P

Parafrase

Bruce Leasure
Saint Paul, MN, USA

Synonyms

Parallelization

Discussion

Parafrase was a successful project allowing experimentation in source-to-source translation of FORTRAN programs. Parafrase was the work of David J. Kuck and his students at the University of Illinois at Urbana-Champaign. It was supported largely by NSF. The name of system is due to Stott Parker. A wide variety of optimization strategies were developed by researchers, and the performance achieved was measured using a broad cross section of applications, across a variety of theoretical and actual machines. Parafrase was successful because it reduced the effort required of the researcher to implement an optimization strategy, and to perform experiments.

Parafrase was structured much like a multi-pass compiler of the same era. The FORTRAN program being compiled was processed one routine at a time. A scanner consumed the source code for a FORTRAN routine and built an equivalent representation in the Internal Language (IL). Then a series of passes that transformed the IL in various ways was run. The researcher would select the passes based on the goals of the research. Then lastly, a FORTRAN code regenerator pass was run. This entire process was repeated for each of the routines in the input program. The FORTRAN program output by Parafrase could be compiled by a FORTRAN compiler and run on any machine.

Because the ultimate goal of Parafrase was to regenerate optimized FORTRAN code, the IL used by Parafrase was considerably different than the IL found in a compiler. The IL used by Parafrase was a relatively straight forward representation of FORTRAN source, with some additional look-aside tables to hold summary information about variables and loop structures. Each pass was required to accept and generate this common IL.

Having an IL that was very close to FORTRAN enabled Parafrase to provide an easily understandable IL dump by simply pretty printing the FORTRAN program that was represented in the data structures. The pretty printer could be invoked by Parafrase at the end of a pass, or called by the pass itself to display intermediate results. This made it easy for the researcher to see what an optimization was doing because the researcher only had to understand FORTRAN, not the details of some unusual IL.

Parafrase also included an IL verification phase that could be enabled to run after any pass to ensure that that pass was following the rules. This was especially useful when identifying which pass was incorrectly processing a FORTRAN routine.

Because all of the passes were required to accept and generate the common IL, the passes could be run in any order. This flexibility allowed the researcher to construct a specialized ordering of the passes to achieve specific goals. Very early in Parafrase's life, the ability to specify the order of the passes in a text file was added. This enabled the researcher to adjust the order of the passes without having to rebuild the executable image of Parafrase.

Parafrase was implemented in PL/I and run on an IBM System/360 before virtual memory. With all of the passes and the limited amount of physical memory available, the executable had to be built using the technique of overlays. By placing each pass in its own overlay, the passes would all share the same memory address space, thus reducing the memory foot print. Of course, it was a little more complicated than that, and the obscure interface to IBM's link editor to build the overlays was understood by only a few people. A build tool was created for Parafrase to automatically

David Padua (ed.), *Encyclopedia of Parallel Computing*, DOI 10.1007/978-0-387-09766-4,
© Springer Science+Business Media, LLC 2011

identify an appropriate overlay structure, to create the appropriate link editor commands, and to build the overlay executable, thus relieving the researcher from understanding the messy details of this process.

When researchers started to use Parafrase on more than just toy programs, it was quickly discovered that a scanner handling ANSI FORTRAN was not nearly strong enough. The various extensions to FORTAN promoted by computer manufacturers had changed common use. The only solution was to add these extensions to the Parafrase scanner, and to extend the IL where needed to include them. Eventually, VAX, CDC, HP, IBM, and other manufactures extensions were implemented. Some syntax extensions were just syntactic sugar, and were translated away by the scanner. Others required unique extensions to the IL.

Another unexpected discovery when processing non-toy programs was that most FORTRAN programs contained hand optimized code targeting a particular machine. For most of the research performed with Parafrase, the ideal input program was one that was not hand optimized.

Consider the case of exploring the impact of vectorization on a program. If the loops in the program that could be vectorized were unrolled by hand in the original code, then the vector length and the reference pattern for each vector operand would be different than if the loops were not unrolled in the original program. Vector lengths would be shorter, and operands would be less likely to be contiguous. On most vector capable machines, these changes would seriously impact performance.

Consequently, a collection of de-optimization passes were written for Parafrase. It was impossible to write de-optimization passes for every hand optimization. There were just too many. Instead, we kept track of how many important loops exhibited a particular hand optimization, and whenever a sufficiently large number of loops exhibited a particular hand optimization, we wrote a de-optimization pass to address the issue.

Parafrase ended up having de-optimization passes and optimization passes that performed inverses of each other for many of the common optimizations. For example: Loop rerolling and loop unrolling, forward substitution and code floating, and loop distribution and loop fusion.

The research efforts supported by Parafrase dealt with optimization techniques for supporting new types of hardware acceleration devices: vector processors, streaming memory systems, cache memory systems, multiple functional units, parallel processors, and memory banks. At the time, the impact of any one of these acceleration devices on the performance of ordinary programs was not well understood.

The optimization techniques to automatically exploit these acceleration devices were not well understood. Parafrase provided a reasonable vehicle to explore optimization techniques.

Parafrase. Table 1 Theses related to Parafrase

Year	Author	Topics
1971	Yoichi Muraoka	Arithmetic expressions, loop dependence testing, wave fronts, scheduling
1975	Steve S.C. Chen	m-th order linear recurrences
1976	Ross A. Towle	Dependence testing, loops with branching, parallel parsing time
1976	Bruce Leasure	Design of Parafrase
1978	Walid Abu Sufah	Virtual memory optimization, name partitioning, loop reindexing
1979	Utpal Banerjee	Data dependence tests for multi-loop programs
1979	David A. Padua	Clustered system loop transforms, loop pipelining, scheduling
1980	Robert H. Kuhn	Vector optimization, decision tree optimization
1982	Michael J. Wolfe	Direction-vector based optimization, recurrences, while loops
1984	Ron G. Cytron	Doacross loop optimization and scheduling
1985	Alex Veidenbaum	Blocks of assignment statements, coarse grain optimization
1986	Constantine Polychronopoulos	Loop coalescing, subscript blocking, static and dynamic scheduling (GSS)

The history of research can be traced from the early 1970s to the mid 1980s through thesis topics, shown in Table 1, and published papers listed in the References. Twelve theses and 11 papers are listed from a larger collection of work on the Parafrase system.

Bibliography

1. Kuck D, Muraoka Y, Chen SC (1972) On the number of operations simultaneously executable in Fortran-like programs and their resulting speedup. IEEE Trans Comput C-21(12):1293–1310
2. Kuck D, Budnik P, Chen SC, Davis E Jr, Han J, Kraska P, Lawrie D, Muraoka Y, Strebendt R, Towle R (1974) Measurements of parallelism in ordinary FORTRAN programs. Computer 7(1):37–46
3. Kuck DJ (1976) Parallel processing of ordinary programs. Adv Comput 15:119–179, Rubinoff M, Yovits MC (eds). Academic Press, New York
4. Abu-Sufah W, Kuck D, Lawrie D (1979) Automatic program transformations for virtual memory computers. Proceedings of the 1979 national computer conference, AFIPS Press, June 1979, pp 969–974
5. Kuck DJ, Padua DA (1979) High-speed multiprocessors and their compilers. Proceedings of the 1979 international conference on parallel processing, Aug 1979, pp 5–16
6. Padua DA, Kuck DJ, Lawrie DH (1980) High-speed multiprocessors and compilation techniques, special issue on parallel processing. IEEE Trans Comput C-29(9):763–776
7. Kuck DJ, Kuhn RH, Leasure B, Wolfe M (1980) The structure of an advanced vectorizer for pipelined processors. Proceedings of COMPSAC 80, The 4th international computer software and applications Conference, Chicago, IL, Oct 1980, pp 709–715
8. Kuck DJ, Kuhn RH, Padua DA, Leasure B, Wolfe M. Dependence graphs and compiler optimizations. Proceedings of the 8th ACM symposium on principles of programming languages (POPL), Williamsburg, VA, Jan 1981, pp 207–218
9. Cytron R, Kuck DJ, Veidenbaum AV (1985) The effect of restructuring compilers on program performance for high-speed computers. In: Duff IS, Reid JK (eds) Special issue of computer physics communications devoted to the proceedings of the conference on vector and parallel processors in computational science II, vol 37 Elsevier Science Publishers B V (North-Holland Physics Publ.), Oxford, England, pp 39–48
10. Lee G, Kruskal CP, Kuck DJ (1985) An empirical study of automatic restructuring of nonnumerical programs for parallel processors. Special issue on parallel processing. IEEE Trans Comput C-34(10):927–933
11. Constantine Polychronopoulos, Kuck D, Padua D (1989) Utilizing multidimensional loop parallelism on large-scale parallel processor systems. IEEE Trans Comput 38(9):1285–1296

Parallel Communication Models

►Bandwidth-Latency Models (BSP, LogP)

Parallel Computing

David J. Kuck
Intel Corporation, Champaign, IL, USA

Definition

Parallel computing covers a broad range of topics, including algorithms and applications, programming languages, operating systems, and computer architecture. Each of these must be specialized to support parallel computing, and all must be designed and implemented coherently to provide highly efficient parallel computations.

Discussion

Introduction and History

All computations transform data – logically and arithmetically – following a series of algorithms. The broad goals of parallel computing are to improve the speed or functional ease with which this is done. Speed and functionality goals can usually be met, but in many practical cases, difficulties arise that present far more complexity than does traditional sequential computing. The following gives an overview of background issues and the current state of parallel computing.

The long and diverse history of computing is intertwined with parallelism. There will be no attempt here to formulate a rigorous definition of parallel computing. Instead, an introduction to parallel computing can be provided by sketching the many uses for parallelism over time. Computer system *performance* has always been the key driver of *hardware parallelism*. On the other hand, *software parallelism* has been driven by both performance and application functionality.

The baseline of parallelism is serial computing, and in the beginning, some computers operated in bit-serial fashion. One bit, the minimal information unit in a computer, was processed at a time. To improve speed, multiple bits were fetched from memory and transformed in parallel, a process that continued up to word-level computation (32 or 64 bits). This required making all data paths in a system – memory, processor, and interconnection – one word wide. Fast parallel algorithms for individual arithmetic operations occupied the research agenda in early computing, into the 1960s.

P

Some early machines used one hardware (HW) technology to implement all parts of a computer, from processor to memory. Over time, as technology advanced and more performance was demanded, distinct technologies were used in different parts of machines – e.g., transistors for processing and magnetic cores for memory. As processor technology speeds increased more quickly than memory, parallel memory units were introduced to match the data rates (i.e., bandwidths measured in bits per second) of processors. As one part of the processor became a performance bottleneck relative to others, multiple units (e.g., 2 adders) were introduced within the processor.

The idea of performing more than one arithmetic operation in parallel is an obvious way to improve performance, and was mentioned by the first computer designer, Charles Babbage (who designed a mechanical digital computer in the 1840s [1, 2]). Eventually, reasonably balanced systems for various types of computation became well understood, and parallelism as above was employed to produce very efficient uniprocessor systems. *Simultaneous operation* on whole words and *overlap* of various operations are the basic principles that support parallelism.

After sufficient HW has been invested to produce a good architecture using various types of low-level parallelism, the clock speed (which determines the time of a computer's most basic steps) can be increased to speed up a system. From the beginning, new device technology allowed faster clocks; from the 1960s on, this and architectural refinements drove system performance. But at the high end of performance demands, the ability to change the internal structure of a system by simply adding low-level parallelism became increasingly difficult and yielded slower growth. Architectural refinements were added to keep systems balanced (e.g., cache memory hierarchies) as technology advances drove processor performance faster than off-chip memories.

Parallel Architectures

System clock speed determines the bandwidth (bits/sec) of various computer elements, and the clock speed depends on the basic technology available. Physical issues underlie technology, including:

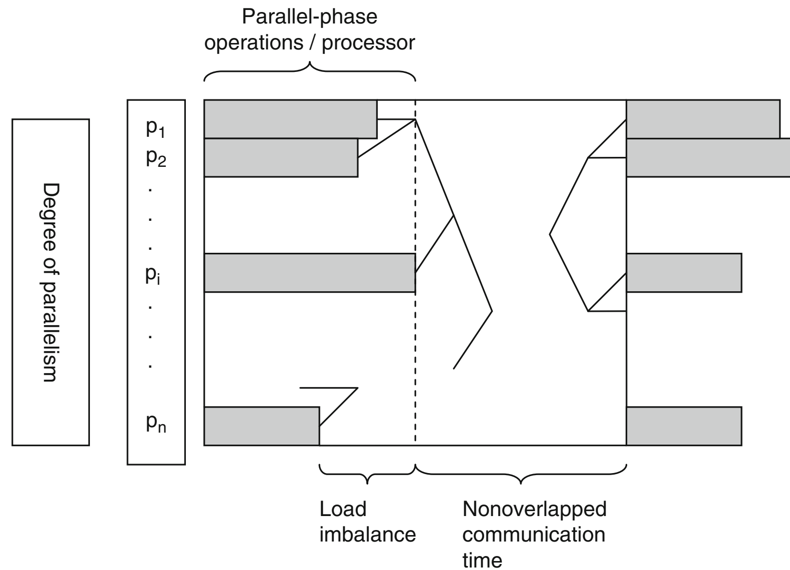
- Transistor density in an integrated circuit limits clock speed
- The speed of light limits data transmission time or latency (measured in seconds)
- Manufacturing costs of HW limit the complexity that can be built into a system

To avoid limits imposed by the first and last of these, parallel processors can be used, each carrying out part of an overall computation. In other words, transistor density increases can be frozen as can the complexity of faster uniprocessor manufacturing, by moving to parallel processing. Although this forces the system size to grow, which in turn exacerbates latency issues, two of the three physical issues are controlled. The number of parallel units being used is often expressed as the *degree of parallelism, n*, see Fig. 1.

Parallelism also raises a new architectural issue: How does one interconnect the processors and the processors to memory? Ideally one would choose direct connection of each unit to all others (a crossbar switch), but this raises severe cost and design issues as the degree of parallelism grows. At the other extreme, the bus can be extended from within each processor to all others, a simple but low-performance idea. Between these extremes, one can choose a fast parallel ring, torus, or shuffle interconnect network pattern, each with many parallel data transmission paths and various performance and manufacturing trade-offs. For details and more references see [1, 3].

From the 1960s through the 1990s, a host of research and commercially available parallel systems were built, exploring a wide variety of architectural trade-offs. At the same time, beginning in the 1960s, several related approaches to computing emerged. Multiprocessing, the connection of whole computers to increase system throughput, began. The distinction can be captured by viewing *parallel computing* as turnaround computing – whose goal is the fast turnaround of one job at a time. In contrast, *multiprocessing* can be viewed as throughput computing – the goal being to complete a large volume of similar transactions quickly.

The third approach, *distributed computing* arose for a totally different reason [4]. Large companies bought multiple computer systems at different sites, and wanted to exchange results among them – e.g., financial information for the whole enterprise – via telephone line communications. Distributed computing in general puts less demand on communication than parallel computing or multiprocessing. By the late 1960s, this idea



Parallel Computing. Fig. 1 Key parallel computation parameters

grew into the ARPANET among government contractors and agencies, and was followed by new concepts that led to the Internet.

At the HW technology level, transistor speeds increased continually and casualties developed – by the early 1990s ECL (emitter coupled logic) could not be pushed faster, and was supplanted by traditionally slower CMOS transistors. By the first decade of the twenty-first century, it became clear that silicon transistors, whose size continued to decrease (Moore's Law [5]) and whose power dissipation density concomitantly increased with clock speed, were hitting a "power wall." The result was that clock speeds could no longer increase, even though densities could continue to grow, allowing more processors on a single semiconductor chip. Thus, parallel computing reached the mainstream, and by 2010 has become ubiquitous.

Parallel Software Concepts

Over the decades of parallel architecture development, research into algorithms and parallel programming tools enjoyed broad growth. However, software (SW) application development is a very diverse subject, and parallel programming has not matured at the rate that parallel HW has arrived in the mainstream of computing. Roughly speaking, parallel application development suffers from all the problems inherent in sequential programming, plus a number of additional problems arising from parallelism.

As is obvious from the release delays, security weaknesses, and other flaws in sequential applications, SW development methods are still maturing. Some decades after sequential computing HW reached its eventual complexity (except perhaps for cache hierarchy management), SW development remains labor intensive, with results that are error prone, and hard to test. Improving parallel SW development tools remains an important activity.

Application Software Development

In 2010, various programming languages are able to capture any given algorithm in many forms. Some algorithms and applications are naturally parallel. For example, little dependence exists among tasks in multiprogramming and distributed computing, and any language that suits their problem domain will do. Other applications contain data and control dependences that *appear* to force sequential execution of program phases. While a continuum of ideas exists to deal with parallel programming, the following breakdown captures some key points, concluding with the unexploited parallelism remaining in useful parallel applications. This section is an introduction; more details appear in the following sections.

Parallelization Methods

1. For some apparently nonparallel algorithms and programs, effective SW procedures lead to good parallel performance:
   (a) Compilers can extract some parallelism from sequential constructs by language-level transforms [6].
   (b) Parallel libraries allow developers to express parts of programs in parallel terms at the language level, without the need to understand the underlying parallel algorithms [7].
   (c) For other program constructs, compilation is too hard and libraries are not available, so a language or algorithm change is necessary for compiler success, forcing the application developer to think through parallelism detail. Three programming models will be discussed in detail below as found in [8–10].

Thus, there are several ways to adjust some programs to be suitable for efficient parallel execution, but in other cases parallelism is harder to obtain.

Parallelization Impediments

2. For the most difficult applications, algorithms and their programs, there are inherent, nonparallelizable issues:
   (a) Data and control dependences arise that cannot practically be removed by a combination of compiler and run-time HW checks. For example, too much branching, or too many pointers or run-time parameter tests exist.
   (b) Architecture-related clashes arise, e.g., there is too much nonoverlapped interprocessor communication or imbalance between the sizes of parallel tasks that waste parallel computation time, see Fig. 1.

In case 2, the best approach is to rethink the algorithms and program, and restructure the program or use entirely different algorithms. These steps can be very difficult and nonintuitive, unless a developer is specifically driven by the parallel architectural constraints imposed, understands parallel languages, and has a deep understanding of the underlying problem and various algorithms for solving it.

Unexploited Parallelism in "Well-Parallelized" Applications

3. Consider the amount of "unexploited" parallelism remaining in an application that is already regarded as well parallelized for a given system. In other words, how far do parallel applications in use deviate from the perfect exploitation of parallelism in their algorithms? This question exposes language and compiler issues. Assume that an algorithm and program contain reduction operations in the form of Eq. 1, which sums the elements a_i of array a into scalar x. If the compiler cannot parallelize this,

$$x = 0; \quad \text{for all } i\ (x = x + a_i) \tag{1}$$

   then unexploited parallelism can be recovered by using a language (extension) that contains a parallel reduction operator. Much language research is directed toward filling such language gaps in ways that are easy to use in specific algorithms or application domains, e.g., games, image processing, and transaction processing.

Figure 1 summarizes several details that determine parallel system performance. The shaded areas represent useful work (their unequal lengths show that not all processor steps take the same amount of time), and the trees represent communication or synchronization, e.g., join/fork or message-passing patterns. Performance increases with the degree of parallelism and parallel-phase operations per processor, which increase the use of parallel resources. Performance decreases due to load imbalance and nonoverlapped communication, which waste resources relative to decreasing elapsed time in Eq. 2.

System Architecture Performance Criteria

Let *computation time* be defined as a measure of the useful processing steps in a source program, i.e., corresponding to necessary steps in the original algorithms and their program counterparts. *Communication time* includes the total HW interconnect time and system SW overhead required in a program and its resulting computation to support the desired computation time.

Our overall objective is to *minimize total elapsed time*, in Eq. 2 (cf. Fig. 1).

$$\begin{aligned}\text{Total elapsed time} &= \text{Computation time} \\ &+ \text{Nonoverlapped communication time}\end{aligned} \tag{2}$$

In general, the diversity of parallel architectures, especially issues arising from major increases in processor count, change some of the inherent needs of algorithms to satisfy the constraints outlined above. Furthermore, each data size to which a given algorithm applies may force algorithm adjustments for machine size. Thus, for a given algorithm type, adaptations to machine parameters may be necessary.

As outlined above, algorithms, languages, run-time libraries, and architecture are closely linked in obtaining top performance. Good parallel performance across all types of problems can usually be obtained by an approach that includes all of these factors.

A universally important performance criterion that follows from Eq. 2 is shown in Eq. 3 (where *communication time* includes overlapped and nonoverlapped time).

$$\frac{\text{Computation time}}{\text{Communication time}} \geq 1 \tag{3}$$

Maintaining this inequality, averaged over time windows throughout a computation, means that the HW design and application SW running may be brought into balance by overlapping communication with meaningful computation.

The ratio can be used as a guideline in system and SW design. If the ratio of Eq. 3 is less than 1, increasing the numerator, while decreasing the denominator can reduce overall time and bring the HW use into balance. For example, if the ratio is less than 1, performance may be improved by using an algorithm with redundant operations that reduces the total data communication time. Such a trade-off causes no problems because the key ratio was too low to begin with.

Parallel Performance

A successfully performing parallel computation must satisfy four basic requirements:

1. Algorithmic parallelism
2. Program parallelism
3. Data size and parallelism
4. Architecture balance and parallelism

These success requirements have many interpretations and interrelations, which are sketched below. They include mathematical issues at the algorithmic level, programming languages that well-suit important application areas, accommodating the data structures and sizes that occur naturally, and then synthesizing these requirements into an architectural design that is well suited to efficient solution of the problems defined by the original applications. Many books discuss various aspects of performance for a range of algorithms and applications, ranging from broad coverage [11] to emphases on nonnumerical [12] to numerical [13] computations.

1. *Algorithmic parallelism*: The algorithms used must avoid dependences among operations that force one step to follow another. The essence of parallelism is to be able to do large numbers of operations simultaneously, and this must begin at the algorithm and data structure level. A simple case is an interactive program that has several sequential algorithms, each of which can be run simultaneously – e.g., a game may use three processors working independently on processing user input, readying the next frame, and displaying the current frame.

A more scalable algorithm that may require any number of processors is one that operates on arrays of many numbers, or files of many records. For example, if computing x_i requires a_i and b_i, $1 \leq i \leq n$, as in Eq. 4, the process can be carried out for any number of values at once. If it requires x_{i-1}, as in Eq. 5, this is a linear recurrence, which

$$\text{for all } i\ (x_i = a_i + b_i) \tag{4}$$

$$x_0 = 0; \quad \text{for all } i\ (x_i = x_{i-1} + a_i) \tag{5}$$

appears to have sequential dependences but can be transformed and computed by reasonable parallel algorithms in $O(\log_2 n)$ steps. On the other hand, nonlinear recurrences, e.g., Eq. 6 ($1 \leq i \leq n$), usually present insurmountable mathematical problems

$$x_{-2} = 2; x_{-1} = 1; x_0 = 1; \quad \text{for all } i \left(x_i = \frac{a_i (x_{i-1})^2 x_{i-2}}{b_i x_{i-3}} \right) \tag{6}$$

for parallelization, so a different algorithm must be found.

2. *Program parallelism*: Given a set of parallel algorithms, expressing them effectively in an appropriate parallel language is relatively straightforward. However, the structure of an algorithm and architecture

will force the choice of language, as the following example illustrates.

Game Example: Consider the many algorithms necessary to support an interactive computer game. At the highest level, communication from the players must be coordinated with the internals of the game. These include a changing scene driven by the game logic as well as modifications made by players' actions. High-quality graphics depends on computationally intense solutions of complex physics equations for mechanical structures, fluid flow, cloth and hair motion, etc. An important consideration for game writers is how to use all of the computing power available to create appealing graphics and game experiences. Finally, each new frame must be composed and displayed. All of this must occur in real time.

Three *programming models* form the core of most programming languages and methods (cf. Fig. 1).

(a) *Message passing* among communicating sequential (or parallel) processes may be the appropriate programming model at the highest level of game SW. Each process runs under independent control of the operating system on one or more processors, with the added feature that processes occasionally may send data to, and receive data from, one another. To keep the program logic valid, the system SW must provide for processes interrupting one another, and later expecting acknowledgements back from other processes. Communication delays are typical of performance problems in message-passing programs. A good exposition of MPI, a popular message-passing language for scalable systems, is found in [9].

(b) The *fork-join parallel* model matches well with the execution of high-level tasks internal to a game. Any number of tasks can be dispatched with fork statements to separate processors, in the form of threads, and when their independent work is finished, they combine in join statements. *Threads* share memory and may be controlled by each application without OS intervention in scheduling. The degree of parallelism is limited by the number of parallel tasks. For a description of OpenMP and its use in shared memory programming, see [8].

(c) *Data parallel languages*, embody a third type of program parallelism. They normally express a great deal of low-level parallelism in the data, e.g., application-specific array oriented languages can be used to express parallelism directly in the form required by an application algorithm. Equation 4 is an example, and may occur in rendering an image for display, where each pixel is computed independently of all others. Another example is movie making, where each frame can be rendered independently of the others. An introduction to Cuda and OpenCL, two recent data parallel languages, is given in [10].

3. *Data size and parallelism*: The degree of parallelism is also limited by data dependence and data size. The number of independent data structures and the size of each are indicators of the degree of available parallelism in a computation. A successful parallel computation requires data locality in that program references stay relatively confined to the data available in each processor – otherwise too much overhead time can be consumed, ruining parallel performance.

The number of *parallel-phase operations per processor* must be sufficient to dominate the communication time per phase, see Fig. 1. Equation 3 can be used to explain the reason why. In parallel algorithms, communication among processors (or processor-memory communication) occurs with some frequency; in the easy case, the frequency is low. But as the frequency grows, so does the communication in any time window. Equation 3 says that as the communication time exceeds computation time, performance can suffer. If there is little data per processor, then the computation time window without communication is expected to be small. Thus a small numerator leads to a large denominator, driving the ratio to violate the inequality.

4. *Architecture balance and parallelism*: The architecture must have a sufficient number of processors, sufficiently fast global memory access, and interprocessor communication of data and control information that allows parallel scalability. As systems grow in degree of parallelism, the memory and communication systems define the success of a design.

Of course, these qualitative statements imply many difficult technical design choices (mentioned earlier) that must provide a good match to the programs and data sets to be run. In systems with low degrees of parallelism, including modern multicore chips, all of the processors generate addresses for a single, shared address space. This *shared-memory* parallelism (SMP) allows them to communicate quickly and simply via memory. But as the degree of parallelism increases and parallelism grows, *distributed-memory* parallel (DMP) systems are used.

Whether DMP clusters of smaller SMP systems occupy a single large computer room, or span different cities, large systems have nonuniform memory access time (NUMA), so called because of the very large range of overheads involved in communicating with local (SMP) and global (DMP) memory. These systems present greater programming challenges, in order to minimize and overlap communication and computation time (recall Eq. 2) and to schedule tasks properly (load balancing) to minimize wasted resources (cf. Fig. 1). The great variety in real-world computations has led designers down many successful paths from the 1980s to the present [14, 15].

Parallel Program Correctness

Correctness is harder to determine for a parallel program than for a sequential program, and fixing bugs is concomitantly much harder. Whole new classes of bugs arise in parallel programs, for various reasons. A major problem that arises at the hardware level is *non-determinism* of execution. Because the system is large and some clocks may be out of sync, slight timing variations may exist from run to run of a given program.

Even if in sync, different interactions with the OS (e.g., on which processor a process is scheduled) or with other applications, or slightly different data-dependent execution paths can cause a program to execute in different ways each time it runs. These nondeterministic issues make error states nearly impossible to capture or recreate. This is obvious for reactive or interactive programs, whose inputs are difficult to recreate exactly (precisely when did a keystroke that triggered a process occur?).

In some parallel programs, synchronization points occur to coordinate the work of distinct processors, e.g., at *join* points and at the end of *data parallel* operations. These are necessary to ensure that the computations on all processors are complete for a given parallel phase of the overall program, before continuing. In some programming languages, message passing is used to send data and synchronize control of various processors. This is common in distributed memory machines. Software developers often find it hard to get the logic of such programs exactly right, given the variety of inputs some programs have, and the fact that multiple developers may have worked on a given section of code.

The logic of some programs may require *critical sections*, in which a data structure is accessed by only one computational process at a time. For example, if work tasks are being chosen from a queue, to assure that each task is assigned to only one processor, the task assignment algorithm would include a critical section. Software *locks* may be used to allow just one process at a time to execute a critical section.

When combinations of the above occur, various anomalies may arise, including program *deadlock*, in which, e.g., two processors halt, each waiting for the other. In what is perhaps the worst case, poorly synchronized programs actually do not halt, but spend so much time waiting for one another and occasionally proceeding, that the effect is to lead naïve developers to confuse a complex correctness bug with a performance problem. The result may be fruitless developer time spent using performance tools and techniques to find a correctness problem, which needs other approaches.

Future

The era of ubiquitous parallel processing hardware has arrived, but the development of parallel application SW is lagging. In 2010, single chips contain eight shared memory processors (or "cores"). Distributed memory clusters of these, containing from 32 to over 100 K processors are the state of the art in server computers. The use of clustered systems ranges from scientific research and engineering design, to Internet search engine support.

While their clock speeds are frozen, the core count in future chips will continue to provide more performance, based on better architecture, parallel applications algorithms, and SW. Computers based on new technologies and principles (e.g., quantum or biological

principles) could change all of this in some application areas, but it is most likely that parallel SW and architectural issues will continue to dominate much of system design for many decades.

There is a seldom mentioned issue in parallelism. After the challenges of getting all current applications well parallelized are met, how does performance continue to grow? There are only two ways that more parallel processors can produce more performance: larger data sets, or more complexity in algorithms for today's applications. Both of these factors have driven much of computing in the past, but clearly there are limitations for some applications in the future. Those that cannot grow, will be frozen at a performance level, which is fine in some areas. However, application developers will be forced to work harder in the future to exploit parallelism than they did in the past, and new parallel applications will continue to arise.

This leads to a host of new and continuing problems to be solved in order to allow parallelism to continue to provide performance benefits. Parallel algorithm research will continue to provide important basic parallelism. SW researchers will design new development tools to aid the above, work on languages that prevent the commission of serious bug errors, and drive toward hardware support for easier ways to write correct programs or tools to aid in debugging. Architecture and HW research will provide faster systems in accordance with Eqs. 1 and 2. All of these efforts tend to increase the degree of parallelism while reducing parallel computation time; increasing the width and shrinking the length of Fig. 1.

Bibliography

1. Kuck DJ (1978) The structure of computers and computations. Wiley, New York
2. Randell B (1982) The origins of digital computers. Springer, New York
3. Hennessy J, Patterson DA (1996) Computer architecture: a quantitative approach, 2nd edn. Morgan Kaufmann, San Francisco
4. Jia W, Zhou W (2005) Distributed network systems: from concepts to implementations. Springer, New York
5. Noyce RN (1977) Microelectronics. Sci Am 273(3):63–69
6. Kennedy K, Allen R (2002) Optimizing compilers for modern architectures. Morgan-Kaufmann, San Francisco
7. Reinders J (2007) Intel threading building blocks. O'Reilley Media, Sebastapol
8. Chapman B, Jost G, van der Pas R (2008) Using open MP: portable shared memory parallel programming. O'Reilley Media, Sebastapol
9. Gropp W, Lusk E, Skjellum A (1999) Using MPI: portable parallel programming with the message-passing interface, 2nd edn. MIT, Cambridge
10. Kirk D, Hwu W-m (2010) Programming massively parallel processors. Morgan Kaufmann, San Francisco
11. Akl SG (1989) The design and analysis of parallel algorithms. Prentice Hall, Englewood Cliffs
12. Grama A (2003) Introduction to parallel computing. Addison Wesley, Reading
13. Trobec R, Vajtersic M, Zinterhof P (eds) (2009) Parallel computing: numerics, applications, and trends. Springer, New York
14. Kuck DJ (1996) High performance computing. Oxford University Press, New York
15. Culler DE, Singh JP, Gupta A (1999) Parallel computer architecture. Morgan Kaufmann, San Francisco

Parallel I/O Library (PIO)

►Community Climate System Model

Parallel Ocean Program (POP)

►Community Climate System Model

Parallel Operating System

►Operating System Strategies

Parallel Prefix Algorithms

►Reduce and Scan
►Scan for Distributed Memory, Message-Passing Systems

Parallel Prefix Sums

►Reduce and Scan
►Scan for Distributed Memory, Message-Passing Systems

Parallel Random Access Machines (PRAM)

►PRAM (Parallel Random Access Machines)

Parallel Skeletons

Sergei Gorlatch[1], Murray Cole[2]
[1]Westfälische Wilhelms-Universität Münster, Münster, Germany
[2]University of Edinburgh, Edinburgh, UK

Synonyms

Algorithmic skeletons

Definition

A parallel skeleton is a programming construct (or a function in a library), which abstracts a pattern of parallel computation and interaction. To use a skeleton, the programmer must provide the code and type definitions for various application-specific operations, usually expressed sequentially. The skeleton implementation takes responsibility for composing these operations with control and interaction code in order to effect the specified computation, in parallel, as efficiently as possible. Abstracting from parallelism in this way can greatly simplify and systematize the development of parallel programs and assist in their cost-modeling, transformation, and optimization. Because of their high level of abstraction, skeletons have a natural affinity with the concepts of higher-order function from functional programming and of templates and generics from object-oriented programming, and many concrete skeleton systems exploit these mechanisms.

Discussion

Traditional tools for parallel programming have adopted an approach in which the programmer is required to *micromanage* the interactions between concurrent activities, using mechanisms appropriate to the underlying model. For example, the core of MPI is based around point-to-point exchange of data between processes, with a variety of synchronization modes. Similarly, threading libraries are based around primitives for locking, condition synchronization, atomicity, and so on. This approach is highly flexible, allowing the most intricate of interactions to be expressed. However, in reality many parallel algorithms and applications follow well-understood patterns, either monolithically or in composition. For example, image processing algorithms can often be described as *pipelines*. Similarly, *parameter sweep* applications present a parallel scheduling challenge which is orthogonal to the detail of the application and parameter space.

Programming with Skeletons

The term *algorithmic skeleton* stems from the observation that many parallel applications share common internal interaction patterns. The parallel skeleton approach proposes that such patterns be abstracted as programming language constructs or library operations, in which the implementation is responsible for implicitly providing the control "skeleton," leaving the programmer to describe the application-specific operations that specialize its behavior to solve a particular problem. For example, a pipeline skeleton would require the programmer to describe the computation performed by each stage, while the skeleton would be responsible for scheduling stages to processors, communication, stage replication, and so on. Similarly, in a parameter sweep skeleton, the programmer would be required to provide code for the individual experiment and the required parameter range. The skeleton would decide upon (and perhaps dynamically adjust) the number of workers to use, the granularity of distributed work packages, communication mechanisms, and fault-tolerance. At a more abstract level, a divide-and-conquer skeleton would require the programmer to specify the operations used to divide a problem, to combine subsolutions, to decide whether a problem is (appropriately) divisible, and to solve indivisible problems directly. The skeleton would take on all other responsibilities, from whether to use parallelism at all, to details of dynamic scheduling, granularity, and interaction.

Thus, in contrast to the micromanagement of traditional approaches, skeletons offer the possibility of *macromanagement* – by selection of skeletons, the programmer conveys macro-properties of the intended computation. This is clearly attractive, provided that the skeletons offered are sufficiently comprehensive collectively, while being sufficiently optimizable individually and in composition. Sometimes a skeleton may

have several implementations, each geared to a particular parallel architecture, for example, distributed- or shared-memory, multithreaded, etc. Such customization has the potential for achieving high performance, portable across various target machines.

Application programmers gain from abstraction, which hides much of the complexity of managing massive parallelism. They are provided with a set of basic abstract skeletons, whose parallel implementations have a well-understood behavior and predictable efficiency. To express an application in terms of skeletons is usually simpler than developing a low-level parallel program for it.

This high-level approach changes the program design process in several ways. First, it liberates the user from the practically unmanageable task of making the right design decisions based on numerous, mutually influencing low-level details of a particular application and a particular machine. Second, by providing standard implementations, it increases confidence in the correctness of the target programs, for which traditional debugging is too hard to be practical on massively parallel machines. Third, it offers predictability instead of an *a posteriori* approach to performance evaluation, in which a laboriously developed parallel program may have to be abandoned because of inadequate efficiency. Fourth, it provides semantically sound methods for program composition and refinement, which open up new perspectives in software engineering (in particular, for reusability). And last but not least, abstraction, that is, going from the specific to the general, gives new insights into the basic principles of parallel programming.

An important feature of the skeleton-based methodology is that the underlying formal framework remains largely invisible to application programmers. The programmers are given a set of methods for instantiating, composing, and implementing diverse skeletons, but the development of these methods is delegated to the community of implementers.

In order to understand the spectrum of skeleton programming research, it is helpful to distinguish between skeletons that are predominantly data-parallel in nature (with an emphasis on transformational approaches), skeletons that are predominantly task-parallel or related to algorithmic classes, and the concrete skeleton-based systems that have embedded these concepts in real frameworks.

Data-Parallel Skeletons and Transformational Programming

The formal background for data-parallel skeletons can be built in the functional setting of the Bird–Meertens formalism (BMF), in which skeletons are viewed as higher-order functions (functionals) on regular bulk data structures such as lists, arrays, and trees.

The simplest – and at the same time the "most parallel" – functional is *map*, which applies a unary function f to each element of a list, that is,

$$map\ f\ [x_1, x_2, \ldots, x_n] = [f\ x_1, f\ x_2, \ldots, f\ x_n] \qquad (1)$$

Map has the following natural data-parallel interpretation: each processor of a parallel machine computes function f on the piece of data residing in that processor, in parallel with the computations performed in all other processors.

There are also the functionals *red* (reduction) and *scan* (parallel prefix), each with an associative operator $\oplus$ as parameter:

$$red\ (\oplus)\ [x_1, x_2, \ldots, x_n] = x_1 \oplus x_2 \oplus \ldots \oplus x_n \qquad (2)$$

$$scan\ (\oplus)\ [x_1, x_2, \ldots, x_n] = [\,x_1,\ x_1 \oplus x_2,\ \ldots,\ x_1 \oplus \ldots \oplus x_n\,] \qquad (3)$$

Reduction can be computed in parallel in a tree-like manner with logarithmic time complexity, owing to the associativity of the base operation. There are also parallel algorithms for computing the scan functional with logarithmic time complexity, despite an apparently sequential data dependence between the elements of the resulting list.

Individual functions are composed in BMF by means of backward functional composition $\circ$, such that $(f \circ g)\, x = f\,(g\, x)$, which represents the sequential execution order on (parallel) stages.

Special functions, called *homomorphisms*, possess the common property of being well-parallelizable in a data-parallel manner.

Definition 1 (**List Homomorphism**) *A function h on lists is called a* homomorphism *with combine operation* $\circledast$*, iff for arbitrary lists* x, y*:*

$$h\,(x +\!\!\!+\, y) = (h\, x) \circledast (h\, y) \qquad (4)$$

Definition 1 describes a class of functions, operation $\circledast$ being a parameter, which is why it can be viewed as

defining a skeleton. Both map and reduction can obviously be obtained by an appropriate instantiation of this skeleton.

The key property of homomorphisms is given by the following theorem:

Theorem 1 (Factorization) *A function h on lists is a homomorphism with combine operation ⊛, iff it can be factorised as follows:*

$$h = red(\circledast) \circ map\ \phi \tag{5}$$

where $\phi\, a \;=\; h\,[a]$.

In this theorem, homomorphism has one more parameter beside ⊛, namely function ϕ. The practical importance of the theorem lies in the fact that the right-hand side of the equation (5) is a good candidate for parallel implementation. This term consists of two stages. In the first stage, function ϕ is applied in parallel on each processor (*map* functional). The second stage constructs the end result from the partial results in the processors by applying the *red* functional. Therefore, if a given problem can be expressed as a homomorphism instance then this problem can be solved in a standard manner as two consecutive parallel stages – map and reduction.

The standard two-stage implementation (5) may be time-optimal, but only under an assumption that makes it impractical: the required number of processors must grow linearly with the size of the data. A more practical approach is to consider a bounded number p of processors, with a data block assigned to each of them. Let $[\alpha]_p$ denote the type of lists of length p, and subscript functions defined on such lists with p, for example, map_p. The partitioning of an arbitrary list into p sublists, called *blocks*, is done by the *distribution function*, $dist\,(p)\;:\;[\alpha]\;\rightarrow\;[\,[\alpha]\,]_p$. The following obvious equality relates distribution to its inverse, flattening: $red\,(+\!\!+)\;\circ\;dist\,(p)\;=\;id$.

Theorem 2 (Promotion) *If h is a homomorphism w.r.t. ⊛, then*

$$h \circledast red\,(+\!\!+) \;=\; red\,(\circledast) \circ map\ h \tag{6}$$

This general result about homomorphisms is useful for parallelisation via data partitioning: from (6), a standard distributed implementation of a homomorphism h on p processors follows:

$$h \;=\; red\,(\circledast) \;\circ\; map_p\, h \;\circ\; dist\,(p) \tag{7}$$

Sometimes, it can be assumed that data is distributed in advance: either the distribution is taken care of by the operating system, or the distributed data are produced and consumed by other stages of a larger application.

The development of programs using skeletons differs fundamentally from the traditional process of parallel programming. Skeletons are amenable to formal transformation, that is, the rewriting of programs in the course of development, while ensuring preservation of the program's semantics. The transformational design process starts by formulating an initial version of the program in terms of the available set of skeletons. This initial version is usually relatively simple and its correctness is obvious, but its performance may be far from optimal. Program transformation rules are then applied to improve performance or other desirable properties of the program. The rules applied are semantics-preserving, guaranteeing the correctness of the improved program with respect to the initial version. Once rules for skeletons have been established (and proved by, say, induction), they can be used in different contexts of the skeletons' use without having to be reproved.

For example, in the SAT programming methodology [10] based on skeletons and collective operations of MPI, a program is a sequence of stages that are either a computation or a collective communication. The developer can estimate the impact of every single transformation on the target program's performance. The approach is based on reasoning about how individual stages can be composed into a complete program, with the ultimate goal of systematically finding the best composition. The following example of a transformation rule from [8] is expressed in a simplified C+MPI notation. It states that, if binary operators ⊗ and ⊕ are associative and ⊗ distributes over ⊕, then the following transformation of a composition of the collective operations scan and reduction is applicable:

```
[ MPI_Scan (⊗);
  MPI_Reduce (⊕);      ⟹
```

```
Make_pair;
MPI_Reduce (f(⊗,⊕));
if my_pid==ROOT then Take_first;
```
(8)

Here, the functions `Make_pair` and `Take_first` implement simple data arrangements that are executed locally in the processes, that is, without interprocess communication. The binary operator $f(\otimes,\oplus)$ on the right-hand side is built using the operators from the left-hand side of the transformation.

Rule (8) and other, similar transformation rules have the following important properties: (1) they are formulated and proved formally as mathematical theorems; (2) they are parameterized in one or more operators, for example, $\oplus$ and $\otimes$, and are therefore usable for a wide variety of applications; (3) they are valid for all possible implementations of the collective operations involved; (4) they can be applied independently of the parallel target architecture.

Task- and Algorithm-Oriented Skeletons

In contrast to the data-parallel style discussed in the preceding section, there are many instances of skeletons in which the abstraction is best understood by reference to an encapsulated parallel control structure and/or algorithmic paradigm. These can be examined along a number of dimensions, including the linguistic framework within which the skeletons are embedded, the degree of flexibility and control provided through the API, the complexity of the underlying implementation framework and the range of intended target architectures.

While *map* is the simplest data-parallel skeleton, *farm* can be viewed as the simplest task-parallel skeleton. Indeed, in its most straightforward form, *farm* is effectively equivalent to *map*, calling for some operation to be applied independently to each component of a bulk data structure. More subtly, the use of *farm* often carries the implication that the execution cost of these applications is unpredictable and variable, and therefore that some form of dynamic scheduling will be appropriate – the programmer is providing a high-level, application-specific hint to assist the implementation. Typical *farm* implementations will employ centralized or decentralized master–worker approaches, with internal optimizations which try to find an appropriate number of workers and an appropriate granularity of task distribution (trading interaction overhead against load balance). From the programmer's perspective, all that must be provided is code for the operation to be applied to each task, and a source of tasks, which could be a data structure such as an array or a stream emerging from some other part of the program. The *bag-of-tasks* skeleton extends the simple *farm* with a facility for generating new tasks dynamically.

Pipeline skeletons capture the pattern in which a stream of data is processed by a sequence of "stages," with performance derived from parallelism both between stages, and where applicable, within stages. Even such a simple structure allows considerable flexibility in both API design and internal optimization. For example, the simplest pipeline specification might dictate that each item in the stream is processed by each stage, that each such operation produces exactly one result, and that all stages are stateless. For such a pipeline, the programmer is required to provide a sequential function corresponding to each stage. More flexible APIs may admit the possibility of stateful stages, of stages in which the relationship between inputs and outputs is no longer one-for-one (e.g., filter stages, source, and sink stages), of bypassing stages under certain circumstances, and even of controlled sharing of state between stages. From the implementation perspective, internal decisions include the selection of the number of implementing processes or threads, allocation of operations to processes or threads, whether statically or dynamically, and correct choice of synchronization mechanism.

The *divide-and-conquer* paradigm underpins many algorithms: the initial problem is divided into a number of sub-instances of the same problem to be solved recursively, with subsolutions finally combined to "conquer" the original problem. For the situation in which the sub-instances may be solved independently the opportunities for parallelism are clear. Within this context, there is considerable scope to explore a skeleton design space in which constraints in the API are traded against performance optimizations within the implementation. A very generic API would simply require the programmer to provide operations for "divide" and "conquer," together with a test determining whether an

instance should be solved recursively, and a direct solution method for those instances failing this test. Less-flexible APIs, could, for example, require the degree of division to be fixed (every call of divide returns the same number of sub problems), the depth of recursion to be uniform across all branches of divide tree, or even for some aspects of the "divide" or "conquer" operations to be structurally constrained. Each such constraint provides useful information, which can be exploited within the implementation.

Wavefront computations are to be found at the core of a diverse set of applications, ranging from numerical solvers in particle physics to dynamic programming approaches in bioinformatics. From the application perspective, a multidimensional table is computed from initial boundary information and a simple stencil that defines each entry as a function of neighboring entries. Constraints on the form of the stencil allow the table to be generated with a "wavefront" of computations, flowing from the known entries to the unknown, with consequent scope for parallelism. A *wavefront* skeleton may require the programmer to describe the stencil and the operation performed within it, leaving the implementation to determine and schedule an appropriate level of parallelism. As with *divide-and-conquer*, specific *wavefront* skeleton APIs may impose further constraints on the form of these components.

Branch-and-bound is an algorithmic technique for searching optimization spaces, with built-in pruning heuristics, and scope for parallelization of the search. Efficient parallelization is difficult, since although results are ultimately deterministic, the success of pruning is highly sensitive to the order in which points in the space are examined. This makes it both promising and challenging to the skeleton designer. With respect to the API, a *branch-and-bound* skeleton can be characterized by small number of parameters, capturing operations for generating new points in the search space, bounding the value of such a point, comparing bounds, and determining whether a point corresponds to a solution.

Skeleton-Based Systems

Past and ongoing research projects have embedded selections of the above skeletons into a range of programming frameworks, targeting a range of platforms. The P3L language [15] was a notable early example in which skeletons became first-class language constructs, together with sequential "skeletons" for iteration and composition. Subsequent projects within the Pisa group have taken similar skeletons into object-oriented contexts, with a focus on distributed and Grid implementations. In contrast, Muesli [3] allows data-parallel skeletons to be composed and called from within a layer which itself composes task-parallel skeletons, all within a C++ template-based library. Several systems have taken a functional approach. Hermann's HDC [12] focuses exclusively on a collection of divide-and-conquer skeletons within Haskell, and implemented on top of MPI, while the Eden skeleton library [14] and related work [11] have implemented skeletons on top of both implicitly and explicitly parallel functional languages. The COPS project [2] presents a layered interface in which expert programmers can also have access to the implementation of the provided "templates," all embedded in Java with both threaded and distributed RMI-based implementations. The SkeTo project offers a BMF-based collection of data-parallel skeletons for lists, matrices, and trees, implemented in C with MPI. Domain-specific skeletons are represented within the Mallba project [1] (focusing on combinatorial search) and Skipper and QUAFF projects (image processing). Higher-Order Components (HOCs) [5], Enhance [19], and Aspara [6] extend the idea of skeletons toward the area of distributed computing and Grids. HOCs implement generic parallel and distributed processing patterns, together with the required middleware support and are offered to the user via a high-level service interface. Users only have to provide the application-specific pieces of their programs as parameters, while low-level implementation details such as transfer of data across the grid are handled by the HOC implementations. HOCs have become an optional extension of the popular Globus middleware for Grids.

Skeleton principles are also evident in a number of other emerging parallel programming tools. Most notably, the MapReduce paradigm (related to, but distinct from the similarly named BMF skeletons) represents a pattern common to many applications in Google's infrastructure. Emphasis in the original implementation was placed on load balancing and fault-tolerance within an unreliable massively parallel computational resource, a task strongly facilitated by the structurally constraining nature of the skeleton. Thain's Cloud Computing Abstractions [18] implement

a range of distributed patterns with applications in Biometrics and Genomics. As with MapReduce, these are trivial to implement sequentially, and relatively straightforward on a reliable, homogeneous parallel architecture. The strength of the approach becomes apparent when ported to less predictable (or reliable) targets, with no additional effort on the part of the programmer. Finally, skeletal approaches can also be discerned in MPI's collective operations [9], where `MPI_Reduce` and `MPI_Scan` are parameterized by operations as well as data, and Intel's Threading Building Blocks [17], which in particular includes a *pipeline* skeleton.

Related Entries

►Collective Communication
►Eden
►Glasgow Parallel Haskell (GpH)
►NESL
►Reduce and Scan
►Scan for Distributed Memory, Message-Passing Systems

Bibliographic Notes and Further Reading

The term "algorithmic skeleton" was originally introduced by Cole [4]. A considerable body of work now exists. Helpful snapshots can be obtained by consulting the 2003 book edited by Rabhi and Gorlatch [16], the September 2006 special edition of the journal Parallel Computing [13], and the recent survey by Gonzalez-Velez and Leyton [7].

Bibliography

1. Alba E, Almeida F, Blesa MJ, Cabeza J, Cotta C, Díaz M, Dorta I, Gabarró J, León C, Luna J, Moreno LM, Pablos C, Petit J, Rojas A, Xhafa F (2002) MALLBA: a library of skeletons for combinatorial optimisation. Euro-Par '02: Proceedings of the 8th International Euro-Par Conference on Parallel Processing, Springer, London, pp 927–932
2. Anvik J, Schaeffer J, Szafron D, Tan K (2003) Why not use a pattern-based parallel programming system? Euro-Par, Klagenfurt Austria, pp 81–86
3. Ciechanowicz P, Poldner M, Kuchen H (2009) The Münster skeleton library muesli – a comprehensive overview. Technical report, University of Münster
4. Cole M (1989) Algorithmic skeletons: structured management of parallel computation. MIT Press, Cambridge
5. Dünnweber J, Gorlatch S (2009) Higher-order components for grid programming: making grids more usable. Springer, New York
6. Gonzalez-Velez H, Cole M (2010) Adaptive structured parallelism for distributed heterogeneous architectures: A methodological approach with pipelines and farms. Concurrency Comput Pract Exp 22(15):2073–2094
7. Gonzalez-Velez H, Leyton M (2010) A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. Software Pract Exp 40:1135–1160
8. Gorlatch S (2000) Towards formally-based design of message passing programs. IEEE Trans Softw Eng 26(3):276–288
9. Gorlatch S (2004) Send-receive considered harmful: Myths and realities of message passing. ACM TOPLAS 26(1):47–56
10. Gorlatch S, Lengauer C (2000) Abstraction and performance in the design of parallel programs: an overview of the SAT approach. Acta Inf 36(9/10):761–803
11. Hammond K, Berthold J, Loogen R (2003) Automatic skeletons in template Haskell. Parallel Process Lett 13(3):413–424
12. Herrmann CA, Lengauer C (2000) HDC: a higher-order language for divide-and-conquer. Parallel Process Lett 10(2/3):239–250
13. Kuchen H, Cole M (eds) (2006) Algorithmic skeletons. Parallel Comput 32(7/8):604–615
14. Loogen R, Ortega Y, Pena R, Priebe S, Rubio F (2003) Parallelism abstractions in Eden. In: Rabhi F, Gorlatch S (eds) Patterns and skeletons for parallel and distributed computing. Springer, London, pp 95–128
15. Pelagatti S (1997) Structured development of parallel programs. Taylor & Francis, London
16. Rabhi FA, Gorlatch S (eds) Patterns and skeletons for parallel and distributed computing. Springer, London, ISBN 1-85233-506-8
17. Reinders J (2007) Intel threading building blocks. O'Reilly, Sebastopol CA
18. Thain D, Moretti C (2009) Abstractions for cloud computing with condor. In: Ahson S, Ilyas M (eds) Cloud computing and software services. CRC Press, Boca Raton
19. Yaikhom G, Cole M, Gilmore S, Hillston J (2007) A structural approach for modelling performance of systems using skeletons. Electr Notes Theor Comput Sci 190(3):167–183

Parallel Tools Platform

Gregory R. Watson
IBM, Yorktown Heights, NY, USA

Synonyms

PTP

Definition

The Parallel Tools Platform (PTP) is an integrated development environment (IDE) for developing parallel programs using the C, C++, Fortran, and Unified Parallel C (UPC) languages. PTP builds on the Eclipse platform

by adding features such as advanced help, static analysis, parallel debugging, remote projects, and remote launching and monitoring. It also provides a framework for integrating other non-Eclipse tools into the Eclipse platform. The Parallel Tools Platform is not restricted to any particular programming model, but most tools to date are designed to support course-grained parallelism.

Discussion

Challenges

The challenges facing a developer writing parallel programs largely fall into the following three areas:

Coding: The programmer must translate the mathematical model of a problem into an algorithm that is implemented using a particular programming language and parallel programming model. This process of translation involves a large degree of effort, since there is often a significant mismatch between the model and an implementation that fully exploits the available parallelism.
Testing and debugging: Once an application program has been created, its correctness and accuracy must be validated. This involves a cycle of comparing the program output to known values using a variety of input data sets in order to verify that the results are correct. Coding errors, logic errors, and nondeterministic behavior all have to be addressed during this process, typically by employing an interactive debugger that allows program state to be inspected at key points during execution.
Performance optimization: A correctly functioning program may still not optimally utilize the available hardware resources. A variety of tools are typically employed to examine program behavior in order to identify bottlenecks and other performance related issues. These tools usually collect relevant information about the program execution, either directly, or in an aggregated form, and provide a means of analyzing and interpreting the data in order to identify actions that can be taken to improve performance. This is important for many situations when the speed of execution is time critical, such as in weather applications.

These activities are made significantly more difficult when applications are being developed for architectures that employ concurrent execution (or parallelism) in order to achieve extremely high levels of performance. Parallel programming introduces many additional complexities that impact on all aspects of the development process. The nature of these complexities depends on the particular programming model employed, but includes issues such as deciding how the algorithm or data is to be partitioned, the need to deal with *communication* in addition to computation, and the inherent nondeterminism introduced by concurrent execution.

Productivity

The quest to maximize productivity has received a much greater emphasis in recent years, where productivity is defined as both the speed and ease at which a correctly functioning application can be *developed* combined with the level of *performance* achieved by fully exploiting the available parallelism. Improving productivity is therefore dependent on a wide range of factors. One approach for improving the application development process, and that has seen wide success across a range of computing disciplines, is the use of an integrated development environment (IDE).

An IDE combines a core set of tools into a single package that provides a consistent user interface across a variety of systems and architectures. Many IDEs also provide a plug-in capability so that the core set of tools can be extended as new tools are developed or extra functionality is required. IDEs tend to improve developer productivity because the required tools are always available regardless of the environment, and also because they are integrated together, tools can share data and other information in order to streamline the developer workflow. Scientific computing is one of the few areas that have largely shunned the use of IDEs. Recently, however, the benefits of using an IDE for developing scientific programs have become more apparent to the community. This is probably because the scope of the programming task for the new generation of supercomputers is looking increasingly daunting, combined with an influx of new developers who have already had exposure to, and understand the advantages of, IDEs.

Productive Parallel Programming

The goal of the Parallel Tools Platform is to address the challenges facing developers of parallel programs in a

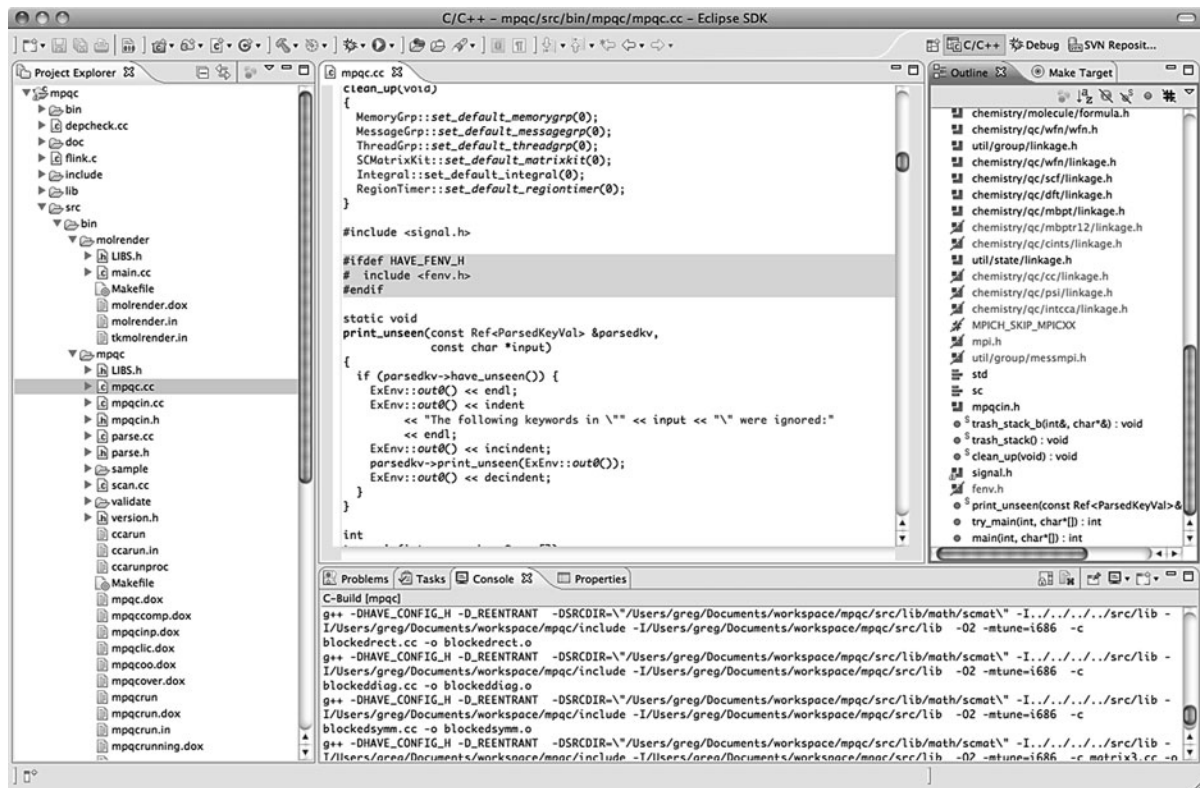
manner that leads to improved productivity. The strategy employed by PTP to achieve this is to combine the productivity enhancements inherent in an integrated development environment with a number of key tools that specifically target the parallel application developers' capacity to deliver correct applications that are optimized for their target platforms.

C, C++, Fortran, and UPC Programming in Eclipse

The Eclipse platform is an open source, vendor neutral, portable, extensible platform for tool integration. It employs a plug-in architecture that allows tools to be integrated directly with the platform, and provides a range of core functionality that is suited to a variety of application development activities. This core functionality includes support for multi-developer projects, an integrated help system, multiple language support, project resource management, advanced editing features, incremental builds, and a range of other features. Eclipse functionality is extensible using the plug-in mechanism, and there is a rich community of developers and a large number of both commercial and open source plug-ins available. The IDE is also highly portable, so it is available on a wide variety of hardware platforms.

The main interface provided by Eclipse is the *workbench* shown in Fig. 1. The workbench arranges *views* (text editors, project explorers, consoles, and other user interface components) into groupings known as *perspectives*. There is typically a perspective available for particular types of activities (coding, debugging, etc.) Eclipse also permits views to be shared between different perspectives. Along with a traditional menu bar, the workbench provides a toolbar containing a variety of actions that are associated with the currently active perspective. There is also a status bar and a range of other user interface decorations. Individual views may also have toolbars and menus that vary depending on context.

Eclipse provides a number of advanced coding and editing features that are common across any language



Parallel Tools Platform. Fig. 1 The eclipse workbench showing the C/C++ development perspective

that it supports. Many of these features rely on core functionality that is provided as standard in Eclipse, but is typically not available in other development environments. Such features include:

Context-sensitive help: In order to provide user assistance that is targeted to a particular context, this feature can deliver help documentation on demand (for example, when the user hits a key) or automatically display information such as prototype definitions or code snippets that pop up when the cursor is placed over a particular piece of text.
Searching: In addition to the usual string or pattern matching searches available in many editors, Eclipse allows searching on language elements (such as types, variables, functions etc.), the ability to limit searches to contexts (such as declarations, references, etc.), and to specify the scope of the search (such as a subset of the files in a project).
Content assistance: Reducing the amount of typing that a developer is required to do can be a powerful means of improving productivity. By inferring what the developer will type based on context and scope information, Eclipse's *code completion* makes suggestions using the first few letters of the language element currently being entered (e.g., a function name, variable name, etc.), via either manual or automatic activation. Frequently used snippets of code can also be saved as *code templates*. These templates can then be inserted into the text according to the current scope in the same manner as other code completions.
Refactoring: Improving code by changing its structure without changing behavior is a common, but error-prone, task in programming. By providing a range of common but sophisticated refactorings that can be automatically performed, Eclipse is able to prevent many of the errors that are introduced by manual changes. Typical refactorings include the ability to rename language elements, such as variables or functions, extract sections of code into a method or function, and automatically declare an interface, as well as many others.
Wizards: Another method of reducing the amount of repetitive work is to automate many of the common tasks that a developer must undertake. Eclipse provides a variety of wizards for activities such as creating new projects, classes, or files, importing projects into Eclipse, exporting projects from Eclipse, and project conversion.

In addition to these powerful features, Eclipse provides support for application building, either using project supplied build scripts (for example Makefiles), or using an internal build system to track file dependencies and manage external tools such as compliers and linkers. The build support is also able to process errors and warnings generated by external tools, and map these back to messages displayed in the user interface, and markers that are inserted into the editor views.

The Eclipse C/C++ *Development Tools (CDT)* provide C, C++, and UPC language support, while the Parallel Tools Platform *Photran* feature adds Fortran support to Eclipse. Support for a wide range of other languages also available through a variety of Eclipse and third-party plug-ins.

Code and Static Analysis for Parallel Programs

In addition to the advanced editing features that are a standard part of Eclipse, PTP adds a range of enhanced features that are specifically targeted toward parallel programming.

The first of these provides additional high-level help documentation for language elements used by common programming models, such as the message passing interface (MPI), the OpenMP Application Programming Interface, IBM's Low-level messaging Application Programming Interface (LAPI), and UPC. This documentation can be accessed using Eclipse's standard context sensitive help feature during coding, and will display prototype information along with a detailed description of the element.

Another feature provided by PTP is cataloging and navigation of MPI application programming interfaces and OpenMP pragma statements in a parallel program. This enables the developer to see the elements that have been used in the program at a single glance, and to navigate to the source code line that contains a selected element.

PTP also includes a Barrier Analysis tool that provides a more sophisticated analysis of MPI programs. This tool locates all calls to MPI *barrier* functions (synchronization points) in a program, and computes logical paths through the program to determine if it

is possible for a deadlock to occur. This can happen because all tasks must synchronize on an MPI barrier, but logic errors may enable one or more tasks to miss execution of the barrier. The advantage of this type of analysis is that it can be performed without the need to execute the program.

Launching and Monitoring Parallel Programs

One hindrance to developer productivity is the seemingly basic ability to easily launch an application on a target parallel system. Most high-performance computing systems employ complex environments to monitor and control application launching. Many of these systems are also highly centralized assets that are tightly controlled, and they will typically restrict direct user access using a batch scheduler system. A developer's ability to interact with these systems will therefore be limited by their ability to deal with the additional complexity that this introduces. This is particularly the case where programs are being ported from one system to another, or where the developer has access to multiple systems and architectures.

PTP's approach to this issue is to provide a single, uniform interface that allows the developer to launch and monitor applications on a wide range of systems, without needing to understand the intricacies of each particular system. This is achieved through two features:

Parallel runtime perspective: This perspective provides a number of views that give the developer a snapshot of system activity at any particular time. Importantly, the views are independent of the type of system that is being targeted, so the developer does not need to be concerned with the underlying system details. The perspective is comprised of the *Resource Manager View*, which lists the systems that are available to the developer for launching and debugging applications, the *Machines View*, which shows the status of the system and its computing elements, and the *Jobs View*, which lists pending, running, and completed jobs on the system. A console is also available to display the program-generated output.
Parallel application run configuration: This utilizes the standard Eclipse *Run Configuration* dialogs, which are used to enter all the parameters necessary for a successful program launch. Run configurations are automatically saved so they can be easily used to relaunch an application during testing or debugging activities. PTP provides an extended run configuration for parallel programs that allows system-specific information about the parallel run to be supplied.

Parallel Debugging

Developing complex parallel programs is a difficult task, and it is particularly challenging to ensure that they operate correctly. Finding errors in a parallel program is complicated because the many concurrent threads of execution make it difficult to observe the program operation in a deterministic manner. Managing large applications running on many processors may present a problem to the developer as well since the sheer volume of information may be overwhelming. The very act of debugging the program may also disturb its operation enough to make identifying the cause of an error impossible.

The PTP debugger attempts to address some of these issues. In particular, the debugger provides the basic functionality needed to step through the execution of the program examining variables and program state along the way, and an easy to use interface that enables the developer to quickly obtain pertinent information about an executing application. The debugger is invoked in the same manner as any other application launch, by clicking a button. Eclipse will automatically switch to the *Parallel Debug Perspective* when the debugging session is ready.

The Parallel Debugger Perspective comprises a number of views to facilitate the debugging task. The *Parallel Debug View* provides a high level view of the application, showing all the tasks currently executing. Debug operations on groups of processes, such as single stepping, can be performed using this view. The *Debug View* provides information about individual threads of execution, including a stack frame showing the current location. The *Variables View* shows the local variables from the currently selected stack frame in the Debug View, and can be used to inspect variables from a range of tasks. Other views are also available to inspect different aspects of the program state.

Utilizing External Tools

Although PTP provides a range of tools for developing parallel programs, there are many other tools available that could be used to the developer's advantage if they were accessible from within the Eclipse environment. To facilitate this, PTP provides an external tools framework that allows non-Eclipse tools to be integrated in a manner that preserves the developer's workflow. The framework defines three main integration points during the application development workflow: *compile, execute*, and *analyze*. The *compile* integration point specifies how the normal build commands are to be modified during the build process to perform any tool-specific actions. An example of this might be to instrument the application to collect tracing or profiling information. The *execute* integration point specifies how the command used for launching of the application executable can be modified to perform any tool-specific actions. An example of this might be to pass the application executable to a tool that performs data collection during the execution. The *analyze* integration point allows an external tool to be launched once the program execution is complete, such as a tool to analyze performance data generated during execution.

In addition to these integration points, the external tools framework provides a *Feedback View* that enables externally generated information to be mapped back to source files and lines within the Eclipse environment. Eclipse can then display this information in the form of sortable tables, or by annotating the source code directly with markers or other forms of highlighting.

Remote Development

To be an effective enhancement to conventional development processes, PTP needs to support a broad range of environments and systems. Many of these systems are scarce resources, so access is often tightly controlled from remote locations. Ideally, application developers for these systems need to be able to access these resources for testing, debugging, and performance optimization as if they were local. This enables the development process to be significantly streamlined, since the developer does not need to be concerned with copying executables and/or data files from system to system.

PTP addresses this requirement by adding a *Remote Development Tools (RDT)* feature. This enables a project to be physically located on a remote machine, but allows access to the project and its source files for editing and building as if they were local. RDT takes care of all the activities necessary to make this process appear as transparent as possible to the developer. This can also be combined with PTP's ability to launch and debug programs on a remote target system, so the developer is able to take advantage of a fully remote enabled environment.

The Parallel Tools Platform Today

PTP is an evolving project. Started in 2005, it has an active and growing developer community that spans a range of government, academic, and commercial organizations. The number of developers contributing to the project continues to increase, including the renowned National Center for Supercomputing Applications, which has recently begun to actively participate in PTP development.

Another promising development is the contribution of value-added components that enhance the functionality of PTP. Contributions have been made by the University of Oregon to add functionality to support their performance analysis tool, called Tuning and Analysis Utilities (TAU), the University of Utah to support their tool for formal verification of MPI programs, called In-situ Partial Order (ISP), and the University of Florida to support their tool for performance analysis of partitioned global-address-space (PGAS) programming models, called Parallel Performance Wizard (PPW).

PTP's main goal to date has been to demonstrate that Eclipse is a viable alternative for the development of scientific applications for large-scale computer systems. The list of features that are now available, and the growing developer and contributor community, demonstrate that this goal has been met.

Future Directions

A comprehensive IDE for developing applications for a broad range of parallel platforms is a huge undertaking. PTP has begun by addressing many of the most pressing issues, such as programmer assistance, uniform access to remote systems, parallel debugging, and external tool integration. However there are still a large number of

issues that remain to be dealt with, and these will form the basis of much of the ongoing development over the coming years. Some of these issues include:

Scalability: Both application and systems sizes are increasing dramatically. The next generation of petascale systems will have hundreds of thousands of processing elements, and executions that exceed one million threads will be likely. The applications themselves are growing continuously, with millions of lines of code now becoming increasingly common. To continue to be effective, PTP will need to support such scales without unduly impacting developer productivity.

Debugging paradigms: Conventional debugging approaches have been successfully applied to parallel programs for some years. However, as program sizes increase, it is likely that these techniques will break down. One reason for this is that the sheer volume of information (for example, a million executing threads) is likely to overwhelm the developer. New debugging paradigms will need to be introduced that address this information overload, yet still allows the developer to accurately pinpoint the location of errors within a program.

Static analysis tools: Eclipse has an enormously powerful infrastructure available to the tool developer, and PTP has only really touched the surface of possible tools to exploit this. There is significant scope for much more powerful static analysis and refactoring tools that could have a very beneficial impact on developer productivity, and work is actively underway to enhance this capability.

Remote development: To be a truly effective remote development platform, PTP needs to support additional paradigms for developing applications remotely. In particular, the ability to work off-line and support for synchronizing with a remote repository are two areas that would enhance developer productivity significantly.

Multi-core architectures: The computing industry is beginning to see a real convergence between multi-core systems, which have already reached hundreds or even thousands of cores, and parallel architectures, which already employ multi-core programming elements. Many of the techniques used for parallel programming may also be beneficial for applications that wish to exploit multi-core technology. PTP is the logical home for exploring this convergence, and work is beginning to take place in this area.

Related Entries

►Compilers
►Debugging
►Fortran 90 and its Successors
►MPI (Message Passing Interface)
►OpenMP
►PGAS (Partitioned Global Address Space) Languages
►Performance Analysis Tools
►Scheduling
►TAU
►UPC

Bibliographic Notes and Further Reading

The idea of using computers to aid in the software engineering process extends back to the late 1960s; however, it was not until the 1980s that the market for computer-aided software engineering or CASE-based tools became significant (the term CASE was not coined until 1982). CASE covers a very broad area of software engineering practices, whereas integrated development environments, which are one class of CASE tools, tend to focus on the relatively small set of software engineering practices associated with the edit-build-test-debug development lifecycle.

There have been a number of papers describing the relationship between productivity and the use of IDEs, and the productivity and quality improvements that can be obtained by such environments is well documented [5, 6, 9, 10]. Their success is also reflected in the best practice use of IDEs for most commercial software development today.

Van De Vanter, et al. [11] have asserted that there are many reasons why existing developer tools are not going meet the productivity requirements of high performance computing: they are difficult to learn, they may not scale as machine sizes increase, they are different across different platforms, they are hard to develop or port to new platforms, they are often poorly supported, and they can be expensive.

The use of IDEs for parallel computing has also been explored before [1–4, 8], but all of these appear to have suffered from low levels of use, were academic research projects, or were too expensive to develop and maintain, and so have gradually died out. PTP is the first time that an open-source IDE has been tailored specifically for scientific and parallel computing [12].

The non-for-profit Eclipse Foundation was created in 2004 to oversee the stewardship of the open-source Eclipse platform. Since it was established, the Foundation membership has grown to exceed 150 organizations. Many of these companies use Eclipse as the core of a commercial product offering, after having abandoned their own proprietary IDE technology and switched to the Eclipse platform. Eclipse is widely used across many computing fields, and arguably has the largest number of users of any IDE in history. It is also now one of the largest open-source projects and there have been over two million downloads of the latest version alone [7].

Bibliography

1. Bemmerl T (1992) TOPSYS for programming distributed multiprocessor computing environments. In: Proceedings of computer systems and software engineering, The Hague, pp 175–180
2. Brode BQ, Warber CR (1998) DEEP: a development environment for parallel programs. In: Proceedings of the international parallel processing symposium, Orlando, pp 588–593
3. Callahan D, Cooper K, Hood R, Kennedy K, Torczon L (1987) ParaScope: a parallel programming environment. In: Proceedings of the first international conference on supercomputing, Athens, Greece, June 1987
4. Cownie J, Dunlop A, Hellberg S, Hey AJG, Pritchard D (1994) Portable parallel programming environments – the ESPRIT PPPE project, massively parallel processing applications and development, The Netherlands, June 1994
5. Frazer A (1993) CASE and its contribution to quality. The Institution of Electrical Engineers, London
6. Granger MJ, Pick RA (1991) Computer-aided software engineering's impact on the software development process: an experiment. In: Proceedings of the 24th Hawaii international conference on system sciences, Jan 1991, pp 28–35
7. http://eclipse.org/downloads
8. Kacsuk P, Cunha JC, Dózsa G, Lourencço J et al (1997) A graphical development and debugging environment for parallel programs. Parallel Comput 22(13):1747–1770
9. Luckey PH, Pittman RM (1991) Improving software quality utilizing an integrated CASE environment. In: Proceedings of the IEEE national aerospace and electronics conference, Dayton, May 1991, pp 665–671
10. Norman RJ, Nunamaker JF Jr (1989) Integrated development environments: technological and behavioral productivity perceptions. In: Proceedings of the annual Hawaii international conference on system sciences, Kailua-Kona, Jan 1989, pp 996–1003
11. Van De Vanter ML, Post DE, Zosel ME (2005) HPC needs a tool strategy. In: Proceedings of the second international workshop on software engineering for high performance computing system applications, ACM, May 2005, pp 55–59
12. Watson GR, DeBardeleben NA (2006) Developing scientific applications using eclipse. Comput Sci Eng 9(4):50–61

Parallelism Detection in Nested Loops, Optimal

Alain Darte
École Normale Supérieure de Lyon, Lyon, France

Synonyms

Detection of DOALL loops; Nested loops scheduling; Parallelization

Definition

Loops are a fundamental control structure in imperative programming languages. Being able to analyze, transform, and optimize loops is a key feature for compilers to handle repetitive schemes with a complexity proportional to the program size and not to the number of operations it describes. This is true for the generation of optimized software as well as for the generation of hardware, for both sequential and parallel execution.

Exploiting parallelism is a difficult task that depends, among others, on the target architecture, on the structure of computations in the program, and on the way data are mapped, used, and communicated. In contrast, **detecting parallelism** that can be expressed as loops, i.e., transforming Fortran-like DO loops into DOALL loops (loops whose iterations are all independent) depends only on the dependences between computations in the program being analyzed. Intuitively, an algorithm is **optimal** for parallelism detection in loops, if it transforms the loops of a program so that each statement is surrounded, after transformation, by a maximal number of parallel (DOALL) loops. However, such a notion of optimality needs to be defined with care (see the discussion hereafter) to avoid inconsistent or inaccurate claims.

Discussion

Optimal Parallelism Detection in Loops

How to define optimal parallelism detection in loops? An optimality criterion solely based on the *number* of

parallel loops – thus not on the number of iterations – has no meaning for loops with constant bounds as they can always be unrolled or strip-mined. For example, it is true that hyperplane scheduling can always be applied on n perfectly nested loops with constant bounds, resulting in a code with one outer sequential loop and $(n-1)$ parallel loops. However, it does not mean that it exhibits any parallelism as, for a purely sequential code, each parallel loop has then a single iteration. Similarly, an optimality criterion should not lead to inconsistencies due to the way loops are generated, e.g., n nested loops with constant loop bounds can be fully unrolled leading to a code with no loop while applying loop tiling leads to $2n$ loops.

One way to define **optimality** is **with respect to the execution time** of the parallelized code **on an ideal** PRAM-like **machine**, with an unlimited number of computation units, where any instruction takes a single unit of time. An algorithm for parallelism detection is then optimal if the corresponding code with DOALL loops has an optimal execution time for this ideal machine, which is the maximal length of a path in the dependence DAG (directed acyclic graph) obtained by fully unrolling the program. However, a full unroll of the code is too costly, often undesirable as it loses the loop structure, and sometimes impossible, e.g., when the loop bounds are parameterized. A more practical definition of optimality is to assume that each loop has a parameterized number of iterations, of order N, and to say that an algorithm is optimal if the execution time of the parallelized program on the ideal machine is $O(N^d)$ where d is minimal. Optimality can then be proved by exhibiting a dependence path in the unroll program whose length is not $O(N^{d-1})$. A more accurate definition of optimality can be given by applying the same reasoning for each statement, considering that any other statement takes no time on the ideal machine, i.e., does not count in the length of a dependence path.

In general, algorithms for parallelism detection transform the code so that each statement is surrounded by the same number of loops before and after transformation. This is true, in particular, for algorithms based on unimodular transformations. In this case, one retrieves the intuitive notion of optimality which states that parallelism detection is optimal if each statement is surrounded, after transformation, by a maximal number of parallel loops. The only constraint that a parallelism detection algorithm must respect is that the partial order of operations defined by the dependences in the program are preserved. How these dependences are computed and abstracted is not part of the algorithm, it is its input. In other words, the **optimality** of an algorithm can only be defined **with respect to the dependence abstraction** it uses. To show that it is optimal, one needs to exhibit a dependence path of the required length, not in the original program, but in its abstraction, i.e., a dependence path in an over-approximation of the actual dependence graph.

A more complete discussion on the definition of optimality for parallelism detection algorithms is provided in [10, 12, 13]. The rest of this essay, with large parts borrowed from [7], shows that it is possible to give, for each classic dependence abstraction (dependence level, uniform dependence vector, direction vector, dependence cone, dependence polyhedron), an algorithm which is optimal with respect to this abstraction. Moreover, this algorithm is a specialization of a more generic algorithm [13], inspired by the decomposition of Karp, Miller, and Winograd for checking the computability of a system of uniform recurrence equations [18]. Notice that the previous optimality criterion makes no distinction between an outer parallel loop and an inner parallel loop. If a parallel loop can always be pushed down, i.e., interchanged with inner loops, the converse is not true. However, detecting parallel loops that contain sequential loops and detecting permutable loops (the base for loop tiling) are two similar problems that can be achieved by variations of the algorithms mentioned hereafter. But optimality for these goals is more difficult to define. Here is a summary of the main optimality results:

Dependence level The algorithm of Kennedy and Allen [2] is optimal, which implies that loop distribution is sufficient to detect maximal parallelism for dependence levels.

Uniform dependences Hyperplane scheduling (as defined by Lamport [19]) is optimal, for perfectly nested loops, which implies that unimodular transformations with one outermost sequential loop is sufficient for uniform dependences.

Direction vectors An adaptation of the algorithm of Wolf and Lam [25] (which detects permutable loops) is optimal, which implies that unimodular

transformations (loop interchange, loop reversal, loop skewing) are sufficient for direction vectors.

Dependence polyhedron The algorithm of Darte and Vivien [13] is optimal, which implies that unimodular transformations, combined with loop distribution and loop shifting, are sufficient for dependence polyhedra.

Affine dependences The algorithm of Feautrier [16] is optimal among all affine transformations, but these transformations are not sufficient to detect maximal parallelism for affine dependences.

This essay is organized as follows. The first section recalls the model of system of uniform recurrence equations (SURE) and the link between the computability and scheduling problems of such a system. The second section shows how, thanks to a uniformization of dependence distance abstractions, the detection of parallelism in nested DO loops can be transferred to the model of SURE so as to derive optimality results. The last section illustrates yet another connection: the multi-dimensional scheduling techniques used to analyze a SURE and to detect parallelism in nested DO loops can also be used to prove the termination of some WHILE loops.

The Organization of Computations in a System of Uniform Recurrence Equations

In 1967, Karp, Miller, and Winograd introduced a model (system of uniform recurrence equations or SURE) to describe a set of regular computations as mathematical equations [18]. The goal of their paper, entitled "The Organization of Computations for Uniform Recurrence Equations," was to study the parallelism that such a description contains implicitly, motivated by "the recent [at this time] development of computers capable of performing many operations concurrently," in particular for regular applications such as solving partial differential equations by finite-difference methods. This work was purely theoretical: the paper does not even have a conclusion section, which would possibly mention some applications. Nevertheless, it turned out to be very prophetic, when considering the large number of developments for which it served as foundations. For example, it has some connections with accessibility problems in vector addition systems and Petri nets. The developments of systolic arrays and of high-level synthesis from systems of recurrence equations are directly inspired from it. Most scheduling methods developed for automatically parallelizing DO loops are based on it, directly or indirectly. As a by-product, the theory of unimodular transformations and the different parallelism detection algorithms [2, 13, 16, 19, 25] can be seen as extensions and/or specializations of the technique of Karp, Miller, and Winograd, adapted to specific dependence abstractions and objectives. Last but not least, it has also some connections with techniques to prove the termination of imperative programs, either with the insertion of counters or by the derivation of affine ranking functions [1, 5, 17, 22].

Definition of a SURE

A SURE is a finite set of m equations of the form

$$a_i(p) = f_i(a_{i_1}(p - d_{i_1,i}), \ldots, a_{i_{m_i}}(p - d_{i_{m_i},i})) \quad (1)$$

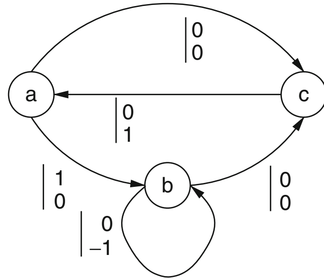
where $p \in \mathbb{Z}^n$ is called an **iteration vector**, $d_{i,j} \in \mathbb{Z}^n$ is called a **dependence vector**, f_i is a strict function with m_i arguments whose properties are not considered (only the structure of computations is analyzed, not what they do). The value $a_i(p)$ has to be computed for all integral points p in a subset $\mathcal{P}$ of $\mathbb{Z}^n$, called the **evaluation region**. Values are supposed to be given at $p - d_{i,j} \notin \mathcal{P}$ (input variables) wherever required for the evaluation at $p \in \mathcal{P}$.

The value $a_i(p)$ is said to depend on $a_{i_j}(p - d_{i_j,i})$, for $1 \le j \le i_m$. These dependence relations define a graph Γ, called the **expanded dependence graph** (EDG). Its set of vertices is $\{1, \ldots, m\} \times \mathcal{P}$ and there is an edge from (j, q) to (i, p) if $a_i(p)$ depends on $a_j(q)$. A SURE can be equivalently defined by a weighted-directed multigraph $G = (V, E, w)$, called the **reduced dependence graph** (RDG), as follows:

- For each a_i, there is a vertex v_i in V.
- If $a_i(p)$ depends on $a_j(p - d_{j,i})$, G has an edge $e = (v_j, v_i)$ in E from v_j to v_i with **weight** $w(e) = d_{j,i}$.

A SURE is then defined by an RDG $G = (V, E, w)$ and an evaluation region $\mathcal{P}$. For example, the following system, to be evaluated for $\mathcal{P} = \{(i, j) \mid 1 \le i, j \le N\}$,

$$\begin{cases} a(i,j) = c(i,j-1) \\ b(i,j) = a(i-1,j) + b(i,j+1) \\ c(i,j) = a(i,j) + b(i,j) \end{cases}$$



Parallelism Detection in Nested Loops, Optimal. Fig. 1
Corresponding RDG for the SURE example

defines a SURE with three equations on a square of size N, corresponding to the RDG of Fig. 1.

Computability: Definition and Properties

The definition of a SURE, either by Equation (1), or by the reduced dependence graph, is implicit: to compute a value $a_i(p)$, one must first compute all values that appear in the right-hand side of the definition of $a_i(p)$. These values are computed the same way, by evaluating the right-hand side of their definition, and so on. Thus, two questions arise:

Computability problem Does this recursive process end, i.e., are all values $a_i(p)$ computable?

Scheduling problem If the system is computable, how to organize the computations?

A **schedule** is a function θ from the vertices of the EDG Γ to $\mathbb{N}$ such that $\theta(j,q) < \theta(i,p)$ whenever (i,q) depends on (j,q). A SURE is **computable** (or explicitly defined) if there exists a schedule. It is not computable if there exists a vertex (i,p) in the EDG Γ such that the length of a dependence path leading to it is unbounded. Then, either $a_i(p)$ depends on itself (possibly through other computations) or it needs to wait "infinite" time before the complete evaluation of its right-hand side. In this case, because the in-degree in the EDG Γ is finite, there is an infinite path leading to (i,p).

To make the analysis simpler, the following discussion focuses on (parametric) bounded evaluation regions. An RDG G is said computable if all SUREs defined from G on bounded evaluation regions are computable.

Theorem 1 *An RDG is computable if and only if (iff) it has no cycle of zero weight.*

More precisely, the fact that G has no cycle of zero weight is a sufficient condition for a SURE defined from G to be computable, whatever the bounded evaluation region. However, it is a necessary condition only if the evaluation region is "sufficiently large" (see [9, 18]).

The case of a single equation

A uniform recurrence equation (URE) is defined by an evaluation region $\mathcal{P} \subseteq \mathbb{Z}^n$ and an RDG G with a single vertex, i.e., by a single equation $a(p) = f(a(p-d_1),\ldots,a(p-d_m))$. Let D be the $n \times m$ matrix whose columns are the dependence vectors. Consider the following sets:

$$T(D) = \{t \in \mathbb{Z}^n \mid tD \geq 1\}$$

$$Q(D) = \{q \in \mathbb{Z}^m \mid q \geq 0,\ q \neq 0,\ Dq = 0\}$$

and $T_{\mathbb{Q}}(D)$ and $Q_{\mathbb{Q}}(D)$ their rational relaxations. G is computable iff it has no cycle of zero weight, i.e., $Q(D)$ is empty. Farkas Lemma [23] shows the following result:

Theorem 2 $Q(D) = \emptyset \Leftrightarrow Q_{\mathbb{Q}}(D) = \emptyset \Leftrightarrow T_{\mathbb{Q}}(D) \neq \emptyset \Leftrightarrow T(D) \neq \emptyset$.

This property shows that checking if an RDG is computable can be done in polynomial time by checking that $T_{\mathbb{Q}}(D)$ is non empty, i.e., that the cone generated by the dependence vectors is strictly included in a half-space. Each $t \in T(D)$ corresponds to a **separating hyperplane** and a schedule θ_t defined by $\theta_t(p) = t.p + K$ where $t.p$ is the vector product and $K = -\min_{p\in\mathcal{P}} t.p$. Indeed, if q depends on p, $q = p + d_i$, then $\theta_t(q) = t.q + K = t.p + t.d_i + K > \theta_t(p)$. Thus, an RDG is computable iff there is an **affine schedule**, i.e., a way of computing the URE by a regular schedule, whose definition does not depend on the evaluation region $\mathcal{P}$.

Given a vector $t \in T_{\mathbb{Q}}(D)$, the **latency** of the schedule θ_t, i.e., the total number of sequential steps it induces, can be rounded to $L(\theta_t) = \max_{p,q\in\mathcal{P}} t.(p-q)$. Finding the "fastest" linear schedule means solving a min-max optimization problem $L_{\min} = \min\{L(\theta_t) \mid t \in T_{\mathbb{Q}}(D)\}$. If $\mathcal{P}$ is a polytope $\{p \mid Ap \leq b\}$, then $L(\theta_t) = \max\{t.(p-q) \mid Ap \leq b,\ Aq \leq b\}$. The duality theorem of linear programming [23] leads to:

$$\begin{aligned} L(\theta_t) &= \max\{t.(p-q) \mid Ap \leq b,\ Aq \leq b\} \\ &= \min\{(t_1+t_2).b \mid t_1,t_2 \geq 0,\ t_1A = t,\ t_2A = -t\} \\ L_{\min} &= \min\{(t_1+t_2).b \mid t_1,t_2 \geq 0,\ t_1A = -t_2A = t,\ tD \geq 1\} \end{aligned}$$

Solving the previous linear program gives a way to produce a vector t in $T_{\mathbb{Q}}(D)$ from which a fast schedule can be built. Its performance can be characterized as follows:

Theorem 3 *If the evaluation region $\mathcal{P}$ is sufficiently large, the difference between $L_{\min}$ (the latency of the fastest affine schedule) and the longest dependence path in Γ (the latency of the fastest schedule) is bounded by a constant that does not depend on the domain size.*

Theorems 2 and 3 together show that only two cases can occur for a URE defined on bounded regions: either the URE is not computable as soon as the evaluation region is large enough, or there is an affine schedule. In the latter case, for a URE defined on polyhedra $\{Ap \leq Nb\}$, the length of the longest path is $kN + O(1)$ for some positive rational k and an affine schedule with latency $kN+O(1)$ can be derived. See more details in [9].

The case of several equations

For one equation, a vector $q \in Q(D)$ can be interpreted directly as a cycle in the RDG since all edges are connected to the same vertex: they can be used in any order. For several equations, it is more difficult to express a cycle by linear constraints and to ensure that edges are traversed in a specific order. To detect cycles of zero weight in $G = (V, E, w)$, the key is to consider G' **the subgraph of zero-weight multicycles (union of cycles)**, i.e., the subgraph of G generated by the edges that belong to a union of cycles whose total weight is zero. The following theorem identifies the links between G and G'.

Theorem 4 *(a) G contains a zero-weight cycle iff its subgraph G' does. (b) G contains a zero-weight cycle iff one of its strongly connected components (SCCs) does. (c) If G' is strongly connected, G has a zero-weight cycle.*

These properties give the hint for solving the problem with a recursive search. To detect a zero-weight cycle in G, it is sufficient to consider each SCC of G'. If G' is empty, G has no zero-weight multicycle, thus no zero-weight cycle. If G' has more than one SCC, then G has a zero-weight multicycle if at least one SCC has a zero-weight cycle. It remains to solve the terminating case, i.e., when G' is strongly connected, in which case G has a zero-weight cycle. Finally, this leads to the decomposition of Karp, Miller, and Winograd.

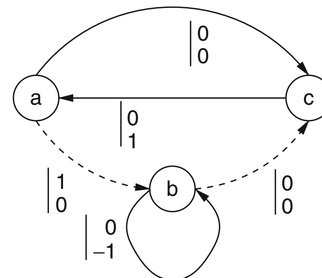
(Decomposition of Karp, Miller, and Winograd) Boolean KMW(G):

- Build the subgraph G' of zero-weight multicycles in G.
- Compute $G'_1, \ldots, G'_s$, the s SCCs of G'.
  - If $s = 0$, G' is empty, return TRUE.
  - If $s = 1$, G' is strongly connected, return FALSE.
  - Otherwise return $\wedge_i$KMW(G'_i) (logical AND).

Then, G is computable iff KMW(G) returns TRUE.

Consider the SURE whose RDG is depicted in Fig. 1. The two edges (a, b) and (b, c) cannot belong to a zero-weight multicycle, since the weight of any multicycle that traverses them has a positive first component. The self-dependence on b and the cycle formed by the edges (a, c) and (c, a) define a zero-weight multicycle, thus G' is the subgraph of G obtained by deleting the two edges from (a, b) and (b, c) (see Fig. 2). It has two SCCs. For both, the subgraph of zero-weight multicycle is empty, thus the decomposition stops: both SCCs are computable, thus G' is computable, and finally G is computable too.

Let us now focus on the construction of the subgraph G'. The **cycle vector** associated to a cycle in G is a vector q, with $|E|$ components, such that q_e is the number of times the edge e is traversed in the cycle. The **connection matrix** is a $|V| \times |E|$ matrix C such that $C_{v,e} = 1$ (resp. $C_{v,e} = -1$) if the edge e leaves (resp. enters) vertex v, and $C_{v,e} = 0$ otherwise. A vector $q \geq 0$ such that $Cq = 0$ represents a union of cycles, which is a cycle if the subgraph of G generated by the edges e such that $q_e > 0$ is connected. Let W be the $n \times |E|$ **weight**



Parallelism Detection in Nested Loops, Optimal. Fig. 2 As dotted lines, edges that do not belong to G'

matrix whose columns are the edges weights of G. Then, the edges of G' are exactly the edges e for which $v_e = 0$ in any optimal solution of the following linear program:

$$\min\left\{ \textstyle\sum_e v_e \mid q \geq 0,\ v \geq 0,\ q + v \geq 1,\ Cq = 0,\ Wq = 0 \right\}$$

Now, to better understand what is behind this linear program, let us interpret its dual. After algebraic manipulations, it can be written as:

$$\max\left\{ \textstyle\sum_e z_e \mid 0 \leq z \leq 1, t.w(e) + \rho_{y_e} - \rho_{x_e} \geq z_e,\ \forall e \in E \right\}$$

where e goes from x_e to y_e. The complementary slackness theorem [23] shows interesting properties in the dual.

Theorem 5 *For any optimal solution (z, t, ρ) of the dual:*

$$e \in G' \Leftrightarrow t.w(e) + \rho_{y_e} - \rho_{x_e} = 0 \qquad (2)$$

$$e \notin G' \Leftrightarrow t.w(e) + \rho_{y_e} - \rho_{x_e} \geq 1 \qquad (3)$$

Theorem 5 shows how the constraints $t.D \geq 1$ obtained for linear scheduling in the case of a URE (remember the set $T(D)$ in Theorem 2) can be generalized to the case of a SURE. For a URE, t was interpreted as a vector normal to a hyperplane that separates the space into two half-spaces, all dependence vectors being strictly in the same half-space. Here, the vector t defines (up to a translation given by the constants ρ) a hyperplane which is a **strictly separating hyperplane** for the edges not in G', see Inequality (3), and a **weakly separating hyperplane** for the edges in G', see Equality (2). Furthermore, for each subgraph G that appears in the decomposition, t defines a hyperplane that is the "most often strict": the number of edges, for which such a hyperplane is strict, is maximal (since $\sum_e z_e$ is maximal). See more details in [11].

Define the **depth** d of $G = (V, E, w)$ as the maximal number of recursive calls generated by the initial call KMW(G) (counting the first one), except if G is acyclic, in which case $d = 0$. The depth d is a measure of the parallelism described by G: it is related both to the length of the longest paths (intrinsic sequentiality) and to the minimal latency of particular schedules, called shifted-linear **multi-dimensional schedules**, i.e., mappings from $V \times \mathbb{Z}^n$ to $\mathbb{Z}^d$. The execution order is the lexicographic order $\leq_{lex}$ on vectors of dimension d: the components of the schedule can be interpreted as hours, minutes, seconds, …, described by nested loops starting from the outermost one.

For each $v \in V$, involved in d_v recursive calls, a sequence of vectors $t_v^1, \ldots, t_v^{d_v}$ and of constants $\rho_v^1, \ldots, \rho_v^{d_v}$ can be built by considering the dual program during the decomposition algorithm. The two sequences can be completed with zeros, if needed, to get sequences of length d.

Theorem 6 *Let $G = (V, E, w)$ be a computable, strongly connected RDG of depth d. If the evaluation region is a n-dimensional cube of size N, the mapping defined by $\theta(v,p) = (t_v^1.p + \rho_v^1, \ldots, t_v^{d_v}.p + \rho_v^{d_v}, 0, \ldots, 0)$ defines a multi-dimensional schedule with latency $O(N^d)$. Furthermore, the associated EDG Γ contains a dependence path of length $\Omega(N^d)$, whose projection onto G visits $\Omega(N^{d_v})$ times each vertex $v \in V$.*

A dependence path of length $\Omega(N^d)$ can be built in Γ, following the hierarchical structure of G', by traversing order N times a cycle that visits all vertices of G and by plugging, during this traversal, each SCC of G' order of N times, in a recursive manner. Consider the example of Fig. 2 again. Go from $a(1,1)$ to $a(1, N-1)$, following $(N-1)$ times the cycle between a and c. Then, go to $b(2, N-1)$ following the edge (a, b), and to $b(2,1)$ following $(N-1)$ times the self-loop on b. Finally, go to $c(2,1)$ and $a(2,2)$ following the edges (b,c) and (c,a) once. This makes a path of length $2(N-1)+1+$(N-1)$+2 = 3N$. This pattern can be repeated $(N-1)$ more times, leading to $a(N,N)$, for a path of length $3N^2$.

In terms of scheduling, solving the constraints of Theorem 5 shows that $a(i,j)$ can be computed at time step $(2i+1, 2j)$, $b(i,j)$ at step $(2i, -j)$, and $c(i,j)$ at step $(2i+1, 2j+1)$. Indeed, for the first level of the decomposition, the constraints amount to look for a vector t such that $t.(0,1) = t.(0,-1) = 0$ (for the two cycles of G') and $t.(1,0) \geq 2$ (for the cycle (a,b,c,a) with two edges not in G'), which leads, e.g., to $t = (2,0)$ and suitable constants ρ_a, ρ_b, and ρ_c. The second level leads to $t = (0,2)$ for a and c, and $t = (0,-1)$ for b. This explicit schedule

corresponds to the code below. It is purely sequential (2-dimensional schedule, for a dependence of length order N^2, in a square of size N). In general, if a statement is surrounded in the initial code by n loops and scheduled with a d-dimensional schedule, it will be surrounded by d sequential loops and $(n-d)$ parallel loops in the resulting code.

```
DO i=1, N
   DO j=N, 1, -1
      b(i,j) = a(i-1,j) + b(i,j+1)
   ENDDO
   DO j=1, N
      a(i,j) = c(i,j-1)
      c(i,j) = a(i,j) + b(i,j)
   ENDDO
ENDDO
```

Loop Transformations and Automatic Loop Parallelization

Linear programming methods, optimizations on polytopes, manipulations of integral matrices, are now commonly used in the field of automatic parallelization and program transformations, in particular for imperative codes with DO loops. The key is to represent, analyze, and transform loops without unrolling them, in an abstract way, thanks to polyhedral representations and transformations. This way, compilation methods can be developed with a complexity that depends on the (textual) size of the program and not on the number of operations it describes. DO loops are indeed, as SUREs, a condensed way for representing repetitive computations. Such an approach started in 1974 when Lamport introduced the hyperplane method [19] to parallelize perfectly nested loops. Similar techniques were applied for the automatic synthesis of systolic arrays from uniform recurrence equations. In 1988, Feautrier introduced PIP [14], a software tool for parametric (integer) linear programming, and he demonstrated its interest for dependence analysis [15], loop parallelization [16], code generation [4], etc. This work initiated many more developments based on polytopes for detecting dependences, scheduling computations, mapping data and communications, generating code for computations and communications, etc.

The following presentation, borrowed from [10], recalls the link between the decomposition of Karp, Miller, and Winograd, and different algorithms for transforming (sequential) DO loops into DOALL loops, i.e., loops whose iterations can be computed in any order, in particular in parallel. These algorithms perform high-level source-to-source transformations, in the same way a user inserts parallelization directives in OpenMP. Exploiting the parallelism is another story, which requires data and communication optimizations, depending on the target architecture.

Representation of DO Loops

Loop transformations apply to codes defined by nested loops, for which the control structure is simple enough to be captured with polytopes. Each loop has its own loop counter that takes, if the loop step is 1, any integer value from the lower to the upper bound, which are defined by affine expressions of the surrounding loop counters. The iterations of n perfectly nested loops can thus be represented by an **iteration domain**, set of all integer vectors in a polyhedron. When running the program, each statement S is executed for each value of the surrounding loop counters, represented by an **iteration vector** p. Such an execution is an **operation**, denoted $S(p)$.

Thus, as for SUREs, nested loops have an evaluation region, the iteration domain. However, unlike SUREs, the schedule is explicit and defines the semantics, while the dependences are implicit and must be precomputed. Indeed, because each inner loop is scanned for each iteration of an outer loop, the operations $S(p)$ are carried out in the predefined **sequential order** $<_{seq}$, given by the lexicographic order defined on iteration vectors plus the textual order:

$$S(p) <_{seq} T(q) \Leftrightarrow (\widetilde{p} <_{lex} \widetilde{q}) \text{ or } (\widetilde{p} = \widetilde{q} \text{ and } S <_{text} T)$$

where $\widetilde{p}$ and $\widetilde{q}$ are the vectors p and q restricted to the loop counters that surround both S and T. There is a **data dependence** from $S(p)$ to $T(q)$ (denoted $S(p) \Rightarrow T(q)$) if both operations access the same memory location, at least one access is a write, and $S(p) <_{seq} T(q)$. As for SUREs, the relation $\Rightarrow$ defines a partial order between operations, i.e., an **expanded**

dependence graph (EDG). To keep the program semantics, code transformations should preserve this partial order. In general, instead of representing all pairs $(S(p), T(q))$ for which $S(p) \Rightarrow T(q)$, the dependences are approximated by a **reduced dependence graph** (RDG), with one vertex per statement, where the weight $w(e)$ of each edge e describes a set D_e of **dependence distances** $\widetilde{q} - \widetilde{p}$, in a conservative way: if $S(p) \Rightarrow T(q)$ in the EDG, then there exists $e = (S, T)$ in the RDG such that $\widetilde{q} - \widetilde{p} \in D_e$. In other words, the RDG describes a superset of the EDG called **apparent dependence graph** (ADG). All loop transformations algorithms have to respect the dependences in the ADG. Thus, their properties, in particular their optimality for detecting parallelism, need to be analyzed with respect to the ADG (and not the EDG, which is not provided), i.e., with respect to the dependence abstraction used.

Approximations of Distances: Dependence Level and Direction Vector

The simplest way to represent a dependence distance is to use the abstraction by dependence level. A dependence between $S(p)$ and $T(q)$ is **loop-independent** if it occurs for a fixed iteration of each loop surrounding both S and T (i.e., $\widetilde{q} = \widetilde{p}$). Otherwise, it is **loop-carried** and its **level** is the index of the first nonzero component of $\widetilde{q} - \widetilde{p}$. Then, all iterations of a loop L at depth k can be executed in any order, i.e., L is parallel, if there is no dependence at level k in the RDG that corresponds to the code surrounded by L.

The main idea of Allen and Kennedy's parallelization algorithm [2] is to use loop distribution to reduce the number of statements within a loop, and thus the number of potential dependences. Briefly speaking, loop distribution separates, in different loops, the statements of the different SCCs of the RDG. Then, each SCC is treated separately and, according to the dependence levels, the outermost loop is marked as a DOALL or DOSEQ loop. Inner loops are treated the same way, recursively. Here is a sketch of the algorithm (different but equivalent to the original formulation). The initial call is AK(G, 1), where G is the RDG with dependence levels.

A careful analysis [12] reveals that this algorithm is nothing but the decomposition of Karp, Miller, and Winograd, applied to an RDG with levels, except that parallel loops are generated at the outermost level, when possible. Indeed, if schedules are searched as in Theorem 5, considering that $w(e)$ corresponds to any distance vector represented by the dependence level of the edge e, then the only valid schedules are the elementary schedules that correspond to loop distribution/parallelization. This can also be understood with the uniformization principle mentioned in the next section. Moreover, using a proof technique similar to the one used in Theorem 6, one can even prove the following optimality result.

(Algorithm of Allen and Kennedy)
AK(G, k):

- Remove from G all edges of level $< k$.
- Compute the SCCs of G.
- For each SCC C in topological order, do:
  - If C is reduced to a single statement S, with no edge, generate DOALL loops in all remaining dimensions, and generate code for S.
  - Else
    - Let l be the minimal dependence level in C.
    - Generate DOALL loops from level k to level $l-1$, and a DOSEQ loop for level l.
    - Call AK(C, $l+1$).

Theorem 7 *Allen and Kennedy's algorithm is optimal for parallelism extraction in an RDG labeled with dependence levels.*

Here, the optimality means the following. Let d_S be the number of DOSEQ loops generated around a statement S. Assume that each loop has order N iterations. Then, it is possible to build, in the ADG corresponding to the RDG, a dependence path that visits $\Omega(N^{d_S})$ times the statement S. It is even possible to build a code, with the same RDG, whose EDG contains such a path. In other words, without any other information, there is no way to extract more parallelism.

Another popular dependence abstraction is the **direction vector** whose components belong to $\mathbb{Z} \cup \{*, +, -\} \cup (\mathbb{Z} \times \{+, -\})$. Its ith component is an approximation of the ith components of all possible distance vectors: it is equal to $z+$ (resp. $z-$) if all ith components are at least (resp. at most) $z \in \mathbb{Z}$. It is equal to $*$ if the ith component may take any value and to $z \in \mathbb{Z}$ if it takes the unique value z. The notation $+$ (resp. $-$) is a shortcut for $1+$ (resp. $(-1)-$. A dependence of level k corresponds

to a direction vector $(0, \ldots, 0, +, *, \ldots, *)$ where + is the k-th component. Unlike the dependence level, the direction vector gives information on all dimensions of the distance vectors. But, it is still not powerful enough to express relations between different components.

When loops are perfectly nested and direction vectors are constant, loops are called **uniform**. Since dependences are directed according to the sequential order, the direction vector is always lexico-positive, its first nonzero component is positive, and Theorem 2 applies: there is a linear schedule, and the code can always be rewritten with one outer sequential loop (which carries all dependences) surrounding parallel loops. Lamport's **hyperplane method** [19] is just a different way to build a linear schedule, without requiring linear programming. To reduce the number of sequential steps, Theorem 3 can be applied too. Variants in which not all dependences are carried by the outermost loop lead, among others, to more subtle NP-complete retiming problems (see [8]).

Finally, a more powerful abstraction of dependence distances is to represent them by a **dependence polyhedron**, defined by a set of vertices, a set of rays, and a set of lines. A direction vector is a particular polyhedral representation. For example, the direction vector $(2+, *, (-1)-, 3)$ defines the polyhedron with one vertex $(2, 0, -1, 3)$, two rays $(1, 0, 0, 0)$ and $(0, 0, -1, 0)$, and one line $(0, 1, 0, 0)$. Thanks to a uniformization principle, an RDG with dependence polyhedra can be reinterpreted in terms of SUREs as recalled in the next section.

Uniformization Principle: From Dependence Polyhedra to SUREs

Consider the following code example:

```
DO i=1, N
   DO j=1, N
      a(i,j) = a(i,j-1) + a(j,i)
   ENDDO
ENDDO
```

It has three dependences: a uniform flow dependence of distance $(0,1)$ from $S(i,j)$ to $S(i,j+1)$, a flow dependence from $S(i,j)$ to $S(j,i)$ if $i < j$, and an anti-dependence from $S(j,i)$ to $S(i,j)$ if $j < i$. The latter two dependences correspond to the same dependence distances and can be combined. The uniform dependence has level 2 and the combined dependence has level 1, thus the algorithm of Allen and Kennedy cannot find any parallelism. The corresponding direction vectors are equal to $(0,1)$ and $(+,-)$. In the second dimension, the "1" and the "−" are incompatible and prevent the detection of parallelism. However, there is a linear schedule $\theta(i,j) = 2i + j$, which leads to a transformed code with one parallel loop:

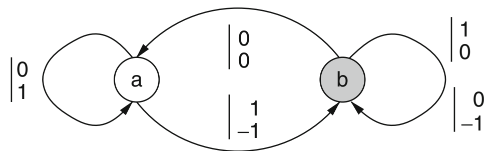
```
DO j = 3, 3N
   DOALL i=max(1,⌈(j-N)/2⌉), min(N, ⌊(j-1)/2⌋)
      a(i,j-2i) = a(j-2i,i) + a(i,j-2i-1)
   ENDDOALL
ENDDO
```

To find this program transformation, one can notice that the set of distance vectors $\{(j-i, i-j) \mid 1 \le j-i \le N-1\}$ can be (over)-approximated by $\mathcal{D} = \{(1,-1) + \lambda(1,-1) \mid \lambda \ge 0\}$, i.e., a polyhedron with one vertex $v = (1,-1)$ and one ray $r = (1,-1)$. Now, as for Theorem 2, consider t such that $t.d \ge 1$ for any dependence vector d. Thus, $t.(0,1) \ge 1$ and $t.d \ge 1$ for all $d \in \mathcal{D}$. The latter inequality is equal to $t.(1,-1) + \lambda t.(1,-1) \ge 1$ with $\lambda \ge 0$, which is equivalent to $t.(1,-1) \ge 1$ and $t.(1,-1) \ge 0$, i.e., $t.v \ge 1$ and $t.r \ge 0$. Therefore, a valid linear schedule is defined by a vector t that satisfies the three inequalities $t.u \ge 1$, $t.v \ge 1$, $t.r \ge 0$, which leads, as desired, to $t = (2,1)$ and $\theta(i,j) = 2i + j$.

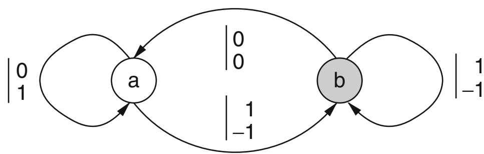
What is important here is the "uniformization" principle, which transforms an inequality on $\mathcal{D}$ into uniform inequalities on v and r. In terms of dependence path, this amounts to consider an edge e, labeled by the distance vector $p = v + \lambda r$, as a path that uses once the "uniform" vector v and λ times the "uniform" vector r. Now, the dependence vectors are not necessarily lexico-positive anymore (e.g., a ray can be equal to $(0,-1)$). Thus, the uniformized dependence graph looks more like the RDG of a SURE than the RDG of a uniform loop nest. However, the constraint imposed on a ray r is weaker, it is $t.r \ge 0$ instead of $t.r \ge 1$, and $t.l = 0$ for a line l. This freedom must be taken into account in the parallelization algorithm. This leads to the algorithm of Darte and Vivien, devoted to an RDG with dependence polyhedra [13]. The technique is simple, it consists in uniformizing the dependences so as to transform the initial RDG into the RDG of a SURE, with some new vertices emulating rays and lines. Despite the

fact that these new vertices must be considered in a special way, all results obtained for SUREs, such as Theorems 5 and 6, and the structure of the subgraph G', can then be transferred to provide an optimal algorithm for parallelism detection in an RDG with dependence polyhedra. Furthermore, thanks to this uniformization principle, the algorithm can be specialized to a particular dependence abstraction, such as dependence level or direction vector, while keeping optimality with respect to the dependence abstraction. For the dependence level abstraction, it is the algorithm of Allen and Kennedy. For direction vectors and a single instruction (or block of instructions considered atomically), it leads to a variant of the algorithm of Wolf and Lam that would generate parallel loops instead of permutable loops (to find permutable loops, one needs to analyze the cone generated by the cycle weights, those in G' form the vector space of this cone).

Figures 3 and 4 depict the uniformized RDGs for the previous example when the non-uniform dependence is represented by a direction vector $(+,-)$ and a dependence polyhedron along $(1,-1)$. In the first case, two self-loops on the new vertex b, of weight $(1,0)$ and $(0,-1)$, are introduced, resulting in a nonempty subgraph G', and no parallelism. In the second case, the self-loop on b has weight $(1,-1)$ and the RDG has no multicycle of zero weight, and thus contains some parallelism. Similarly, the graph of Fig. 1 is the uniformized



Parallelism Detection in Nested Loops, Optimal. Fig. 3
Uniformized graph for direction vectors



Parallelism Detection in Nested Loops, Optimal. Fig. 4
Uniformized graph for polyhedron vector

RDG for the following code, where b is the vertex introduced by the uniformization:

```
DO i=1, N
   DO j=1, N
      a(i,j) = c(i,j-1)
      c(i,j) = a(i,j) + a(i-1,N)
   ENDDO
ENDDO
```

Going Beyond, with the Affine Form of Farkas Lemma

So far, the discussion focused, as in Theorem 6, on loop transformations based on multi-dimensional scheduling functions (with the lexicographic order) called shifted-linear, i.e., of the form $\theta(v,p) = (t_v^1.p + \rho_v^1, \dots, t_v^{d_v}.p + \rho_v^{d_v}, 0, \dots, 0)$ where, for all vertices of the same SCC encountered at depth i of the decomposition, the linear part t_v^i is the same. Different constants (similar to retiming [20]) can be used for different statements however. The fact that the linear part is the same made life easier. Indeed, to get valid scheduling functions, one had to solve inequalities of the form $\theta(T,q)-\theta(S,p) \geq \epsilon$ ($\epsilon = 0$ or 1 as in Theorem 5), i.e., $t.(q-p) + \rho_T - \rho_S \geq \epsilon$. If the dependence distance $q-p$ is constant, one directly ends up with a system of linear inequalities. Otherwise, as just showed, the set of all $q-p$ can be approximated by a polyhedron and linear inequalities involving the vertices, rays, and lines of this polyhedron are obtained.

Now, what if even more general functions are searched, i.e., affine functions with a different linear part for each statement? This is the approach of Feautrier [16]. With $\theta(S,p) = t_S.p + \rho_S$, the constraints that need to be solved are then of the form $t_T.q - t_S.p + \rho_T - \rho_S \geq \epsilon$ for all p and q such that $S(p) \Rightarrow T(q)$. The number of inequalities depends on the number of p and q, which is not practical. However, if the set of pairs (p,q) such that $S(p) \Rightarrow T(q)$ can be described by a polyhedron, the **affine form of Farkas lemma** can be used to simplify the inequalities. This lemma states that $c.p \leq \delta$ for all vectors p in a polyhedron $\{p \mid Ap \leq b\}$ iff $c = y.A$ for some vector $y \geq 0$ such that $y.b \leq \delta$. With this mechanism, an inequality involving all (p,q) in a polyhedron can be replaced by a finite set of inequalities.

The affine form of Farkas lemma is the key tool for writing inequalities in Feautrier's algorithm, which generates general multi-dimensional affine scheduling

functions. The skeleton of the algorithm itself is similar to the decomposition of Karp, Miller, and Winograd: trying to find a function for which as many dependences as possible are satisfied. As for previous algorithms, some optimality result can be formulated, but of a different nature. For affine dependences, i.e., if p is expressed as an affine function of q, for q in a polyhedron, whenever $S(p) \Rightarrow T(q)$, optimal parallelism detection requires more than affine functions, in particular index splitting, i.e., piecewise affine functions. Thus Feautrier's algorithm cannot be optimal with respect to the dependence abstraction it was designed for. However, among all affine functions, it can find one with the "right" parallelism extraction, in other words, the algorithm is optimal with respect to the class of functions it considers [24]. Extensions to detect outer parallel loops, to derive permutable loops for tiling, and to generate codes where not all dependences are carried have also been proposed, see, e.g., [3, 21].

Multi-dimensional Affine Ranking Functions and Program Termination

Recall the graph of Fig. 1. It is the RDG of a SURE with three equations, a, b, c. Starting from such a description of repetitive computations, with explicit evaluation region, implicit schedule, and explicit dependences, an explicit schedule for it was derived. As seen in the previous section, this RDG can also be interpreted as the uniformized RDG of two Fortran-like nested loops where b is a dummy vertex added to emulate the $(1, 0-)$ direction vector. In this case, from a description of repetitive computations, with explicit iteration domain and explicit initial schedule (the sequential order), dependences that were implicit are first computed and abstracted.

Then, another schedule is derived that respects the dependences and expresses, possibly, some parallelism. Now, consider the following C-like code example:

```
y = 0; x = 0;
while (x ≤ N and y ≤ N) {
   if (unknown) {
      x = x + 1;
      while (y ≥ 0 and unknown) y = y − 1;
   }
   y = y + 1;
}
```

This code is yet another description of repetitive computations. Here, the schedule is explicit, it is the sequential schedule. However, loop counters are not specified, no iteration domain is specified, and the program may not terminate. The program is controlled by a parameter N and the integer variables x and y whose values are modified by the program. Now, the RDG of Fig. 1 depicts, not the dependences between computations, but how integer variables, implied in the program control, evolve. Each vertex corresponds to a state (program point + values of variables), edges represent transitions, i.e., modifications of variables. Again, this leads to a model of computations similar to the model of SUREs, for which the techniques and results previously exposed can be useful. In this example, the program can be proved to terminate, after performing $O(N^2)$ operations.

Integer Interpreted Automata and Invariants

The following presentation is borrowed from [1]. To prove the termination of an imperative program, a standard approach is to transform it into an **affine integer interpreted automaton** $(\mathcal{K}, n, k_{init}, \mathcal{T})$ defined by (1) a finite set $\mathcal{K}$ of control points, (2) n integer variables represented by a vector x of size n, (3) an initial control point $k_{init} \in \mathcal{K}$, and (4) a finite set $\mathcal{T}$ of 4-tuples (k, g, a, k'), called **transitions**, where $k \in \mathcal{K}$ (resp. $k' \in \mathcal{K}$) is the source (resp. target) control point. The **guard** $g : \mathbb{Z}^n \mapsto \mathbb{B} = \{\text{true}, \text{false}\}$ is a logical formula expressed with affine inequalities $Gx + g \geq 0$ and the **action** $a : \mathbb{Z}^n \mapsto \mathbb{Z}^n$ assigns, to each variable valuation x, a vector x' of size n, expressed by an affine expression $x' = Ax + a$.

The guard g in the transition $t = (k, g, a, k')$ gives a necessary condition on variables x to traverse the transition t from k to k', and to apply its corresponding action a. If, for two transitions going out of k, the guards describe non-disjoint conditions, the automaton expresses some non-determinism. To approximate non-affine or non-analyzable assignments in the program, the link between x and x' can be described by affine relations instead of functions, which introduces another form of non-determinism. Unlike for SUREs, this non-determinism can be unbounded, i.e., a single transition can give rise to an unbounded number of "successors" x' for a given x.

Unlike for DO loops and SUREs where the range of iteration vectors is explicitly defined, with the iteration domain and the evaluation region respectively, here, the set of all possible values for x at control point k, denoted $\mathcal{R}_k$, is implicit and hard to compute exactly. However, it is possible to over-approximate $\mathcal{R}_k$ by an **invariant** at control point k, i.e., a formula true for all reachable states (k, x). Polyhedral invariants can be computed with **abstract interpretation** techniques, widely studied since the seminal paper of Cousot and Halbwachs [6]. In this case, $\mathcal{R}_k$ is over-approximated by the integer points within a polyhedron $\mathcal{P}_k$, which represents all the information on the variables at control point k that can be deduced from the program by state-of-the-art analysis techniques.

Termination and Ranking Functions

Invariants can only prove partial correctness of a program. The standard technique for proving termination is to consider a **ranking function** to a well-founded set, i.e., a set $\mathcal{W}$ with a (possibly partial) order $\leq$ (the notation $a \prec b$ means $a \leq b$ and $a \neq b$) with no infinite descending chain, i.e., no infinite sequence $(x_i)_{i\in\mathbb{N}}$ with $x_i \in \mathcal{W}$ and $x_{i+1} \prec x_i$ for all $i \in \mathbb{N}$. More precisely, a ranking is a function $\rho : \mathcal{K} \times \mathbb{Z}^n \to \mathcal{W}$, from the automaton states to a well-founded set $(\mathcal{W}, \leq)$, whose values decrease at each transition $t = (k, g, a, k')$:

$$(x \in \mathcal{R}_k) \wedge (g(x) = \text{true}) \wedge (x' = a(x)) \Rightarrow \rho(k', x') \prec \rho(k, x) \qquad (4)$$

The ranking is said to be affine if it is affine in the second parameter (the variables). It is **one-dimensional** if its co-domain is $(\mathbb{N}, \leq)$ and d**-dimensional** (or multi-dimensional of dimension d) if its co-domain is $(\mathbb{N}^d, \leq_d)$, where the order $\leq_d$ is the standard lexicographic order on integer vectors.

Obviously, the existence of a ranking function implies program termination for any valuation v at the initial control point k_{init}. A well-known property is that an integer interpreted automaton terminates for any initial valuation if it has a ranking function. Furthermore, if it terminates and has bounded non-determinism, there is a one-dimensional ranking function (but it is not necessarily affine). The problem is now very similar to the scheduling problem described for SUREs and for DO loops, except that dependences are considered in the opposite direction. In the same way, affine multi-dimensional ranking functions can be derived. Considering rankings with $d > 1$ is mandatory to be able to prove the termination of programs that induce a number of transitions, i.e., a trace length, more than linear in the program parameters. Considering rankings with a different affine function for each control point also extends the set of programs whose termination can be determined, compared, e.g., to shifted-linear rankings or to the technique of [5].

A Greedy Complete Polynomial-Time Procedure

A ranking function ρ of dimension d needs to satisfy two properties. First, as ρ has co-domain $\mathbb{N}^d$, it should assign a nonnegative integer vector to each relevant state:

$$x \in \mathcal{P}_k \Rightarrow \rho(k, x) \geq 0 \text{ (component-wise)} \qquad (5)$$

Second, it should decrease on transitions. Let $\mathcal{Q}_t$ be the polyhedron giving the constraints of a transition $t = (k, g, a, k')$, i.e., $x \in \mathcal{P}_k$, $g(x)$ is true, and $x' = a(x)$. $\mathcal{Q}_t$ is built from the matrices A and G, and the vectors a and g. For an automaton whose actions are general affine relations, $\mathcal{Q}_t$ is directly given by the action definitions. With $\Delta_t(\rho, x, x') = \rho(k, x) - \rho(k', x')$, Inequality (4) then becomes:

$$(x, x') \in \mathcal{Q}_t \Rightarrow \Delta_t(\rho, x, x') \succ_d 0 \qquad (6)$$

which means $\Delta_t(\rho, x, x') \neq 0$ and its first nonzero component is positive. If this component is the i-th, the **level** of $\Delta_t(\rho, x, x')$ is i. A transition t is said to be (fully) **satisfied by the** i**-th component** of ρ (or **at dimension** i) if the maximal level of all $\Delta_t(\rho, x, x')$ is i. To build a ranking ρ, the same greedy mechanism as in [5, 16, 18] can be used. The components of ρ, functions from $\mathcal{K} \times \mathbb{Z}^n$ to $\mathbb{N}$, are built from the first one to the last one. For a component σ of ρ and a transition t not yet satisfied by one of the previous components of ρ, the following constraint is considered:

$$(x, x') \in \mathcal{Q}_t \Rightarrow \Delta_t(\sigma, x, x') \geq \epsilon_t \text{ with } 0 \leq \epsilon_t \leq 1 \qquad (7)$$

and a ranking is selected for which as many transitions as possible have $\epsilon_t = 1$, i.e., are now satisfied. Again, inequalities such as (7) are captured thanks to the affine form of Farkas lemma. The algorithm itself has the same structure as the decomposition of Karp, Miller, and Winograd, and of Feautrier's algorithm.

(Generation of a multi-dimensional affine ranking)

1: $i = 0$; $T = \mathcal{T}$; ▷ Initialize T to all transitions
2: **while** T is not empty **do**
3: Find a 1D affine function σ and values ϵ_t such that all inequalities (5) and (7) are satisfied and as many ϵ_t as possible are equal to 1; ▷ This means maximizing $\sum_{t \in T} \epsilon_t$
4: Let $\rho_i = \sigma$; $i = i + 1$; ▷ σ defines the i-th component of ρ
5: If no transition t with $\epsilon_t = 1$, **return** false ▷ No multi-dimensional affine ranking.
6: Remove from T all transitions t such that $\epsilon_t = 1$; ▷ The transitions have level i
7: **end while**;
8: $d = i$; **return** true; ▷ d-dimensional ranking found

Since nonterminating programs exist, there is no hope of proving that a ranking function always exists. Moreover, there are terminating affine interpreted automata with no multi-dimensional affine ranking. Thus, what can be proved is only that, if a multi-dimensional affine ranking exists, the algorithm finds one, i.e., it is **complete** for the class of multi-dimensional affine rankings. Also, as the sets $\mathcal{R}_k$ are over-approximated by the invariants $\mathcal{P}_k$, completeness has to be understood with respect to these invariants, which means that if the algorithm fails when an affine ranking exists, it is because invariants are not accurate enough.

Theorem 8 *If an affine interpreted automaton, with associated invariants, has a multi-dimensional affine ranking function, then the greedy algorithm finds one. Moreover, the dimension of the generated ranking is minimal.*

Conclusion

In [18], Karp, Miller, and Winograd introduced several new concepts and techniques that gave rise to important developments in the context of loop transformations and program analysis. The key is to represent a repetitive scheme of computations, even infinite, by a finite structure, the reduced dependence graph (and its variants). This allows a compiler to manipulate a program in a time that depends on the structure of the code but not on the number of operations that it describes. In other words, there is no need to unroll loops to understand what they do. Linear programming techniques and polyhedral representations can be used to analyze and optimize such programs, in a parametric way.

This essay mentioned three related problems: determining if a system of uniform recurrence equations is computable, transforming DO loops so as to reveal parallel loops, and proving the termination of programs with IFs and WHILE loops, thanks to affine ranking functions. The link with program termination is still to be explored. Indeed, if the derivation of affine rankings is similar to the derivation of affine schedules, there are many subtle differences, in particular concerning the underlying iteration domains (invariants). One of the most challenging problems is to derive piecewise affine rankings to prove the termination of many more programs. For the detection of parallelism, deriving parallel codes with more data reuse and a better handling of memory transfers is still a challenge.

Related Entries

►Dependence Abstractions
►Dependence Analysis
►Dependences
►Loop Nest Parallelization
►Loops, Parallel
►Parallelization, Automatic
►Polyhedron Model
►Scheduling Algorithms
►Tiling
►Unimodular Transformations

Bibliography

1. Alias C, Darte A, Feautrier P, Gonnord L (2010) Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In 17th International Static Analysis Symposium (SAS'10). Lecture notes in computer science, vol 6337. Springer Verlag, Perpignan, pp 117–133
2. Allen JR, Kennedy K (1987) Automatic translation of Fortran programs to vector form. ACM Trans Program Lang Syst 9(4): 491–542
3. Bondhugula U, Baskaran MM, Krishnamoorthy S, Ramanujam J, Rountev A, Sadayappan P (2008) Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In Compiler Construction (CC'08). Lecture notes in computer science, vol 4959. Springer Verlag, pp 132–146
4. Collard J-F, Feautrier P, Risset T (1995) Construction of DO loops from systems of affine constraints. Parallel Process Lett 5(3): 421–436

P

5. Colón MA, Sipma HB (2002) Practical methods for proving program termination. In 14th International Conference on Computer Aided Verification (CAV). Lecture notes in computer science, vol 2404. Springer Verlag, pp 442–454
6. Cousot P, Halbwachs N (1978) Automatic discovery of linear restraints among variables of a program. In 5th ACM Symposium on Principles of Programming Languages (POPL'78). ACM, Tucson, pp 84–96
7. Darte A (2010) Understanding loops: The influence of the decomposition of Karp, Miller, and Winograd. In 8th ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE'10). IEEE Computer Society, Grenoble, pp 139–148
8. Darte A, Huard G (2002) Complexity of multi-dimensional loop alignment. In 19th International Symposium on Theoretical Aspects of Computer Science (STACS'02), vol 2285. Springer Verlag, pp 179–191
9. Darte A, Khachiyan L, Robert Y (1991) Linear scheduling is nearly optimal. Parallel Process Lett 1(2):73–81
10. Darte A, Robert Y, Vivien F (2000) Scheduling and Automatic Parallelization. Birkhauser. ISBN 0-8176-4149-1
11. Darte A, Vivien F (1995) Revisiting the decomposition of Karp, Miller, and Winograd. Parallel Process Lett 5(4):551–562
12. Darte A, Vivien F (1997) On the optimality of Allen and Kennedy's algorithm for parallelism extraction in nested loops. J Parallel Algorithms Appl 12(1–3):83–112
13. Darte A, Vivien F (1997) Optimal fine and medium grain parallelism detection in polyhedral reduced dependence graphs. Int J Parallel Program 25(6):447–497
14. Feautrier P (1988) Parametric integer programming. RAIRO Rech Opérationnelle 22:243–268
15. Feautrier P (1991) Dataflow analysis of array and scalar references. Int J Parallel Program 20(1):23–51
16. Feautrier P (1992) Some efficient solutions to the affine scheduling problem, part II: Multi-dimensional time. Int J Parallel Program 21(6):389–420
17. Gulwani S, Mehra KK, Chilimbi T (2009) SPEED: Precise and efficient static estimation of program computational complexity. In 36th ACM Symposium on Principles of Programming Languages (POPL'09). ACM, Savannah, pp 127–139
18. Karp RM, Miller RE, Winograd S (1967) The organization of computations for uniform recurrence equations. J ACM 14(3):563–590
19. Lamport L (1974) The parallel execution of DO loops. Commun ACM 17(2):83–93
20. Leiserson CE, Saxe JB (1991) Retiming synchronous circuitry. Algorithmica 6(1):5–35
21. Lim AW, Lam MS (1997) Maximizing parallelism and minimizing synchronization with affine transforms. In 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97). ACM, New York, pp 201–214
22. Podelski A, Rybalchenko A (2004) A complete method for the synthesis of linear ranking functions. In Verification, Model Checking, and Abstract Interpretation (VMCAI'03). Lecture notes in computer science, vol 2937. Springer Verlag, pp 239–251
23. Schrijver A (1986) Theory of Linear and Integer Programming. Wiley, New York
24. Vivien F (2003) On the optimality of Feautrier's scheduling algorithm. Concurr Comput 15(11–12):1047–1068
25. Wolf ME, Lam MS (1991) A loop transformation theory and an algorithm to maximize parallelism. IEEE Trans Parallel Distributed Syst 2(4):452–471

Parallelization

►FORGE
►Loop Nest Parallelization
►Parafrase
►Parallelism Detection in Nested Loops, Optimal
►Parallelization, Automatic
►Parallelization, Basic Block
►Polaris
►Run Time Parallelization
►Speculative Parallelization of Loops

Parallelization, Automatic

David Padua
University of Illinois at Urbana-Champaign, Urbana, IL, USA

Synonyms

Parallelization

Definition

Autoparallelization is the translation of a sequential program by a compiler into a parallel form that outputs the same values as the original program. For some authors, autoparallelization means only translation for multiprocessors. However, this definition is more general and includes translation for instruction level, vector, or any other form of parallelism.

Discussion

Introduction

The compilers of most parallel machines are autoparallelizers, and this has been the case since the earliest parallel supercomputers, the Illiac IV and the TI ASC, were introduced in the 1960s. Today, there are autoparallelizers for vector processors, VLIW processors, microprocessor vector extensions, and multiprocessors.

Autoparallelization is for productivity. Autoparallelizers, when they succeed, enable the programming of parallel machines with conventional languages such as Fortran or C. In this programming paradigm, code is not complicated by parallel constructs and the obfuscation typical of manual tuning.

Inserting explicit parallel constructs and tuning is not only time-consuming but also produces non-portable, machine-dependent code. For example, codes written for multiprocessors and those for SIMD machines have different syntax and organization. On the other hand, with the support of autoparallelization, conventional codes could be portable across machine classes.

Explicit parallelism introduces opportunities for program defects that do not arise in sequential programming. With autoparallelization, the code has sequential semantics. There is no possibility of deadlock and programs are determinate. The downside is that it is not possible to implement asynchronous algorithms, although this limitation does not affect the vast majority of applications.

Requirements for Autoparallelization

A parallelizing compiler must analyze the program to detect implicit parallelism and identify opportunities for restructuring transformations, and then apply a sequence of transformations.

Detection of implicit parallelism can be accomplished by (1) computing the dependences to determine where the sequential order of the source program can be relaxed, and (2) analyzing the semantics of code segments to enable the selection of alternative parallel algorithms.

The transformation process is restricted by the information provided by this analysis and is guided by heuristics supported by static prediction of execution time or program profiling.

Dependence Analysis

The dependence relation is a partial order between operations in the program that is computed by analyzing variable and array element accesses. Executing the program following this partial order guarantees that the program will produce the same output as the original code. For example, in

```
for (i=0; i < n; i++) {a[i] += 1;}
for (j=0; j < n; j++) {b[j] = a[j]*2;}
```

corresponding iterations of the first and the second loop must be executed in the specified order. However, these pairs of iterations do not interact with other pairs and therefore do not have to execute in the original order to produce the intended result. Only corresponding iterations of these two loops are ordered. By determining what orders must be enforced, dependence analysis tells us which reordering is valid and what can be done in parallel: two operations that are not related by the partial order resulting from the dependence analysis can be reordered or executed in parallel with each other.

Dependence analysis can be done statically, by a compiler; or dynamically, during program execution.

Static analysis is discussed next, while dynamic analysis is discussed below under the heading of “Runtime Resolution”.

How close the dependences generated by static dependence analysis are to the minimum number of ordered pairs required for correctness depends on the information available at compile time and the algorithms used for the analysis. The loops above are examples of loops that can be analyzed statically with total accuracy because (1) all the information needed for an accurate analysis is available statically, and (2) the subscript expressions are simple, so that most analysis algorithms can analyze them accurately. Accuracy is tremendously important because when the set of dependences computed by a test is not accurate, spurious dependences must be assumed and this may preclude valid transformations including conversion into parallel form.

There are numerous algorithms for dependence analysis that have been developed through the years. They typically trade off accuracy for speed of analysis. For example, some fast tests do not make use of information about the values of the loop indices, while others require them. Ignoring the loop limits works well in some cases. The loops above are an example of this situation. The value of n in these loops is not required to do an accurate analysis. However, in other cases, knowledge of the loop limits is needed. Consider the loop

```
for (i=10; i<15; i++ ) {a[i]+=a[i-8];}
```

The loop limits, 10 and 14, are necessary to determine that no ordering needs to be enforced between loop iterations since, for these values, the iterations do not

interact with each other. A test that ignores the loop limits will report that (some) iterations in this loop must be executed in order.

Some of the most popular dependence tests require for accuracy that the subscript expressions be affine expression of the loop indices and the values of the coefficients and the constant be known at compile time. For example, a test that requires knowledge of the numerical values of coefficients would have to assume that iterations of the loop

```
if (m >0) {
    for (i=0; i < n; i+=2){
            a[m*i]+=a[m*i+1];
            }
    }
```

must be executed in order, while a test with symbolic capabilities would be able to determine that the iterations do not have to be executed in any particular order to obtain correct results. Table 1 presents the main characteristics of a few dependence tests.

When the needed information is not available at compile time or the analysis algorithm is inaccurate, the decision can be postponed to execution time (see “Transformations for Runtime Resolution” below). For example, the loop

```
for (i=0; i <n; i++) {a[i+k]+=a[i];}
```

can be transformed into an array operation as long as k is negative, but the compiler will not know that this is the case if k happens to be a function of the input to the program or if the value propagation analysis conducted by the compiler cannot decide that k is negative. A similar situation arises in the loop

```
for (i=0; i < n; i++) {a[m[i]]+=a[i];}
```

where m[i] must be $\leq$ i or $\geq$ n and all the m[i]’s be different for a transformation into vector operation to be valid. But this will only be known to the compiler, if it can propagate array values and these values are available in the source code. Otherwise, dependences must be assumed or the analysis postponed to execution time.

Semantic Analysis

Semantic analysis identifies operators or code sequences that have a parallel implementation. A good example is the analysis of array operations. For example, in the Fortran statement

```
a(1:n) = sin(a(2:n+1))
```

the n evaluations of sin can proceed in parallel since their parameters do not depend on each other.

Although array operations like this can be interpreted as parallel operations, most Fortran 90 compilers at the time of the writing of this entry do not parallelize directly array operations, but instead translate them into loops which are analyzed by later passes for parallelization. So, in effect, they rely on semantic analysis.

The compiler can also apply semantic analysis to sequence of statements with the help of a database of patterns. For example,

```
for (i=0; i <n; i++) {s+=a[i];}
```

cannot be parallelized by relying exclusively on dependence analysis, because this analysis will only state the obvious: that each iteration requires the result of the previous one (the values of sum) to proceed. However, accumulations like this can be parallelized, if assuming that + is associative is acceptable, and are frequently found in real programs. Therefore, this pattern is a natural candidate for inclusion in this database.

Other frequently found patterns include: finding the minimum or maximum of an array, and linear recurrences such as

```
x[i]=a[i]*x[i-1]+b[i]
```

Parallelization, Automatic. Table 1 Characteristics of a few dependence tests

Test name	# of loop indices in subscript	Subscript expressions must be affine?	Uses loop bounds?	Ref.
ZIV	0 (constant)	Y	N/A	[5]
SIV	1	Y	Y	[5]
GCD	Any	Y	N	[2]
Banerjee	Any	Y	Y	[2]
Access Region	Any	N	Y	[9]

Some compilers have been known to recognize more complex patterns such a matrix–matrix multiplication.

Once the compiler knows the type of operation, it can choose to replace the code sequence with a parallel version of the operation.

Program Transformations

Program transformations are used to

1. Reduce the number of dependences
2. Generate code for runtime resolution, that is, code that at runtime decides whether to execute in parallel
3. Schedule operations to improve locality or parallelism

Transformations for Reducing the Number of Dependences

This class of transformations aims at reducing the number of ordered pairs to improve parallelism and enable reordering. Induction variable substitution and privatization are two of the most important examples in this class. Induction variables are those that assume values that form an arithmetic sequence. Their computation creates a linear order that must be enforced. In addition, using induction variables in subscripts hinders the dependence analysis of other computations. For example, the loop

```
for (i=0; i<n; i++){j+=2; a[j]=a[j]*2;}
```

cannot be parallelized in this form since j+=2 must be executed in order. Furthermore, dependence analysis cannot know that each iteration of the loop accesses a different element unless it knows that j takes a different value in each iteration. Fortunately, in this example, as in most cases, the induction variable can be eliminated to increase parallelism and improve accuracy of analysis. Thus, here j may be represented in terms of the loop index and forward substituted:

```
for (i=0; i<n; i++) {a[j+2*i+1]=a[j+2*i+1]*2;}
```

The effect of this transformation is that the chain of dependences resulting from the j++ statement goes away with the statement. Also, the removal of the increment makes j a loop invariant and this enables an accurate dependence analysis at compile time.

The identification of induction variables was originally developed for strength reduction, which replaces operations with less expensive ones. A typical strength reduction is to replace multiplications with additions. For parallelism, the replacement goes in the opposite direction. For example, additions are replaced by multiplications as shown in the last example. Induction variable identification relies on conventional compiler data-flow analysis.

Privatization can be applied when the a variable is used to carry values from one statement to another within the one iteration of the loop. For example, in

```
for(i=0;i<n; i++){a=b[i]*2;c[i]= a*c[i]}
```

the use of a single variable, a, in all iterations demands that the iterations be executed in order to guarantee correct results because, a should not be reassigned until its value has been obtained by the second statement of the loop body. The privatization transformation simply makes a private to the loop iteration and thus eliminates a reason to execute the iterations in order.

An alternative to privatization is expansion. This transformation converts the scalar into an array and has the same effect on the dependence as privatization. For the previous loop, this would be the result:

```
for (i=0;i<n;i++) {
     a1[i]=b[i]*2;
       c[i]=a1[i]*c[i]
     }
a=a1[n-1];
```

Privatization is applied when generating code for multiprocessors, and expansion is necessary for vectorization. The main difficulty with expansion is the increase in memory requirements. While privatization increases the memory requirements proportionally to the number of processors, expansion does so proportionally to the number of iterations, a number that is typically much higher.

However, expansion can be applied together with a transformation called stripmining to reduce the amount of additional memory.

Privatization and expansion require analysis to determine that the variable being privatized or expanded is never used to pass information across iterations of the loop. This analysis can be done using conventional data flow analysis techniques.

Transformations for Runtime Resolution

In its simplest form, runtime resolution transformations generate if statements to select between a parallel or serial version of the code. For example,

```
do i=m,n
            a(i+k)=a(i)*2
end do
```

as discussed above, can be vectorized if $k \leq 0$. The compiler may then generate a two-version code

```
if (k<=0) then
          a(k+m:k+n)+=a(m:n)
else
            do i=m,n
            a(i+k)=a(i)*2
            end do
end if
```

Two-version code can be also be used in other situations. Thus, if the loop contains an assignment statement that accesses memory through pointers in the right- and left-hand sides, such as the loop

```
for (i=0; i <n; i++) {*(a+i)=*(b+i)+2;}
```

the if statement should check that address a is either less than address b or greater than address (b+n-1).

More complex runtime resolution would be needed for loops like

```
for (i=0; i <n; i++) {a[m[i]]+=a[i];}
```

where the m[i]'s must be $\leq$ i or $\geq$ n and all distinct for vectorization to be possible, or m[i] either = i or outside the values in the iteration space and all distinct for transformation into a parallel loop. In

```
for (i=0; i <n; i++) {a[m[i]]+=a[q[i]];}
```

the m[i]'s and q[i]'s must be such that m[i] $\leq$ q[i] and the m[i]'s all distinct for vectorization, or m[i] $\neq$ q[j] whenever i $\neq$ j for parallelization. Two-version loops can be generated also in this case, but the if condition is somewhat more complex as it must analyze a collection of addresses. In this last case, the technique is called inspector-executor. Another approach to runtime resolution is speculation, which attempts to execute in parallel and optimistically expects that there will be no conflicts between the different components executing in parallel. During the execution of the speculative parallel code or at the end, the memory references are checked to make sure that the parallel execution was correct.

If it was not, the execution is undone and the components executed at a later time either in the right order or again speculatively, in parallel.

Run time resolution is also used to check for profitability, i.e., that parallel execution will make execution faster. For example, if the number of iterations of a parallel loop is not known at compile time, runtime resolution can be used to decide whether to execute a loop in parallel as a function of the number of iterations. Also, runtime resolution can be used to guarantee that vector operations are only executed if the operands are or can be properly aligned in memory when this is required for performance. For example, SSE vector operations sometimes perform better when the operands are aligned on double word boundaries.

Scheduling Transformations

An important class are the transformations that schedule the execution of program operations or partition these operations into groups. To enforce the order, the compiler typically uses the barriers implicit in array operations or multiprocessor synchronization instructions. One such transformation is stripmining. It partitions the iterations of a loop into blocks by augmenting the increment of the loop index and adding an inner loop as follows:

```
for (i=0; i <n; i++) {a[i]=a[i]+1;}

                         ↓

for (i=0; i < (n/q)*q; i+=q)
     for(j=i; j< i+q, j++) {
          a[j]=a[j]+1;
          }
for (i=(n/q)*q; i< n, i++) {a[i]=a[i]+1;}
```

Stripmining is useful to enhance locality and reduce the amount of memory required by the program. In particular, it can be used to reduce the memory consumed by expansion. If the goal is vectorization and the size of the vector register is q, this transformation will not reduce the amount of parallelism.

Another type of loop partitioning transformation is that developed for a class of autoparallelizing compilers targeting distributed memory operations. These compilers, including High-Performance Fortran and Vienna Fortran, flourished in the 1990s but are no longer used. The goal of partitioning was to organize loop iterations groups so that each group could

be scheduled in the node containing the data to be manipulated.

An important sequencing transformation is loop interchange, which changes the order of execution by exchanging loop headers. This transformation can be useful to reduce the overhead when compiling for multiprocessors and to enhance memory behavior by reducing the number of cache misses. For example, the loop

```
for (i=0; i < (n/q)*q; i+=q)
     for(j=i; j< i+q, j++) {
         a[j]=a[j]+1;
         }
```

can be correctly transformed by loop interchange into

```
for (j=0; j<n; j++)
     for(i=0; i<n, i++){
         a[i][j]=a[i-1][j]+1;
         }
```

The outer loop of the original nest cannot be executed in parallel. If nothing else is done, the only option of the compiler targeting a multiprocessor is to transform the inner loop into parallel form and while this could lead to speedups, the result would suffer of the parallel loop initiation overhead once per iteration of the outer loop. Exchanging the loop headers makes the iteration of the outer loop independent so that the outer loop can be executed in parallel and the overhead is only paid once per execution of the whole loop. Furthermore, the resulting loop has a better locality since the array is traverse in the order it is stored, so that the elements of the array in a cache line are accessed in consecutive order, improving in this way spatial locality.

A third example of sequencing transformation is instruction level parallelization. Consider, for example, a VLIW machine with a fixed point and a floating point unit. The sequence

```
r1=r2+r3
r4=r4+r5
f1=f1+f2
f3=f4+f5
```

contains two fixed point operations (those operating on the r registers) and two floating point operations. Exchanging the second and the third operation is necessary to enable the creation of two (VLIW) instructions each making use of both computational units.

In some cases, the partitioning and sequencing of the operations is not completely determined at compile time. For example the sum reduction

```
for (i=0; i <n; i++) {s+=a[i];}
```

once identified as such by semantic analysis, may be transformed into a form in which subsets of iterations are executed by different threads and the elements of a are accumulated into different variables, one per thread of execution. These variables are then added to obtain the final sum. The number of these threads can be left undefined until execution time. In OpenMP notation, this can be represented as follows:

```
#pragma omp parallel
{float sp=0;
#pragma omp for
for (i=0; i <n; i++){
  sp+=a[i];}
#pragma omp single
{s+=sp;}
}
```

or, more simply,

```
#pragma omp parallel for reduction (+: sum)
 for (i=0; i <n; i++) {
   sp+=a[i];}
```

It should be pointed out that in this example, it has been assumed that floating point addition is associative, but because of the finite precision of machines, it is not. In some cases, it is correct to do this transformation, even if the result obtained is not exactly the same as that of the original program.

However, this is not always the case and transformations like this require authorization from the programmer. Table 2 contains a list of important transformations not discussed above.

Autoparallelization Today

Most of today's compilers that target parallel machines are autoparallelizers. They can generate code for multiprocessors and vector code. Although autoparallelization techniques have become the norm, the few empirical studies that exist as well as anecdotal evidence indicate that these compilers often fail to generate high-quality parallel code. There are two reasons for this. First, sometimes the compiler fails to find parallelism due to limitations of its dependence/semantic analysis or transformation modules. In other cases, it is unable to generate good quality code because of limitations in

Parallelization, Automatic. Table 2 An incomplete list of transformations for autoparallelization

Name	Description	Example of use
Alignment	Reorganizes computation so that values produced in one iteration are consumed by the same iteration	Reduce synchronization costs
Distribution	Partitions a loop into multiple loops	Separates sequential from parallel parts
Fusion	Merges two loops	Reduce parallel loop initiation overhead
Skewing	Partitions the set of iterations into groups that are not related by dependences (i.e. are not ordered)	Enhance parallelism
Node Splitting	Breaks a statement into two	Reduce dependence cycles and thus enable transformations
Software pipelining	Reorders and partitions the executions of operations in a loop into groups that are independent from each other	Enhance instruction level parallelism
Tiling	Partitions the set of iterations of a multiply nested loop into blocks or tiles	Enhance locality
Trace scheduling	Reorder and partition the executions of operations in a loop into groups that are independent from each other	Enhance instruction level parallelism
Unroll and Jam	Partitions the set of iterations of a multiply nested loop into blocks or tiles with reuse of values	Enhance locality

its profitability analysis. That is, the compiler incorrectly assumes that transforming into parallel form would slow the program down.

To circumvent these limitations, compilers accept directives from programmers to help the analysis or guide the transformation and code generation process. A few vectorization directives for the Intel C++ compiler and IBM XLC compiler are shown in Table 3. The programmer can also influence the result by modifying the program into a form that can be recognized by the compiler.

Despite their limitations, autoparallelizers today contribute to productivity by

1. Saving labor. As mentioned, manual intervention in the form of directives or rewriting is typically necessary, but programmers can often rely on the autoparallelizing compiler for some sections of code and in some cases the whole program.
2. Portability. Sequential code complemented with directives is portable across classes of machines with the support of compilers. Portability is after all one of the purposes of compilers and autoparallelization brings this capability to the parallel realm.
3. As a training mechanism. Programmers can learn about what can and cannot be parallelized by interacting with an autoparallelizer. Thus, the compiler report to the programmer is not only useful for manual intervention, but also for learning.

Future Directions

Autoparallelization has only been partially successful. As previously mentioned, in many cases today's compilers fail to recognize the existence of parallelism, or having recognized the parallelism, incorrectly assume that transforming into parallel form is not profitable. Although autoparallelization is useful and

Parallelization, Automatic. Table 3 Vectorization directives for the IBM (XLC) and Intel (ICC) compilers

Vectorization directive	Purpose
#**pragma vector always** *(ICC)*	Vectorize the following loop whenever dependences allow it, disregarding profitability ananlysis
#**pragma nosimd** *(XLC)* #**pragma novector** *(ICC)*	Preclude vectorization of the following loop
__assume_aligned (A, 16); *ICC* **__alignx(16, A)**; *(XLC)*	The compiler is told to assume that the vector (**A** in the examples) start at addresses that are a multiple of a given constant (16 in the examples)

effective when guided by user directives, there is clearly much room for improvement. Research in the area has decreased notably in the recent past, but it is likely that there will be more work in the area due to the renewed interest in parallelism that multicores have initiated. Two promising lines of future studies are

1. Empirical evaluation of compilers to improve parallelism detection, code generation, compiler feedback, and parallelization directives. Evaluating compilers using real applications is necessary to make advances in autoparallelization of conventional languages. Although there has been some work done in this area, much more needs to be done. This type of work is labor intensive since the best and perhaps the only way to do it is for an expert programmer to compare what the compiler does with the best code that the programmer can produce. This process is likely to converge since code patterns repeat across applications [8]. These costs and risks are worthwhile given the importance of the topic and the potential for an immense impact on productivity.
2. Study programming notations and their impact on autoparallelization. Higher level notations, such as those used for array operations, tend to facilitate the task of a compiler while at the same time improving productivity. Language-compiler codesign is an important and promising direction not only for autoparallelization but for compiler optimization in general.

Related Entries

►Banerjee's Dependence Test
►Code Generation
►Dependence Analysis
►Dependences
►GCD Test
►HPF (High Performance Fortran)
►Loop Nest Parallelization
►Modulo Scheduling and Loop Pipelining
►Omega Test
►Parallelization, Basic Block
►Run Time Parallelization
►Scheduling Algorithms
►Semantic Independence
►Speculative Parallelization of Loops
►Speculation, Thread-Level
►Trace Scheduling
►Unimodular Transformations

Bibliographic Notes And Further Reading

As mentioned in the introduction, work on autoparallelization started in the 1960s with the introduction of Illiac IV and the Texas Instrument Advanced Scientific Computer (ASC). The Paralyzer, an autoparallelizer for Illiac IV developed by Massachusetts Computer Associates, is discussed in [10].

This is the earliest description of a commercial autoparallelizer in the literature. Since then, there have been numerous papers and books describing commercial autoparallelizers. For example, [11] describes an IBM vectorizer of the 1980s, [3] discusses Intel's vectorizer for their multimedia extension, and [13] describes the IBM XLC compiler autoparallelization features.

Many of the autoparallelization techniques were developed at universities. Pioneering work was done by David Kuck and his students at the University of Illinois [6, 7] . The field has benefited from the contributions of numerous researchers. The contributions of Ken Kennedy and his coworkers [1] at Rice University have been particularly influential.

There have been only a few papers evaluating the effectiveness of autoparallelizers. In [8], different vectorizing compilers are compared using a collection of snippets, and in [4] the effectiveness of parallelizing compilers is discussed using the Perfect Benchmarks.

More information on autoparallelization, can be found in the related entries or in books devoted to this subject [2, 5, 12, 14]. Reference [5] contains a discussion of compiler techniques for High-Performance Fortran.

Bibliography

1. Allen R, Kennedy K (1987) Automatic translation of FORTRAN programs to vector form. ACM Trans Program Lang Syst 9(4):491–542. DOI= http://doi.acm.org/10.1145/29873.29875
2. Banerjee UK (1997) Loop transformations for restructuring compilers: dependence analysis. Kluwer Academic, Norwell
3. Bik AJC (May 2004) The software vectorization handbook. Intel, Hillsboro
4. Eigenmann R, Hoeflinger J, Padua D (Jan 1998) On the automatic parallelization of the perfect Benchmarks®. IEEE Trans Parallel Distrib Syst 9(1):5–23. DOI=http://dx.doi.org/10.1109/71.655238
5. Kennedy K, Allen JR (2002) Optimizing compilers for modern architectures: a dependence-based approach. Morgan Kaufmann, San Francisco
6. Kuck DJ (1976) Parallel processing of ordinary programs. Adv Comput 15:119–179
7. Kuck DJ, Kuhn RH, Padua DA, Leasure B, Wolfe M (1981) Dependence graphs and compiler optimizations. In: Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on principles of programming languages. POPL '81. Williamsburg, 26–28 Jan 1981, ACM, New York, pp 207–218. DOI=http://doi.acm.org/10.1145/567532.567555
8. Levine D, Callahan, D, Dongarra, J (1991) A comparative study of automatic vectorizing compilers. Parallel Comput 17:1223–1244
9. Paek Y, Hoeflinger J, Padua D (Jan 2002) Efficient and precise array access analysis. ACM Trans Program Lang Syst 24(1) 65–109. DOI=http://doi.acm.org/10.1145/509705.509708
10. Presberg DL (1975) The Paralyzer: Ivtran's parallelism analyzer and synthesizer. In: Proceedings of the conference on programming languages and compilers for parallel and vector machines, New York, 18–19 March 1975, pp 9–16. DOI=http://doi.acm.org/10.1145/800026.808396
11. Scarborough RG, Kolsky HG (March 1986) A vectorizing Fortran compiler. IBM J Res Dev 30(2):163–171
12. Wolfe M (1996)High performance compilers for parallel computing. Addison-Wesley, Reading
13. Zhang G, Unnikrishnan P, Ren J (2004) Experiments with auto-parallelizing SPEC2000FP benchmarks. LCPC, pp 348–362
14. Zima H, Chapman B (1991) Supercompilers for parallel and vector computers. ACM, New York

Parallelization, Basic Block

Utpal Banerjee
University of California at Irvine, Irvine, CA, USA

Synonyms

Parallelization

Definition

A *basic block* in a program is a sequence of consecutive operations, such that control flow enters at the beginning and leaves at the end without halt. *Basic block parallelization* consists of techniques that allow execution of operations in a basic block in an overlapped manner without changing the final results.

Discussion

Introduction

The operations in a basic block are to be executed in the prescribed sequential order. This execution order imposes a *dependence structure* on the set of operations, based on how they access different memory locations. A new order of execution is *valid* if whenever an operation B depends on an operation A in the block, execution of B in the new order does not start until after the execution of A has ended. The basic assumption is that executing the operations in any valid order will not change the final results expected from the basic block.

To reduce the total execution time of the block, one needs to find a new valid order where operations are overlapped. Among all such orders, one must choose only those that are compatible with the physical resources of the given machine. Even when the simultaneous processing of two or more operations is permissible by dependence considerations alone, there may not be enough resources available to process them simultaneously.

There are many algorithms for basic block parallelization. This essay presents four of them: two for a hypothetical machine with unlimited resources, and two for a machine with limited resources. It starts with a section on basic concepts, and after developing the algorithms, ends with a simple example that compares

the actions of all four on a given basic block. References to more algorithms are given in the bibliographic notes.

Basic Concepts

By an *operation* one means an atomic operation that a machine can perform. An *assignment operation* reads one or more memory locations and writes one location. It has the general form:

$$A: \quad x = E$$

where A is a label, x a variable, and E an expression. Such an operation reads the memory locations specified in E, and writes the location x. A *basic block* in a program is a sequence of assignment operations, where flow of control enters at the top and leaves at the bottom. There are no entry points except at the beginning, and no branches, except possibly at the end. The object of study in this essay is a basic block of n operations. The set of those operations is denoted by $\mathcal{B}$. There is a mapping $c : \mathcal{B} \to \{0, 1, 2, \ldots\}$ that gives the cycle times of the operations.

An operation B in the block *depends* on another operation A, and one writes $A \,\delta\, B$, if A is executed before B, and one of the following holds:

1. B reads the memory location written by A.
2. B writes a location read by A.
3. A and B both write the same location.

An operation B is *indirectly dependent* on an operation A, and one writes $A \,\bar{\delta}\, B$, if there exists a sequence of operations $A_1, A_2, \ldots, A_k$, such that

$$A = A_1, A_1 \,\delta\, A_2, \ldots, A_{k-1} \,\delta\, A_k, A_k = B.$$

Two operations A and B are *mutually independent* if $A \,\bar{\delta}\, B$ and $B \,\bar{\delta}\, A$ are both false. The *dependence graph* of the basic block is a directed acyclic graph, such that the nodes correspond to the operations, and there is an edge from a node A to a node B if and only if $A \,\delta\, B$. Thus, $A \,\bar{\delta}\, B$ means there is a directed path from the node A to the node B in the dependence graph.

A new execution order for the operations in $\mathcal{B}$ is *valid* if whenever A, B in $\mathcal{B}$ are such that $A \,\delta\, B$, B is executed after A in the new order. If the prescribed sequential order of execution for $\mathcal{B}$ is changed to any valid order, the results would still be the same. The goal is to find a valid execution order where operations are overlapped as much as possible. However, any such order must also be compatible with the resources of the given machine.

Control steps for execution of the basic block are numbered $1, 2, 3, \ldots$. *Scheduling* an operation in the block means assigning a control step to it where it can start executing. An *instruction* for a given machine is a set of operations that the machine can perform simultaneously. An instruction may be empty. *Scheduling* the basic block means creating a sequence of m instructions $(\mathbf{I}_1, \mathbf{I}_2, \ldots, \mathbf{I}_m)$, where $\mathbf{I}_k$ starts in control step k, such that

1. Each instruction consists of operations in $\mathcal{B}$, and each operation in $\mathcal{B}$ appears in exactly one instruction.
2. The operations in each instruction are pairwise mutually independent.
3. If an operation B in an instruction $\mathbf{I}_k$ depends on an operation A in an instruction $\mathbf{I}_j$, then $j + c(A) \le k$.
4. In any control step, the given machine has enough resources to process simultaneously all operations being executed.

Such a sequence of instructions is a *schedule* for the basic block. A schedule for the block can be specified indirectly by scheduling each individual operation. Then all operations starting in control step k constitute the instruction $\mathbf{I}_k$.

The *weight* of a path $(A_1, A_2, \ldots, A_k)$ in the dependence graph for the basic block $\mathcal{B}$ is the expression $[c(A_1) + c(A_2) + \cdots + c(A_k)]$. A path is *critical* if it has the greatest possible weight among all paths in the graph. Let T_0 denote the weight of a critical path. Then, any schedule for $\mathcal{B}$ will need at least T_0 cycles to finish.

For each operation $A \in \mathcal{B}$, the set of all immediate predecessors (in the dependence graph) is denoted by $\text{Pred}(A)$ and the set of all immediate successors by $\text{Succ}(A)$:

$$\text{Pred}(A) = \{B \in \mathcal{B} : B \,\delta\, A\}, \;\; \text{Succ}(A) = \{B \in \mathcal{B} : A \,\delta\, B\}.$$

The number of members of a set S is denoted by $|S|$.

Unlimited Resources

In this section, it is assumed that the given machine has an unlimited supply of resources (functional units, registers, etc.). Consequently, operations in the basic block

can be scheduled subject only to the dependence constraints between them. The two algorithms considered in this section complete the execution of $\mathcal{B}$ in exactly T_0 cycles.

The ASAP algorithm schedules an operation *as soon as possible* so that the basic block can be processed in the shortest possible time. It creates a simple function $\ell : \mathcal{B} \to \{1, 2, \ldots\}$ such that $\ell(A)$ is the earliest possible step when A can start executing. The ALAP algorithm schedules an operation *as late as possible* within the constraint of executing $\mathcal{B}$ in the shortest possible time. It creates a function $L : \mathcal{B} \to \{1, 2, \ldots\}$ such that $L(A)$ is the latest possible step when A can start executing. The *ASAP label* of A is $\ell(A)$ and its *ALAP label* is $L(A)$. It is clear that $\ell(A) \le L(A)$ for each operation A. The range of consecutive integers from which the control step for A may be chosen is $\{\ell(A), \ell(A) + 1, \ldots, L(A)\}$.

ASAP Algorithm

The goal of the ASAP algorithm (Fig. 1) is to compute the ASAP label ℓ for each operation in the given basic block $\mathcal{B}$. If A and B are two operations such that $A\,\delta\,B$, then after starting A, one must wait at least until A has finished before starting B. This means one must have $\ell(B) \ge \ell(A) + c(A)$.

Algorithm 1 Given a basic block $\mathcal{B}$, its dependence graph, and a machine with unlimited resources, this algorithm computes the ASAP label ℓ of each operation, and the total number m of instructions needed to replace $\mathcal{B}$. For each $A \in \mathcal{B}$, the sets $\mathrm{Pred}(A)$ and $\mathrm{Succ}(A)$ are assumed to be known.

```
m ← 1
V ← B
for each operation A ∈ V do
    Pcount(A) ← |Pred(A)|
    ℓ(A) ← 1
endfor
while V ≠ ∅ do
    for each operation A ∈ V do
        if Pcount(A) = 0 then
            for each B ∈ Succ(A) do
                Pcount(B) ← Pcount(B) − 1
                ℓ(B) ← max{ℓ(B), ℓ(A) + c(A)}
            endfor
            V ← V − {A}
            m ← max{m, ℓ(A)}
        endif
    endfor
endwhile
```

Parallelization, Basic Block. Fig. 1 The ASAP algorithm

An operation is scheduled only after all its predecessors have been scheduled. An integer-valued function Pcount on $\mathcal{B}$ is defined as follows: at any point in the algorithm, $\mathrm{Pcount}(A)$ is the number of immediate predecessors of an operation A that have not been scheduled yet.

Initially, all operations are assigned the control step 1, that is, $\ell(A)$ is initialized to 1 for each $A \in \mathcal{B}$. Operations A for which $\mathrm{Pcount}(A) = 0$ keep this value of $\ell(A)$; they have been scheduled. If $\mathrm{Pcount}(A) = 0$ and B is a successor of A, then reduce $\mathrm{Pcount}(B)$ by 1, and increase $\ell(B)$ to $[\ell(A) + c(A)]$ if it is smaller. The operations whose Pcount is now zero keep their ASAP label; they have been scheduled. This process continues until all operations in $\mathcal{B}$ have been scheduled.

The earliest control step where an operation B can start is given by

$$\ell(B) = \max_{A \in \mathrm{Pred}(B)} [\ell(A) + c(A)].$$

The total number of cycles needed to execute the block $\mathcal{B}$ is

$$\max_{A \in \mathcal{B}} [\ell(A) + c(A)] - 1$$

which is clearly equal to the weight T_0 of a critical path in the dependence graph. The total number of instructions needed to replace the basic block is given by $m = \max_{A \in \mathcal{B}} \ell(A)$.

ALAP Algorithm

The goal of the ALAP algorithm (Fig. 2) is to compute the ALAP label L for each operation in the given basic block $\mathcal{B}$. The entire block has to be completed in the shortest possible time in such a way that each operation starts as late as possible. For each operation A, let $f(A)$ denote the number of cycles from the point when A is scheduled to start to the point when execution of the entire block has been completed. The idea then is to minimize $f(A)$ for each A. When operation A starts, $[L(A) - 1]$ cycles have already elapsed. Hence, the total number of cycles T needed to complete $\mathcal{B}$ is $[L(A) - 1 + f(A)]$, so that

$$L(A) = T + 1 - f(A). \quad (1)$$

The ALAP algorithm first computes T, and $f(A)$ for each A, and then evaluates $L(A)$ from this equation. If A and B are two operations such that $A\,\delta\,B$, then after starting A, one must wait at least until A has finished

Algorithm 2 Given a basic block $\mathcal{B}$, its dependence graph, and a machine with unlimited resources, this algorithm computes the ALAP label L of each operation, and the total number m of instructions needed to replace $\mathcal{B}$. For each $A \in \mathcal{B}$, the sets $\text{Pred}(A)$ and $\text{Succ}(A)$ are assumed to be known.

```
T ← 0
V ← B
for each operation A ∈ V do
    Scount(A) ← |Succ(A)|
    f(A) ← 0
endfor
while V ≠ ∅ do
    for each operation A ∈ V do
        if Scount(A) = 0 then
            f(A) ← f(A) + c(A)
            T ← max{T, f(A)}
            for each B ∈ Pred(A) do
                Scount(B) ← Scount(B) − 1
                f(B) ← max{f(B), f(A)}
            endfor
            V ← V − {A}
        endif
    endfor
endwhile
for each operation A ∈ V do
    L(A) ← T + 1 − f(A)
endfor
m ← max_{A∈B} L(A)
```

Parallelization, Basic Block. Fig. 2 The ALAP algorithm

before starting B. This means $L(B) \geq L(A) + c(A)$, or $f(A) \geq f(B) + c(A)$ by (1). Thus, the minimum possible value for $f(A)$ is

$$f(A) = c(A) + \max_{B \in \text{Succ}(A)} f(B).$$

An operation is scheduled only after all its successors have been scheduled. An integer-valued function Scount on $\mathcal{B}$ is defined as follows: at any point in the algorithm, $\text{Scount}(B)$ is the number of immediate successors of an operation B, that have not been scheduled yet.

Initialize T to 0, and $f(A)$ to 0 for each $A \in \mathcal{B}$. If an operation A has $\text{Scount}(A) = 0$, then $f(A)$ is increased by $c(A)$ to reach $f(A) = 0 + c(A) = c(A)$. This operation has now been scheduled. The value of T is also increased to $f(A)$ if $T < f(A)$. If $\text{Succ}(A) = 0$ and B is a predecessor of A, then reduce $\text{Scount}(B)$ by 1, and increase $f(B)$ to $f(A)$ if $f(B) < f(A)$. The operations A for which $\text{Scount}(A)$ is now zero are handled next. This process continues until all operations in $\mathcal{B}$ have been processed. When the final value of $f(A)$ for each operation A and the final value of T are known, the ALAP labels are found from the equation $L(A) = T + 1 - f(A)$.

The total number of cycles needed to execute the block is $T = \max_{A \in \mathcal{B}} f(A)$ which is equal to the weight T_0 of a critical path in the dependence graph. The total number of instructions needed to replace the basic block is given by $m = \max_{A \in \mathcal{B}} L(A)$.

Limited Resources

In this section, the reality is acknowledged that any given machine has limited amount of functional resources. While scheduling the operations in the basic block $\mathcal{B}$, one now needs to worry about the potential resource conflicts between two operations, in addition to the dependence constraints that may exist between them. For simplicity, a pipelined implementation is assumed for each multi-cycle operation. Register allocation is not treated here; it should be done either before or after scheduling.

List Scheduling

List Scheduling employs a greedy approach to schedule as many operations as possible among those whose predecessors have been scheduled. Each operation is assigned a priority. Operations that are ready to be scheduled are placed on a *ready list* ordered by their priorities. At each control step, the operation with the highest priority is scheduled first. If there are two or more operations with the same priority, then a selection is made at random. List scheduling encompasses a family of different algorithms based on the choice of the priority function. In the algorithm described here (Fig. 3), the priority of an operation A is defined by the difference $\mu(A) = L(A) - \ell(A)$ between its ALAP and ASAP labels, called the *mobility* of the operation. An operation with a lower mobility has a higher priority.

First, Algorithm 1 and Algorithm 2 are used to find the ASAP and ALAP labels of each operation in the given basic block. Operations without predecessors are placed on a ready list $\mathcal{S}$ and arranged in the order of increasing mobility. They are taken from the ready list and scheduled one by one subject to the availability of machine resources. After one round, if there is still an operation A left over in $\mathcal{S}$, then its ASAP label is increased by 1 without exceeding its ALAP label. This will reduce the mobility of A, if it is not already zero.

Algorithm 3 Given a basic block $\mathcal{B}$, its dependence graph, and a machine with limited resources, this algorithm finds a schedule for $\mathcal{B}$. For each $A \in \mathcal{B}$, the sets $\mathrm{Pred}(A)$ and $\mathrm{Succ}(A)$ are assumed to be known.

```
V ← B
k ← 1
for each operation A ∈ V do
    Compute the labels ℓ(A) and L(A) by Algorithm 1 and Algorithm 2
endfor
while V ≠ ∅ do
    S ← all operations in V whose predecessors have finished
            executing before control step k
    for each A ∈ S do
        μ(A) ← L(A) − ℓ(A)
    endfor
    Arrange the operations of S in a sequence (A_{r_1}, A_{r_2},..., A_{r_{|S|}})
        in the increasing order of their mobilities
    Create an empty instruction I_k
    for i = 1 to |S| do
        if A_{r_i} does not have resource conflicts with operations in I_k then
            Put A_{r_i} in I_k
            V ← V − {A_{r_i}}
            S ← S − {A_{r_i}}
        endif
    endfor
    k ← k + 1
    for each A ∈ S do
        ℓ(A) ← min{ℓ(A) + 1, L(A)}
    endfor
endwhile
```

Parallelization, Basic Block. Fig. 3 List scheduling algorithm

Linear Analysis

The Linear Algorithm (Fig. 4) checks the operations of the basic block *linearly* in their order of appearance, and puts them into instructions observing dependence constraints and avoiding resource conflicts.

Arrange the operations in the block in their prescribed sequential order: $A_1, A_2, \ldots, A_n$. For an easy description of the algorithm, it is convenient to assume that a control step could be any integer. Start with a sequence of empty instructions $\{\mathbf{I}_k : -\infty < k < \infty\}$ arranged in an imaginary vertical column. The operations $A_1, A_2, \ldots, A_n$ are taken in this order and put in instructions one by one. At any point, all operations already scheduled are in a range of instructions $(\mathbf{I}_t, \mathbf{I}_{t+1}, \ldots, \mathbf{I}_b)$, where $t \le 1$ and $b \ge 1$. The value of t keeps decreasing and the value of b keeps increasing. At the end of the algorithm, the final sequence $(\mathbf{I}_t, \mathbf{I}_{t+1}, \ldots, \mathbf{I}_b)$ can be renumbered to get a sequence of instructions $(\mathbf{I}'_1, \mathbf{I}'_2, \ldots, \mathbf{I}'_m)$, where $m = b - t + 1$.

Both t and b are initialized to 1. The first operation A_1 is put in instruction $\mathbf{I}_1$. Suppose operations $A_1, A_2, \ldots, A_{i-1}$ have already been scheduled, and the time has come to schedule A_i, where $2 \le i \le n$. Let k denote the smallest integer $\ge t$, such that all predecessors of A_i finish executing before control step k. If A_i has no predecessors, then $k = t$. Otherwise, if a predecessor A_r is in an instruction $\mathbf{I}_j$, where $t \le j \le b$, then $k \ge j + c(A_r)$. The exact value of k is found by taking the maximum of all such expressions.

So, operation A_i can be put in instruction $\mathbf{I}_k$ without violating any dependence constraints. If $k \le b$, start checking the instructions $\mathbf{I}_k, \mathbf{I}_{k+1}, \ldots, \mathbf{I}_b$, in this order, for potential resource conflicts. If an instruction is there in this range with which A_i does not conflict, then put A_i in the first such instruction. Otherwise, A_i conflicts with all instructions in the range and $k = b + 1$. If $k > b$ was true before the checking could start, then its value did not change. At this point, if A_i had no predecessors

Algorithm 4 Given a basic block $(A_1, A_2, \ldots, A_n)$ of n operations, its dependence graph, and a machine with limited resources, this algorithm finds a schedule for the block. At any point in the algorithm, all scheduled operations lie in the sequence of instructions $(\mathbf{I}_t, \mathbf{I}_{t+1}, \ldots, \mathbf{I}_b)$, where $t \le 1$ and $b \ge 1$.

```
t ← 1
b ← 1
Put the operation A_1 in instruction I_1
for i = 2 to n do
    k ← earliest control step ≥ t before which all predecessors of A_i finish executing
    while k ≤ b and A_i has a resource conflict with operations in I_k do
        k ← k + 1
    endwhile
    if k ≤ b then
        put A_i in I_k
    else
        if Pred(A_i) = ∅ then
            Put A_i in I_{t−1}
            t ← t − 1
        else
            Put A_i in I_k
            b ← k
        endif
    endif
endfor
```

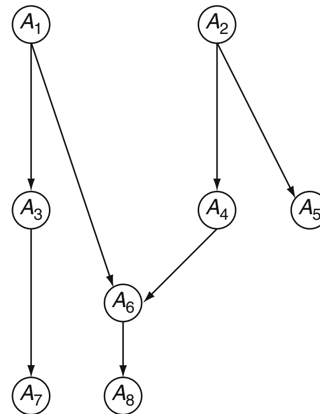
Parallelization, Basic Block. Fig. 4 The linear algorithm

in the first place, then put it at the top in instruction $\mathbf{I}_{t-1}$ and decrease t to $t-1$. If A_i had predecessors, then put it in instruction $\mathbf{I}_k$ and increase b to k. The process ends when all operations in the given basic block have been scheduled.

An Example

Consider a basic block $\mathcal{B}$ consisting of eight operations $A_1, A_2, \ldots, A_8$ arranged in this order. The dependence graph of $\mathcal{B}$ is given in Fig. 5. The type, cycle time, the immediate predecessors, and the immediate successors of each operation are listed in Table 1. It is assumed that the cycle time of an addition is 1, and that of a multiplication is 4. The four algorithms described in this essay are applied to $\mathcal{B}$ one by one. The list scheduling and linear algorithms are customized for a machine with one adder and one multiplier. Note that the critical path in the dependence graph is $A_2 \to A_4 \to A_6 \to A_8$. Since its weight is $T_0 = 10$, the total number of cycles taken by each algorithm to process $\mathcal{B}$ must be at least 10.

ASAP Algorithm. At the beginning, the ASAP label $\ell(A_i)$ of each operation A_i is initialized to 1. Operations A_1 and A_2 have no predecessors. They keep their ASAP labels, that is, they are scheduled to start in control step 1. Since A_3 and A_6 are successors of A_1, and



Parallelization, Basic Block. Fig. 5 Dependence graph of basic block $\mathcal{B}$

A_4 and A_5 are successors of A_2, their ASAP values are increased as follows:

$$\ell(A_3) \leftarrow \max\{\ell(A_3), \ell(A_1) + c(A_1)\} = \max\{1, 2\} = 2$$
$$\ell(A_6) \leftarrow \max\{\ell(A_6), \ell(A_1) + c(A_1)\} = \max\{1, 2\} = 2$$
$$\ell(A_4) \leftarrow \max\{\ell(A_4), \ell(A_2) + c(A_2)\} = \max\{1, 5\} = 5$$
$$\ell(A_5) \leftarrow \max\{\ell(A_5), \ell(A_2) + c(A_2)\} = \max\{1, 5\} = 5.$$

Parallelization, Basic Block. Table 1 Details of the basic block of example 1

OP A	Type	c(A)	Pred(A)	Succ(A)	$\ell(A)$	$f(A)$	$L(A)$	$\mu(A)$
A_1	+	1		A_3, A_6	1	6	5	4
A_2	*	4		A_4, A_5	1	10	1	0
A_3	+	1	A_1	A_7	2	2	9	7
A_4	+	1	A_2	A_6	5	6	5	0
A_5	+	1	A_2		5	1	10	5
A_6	+	1	A_1, A_4	A_8	6	5	6	0
A_7	+	1	A_3		3	1	10	7
A_8	*	4	A_6		7	4	7	0

After A_1 and A_2 have been scheduled, operations A_3, A_4, and A_5 do not have any predecessors remaining to be scheduled. So, they can be scheduled in steps determined by their current ASAP values. Since A_7 is a successor of A_3 and A_6 a successor of A_4, their ASAP values are increased as follows:

$$\ell(A_7) \leftarrow \max\{\ell(A_7), \ell(A_3) + c(A_3)\} = \max\{1, 3\} = 3$$
$$\ell(A_6) \leftarrow \max\{\ell(A_6), \ell(A_4) + c(A_4)\} = \max\{2, 6\} = 6.$$

After A_3 and A_4 have been scheduled, operations A_7 and A_6 do not have any predecessors remaining to be scheduled. So, they can be scheduled in steps determined by their current ASAP values. Since A_8 is a successor of A_6, its ASAP value is increased as follows:

$$\ell(A_8) \leftarrow \max\{\ell(A_8), \ell(A_6) + c(A_6)\} = \max\{1, 7\} = 7.$$

After A_6 has been scheduled, operation A_8 does not have any predecessors remaining to be scheduled. So, A_8 keeps its current ASAP value, and is scheduled to start in step 7. All the ASAP values are shown in Table 1.

The total number of control steps taken to execute $\mathcal{B}$ is

$$\max_{1 \le i \le 8}[\ell(A_i) + c(A_i)] - 1 = 10.$$

The total number of instructions in the schedule is $\max_i \ell(A_i) = 7$. The instructions are $(\mathbf{I}_1, \mathbf{I}_2, \ldots, \mathbf{I}_7)$, where $\mathbf{I}_4$ is empty, and

$$\mathbf{I}_1 = \{A_1, A_2\},\ \mathbf{I}_2 = \{A_3\},\ \mathbf{I}_3 = \{A_7\},$$
$$\mathbf{I}_5 = \{A_4, A_5\},\ \mathbf{I}_6 = \{A_6\},\ \mathbf{I}_7 = \{A_8\}.$$

ALAP Algorithm. At the beginning, T is initialized to 0, and $f(A_i)$ is initialized to 0 for each A_i. Since A_5, A_7, and A_8 have no successors, $f(A_5)$, $f(A_7)$, and $f(A_8)$ are increased as follows:

$$f(A_5) \leftarrow f(A_5) + c(A_5) = 0 + 1 = 1$$
$$f(A_7) \leftarrow f(A_7) + c(A_7) = 0 + 1 = 1$$
$$f(A_8) \leftarrow f(A_8) + c(A_8) = 0 + 4 = 4.$$

These operations have now been scheduled. The value of T is increased to 4. Since A_2 is a predecessor of A_5, A_3 of A_7, and A_6 of A_8, their f values are increased as follows:

$$f(A_2) \leftarrow \max\{f(A_2), f(A_5)\} = \max\{0, 1\} = 1$$
$$f(A_3) \leftarrow \max\{f(A_3), f(A_7)\} = \max\{0, 1\} = 1$$
$$f(A_6) \leftarrow \max\{f(A_6), f(A_8)\} = \max\{0, 4\} = 4.$$

Since A_3 has no successors remaining to be scheduled, $f(A_3)$ is changed as follows:

$$f(A_3) \leftarrow f(A_3) + c(A_3) = 1 + 1 = 2.$$

A_3 is now scheduled. The value of T remains unchanged at 4. Since A_1 is a predecessor of A_3, $f(A_1)$ is increased as follows:

$$f(A_1) \leftarrow \max\{f(A_1), f(A_3)\} = \max\{0, 2\} = 2.$$

Since A_6 has no successors remaining to be scheduled, $f(A_6)$ is changed as follows:

$$f(A_6) \leftarrow f(A_6) + c(A_6) = 4 + 1 = 5.$$

A_6 is now scheduled. The value of T is increased from 4 to 5. Since A_1 and A_4 are predecessors of A_6, their f values are increased as follows:

$$f(A_1) \leftarrow \max\{f(A_1), f(A_6)\} = \max\{2, 5\} = 5$$
$$f(A_4) \leftarrow \max\{f(A_4), f(A_6)\} = \max\{0, 5\} = 5.$$

Now A_1 and A_4 do not have any successors remaining to be scheduled. Their f values are increased again as follows:

$$f(A_1) \leftarrow f(A_1) + c(A_1) = 5 + 1 = 6$$
$$f(A_4) \leftarrow f(A_4) + c(A_4) = 5 + 1 = 6.$$

These two operations are now scheduled. The value of T is increased from 5 to 6. Since A_2 is a predecessor of A_4, $f(A_2)$ is increased from 1 to $f(A_4)$ or 6.

Since A_2 now has no successors remaining to be scheduled, $f(A_2)$ is increased again by $c(A_2)$ to $6+4$, or 10. The value of T is also increased from 6 to 10. Now that the value of T (total number of cycles), and the value of $f(A_i)$ for each A_i are known, the ALAP labels are computed from the equation $L(A) = T+1-f(A)$. The values of $f(A_i)$ and $L(A_i)$ for each A_i are shown in Table 1.

The total number of instructions in the schedule is $\max_i L(A_i) = 10$. The instructions are $(\mathbf{I}_1, \mathbf{I}_2, \ldots, \mathbf{I}_{10})$, where $\mathbf{I}_2, \mathbf{I}_3, \mathbf{I}_4, \mathbf{I}_8$ are empty, and

$$\mathbf{I}_1 = \{A_2\},\ \mathbf{I}_5 = \{A_1, A_4\},\ \mathbf{I}_6 = \{A_6\},$$
$$\mathbf{I}_7 = \{A_8\},\ \mathbf{I}_9 = \{A_3\},\quad \mathbf{I}_{10} = \{A_5, A_7\}.$$

List Scheduling Algorithm. Initially, V has all 8 operations. The initial value of the mobility of each operation is listed in Table 1. At step 1, $\mathcal{S} = \{A_1, A_2\}$. The operations are arranged in the order (A_2, A_1), since $\mu(A_2) < \mu(A_1)$. Both operations can be put in instruction $\mathbf{I}_1$, since there is no resource conflict between them. At step 2, $\mathcal{S} = \{A_3\}$. So, A_3 goes into $\mathbf{I}_2$. Similarly, A_7 goes into $\mathbf{I}_3$. At step 4, $\mathcal{S} = \emptyset$. At step 5, $\mathcal{S} = \{A_4, A_5\}$. The operations are arranged in this order, since $\mu(A_4) < \mu(A_5)$. Only A_4 can be put into $\mathbf{I}_5$. Take A_4 out of $\mathcal{S}$ and decrease $\mu(A_5)$ to 4. At step 6, $\mathcal{S} = \{A_5, A_6\}$. The operations are arranged in the order (A_6, A_5), since $\mu(A_6) < \mu(A_5)$. Only A_6 can be put into $\mathbf{I}_6$. Decrease $\mu(A_5)$ to 3. At step 7, $\mathcal{S} = \{A_5, A_8\}$. The operations are arranged in the order (A_8, A_5), since $\mu(A_8) < \mu(A_5)$. Both operations can be put in instruction $\mathbf{I}_7$ since there is no resource conflict between them. The total number of instructions in the schedule for $\mathcal{B}$ is 7. Those instructions are $(\mathbf{I}_1, \mathbf{I}_2, \ldots, \mathbf{I}_7)$, where

$$\mathbf{I}_1 = \{A_1, A_2\},\ \mathbf{I}_2 = \{A_3\},\ \mathbf{I}_3 = \{A_7\},\quad \mathbf{I}_4 = \emptyset,$$
$$\mathbf{I}_5 = \{A_4\},\quad \mathbf{I}_6 = \{A_6\},\ \mathbf{I}_7 = \{A_5, A_8\}.$$

Total number of cycles $= [7 + \max\{c(A_5), c(A_8)\} - 1] = 10$.

Linear Algorithm. Initially, $t = b = 1$. Put operation A_1 in instruction $\mathbf{I}_1$. Since A_2 has no predecessors and does not conflict with A_1, put A_2 also in $\mathbf{I}_1$. Operation A_3 has only one predecessor, namely, A_1. Since $c(A_1) = 1$, A_3 can be put in instruction $\mathbf{I}_2$. Increase b to 2. The sole predecessor of A_4 is A_2 and $c(A_2) = 4$. Hence, A_4 can go into instruction $\mathbf{I}_5$. Increase b to 5. Operation A_5 also has A_2 as its only predecessor. But, A_5 cannot go into $\mathbf{I}_5$, since it conflicts with A_4. So, put A_5 in $\mathbf{I}_6$ and increase b to 6. Operation A_6 has two predecessors: A_1 and A_4. The earliest instruction that can take A_6 without violating dependence constraints is $\mathbf{I}_6$. But $\mathbf{I}_6$ already has A_5 and it conflicts with A_6. So, put A_6 in $\mathbf{I}_7$ and increase b To 7. Now it is easy to see that A_7 can go into $\mathbf{I}_3$ and A_8 into $\mathbf{I}_8$. Increase b to 8.

The total number of instructions in the schedule for $\mathcal{B}$ is 8. Those instructions are $(\mathbf{I}_1, \mathbf{I}_2, \ldots, \mathbf{I}_8)$, where

$$\mathbf{I}_1 = \{A_1, A_2\},\ \mathbf{I}_2 = \{A_3\},\ \mathbf{I}_3 = \{A_7\},\ \mathbf{I}_4 = \emptyset,$$
$$\mathbf{I}_5 = \{A_4\},\quad \mathbf{I}_6 = \{A_5\},\ \mathbf{I}_7 = \{A_6\},\ \mathbf{I}_8 = \{A_8\}.$$

Total number of cycles $= [8 + c(A_8) - 1] = 11$.

Related Entries

►Code Generation
►Loop Nest Parallelization
►Parallelization, Automatic
►Unimodular Transformations

Bibliographic Notes and Further Reading

Basic block parallelization was first studied in the context of microprogramming. The book by Agerwala and Rauscher [1] gives a good introduction to microprogramming. The theory of job scheduling in operations research turned out to be a rich source of algorithms for the microprogramming research community [3, 4, 7]. For linear analysis and list scheduling as covered in this entry, a good place to start is the paper by Landskov and others [8]. (The algorithms presented above try to factor in explicitly the cycle times of operations.) Other standard references are [2, 5, 6, 10].

Only two algorithms are given in this entry for the limited resource case; there are many more. For example, for Force-directed Scheduling, see [9].

The ten references listed here constitute a small percentage of what is available.

Bibliography

1. Agerwala AK, Rauscher TG (1976) Foundations of microprogramming architecture, software, and applications. Academic, New York
2. Agerwala T (Oct 1976) Microprogram optimization: a survey. IEEE Trans Comput C-25(10):962–973
3. Coffman EG Jr (1976) Computer and job-shop scheduling theory. Wiley, New York
4. Conway RW, Maxwell WL, Miller LW (1967) Theory of scheduling. Addison-Wesley, Reading
5. Dasgupta S, Tartar J (Oct 1976) The identification of maximal parallelism in straightline microprograms. IEEE Trans Comput C-25(10):986–992
6. Gonzalez MJ Jr (Sep 1977) Deterministic processor scheduling. ACM Comput Surv 9(3):173–204
7. Hu TC (Nov-Dec 1961) Parallel sequencing and assembly line problems. Oper Res 9(6):841–848
8. Landskov D, Davidson S, Shriver B, Mallett PW (Sep 1980) Local microcode compaction techniques. Comput Surv 12(3):261–294
9. Paulin PG, Knight JP (June 1989) Force-directed scheduling for the behavioral synthesis of ASIC's. IEEE Trans CAD Integ Circ Syst 8(6):661–679
10. Ramamoorthy CV, Chandy KM, Gonzalez MJ (Feb 1972) Optimal scheduling strategies in a multiprocessor system. IEEE Trans Comput C-21(2):137–146

Parallelization, Loop Nest

►Loop Nest Parallelization

ParaMETIS

►METIS and ParMETIS

PARDISO

Olaf Schenk[1], Klaus Gärtner[2]
[1]University of Basel, Basel, Switzerland
[2]Weierstrass Institute for Applied Analysis and Stochastics, Berlin, Germany

Definition

PARDISO, short for "PARallel DIrect SOlver," is a thread-safe software library for the solution of large sparse linear systems of equations on shared-memory multicore architectures. It is written in Fortran and C and it is available at www.pardiso-project.org. The solver implements an efficient supernodal method, which is a version of Gaussian elimination for large sparse systems of equations, especially those arising, for example, from the finite element method or in nonlinear optimization. It is the only sparse solver package that supports all kinds of matrices such as complex, real, symmetric, nonsymmetric, or indefinite. PARDISO can be called from various environments including MATLAB (via MEX), Python (via pypardiso), C/C++, and Fortran. PARDISO version 4.0.0 was released in October 2009.

Discussion

Introduction

The solution of large sparse linear systems lies at the heart of many calculations in computational science and engineering and is also of increasing importance in computations in the medical imaging and financial sectors. Today, systems of equations with millions to hundreds of millions of unknowns are solved. To do this within reasonable time requires efficient use of powerful parallel computers and advanced combinatorial and numerical algorithms based on direct or approximate direct factorizations. To date, only very limited software for such large systems is generally available. The PARDISO software addresses this issue.

In this chapter, some important combinatorial aspects and main algorithmic features for solving sparse systems will be reviewed. The algorithmic improvements of the past 20 years have reduced the time required to factor general sparse matrices by almost three orders of magnitude. Combined with significant advances in the performance to cost ratio of computing hardware during this period, current sparse solver technology makes it possible to solve those problems quickly and easily, which might have been considered by far too large until recently. This chapter discusses the basic and the latest developments for sparse direct solution methods that have lead to modern *LU* decomposition techniques.

The PARDISO development started in the context of a PhD Project [12] at ETH Zurich in Switzerland.

The first aim was to improve parallel sparse factorization methods for highly ill-conditioned matrices arising in the semiconductor device simulation area. The close collaboration with the simulation community resulted in a large amount of test cases and industrial use of the solver.

The library version of PARDISO is publicly released since March 2004. Ongoing research projects resulted in the current version 4.0.0, available since October 2009. This new version also includes novel state-of-the-art incomplete factorization-type preconditioners. The results of the research have appeared in several scientific journals, including the paper "On Large Scale Diagonalization Techniques for the Anderson Model of Localization" in the SIGEST section of the SIAM Review Journal [13].

It is the purpose of this chapter to describe the main combinatorial and numerical algorithms used in an efficient parallel solver.

Sparse Gaussian Elimination in PARDISO

To introduce the notations, the LU-factorization of a nonsymmetric matrix A with pivoting is described. A simple description of the algorithm for solving sparse linear equations by sparse Gaussian elimination in PARDISO is as follows:

- Compute the triangular factorization $P_r D_r P_f A P_f^T D_c P_c = LU$. Here D_r and D_c are diagonal matrices to equilibrate the system, P_f is a permutation matrix that minimizes the number of nonzeros in the factor, and P_r and P_c are permutation matrices. Premultiplying A by P_r reorders the rows of A, and premultiplying A by P_c reorders the columns of A. P_r and P_c are chosen to enhance sparsity, numerical stability, and parallelism. L is a unit lower triangular matrix with $L_{ii} = 1$ and U is an upper triangular matrix.
- Solve $AX = B$ by evaluating

$$X = A^{-1}B = (P_f^{-1} D_r^{-1} P_r^{-1} LU P_c^{-1} D_c^{-1} P_f^{-T})^{-1} B,$$

$$X = P_f^T D_c P_c U^{-1} L^{-1} P_r D_r P_f B.$$

  This is done efficiently by multiplying from right to left in the last expression: the rows of B are permuted with P_f and scaled by D_r. Multiplying $P_r B$ means permuting the rows of $D_r B$. Multiplying $L^{-1}(P_r D_r B)$ means solving triangular systems of equations with n_r right-hand sides with matrix L by substitution. Similarly, multiplying $U^{-1}(L^{-1}(P_r D_r P_f B))$ means solving triangular systems with U.

For symmetric indefinite matrices, PARDISO performs a numerical factorization into LDL^T, in which the factorization is also stabilized by extended numerical pivoting techniques. In addition to the complete factorization, advanced support for incomplete inverse-based factorization preconditioners, in which the factors are computed approximately, are also included in PARDISO.

The efficient computation of the LU decomposition is of utmost importance in Gaussian elimination. Typically, a compact representation of the elimination tree can be used to derive all information concerning fill-in and numerical dependencies. In particular, the fact that pivoting and the factorization must be interlaced requires a completely different treatment than in the case without pivoting. Consider the situation when computing the LU decomposition column by column.

for $k = 1, \ldots, n$
  update column k of L and U via the equation $A_{1:n,k} = L_{1:n,1:k-1} U_{1:k-1,k} - L_{1:n,k} u_{kk}$
  1. step. solve $L_{1:k-1,1:k-1} U_{1:k-1,k} = A_{1:k-1,k}$
  2. step. $L_{k:n,k} := A_{k:n,k}$
    for $i < k$ such that $u_{ik} \neq 0$
      $L_{k:n,k} := L_{k:n,k} - L_{k:n,i} u_{ik}$
  3. step. $u_{kk} = l_{kk}$
    $L_{k:n,k} = L_{k:n,k} / u_{kk}$

Note that it is easy to add pivoting after step 2 by interchanging l_{kk} with some sufficiently large $|l_{mk}|$, where $m \geqslant k$ before u_{kk} is defined. The art of efficiently computing column k of L and U consists of how the sparse forward solve $L_{1:k-1,1:k-1} U_{1:k-1,k} = A_{1:k-1,k}$ in step 1 is efficiently implemented and, a fast update of $L_{k:n,k}$ in step 2.

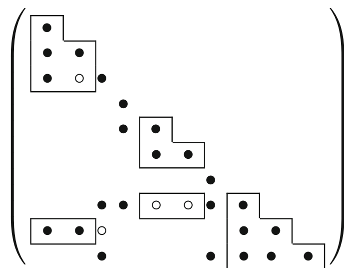
The dense LU decomposition as it is described so far can be implemented in a way that it makes heavily use of level-3 BLAS. One important aspect that allows raising efficiency and speeding up the sparse numerical factorization will be discussed now. It is the recognition

of an underlying block structure with dense submatrices caused by the factorization and the fill. The block structure allows to collect parts of the matrix in dense blocks and to treat them commonly using higher levels of BLAS. As a consequence of the LU decomposition, parts of the triangular factors can be encountered that are dense or become dense by the factorization. This key structure in sparse Gaussian elimination is based a on supernodal representation of the columns and rows in the factors L and U. A sequence $k, k+1, \ldots, k+s-1$ of s subsequent columns in L that form a dense lower triangular matrix is called a supernode.

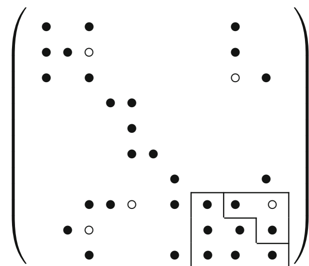
To illustrate the use of supernodes, Figs. 1 and 2 illustrate the underlying dense block structure.

Supernodes in PARDISO are stored in rectangular dense matrices. Beside the storage scheme as dense matrices, the nonzero row indices for these blocks need only be stored once. Next, the use of dense submatrices allows the usage of level-3 BLAS routines. To understand this, one can easily verify that the update process

$$L_{k:n,k} := L_{k:n,k} - L_{k:n,i}u_{ik}$$



PARDISO. Fig. 1 Supernodes in L, symmetric case ("○" denotes fill-in)



PARDISO. Fig. 2 Supernodes in $L+U$, general case ("○" denotes fill-in)

that computes $L_{k:n,k}$ in the algorithm can easily be extended to the block case. Assumung that columns $1, \ldots, k+s-1$ are collected in p supernodes with column indices $\mathcal{K}_1, \ldots, \mathcal{K}_p$. Apparently, the column set is $\mathcal{K}_p = \{1, \ldots, k+s-1\}$. Then updating $L_{k:n,\mathcal{K}_p}$ can be rewritten as

$$L_{k:n,\mathcal{K}_p} := L_{k:n,\mathcal{K}_p} - L_{k:n,\mathcal{K}_i} U_{\mathcal{K}_i,\mathcal{K}_p},$$

where the sum has to be taken over all i in the block version of the elimination tree. Depending on whether one would like to compute the diagonal block $L_{\mathcal{K}_p,\mathcal{K}_p}$ as full or as lower triangular matrix one is able to use level-2 BLAS or even level-3 BLAS subroutines. This allows exploiting machine-specific properties, such as caches to accelerate the computation.

As discussed above, dynamic pivoting has been a central tool by which nonsymmetric sparse linear solvers gain stability. Therefore, improvements in speeding up direct factorization methods were limited to the uncertainties that have arisen from using pivoting. Certain techniques, like the column elimination tree [2], have been useful for predicting the sparsity pattern despite pivoting. However, in the symmetric case, the situation becomes more complicated since only symmetric reorderings, applied to both columns and rows, are required, and no a priori choice of pivots is given. This makes it almost impossible to predict the elimination tree in a sensible manner, and the use of cache-oriented level-3 BLAS is impossible.

With the introduction of symmetric maximum weighted matchings [16] as an alternative to complete pivoting [3], it is now possible to treat symmetric indefinite systems similarly to symmetric positive definite systems. This allows to predict fill using the elimination tree, and thus allows to set up the data structures that are required to predict dense submatrices (also known as supernodes). This in turn means that one is able to exploit level-3 BLAS applied to the supernodes. Consequently, the classical Bunch–Kaufman pivoting approach needs to be performed only inside the supernodes.

This approach has recently been successfully implemented in symmetric indefinite version of PARDISO [16]. As a major consequence of this novel approach, the sparse indefinite solver has been improved to become almost as efficient as its symmetric positive counterpart.

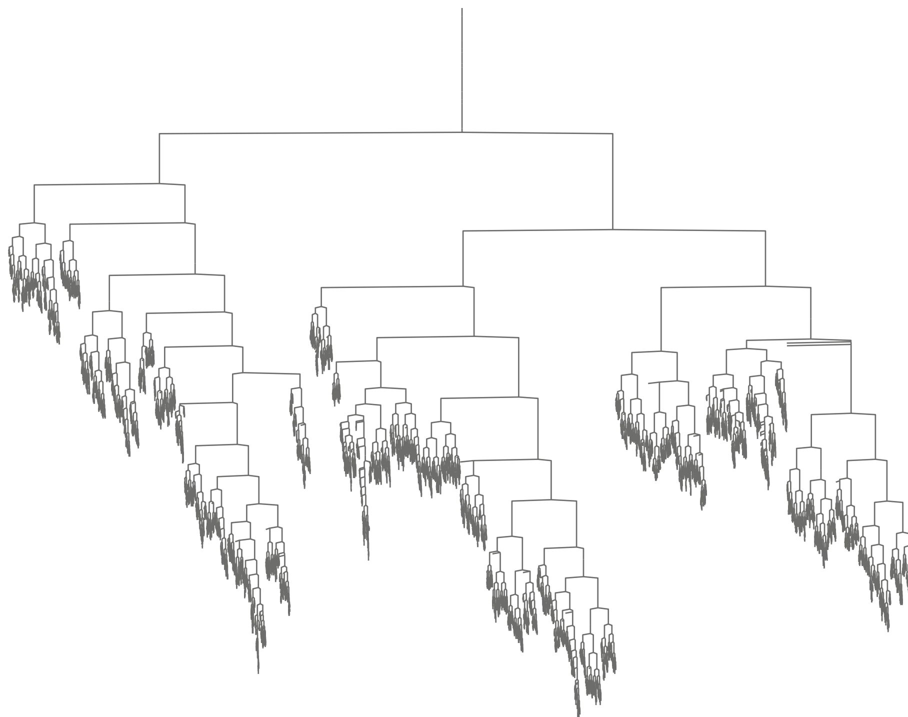
Finally, iterative refinement is the last option to test and enhance the precision of the solution. To encourage the use of L and U as preconditioner in solving continuously dependent families of problems, PARDISO also offers a CG and CGS branch.

Reordering Algorithms and Software in PARDISO

The efficiency of a direct solver depends strongly on the order in which the variables of the matrix are eliminated. This largely determines the computational amount of work executed during the numerical factorization and hence the overall time for the solution. Furthermore, the elimination order of the variables also determines the number of entries in the computed factors and the amount of main storage required. Experiments have shown that the multiple minimum degree (MMD) algorithm of J. Liu [8] is very well suited for 2D problems, but nested dissection type orderings do a much better job on larger problems from 3D discretizations. Thus, the ordering METIS package from Karypis and Kumar [7] and a constrained minimum fill-in orderings developed by Schenk and Gärtner [12] have been added to PARDISO. In a somewhat more loosely coupled way, orderings from other packages such as approximate minimum degree [1] can also be used.

Parallelization Strategies in PARDISO

Parallelism can be expressed by the elimination tree: different branches are independent (see Fig. 3). The length of each vertical line is proportional to the size of a dense diagonal block. The hight counted in levels characterizes the longest chain of sequentially depending steps. Memory writes should be minimized and asynchronous scheduling is possible in PARDISO on SMP and NUMA architectures. It is used to reduce load balance requirements. Hence, the small amount of synchronization data is passed towards the root (or to the right in L, U), while the numerical data is read from the left of the actual supernode. Especially large supernodes close to the root of the elimination tree are split into panels to increase the number of parallel tasks. PARDISO uses METIS by default to generate nested dissection permutations and maps all sparse computations on dense matrix operations to achieve level-3 BLAS performance for the numerical factorization. Scheduling



PARDISO. Fig. 3 Typical elimination tree with a MMD reordering

distinguishes two levels in the elimination tree to reduce synchronization events: updates within the lower level are conflict free, hence must not be synchronized. In PARDISO, complete lower level subtrees of the elimination are mapped onto a single core of the multiprocessing architecture. The columns and rows associated with this complete subtree are factorized independently with respect to other subtrees. The nodes near the root of the elimination tree normally involve more computation than supernodes further away from the root. In practical examples more than 75% of the floating-point computations are performed on the supernodes close to the root the tree. Unfortunately, the number of independent supernodes near the root of the tree is small, and so there is less parallelism to exploit. For example, the root in Fig. 3 has only two neighboring nodes and hence only two processors would perform the corresponding work while the other processors remain idle. The introduction of the panels and the execution of partial updates as tasks for supernodes close to the root increases the parallelism.

An additional constrained that has to be taken into account is that PARDISO is designed to solve a wide range of application problems including symmetric and nonsymmetric matrices, and numerical pivoting is performed within the numerical factorization. This means that only a static analysis of the sparsity pattern and a static allocation of supernodes to cores could be very inefficient if numerical pivoting is required. As a solution, a novel left-right looking factorization has been implemented that uses a dynamic allocation of threads during the numerical factorization. This has the benefit of enabling the linear scalability of the code to perform well on multicore architectures with up to 16 cores. The parallel execution in PARDISO does not change the constant in the leading order of the complexity bound compared with that of the sequential algorithm.

Approximate Sparse Gaussian Factorization in PARDISO

The solver also includes a novel preconditioning solver [13]. The preconditioning approach for symmetric indefinite linear system is based on maximum weighted matchings and algebraic multilevel incomplete LDL^T factorization. These techniques can be seen as a complement to the alternative idea of using more complete pivoting techniques for the highly ill-conditioned symmetric indefinite matrices. In considering how to solve the linear systems in a manner that exploits sparsity as well as symmetry, a diverse range of algorithms is used that includes preprocessing the matrix with symmetric weighted matchings, solving the linear system with Krylov subspace methods, and accelerating the linear system solution with multilevel preconditioners based upon incomplete inverse-based factorizations.

General Software Issues in PARDISO

The PARDISO package was originally designed for structurally symmetric matrices arising in the semiconductor device simulation area, but, in the newer version, all other types of matrices such as real and complex symmetric, real or complex nonsymmetric systems, or complex Hermitian systems are permitted. The PARDISO software is written in C and Fortran. However, in recognition that some users prefer a Matlab or Python programming environment, an appropriate interface has been developed for these programming languages.

It requires OpenMP threading capabilities and makes heavy use of level-3 BLAS and LAPACK subroutines. For the parallel version of PARDISO, a single-threaded version of level-3 BLAS and LAPACK routines is used. PARDISO has been ported to a wide range of computers from previous generations of vector-computers such as Cray or NEC, to all kind of multicore architectures including Intel, AMD, IBM, SUN, and SGI [15] and also to recent throughput manycore processors such as GPUs from NVIDIA [14].

The PARDISO package has an excellent performance relative to other parallel sparse solvers. An extensive evaluation of the performance of the numerical factorization in comparison to a wide range of other sparse direct linear solver packages is given by Gould, Hu, and Scott [4]. Other independent comparisons can be found in [5]. PARDISO is currently heavily used in linear and nonlinear optimization solvers. Recent results can be found in [17]. The current version of the solver and a manual including several examples is available at www.pardiso-project.org.

Example

The example illustrates the complexity gap between numerical factorization and solution in the sparse direct

solution process. As shown in Table 1, the numerical factorization is the most time-consuming phase. This is demonstrated on a 3D eigenvalue problem. The symmetric eigenvalue problem $Ax = \lambda Bx$ is chosen with up to 3.5×10^6 unknowns to be close to the size limits of a today's 128 GB SMP machines.

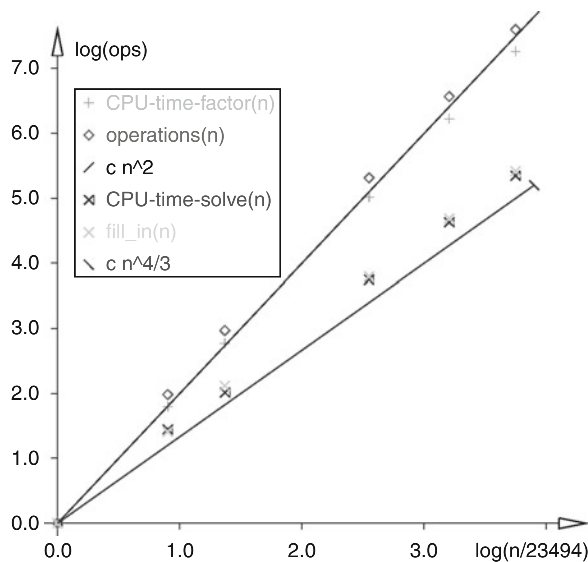
The eigenvalue problem is solved by inverse iteration, polynomial acceleration, a few different spectral shifts, and the same number of factorizations. On average, five eigenvalues are computed per factorization. The spectral shifts can be chosen to result in highly ill-conditioned but sufficiently regular sparse linear systems. The symmetric indefinite linear systems are solved by the Bunch-Kaufman pivoting. The quadratic complexity of the sparse direct factorization is shown in Fig. 4.

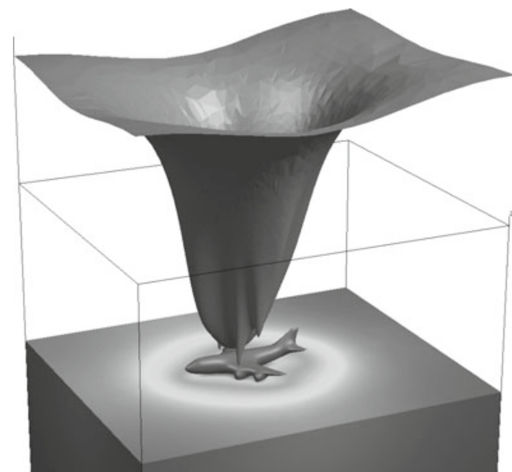
PARDISO. Table 1 The serial computational complexity of the various phases of solving a sparse system of linear equations arising from 2D and 3D constant node-degree graphs with *n* vertices

Phase	Dense	2D complexity	3D complexity
Reordering:	–	O(n)	O(n)
Symbolic factorization	–	$O(n \log n)$	$O(n^{4/3})$
Numerical factorization	$O(n^3)$	$O(n^{3/2})$	$O(n^2)$
Triangular solution	$O(n^2)$	$O(n \log n)$	$O(n^{4/3})$

Future Research Directions

Clearly, the quadratic complexity bound is limiting the problem size in 3D for sparse direct methods. However, from the user perspective, the robustness and the nearly achieved black box behavior are very convenient. The emergence of multicore architectures and scalable petascale architectures does not change both points. Instead, it motivates the development of novel algorithms and techniques that emphasize both concurrency and robustness for the solution of sparse linear systems. Since direct solvers do not generally scale to large problems and machine configurations, efficient application of preconditioned iterative solvers are warranted but involve more a priori knowledge. These hybrid solvers must optimize parallel performance, processor (serial) performance, as well as memory requirements, while being robust across specific classes of applications and systems.





PARDISO. Fig. 4 Unstructured tetrahedral meshes with $n = 23'494, 57'869, 92'328, 300'608, 579'150$ and $996'557$ tetrahedra for a discrete Laplace problem. The *left* graphic shows the measured times for the numerical factorization and the solution, the number of floating-point operations, and the complexity bounds. The first eigenvector is shown in the *right* graphic. The selfsimilar tetrahedral meshes are generated by TetGen, http://www.tetgen.org

The research groups at Purdue University and University of Basel are currently developing a new parallel solver PSPIKE [10] that combines the desirable characteristics of direct methods (robustness) and effective iterative solvers (low computational cost), while alleviating their drawbacks (limited scalability, memory requirements, lack of robustness). The hybrid solver is based on the general sparse solver PARDISO and the Spike family of hybrid solvers [11]. The resulting PSPIKE algorithm is an extension of PARDISO to distributed-memory architectures. Results can be found in, for example, [10].

Related Entries

►BLAS (Basic Linear Algebra Subprograms)
►Graph Partitioning
►Load Balancing, Distributed Memory
►Linear Algebra Software
►Nonuniform Memory Access (NUMA) Machines
►Shared-Memory Multiprocessors
►LAPACK

Further Reading

1. Davis T (2006) Direct methods for sparse linear systems. Society for industrial mathematics, ISBN:0898716136

Bibliography

1. Amestoy R, Davis TA, Duff IS (1996) An approximate minimum degree ordering algorithm. SIAM J Matrix Anal Appl 17:886–905
2. Demmel JW, Eisenstat SC, Gilbert JR, Li XS, Liu JWH (1999) A supernodal approach to sparse partial pivoting. SIAM J Matrix Anal Appl 20:720–755
3. Duff IS, Koster J (1999) The design and use of algorithms for permuting large entries to the diagonal of sparse matrices. SIAM J Matrix Anal Appl 20(4):889–901
4. Gould NIM, Hu Y, Scott JA (2007) A numerical evaluation of sparse direct solvers for the solution of large sparse, symmetric linear systems of equations. ACM Trans Math Software (TOMS) 33(2):1–32
5. Grasedyck L, Hackbusch W, Kriemann R (2008) Performance of H-LU preconditioning for sparse matrices. Comput Methods Appl Math 8(4):336–349
6. Hagemann M, Schenk O (2006) Weighted matchings for preconditioning of symmetric indefinite linear systems. SIAM J Sci Comput 28:403–420
7. Karypis G, Kumar V (1998) A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM J Sci Comput 20:359–392
8. Liu J (1985) Modification of the minimum-degree algorithm by multiple elimination. ACM Trans Math Software 11(2):141–153
9. Ng E, Peyton B (1993) Block sparse Cholesky algorithms on advanced uniprocessor computers. SIAM J Sci Comput 14:1034–1056
10. Manguoglu M, Sameh A, Schenk O (2009) PSPIKE Parallel sparse linear system solver. In: Proceedings of the 15th international Euro-Par conference on parallel processing. Lecture Notes in Computer Science, vol 5704, pp 797–808, DOI 10.1007/978-3-642-03869-3 (vol 14, pp 1034–1056)
11. Polizzi E, Sameh AH (2006) A parallel hybrid banded system solver: the SPIKE algorithm. Parallel Comput 32(2):177–194
12. Schenk O (2000) Scalable parallel sparse LU factorization methods on shared memory multiprocessors. PhD thesis, ETH Zürich
13. Schenk O, Bollhöfer M, Römer RA (2008) On large-scale diagonalization techniques for the Anderson model of localization. SIAM Rev 50:91–112
14. Schenk O, Christen M, Burkhart H (2008) Algorithmic performance studies on graphics processing unit. J Parallel Distrib Comput 28:1360–1369
15. Schenk O, Gärtner K (2004) Solving unsymmetric sparse systems of linear equations with PARDISO. J Future Gener Comput Syst 20(3):475–487
16. Schenk O, Gärtner K (2006) On fast factorization pivoting methods for symmetric indefinite systems. Electron Trans Numer Anal 23:158–179
17. Schenk O, Wächter A, Hagemann M (2007) Matching-based preprocessing algorithms to the solution of saddle-point problems in large-scale nonconvex interior-point optimization. Comput Optim Appl 36(2–3):321–341

PARSEC Benchmarks

The PARSEC benchmarks [1] are mutithreaded codes that represent important applications of multicores. PARSEC stands for Princeton Application Repository for Shared-Memory Computers. Currently, the benchmark suite contains 13 programs: blackscholes, bodytrack, canneal, dedup, facesim, ferret, fluidanimate, freqmine, raytrace, streamcluster, swaptions, vips, and x264.

Bibliography

1. Bienia C (2011) Benchmarking modern multiprocessors. Ph.D. thesis, Princeton University

Partial Computation

►Trace Theory

Particle Dynamics

►*N*-Body Computational Methods

Particle Methods

►*N*-Body Computational Methods

Partitioned Global Address Space (PGAS) Languages

►PGAS (Partitioned Global Address Space) Languages

PASM Parallel Processing System

Howard Jay Siegel, Bobby Dalton Young
Colorado State University, Fort Collins, CO, USA

Definition

PASM was a partitionable mixed-mode parallel system designed and prototyped in the 1980s at Purdue University to study three dimensions of dynamic reorganization: mixed-mode parallelism, partitionability, and flexible interprocessor communications.

Discussion

Introduction

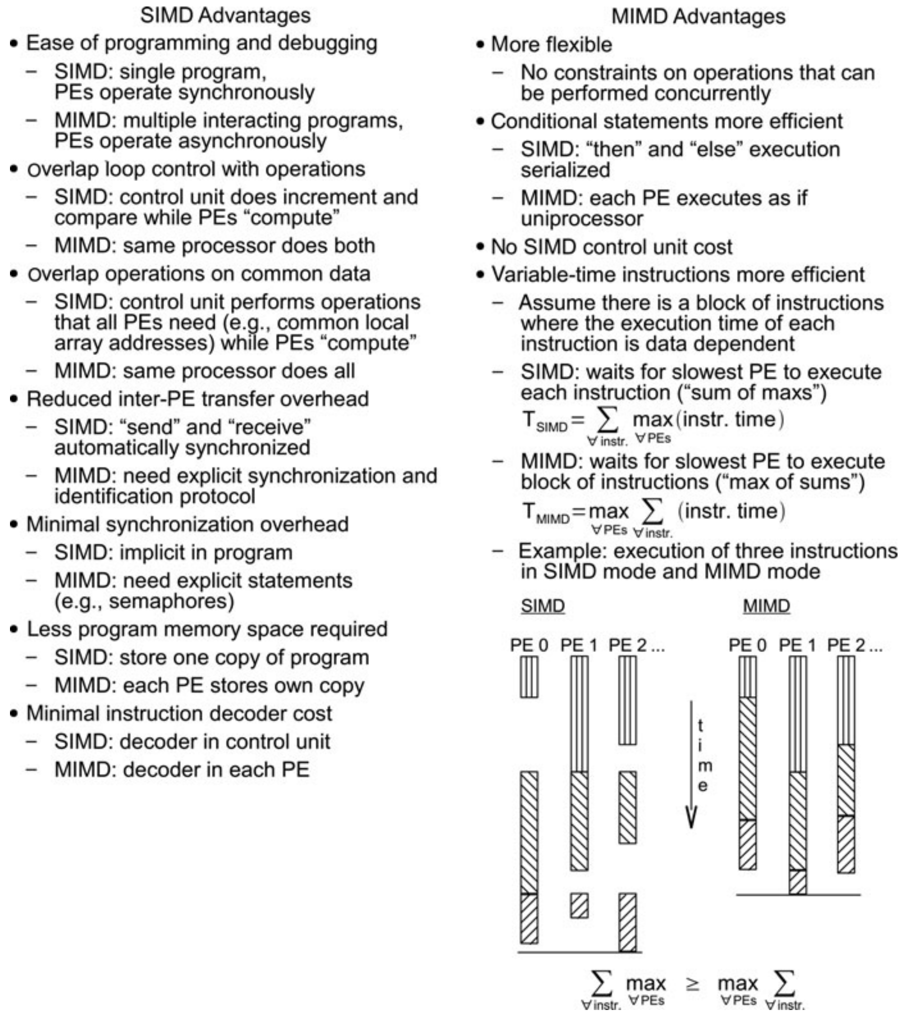
PASM was a partitionable mixed-mode parallel system designed and prototyped in the 1980s at Purdue University [20, 21]. Research was conducted about numerous software, hardware, parallel algorithm, and application aspects of PASM.

In the 1980s, two of the dominant organizations for parallel machines were "SIMD" and "MIMD" [8]. Be forewarned that our use of the term "SIMD" is more general than the way it is used with current multicore systems, where it refers to operating on subfields of a long data word within a single processor. We will use the following definition: An *SIMD* (single instruction stream, multiple data stream) machine consists of N processor-memory pairs, an interconnection network, and a control unit. The *control unit* broadcasts a sequence of instructions to the N processors that execute these instructions in lockstep (this is the "single instruction stream"). All "enabled" (active) processors execute the same instruction at the same time, but each processor does it on its own data. The operands for these instructions are fetched from the local memory associated with each processor (the fetching of operands from the collection of local memories form the "multiple data streams"). The *interconnection network* supports interprocessor communication. Thus, the SIMD definition used here assumes a collection of separate processors operating in lockstep, with all enabled processors following the same single instruction stream, but each processor operating on its own local data, resulting in multiple data streams. Examples of SIMD systems that have been constructed are Illiac IV [6], MasPar MP-1 and MP-2 [5], and MPP [4].

A *multiple-SIMD* system can be structured as a single SIMD machine or as two or more independent SIMD machines of various sizes. The Thinking Machines CM-2 [23] and original design of the Illiac IV [3] are examples of MSIMD systems.

The *MIMD* (multiple instruction stream, multiple data stream) mode of parallelism [8] uses N independent processor-memory pairs that can communicate via an interconnection network (i.e., a multicomputer). Each processor fetches its own instructions and its own operands from its local memory; thus, there are "multiple instruction streams" and "multiple data streams." The nCUBE 2 [9] and IBM RP3 [15] are examples of MIMD systems that have been constructed. One mode of operation possible for MIMD is *SPMD* (single program, multiple data stream), where all processors independently execute the same program on different data sets [13].

A *mixed-mode* system can dynamically change between the SIMD and MIMD modes of parallelism at instruction-level granularity with negligible overhead. This allows different modes of parallelism to be used to execute various portions of an algorithm. Because the mode of parallelism has an impact on performance (see Fig. 1), a mixed-mode system may outperform a single-mode machine with the same number of processors for a given algorithm. A *partitionable mixed-mode* system can dynamically restructure to form independent



PASM Parallel Processing System. Fig. 1 Trade-offs between the SIMD and MIMD modes of parallelism [2]. Processing Elements (PEs) are processor-memory pairs. Examples of variable-time instructions are floating-point operations on the prototype's Motorola MC68000 processors, and function calls (such as floating-point trigonometric operations) on current GPUs

or communicating submachines of various sizes, where each submachine can independently perform mixed-mode parallelism (e.g., TRAC [10]).

PASM is a *PA*rtitionable-*S*IMD/*M*IMD system concept that was developed at Purdue University as a design for a large-scale partitionable mixed-mode machine based on commodity microprocessors [21]. PASM used a flexible multistage interconnection network for interprocessor communication. Thus, PASM could be dynamically reorganized along these three dimensions: partitionability, mode of parallelism, and connections among processors. This ability to be reorganized allowed the system to match the computational structure of a problem, and also provided for fault tolerance in many situations. This fault tolerance is a form of *robustness* [1, 17], where the robust behavior requirement is continued system operation given the uncertainty of any single component failure, and quantified as the portion of the system that can still be used after a fault occurs.

A small-scale prototype was completed in 1987 as a proof-of-concept machine for the PASM design ideas, and was still running in 1995, with over 36,000 hours of execution time logged (Fig. 2) [20]. It was used

PASM Parallel Processing System. Fig. 2 A photograph of the completed PASM prototype circa 1987 and some of the PASM team. *Left* to *right*: Pierre Perrot (Technician), Tom Casavant (Professor), Wayne Nation (PhD Student), H.J. Siegel (Professor, team leader)

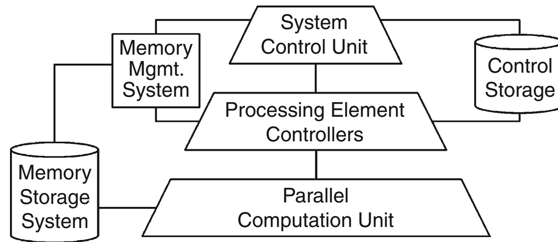
for research and in a parallel programming course at Purdue. The goal of the PASM research team was to design, develop, and build a unique research tool for studying the combined three dimensions of dynamic reorganization mentioned above: mixed-mode parallelism, partitionability, and flexible interprocessor communications.

An overview of the organization of PASM is given in section "▸The Overall PASM Organization." Section "▸The Parallel Computation Unit" describes the processing elements, memory, and interconnection network in the Parallel Computation Unit. The Memory Storage and Memory Management Systems are discussed in section "▸The Memory Storage and Management Systems." Section "▸Using the PASM System" mentions some PASM prototype software tools and application studies. We conclude in section "▸Conclusions" with a list of advantages of the PASM approach. A list of PASM-related publications is available at http://hdl.handle.net/10217/34662.

The Overall PASM Organization

The PASM concept was a distributed-memory machine with a computational engine consisting of processor-memory pairs referred to as *PE*s (*P*rocessing *E*lements). The PASM design concepts could support at least 1,024 PEs. The small-scale prototype built at Purdue University (30 Motorola MC 68,000 processors, 16 in the computational engine) supported experimentation with all three dimensions of reconfigurability mentioned earlier, and produced insights not observed from earlier simulations and theoretical studies.

The PASM system concept consists of six basic components shown in Fig. 3. The *System Control Unit* is the overall system coordinator and is the part of PASM with which the user directly interacts. The *Q PE Controllers* (*PECs*) serve as the PE control units in SIMD mode and may coordinate the PEs in MIMD mode. The *Parallel Computation Unit* contains the $N = 2^n$ PEs, physically numbered 0 to $N - 1$, and the interconnection network used by the PEs. The *Memory Storage System* provides secondary storage for the PEs, and consists of N/Q secondary storage devices, each with an associated processor for file management. It is used to store data files for SIMD mode and both program and data files for MIMD mode. The *Memory Management System* contains multiple processors for controlling the transferring of files between the Memory Storage System and the PEs. *Control Storage* consists of a secondary storage device and an associated file server processor. It supports the PECs and the System Control Unit by holding all code and data used by these components, including the instructions to be broadcast to the PEs from the PECs in SIMD mode. The PASM prototype had a total of 30 processors: $N = 16$ PEs, $Q = 4$ PECs, $N/Q = 4$ Memory Storage System processors, four Memory Management



PASM Parallel Processing System. Fig. 3 The high-level architectural organization of the PASM design concept

System processors, the Control Storage processor, and the System Control Unit.

The tasks to be performed by the System Control Unit include support for program development, job scheduling, general system coordination, management of system configuration and partitioning, assignment of user jobs to submachines, and connection to the host computer network. The hardware needed to combine and synchronize the PECs and PEs to form SIMD submachines of various sizes resides in the System Control Unit. Its functions include combining information from multiple PECs when collective conditionals, discussed in the next section, are performed. It also is responsible for coordinating the loading of the PE memories from the Memory Storage System with the loading of the PEC memories from the Control Storage.

We now describe how the PECs are connected to the PEs, and how this connection scheme is used to form independent mixed-mode submachines. The organization we use provides an efficient PEC/PE interface and supports the partitioning of the PE Interconnection Network.

The PECs, shown in Fig. 4, are the multiple control units required to form a multiple-SIMD system. There are $Q = 2^q$ PECs, physically numbered from 0 to $Q - 1$. Each PEC controls a fixed group of N/Q PEs in the Parallel Computation Unit. A PEC and its associated PEs form a *PEC Group*. PEC *i* is connected to the N/Q PEs whose low-order *q* bits of their PE physical number have the value *I* (for reasons discussed in section "▸The Parallel Computation Unit"). In an $N = 1{,}024$ system, Q may be 32; for the $N = 16$ prototype, $Q = 4$. Figure 5 shows the composition of the physical number of a PE.

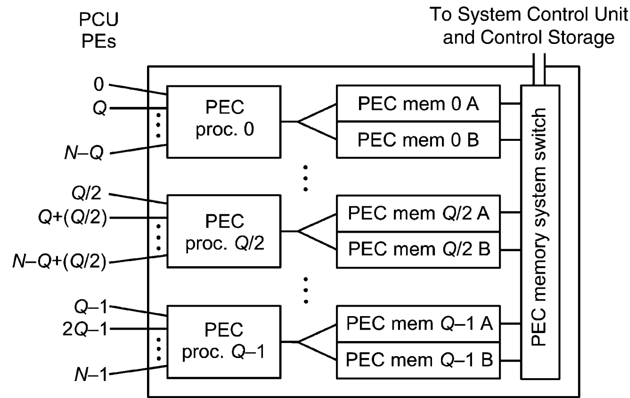
The memory units in each PEC are double-buffered so that computation and memory I/O can be overlapped (see Fig. 4). For example, the PEC processor can execute a job in one memory unit while the next job is preloaded into the other memory unit from PEC secondary storage (the Control Storage). In MIMD mode, a PEC fetches from its memory the instructions and data used to coordinate the operation of its PEs. In SIMD mode, a PEC fetches the instructions and common PE data from its memory units. In general, in SIMD mode, control-flow instructions are executed in the PEC, and data processing instructions are broadcast to the PEC's group of PEs.

Submachines are formed by one or more PEC Groups. The partitioning rule in PASM is that the numbers of all PEs in a submachine of size 2^p must agree in their $n - p$ low-order bit positions (for the reasons discussed in section "▸The Parallel Computation Unit"). The *p* high-order bits of the physical number of a PE correspond to the PE's logical number within the partition. Thus, a submachine containing $R = 2^r$ PEC Groups ($R^*N/Q = 2^p$ PEs), where $0 \leq r \leq q$, is formed by combining the PEs connected to the *R* PECs whose addresses agree in their $q - r$ low-order bits. The PECs within a submachine of R^*N/Q PEs are logically numbered from 0 to $R - 1$. For $R > 1$, the logical number of a PEC is the high-order *r* bits of its physical number. Similarly, the PEs assigned to a submachine are logically numbered from 0 to $(R^*N/Q)-1$ (0 to $2^p - 1$). The logical number of a PE is the high-order $r + n - q = p$ bits of its physical number (see Fig. 5). There is a maximum of *Q* submachines, each of size N/Q.

Each submachine can operate as an independent mixed-mode system. PEs can switch between modes as often as desired; however, all PEs in a submachine must be in the same mode (SIMD or MIMD) at any point in time.

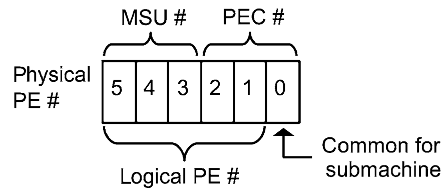
When a submachine is operating in SIMD mode, the *R* PECs must execute and broadcast the same instructions, and all PEs in the submachine must be synchronized. The PECs are given the same instruction simply by loading the memory units of the *R* PECs with the same program. The PEs are synchronized by providing a small amount of special circuitry in the System Control Unit that coordinates instruction broadcasts among these PECs.

Some advantages of this fixed PE to PEC mapping as compared with a dynamic PE to PEC interconnection (e.g., a crossbar switch [14]) include reducing



PASM Parallel Processing System. Fig. 4 PASM Processing Element Controllers (PECs), and how they are connected to the Parallel Computation Unit (PCU) Processing Elements (PEs)

PEC/PE interface hardware, eliminating the overhead of maintaining a record of PE to PEC assignments, scheduling only Q PECs instead of N PEs, allowing the partitioning of the PE interconnection network into independent subnetworks (section "►The Parallel Computation Unit"), and supporting the structure of the efficient parallel primary to secondary memory connections (section "►The Memory Storage and Management Systems"). The main disadvantage to this approach is that the size of each submachine must be a power of two, with a minimum of N/Q PEs. However, for PASM's intended experimental environment, flexibility at a "reasonable" cost is the goal, not maximum PE utilization.

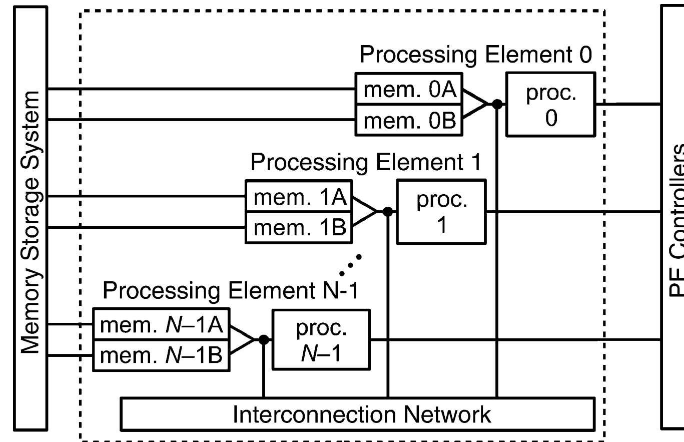


PASM Parallel Processing System. Fig. 5 Physical number of a Processing Element (PE) for a system where $N = 64 = 2^6$ PEs, $Q = 8 = 2^3$ Processing Element Controllers (PECs), and $N/Q = 8 = 2^3$ Memory Storage Units (MSUs). The PE is in a submachine of size $R^* N/Q = 4$ PEC Groups (corresponding to $32 = 2^p = 2^5$ PEs). For this example, the physical numbers of all PEs in this submachine agree in bit position 0

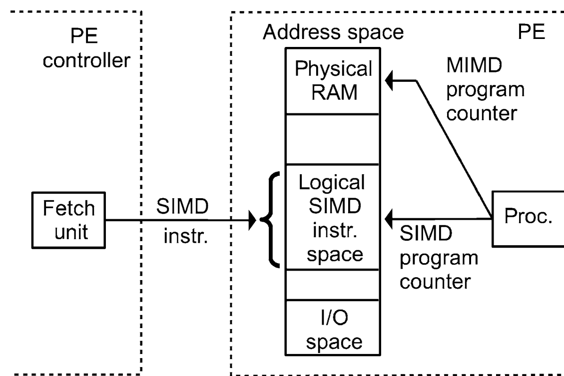
The Parallel Computation Unit

The Parallel Computation Unit, shown in Fig. 6, contains $N = 2^n$ PEs and an interconnection network. Similar to the double-buffering for the PECs, two memory units are used in each PE so that computation and memory I/O can be overlapped. For example, the PE processor can execute a job in one memory unit while the next job is preloaded into the other memory unit from PE secondary storage (the Memory Storage System). These memory units compose the primary memory of the system. In SIMD or MIMD mode, each PE can perform indirect memory addressing using its own PE logical number or local data.

Consider how PASM switches dynamically between the SIMD and MIMD modes of parallelism in our implemented prototype (Fig. 7). In MIMD mode, a PE processor fetches instructions from its local memory by placing its program counter value on the address bus and latching the data placed on the data bus by the memory device holding that instruction word. A PE is forced into SIMD mode by executing a jump to the "logical" *SIMD instruction space,* which is not a physical space. Figure 7 illustrates the address space of a PE, showing the logical SIMD instruction space, physical RAM, and I/O space. Logic in the PE detects read accesses to the logical SIMD instruction space, and any such access sends a SIMD instruction request to the PE's PEC. The SIMD instruction is issued by the submachine's PECs only after all the PEs of a submachine have requested the instruction (recall the System Control Unit has special hardware to synchronize the PECs that are part of the same submachine). The instructions are sent by the submachine PECs' Fetch Units to the PEs (Fig. 7). All enabled PEs in the submachine



PASM Parallel Processing System. Fig. 6 The PASM Parallel Computation Unit



PASM Parallel Processing System. Fig. 7 How the Processing Element (PE) logical address space in the PASM prototype is used to support mixed-mode computation

then execute the instruction simultaneously, each PE on its own data. Thus, all the PEs of the submachine are synchronized when a SIMD instruction is broadcast. The PE continues to access instructions from the SIMD instruction space until an instruction is broadcast that causes each PE to jump to an instruction in its physical RAM (Fig. 7), changing to MIMD mode. Such flexibility in mode switching allows mixed-mode programs to be written that change modes at instruction-level granularity with negligible overhead. The SIMD instruction fetch mechanism also can be used to support a form of MIMD "barrier synchronization" [20].

Masking schemes are used in SIMD mode to enable and disable PEs. The PASM system uses PE-address masks (originated by the PASM project [21]) and data-conditional masks. A *PE-address mask* enables a set of PEs based solely on the logical addresses (numbers) of PEs; it does not depend on local PE data. A PE-address mask has n positions, where each position contains either a 0, 1, or X ("don't care"). A PE is enabled only if the binary representation of its number matches the mask. For example, for $N = 64$, [5{X}{0}] is equivalent to [XXXXX0] and enables all even-numbered PEs. A *negative PE-address mask* enables all PEs whose addresses do not match the mask. For example, for $N = 64$, [4{X}2{0}] = [XXXX00] enables all PEs whose numbers are not multiples of four. PE-address masks are a convenient notation for enabling PEs in a large-scale parallel machine, and proved very useful in image processing applications.

Data-conditional masks enable PEs based on some condition that is dependent on local PE data. The resulting data condition may be true in some PEs and false in other PEs. An example use of data-conditional masking is the `where` statement, the SIMD counterpart to the `if-then-else` statement. When the segment

```
where <data-condition> do <where-part>

elsewhere <else-part>
```

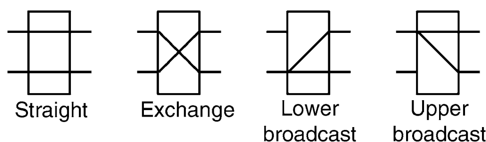
is executed, each PE independently evaluates the `<data-condition>`. The PEs where the condition evaluates true execute the `<where-part>`, and the PEs where the condition evaluates to false are idle. Next, the PEs where the condition evaluates to false

execute the <else-part>, and the PEs where the condition evaluates true are idle. This type of masking is used in many SIMD machines (e.g., CM-2, MP-1). Data-conditional and PE-address masks can be used together.

There are situations in which the combined conditional status of all enabled PEs in a SIMD submachine is needed, e.g., `if-none`, `if-any`, and `if-all`. For example, if the PEs of a submachine are each examining a portion of an image to find certain objects, it may be necessary to know whether any of the PEs have found such an object. Because PASM is a partitionable system, operations such as these require conditional results to be communicated from the PEs to their PECs and subsequently combined among the PECs comprising a SIMD submachine. These operations are efficiently supported by providing a small amount of additional circuitry in the System Control Unit that combines PEC Group results according to the current system partitioning.

The *Interconnection Network* allows PEs to communicate with each other. As described earlier, the partitioning rule in PASM requires that the physical numbers of all PEs in a submachine of 2^p PEs agree in their $n - p$ low-order bit positions (see Fig. 5). Thus, the p high-order bits of a PE's physical number form its logical number within a submachine of 2^p PEs. The low-order partitioning rule was chosen for PASM so that either the multistage cube [18] or the augmented data manipulator (ADM) [18] networks could be used. However, the multistage cube was selected because of its comparative cost-effectiveness.

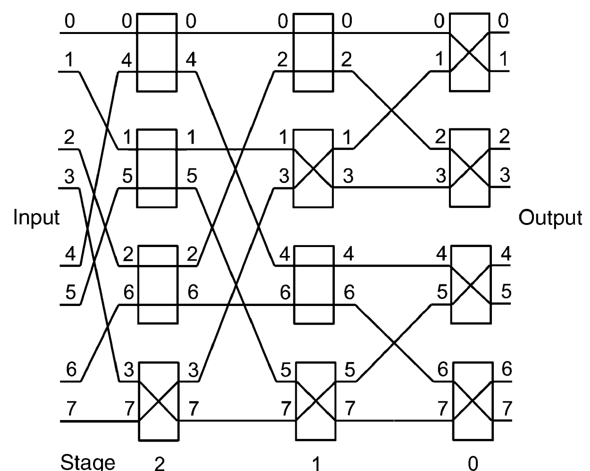
The *multistage cube network* has N input ports, N output ports, and contains $n = \log_2 N$ stages of $N/2$ two-input/two-output *interchange boxes*. Each interchange box can be set to one of the four states shown in Fig. 8. The network can be used in both the SIMD and MIMD modes of parallelism.



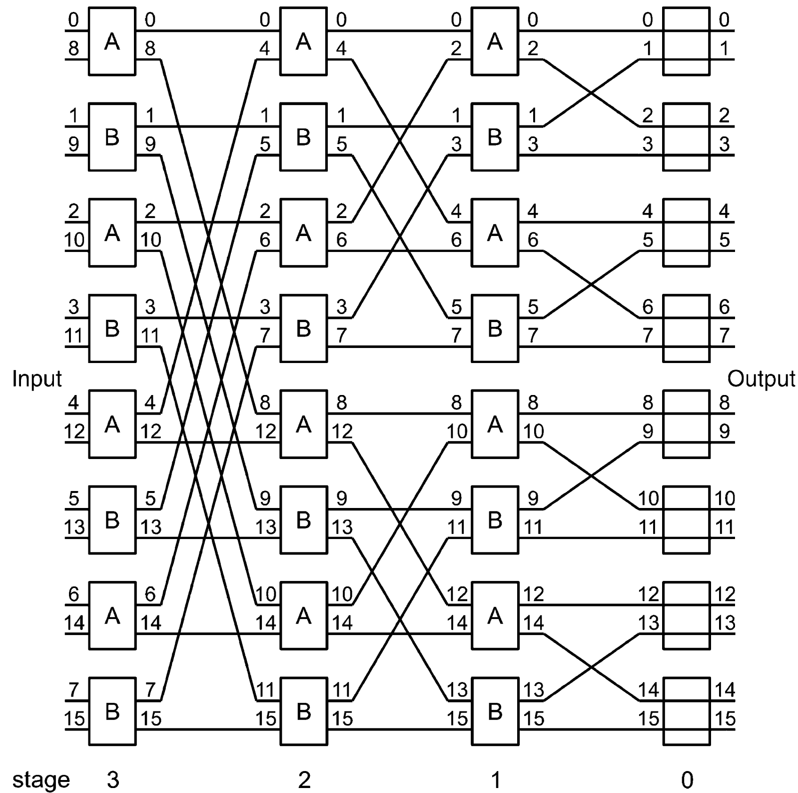
PASM Parallel Processing System. Fig. 8 The four valid states of a multistage cube network interchange box

Figure 9 shows the multistage cube network for $N = 8$. PE i is connected to network input port i and output port i. The stages are numbered consecutively from $n - 1$ at the input stage to 0 at the output stage. The upper interchange box output label is the same as the upper input, and the lower interchange box output label is the same as the lower input. The interconnection pattern between stages is such that at stage j the two links whose labels differ only in bit j are connected to the same interchange box. Figure 9 shows the network set for a "permutation" connection (input $i \rightarrow$ output$(i + 1) \mod 8$) for an $N = 8$ multistage cube network. The network can be controlled in a distributed fashion using routing tags for specifying permutations, one-to-one connections (e.g., PE a to PE b), one PE to many multicast connections, and combinations of these [18].

A network is *partitionable* if it can be divided into independent subnetworks of smaller sizes that have all the properties of the original network [18]. In general, a size N multistage cube network can be partitioned into multiple subnetworks of different sizes, where the size of each subnetwork is a power of two. For example, Fig. 10 shows an $N = 16$ network partitioned into two subnetworks by setting the interchange boxes in stage 0 to straight. Because stage 0 is straight, even-numbered inputs cannot reach odd-numbered outputs, and odd



PASM Parallel Processing System. Fig. 9 A multistage cube network for N = 8 Processing Elements (PEs) set to perform the "permutation" of sending data from PE i to PE $(i + 1) \mod N$ for $0 \leq i < N$



PASM Parallel Processing System. Fig. 10 A multistage cube network for N = 16 Processing Elements (PEs) partitioned into two independent subnetworks, each of size eight. Subnetwork A consists of the even-numbered input and output ports, and subnetwork B the odd

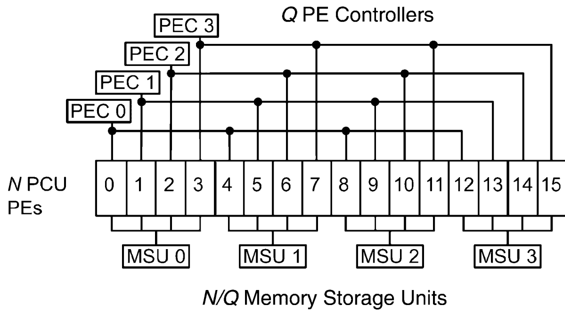
inputs cannot reach even outputs. The two resulting subnetworks are subnetwork *A*, which contains physical ports 0, 2, . . ., 14, and subnetwork *B*, which contains physical ports 1, 3, . . ., 15. Because each of these subnetworks is an independent multistage cube network (of size $N/2 = 8$), either or both subnetworks may be further partitioned by forcing all interchange boxes in stage 1 of the subnetwork to be straight. This process can be applied recursively.

The *Extra Stage Cube network* [18] (designed as a part of the PASM project) is a single-fault tolerant variation of the multistage cube network. There is an extra stage of interchange boxes at the input, labeled stage *n*. Links whose labels differ in the 0th bit position are paired at these extra stage boxes in the same way that they are at stage 0. This Extra Stage Cube network fault-tolerant variation has all of the useful properties described above for the multistage cube, and was constructed in the PASM prototype. It is robust in the sense that it can still provide all of the capabilities of a fully functional multistage network despite the failure of any single network component.

The Memory Storage and Management Systems

The Memory Storage System (Fig. 6) is the secondary storage for the Parallel Computation Unit, storing data files in SIMD mode and both program and data files in MIMD mode (programs for SIMD mode are stored in the Control Storage). The Memory Storage System is comprised of N/Q independent *Memory Storage Units* (*MSUs*), numbered from 0 to $(N/Q) - 1$.

Each of the N/Q MSUs is connected to the memory modules of Q PEs. An example for the prototype size of $N = 16$ and $Q = 4$ is shown in Fig. 11. The benefit of this MSU to PE connection scheme is that the memories of all N/Q PEs connected to any one PEC can be loaded or unloaded in one parallel block transfer. For



PASM Parallel Processing System. Fig. 11 The organization of the PASM Memory Storage System for the prototype size of N = 16 Processing Elements (PEs), Q = 4 Processing Element Controllers (PECs), and N/Q Memory Storage Units (MSUs)

example, in Fig. 11, PEC 2's group of PEs (PEs 2, 6, 10, and 14) are loaded by having MSU 0 load PE 2, MSU 1 load PE 6, MSU 2 load PE 10, and MSU 3 load PE 14, all simultaneously.

In the general case, MSU i is connected to and stores files for the Q PEs whose $n - q$ high-order bits of their PE physical numbers are equal to i (see Fig. 5). This high-order mapping is used so that each of the N/Q PEs connected to a given PEC is connected to a different MSU. Recall that the low-order q bits of the physical number of a PE correspond to the number of the PEC to which it is attached (see Fig. 5). Thus, the full bandwidth of the Memory Storage System can be used when transferring files between the Memory Storage System and the Parallel Computation Unit. If there are only $N/(Q^*D)$ distinct MSUs, where $1 \leq D \leq N/Q$, then this scheme can be scaled so that D parallel block transfers are required to load or unload the memories of the PEs connected to any one PEC. Each MSU contains a mass storage unit and a processor to manage the file system and to transfer files to and from its associated PE memory units.

Similarly, a submachine formed by combining the $(R^*N)/Q$ PEs controlled by R PECs can be loaded or unloaded in R parallel block transfers if there are N/Q MSUs. For example, in Fig. 11, a submachine comprised of the PEs of PEC 0 and PEC 2 can be loaded in $R = 2$ parallel block loads as follows: First PEC 0's PEs are loaded in one parallel block transfer, and then PEC 2's PEs are loaded. If there are only $N/(Q^*D)$ distinct MSUs, only R^*D parallel block transfers are required.

Figure 5 demonstrates which MSU connects to which PE. MSU i is connected to the PE whose high-order $n - q$ bits of its logical number are i (which equal the $n - q$ high-order bits of its physical number). As a result, no matter which PECs are assigned to a task, the data will be in the appropriate MSUs. For example, in Fig. 11, for any submachine of size $N/Q = 4$, MSU 0 is connected to logical PE 0 (which could be physical PE 0, 1, 2, or 3).

The Memory Management System (Fig. 3) is a set of processors that send file system requests from the System Control Unit, PECs, and PEs to the appropriate MSUs; control data transfers between the Memory Storage System and the PEs; supervise input/output operations involving peripheral devices; and enforce consistent file naming and placement across the multiple Memory Storage System disks.

Using the PASM System

ELP (Explicit Language for Parallelism) was a C-based language and associated compiler designed as part of the PASM project for programming mixed-mode parallel machines [13]. ELP provided constructs for both SIMD and MIMD parallelism, and an ELP application program could perform computations that use these parallelism modes in an instruction-level interleaved fashion for mixed-mode operation. ELP provided a vehicle for the exploration of and experimentation with mixed-mode parallelism on the PASM prototype.

CAPS (Coding Aid for the PASM System) was designed to assist in the development and evaluation of application and system software for the PASM prototype [11]. CAPS integrated hardware support and software tools to provide a remote execution and program debugging/monitoring environment for the PASM prototype. The PASM prototype was accessible over the Internet using CAPS, and multiple windows could be displayed to monitor different processors simultaneously.

The programming challenges for partitionable mixed-mode systems are a superset of those for SIMD and MIMD single-mode systems. Application and algorithm research activities to explore PASM's three dimensions of flexibility included theoretical analyses,

simulations, and experiments on the PASM prototype [19]. These studies examined issues such as mapping tasks onto multistage cube network–based parallel processing systems; trade-offs among the SIMD, MIMD, and mixed-mode classes of parallelism; PEC/PE computational overlap in SIMD mode; impact of increasing the number of PEs used; and partitioning for improved performance. Applications considered include edge-guided thresholding, FFTs, global histogramming, image correlation, image smoothing, matrix multiplication, and range-image segmentation (references are given in Armstrong, Watson, and Siegel [2]). Here, we will summarize three application studies that utilized the PASM prototype.

Fineberg, Casavant, and Siegel [7] used the PASM prototype to study the benefits of mixed-mode parallel architectures for bitonic sequence sorting. Four variations of the bitonic sequence sorting algorithm were developed: a SIMD version, a MIMD version, a MIMD version with hardware barrier synchronizations, and a mixed-mode version that allowed switching between SIMD and MIMD modes during execution. The algorithm variations were then coded and profiled on the PASM prototype using varying problem and partition sizes.

The research demonstrated that, by supporting hardware barrier synchronization and mode switching, a machine can achieve better performance than with only SIMD or MIMD parallelism on problems that are computationally similar to bitonic sequence sorting. The authors also proposed a modification to the PASM prototype condition code logic that would increase the performance of the SIMD version.

Saghi, Siegel, and Gray [16] programmed cyclic reduction (a method for solving a general tridiagonal set of irreducible linear algebraic equations) on three parallel systems, including the PASM prototype, to study the trade-offs between SIMD and MIMD parallelism, the effects of increasing the number of processors used on execution time, the impact of the interconnection network on performance, and the advantages of a partitionable system. The cyclic reduction algorithm was implemented using SIMD parallelism on the MasPar MP-1 [5], using MIMD parallelism on the nCUBE 2 [9], and using the same four versions of parallelism on the PASM prototype as mentioned above. The authors also developed a mechanism to predict the algorithm performance on each machine using algorithm analysis and performance measurements from each machine.

By using the PASM prototype as a common basis to compare the modes of parallelism, the authors concluded that a cyclic reduction algorithm using mixed-mode was better than a purely SIMD or purely MIMD algorithm. Execution-time measurements from the other two parallel machines emphasized the need to carefully choose the number of processing elements and distribution of data to obtain efficient computation.

Tan, Siegel, and Siegel [22] used several parallel machines to study block-based motion vector estimation. A SIMD MasPar MP-1 [5], a MIMD Intel Paragon XP/S, a MIMD IBM SP2, and the mixed-mode PASM prototype were used with different data partitioning schemes to perform the estimation, and the results were analyzed to contrast the benefits of each mode of parallelism. In this research, the PASM prototype used SIMD, SPMD, and mixed-mode parallelism.

The research demonstrated a method to analytically predict the performance of various parallel implementations of the algorithm. It also described the impact of the number of processors and mode of parallelism used on algorithm performance. The authors found that, using the PASM prototype, the mixed-mode implementation slightly outperformed the pure SPMD implementation and significantly outperformed the pure SIMD implementation.

Conclusions

Designing, simulating, prototyping, using, and evaluating the PASM partitionable SIMD/MIMD mixed-mode system, based on a multistage cube network, was a great research and educational experience for a large group of faculty and students at Purdue University from the 1970s through the 1990s. As stated earlier, PASM was flexible along three dimensions: partitionability, mode of parallelism, and variable connectivity among PEs. We found that advantages of systems with such flexibility over a pure SIMD machine or a pure MIMD machine included the following [20]:

1. *Multiple simultaneous users*: Because there can be multiple simultaneous independent submachines, there can be multiple simultaneous users of the

system, each executing a different program (not allowed in a pure SIMD machine).
2. *Program development*: Rather than trying to debug a new parallel program on, for example, 1,024 PEs, it can be debugged on a smaller size submachine of 32 PEs, and then extended to 1,024 PEs.
3. *Variable submachine size for increased utilization*: If a task requires only N' of N available PEs, the other $N - N'$ can be used for another task (not allowed in a pure SIMD machine).
4. *Variable submachine size for decreased execution time*: There are some algorithms for which the minimum execution time is obtained when fewer than N PEs are used due to inter-PE communication overhead (e.g., image smoothing [19]); thus, it is desirable to create a submachine consisting of the optimal number of PEs.
5. *Subtask parallelism*: Two independent subtasks that are part of the same job can be executed in parallel, sharing resources if necessary, which may result in improved overall task execution time [12] (not allowed in a pure SIMD machine).
6. *Multiple processing modes*: An algorithm can be executed by using a combination of SIMD and MIMD control with the same set of PEs (mixed-mode parallelism), using the mode that best matches the computations required at each step of the program (Fig. 1).
7. *Matching inter-PE connectivity to the task*: The multistage cube allows different connection patterns among PEs to be established depending on the task (as opposed to, for example, having a fixed mesh network).
8. *System fault tolerance*: If a single PE fails, only those submachines that include the failed PE are affected. This provides some robustness, as mentioned in section "►Introduction," and is not allowed in a pure SIMD machine.
9. *Submachine fault tolerance*: If a PE in a submachine fails, it may be possible to redistribute data and make use of mixed-mode parallelism (i.e., changing from SIMD to MIMD mode) and the variable connectivity (i.e., to establish connection patterns that do not include the faulty PE) so that the job executing on the submachine may continue on that submachine with minimal degradation (this is another form of robustness).

The references cited in this list of advantages, and the complete reading list of PASM-related publications (http://hdl.handle.net/10217/34662), give much more detail about our experiences.

Related Entries

►Connection Machine
►Distributed-Memory Multiprocessor
►Flynn's Taxonomy
►Illiac IV
►MasPar
►MPP
►nCube
►Networks, Multistage

Acknowledgments

The preparation of this entry was supported by the National Science Foundation under grants CNS-0615170 and CNS-0905399, and by the Colorado State University George T. Abell Endowment. The large group of faculty and students who have participated in the PASM project are the coauthors of the papers listed in the PASM-related reading list (http://hdl.handle.net/10217/34662). Numerous agencies supported aspects of PASM-related research: Air Force Office of Scientific Research, Army Research Office, Ballistic Missile Defense Agency, Defense Mapping Agency, Naval Ocean Systems Center, Naval Research Laboratory, National Science Foundation, Office of Naval Research, and Rome Laboratory. IBM provided a grant for much of the prototype equipment. Donations for various parts for the prototype were provided by Amphenol Products, Augat Inc., Belden, Motorola, and Power One.

Bibliographic Notes and Further Reading

This entry is a brief summary of the PASM architecture; details are available in the papers in the PASM-related reading list available at http://hdl.handle.net/10217/34662.

Bibliography

1. Ali S, Maciejewski AA, Siegel HJ, Kim J (2004) Measuring the robustness of a resource allocation. IEEE Trans Parallel Distrib Syst 15(7):630–641
2. Armstrong JB, Watson DW, Siegel HJ (1993) Software issues for the PASM parallel processing system. Software for parallel computation. Springer, Berlin

P

3. Barnes GH, Brown RM, Kato M, Kuck DJ, Slotnick DL, Stokes RA (1968) The ILLIAC IV computer. IEEE Trans Comput C 17(8):746–757
4. Batcher KE (1982) Bit serial parallel processing systems. IEEE Trans Comp C-31(5):377–384
5. Blank T (1990) The MasPar MP-1 architecture. In: Proceedings IEEE compcon spring '90, pp 20–24
6. Bouknight WJ, Denenberg SA, McIntyre DE, Randall JM, Sameh AH, Slotnick DL (1972) The ILLIAC IV system. Proc IEEE 60(4):369–388
7. Fineberg SA, Casavant TL, Siegel HJ (1991) Experimental analysis of a mixed-mode parallel architecture using bitonic sequence sorting. J Parallel Distrib Comput 11(3):239–251
8. Flynn MJ (1966) Very high-speed computing systems. Proc IEEE 54(12):1901–1909
9. Hayes JP, Mudge T (1989) Hypercube supercomputers. Proc IEEE 77(12):1829–1841
10. Lipovski GJ, Malek M (1987) Parallel computing: theory and comparisons. Wiley, New York
11. Lumpp JE, Fineberg SA, Nation WG, Casavant TL, Bronson EC, Siegel HJ, Pero PH, Schwederski T, Marinescu DC (1991) CAPS – a coding aid used with the PASM parallel processing system. Commun ACM 34(11):104–117
12. Nation WG, Maciejewski AA, Siegel HJ (1993) A methodology for exploiting concurrency among independent tasks in partitionable parallel processing systems. J Parallel Distrib Comput 16(3): 271–278
13. Nichols MA, Siegel HJ, Dietz HG (1993) Data management and control-flow aspects of an SIMD/SPMD parallel language/compiler. IEEE Trans Parallel Distrib Syst 4(2):222–234
14. Nutt GJ (1977) Microprocessor implementation of a parallel processor. In: Proceedings 4th annual symposium on computer architecture, pp 147–152
15. Pfister GF, Brantley WC, George DA, Harvey SL, Kleinfelder WJ, Mcauliffe KP, Melton ES, Norton VA, Weiss J (1985) The IBM research parallel processor prototype (RP3): introduction and architecture. In: Proceedings 1985 International conference parallel processing, pp 764–771
16. Saghi G, Siegel HJ, Gray JL (1993) Predicting performance and selecting modes of parallelism: a case study using cyclic reduction on three parallel machines. J Parallel Distrib Comput 19(3): 219–233
17. Shestak V, Smith J, Maciejewski AA, Siegel HJ (2008) Stochastic robustness metric and its use for static resource allocations. J Parallel Distrib Comput 68(8):1157–1173
18. Siegel HJ (1990) Interconnection networks for large-scale parallel processing: theory and case studies, 2nd edn. McGraw-Hill, New York
19. Siegel HJ, Armstrong JB, Watson DW (1992) Mapping computer-vision-related tasks onto reconfigurable parallel-processing systems. IEEE Comput 25(2):54–63
20. Siegel HJ, Schwederski T, Nation WG, Armstrong JB, Wang L, Kuehn JT, Gupta R, Allemang MD, Meyer DG, Watson DW (1996) The design and prototyping of the PASM reconfigurable parallel processing system. Parallel computing: paradigms and applications. International Thomson Computer Press, London
21. Siegel HJ, Siegel LJ, Kemmerer F, Mueller PT Jr, Smalley HE Jr, Smith SD (1981) PASM: a partitionable SIMD/MIMD system for image processing and pattern recognition. IEEE Trans Comput C 30(12):934–947
22. Tan M, Siegel JM, Siegel HJ (1999) Parallel implementations of block-based motion vector estimation for video compression on four parallel processing systems. Int J Parallel Program 27(3): 195–225
23. Tucker LW, Robertson GG (1988) Architecture and applications of the connection machine. IEEE Comput 21(8):26–38

Path Expressions

Roy H. Campbell
University of Illinois at Urbana-Champaign, Urbana, IL, USA

Synonyms

Coordination; Concurrency control; Mutual exclusion; Process synchronization

Definition

A path expression is a declaration of the permitted sequences of operations on an object that is shared by parallel processes.

For example, consider a path expression based on a regular expression notation for a shared buffer. The shared buffer may have operations read and write that may only occur in a sequence specified by the regular expression (write; read)* where a write to the buffer may be followed by a read before it can be repeated. The reads and writes are mutually exclusive. The expression indicates the sequence of actions that may occur independent of the number of processes or the order in which the processes invoke the operations.

Discussion

Path expressions provide a declarative specification of synchronization and coordination of parallel processes performing operations on an object in parallel computing systems [4]. They provide a language mechanism to separate the specification of synchronization from the algorithmic means of providing that synchronization [5]. Although the regular expression is used in our example, other languages that are well defined can be used instead.

In general, the synchronization for an object being manipulated by parallel processes involves mutual exclusion, sequence, and repetition and these may be combined to form path expressions. In more detail, a path expression language based on regular expressions can been described as:

1. A sequence of operations on a resource.
2. A selection from one or more operations on a resource.
3. A repetition of a sequence or selection.
4. A path expression composed of (1), (2), or (3) above.

Other primitives besides these three have been considered including:

1. A burst of parallel sequences of executions: If there is one execution of the operations in a path expression, then there can be parallel executions of the sequences of operations in a path expression up until the event when there are no more parallel executions of those operations occurring. For example, the expression (write, {read})* for a buffer would specify that a write or a burst of parallel reads can occur, but not a read and a write in parallel [4].
2. A concurrency restriction: This allows parallel executions of the path expression up to the limit of the restriction. For example, the expression 5:(write; read) would allow up to 5 parallel instances of a write followed by a read [13].
3. Guarded selection: A path expression selection that describes the synchronization of the parallel execution of its operations when a guard is true. For example, the expression #buffers < 5| (put), #buffers > 0 (get) would describe synchronization where if the variable #buffers is less than 5, a put may occur or if the variable #buffers is greater than 0, a get may occur (see also [2]).

More general forms of path expression may also be described [4]. For example, a general path expression can be composed of independent path expressions such that if all the path expressions allow the execution of an operation, it may occur. The general path expression:

path (A; B)* end
path (A; C)* end

requires an operation A to be followed by operation B and C. Operations B and C may possibly occur in parallel since there are no constraints between B and C but there are constraints between A and B and between A and C.

Implementation

An operation on an object can be considered as a transition that changes the state of the object. When a process invokes an operation on an object, it can be viewed as a request for a transition from one state of the object to another. The implementation of a path expression decides whether to accept the request and proceed or whether to delay the request until an appropriate state and block the process invocation of that operation. When a path expression is built upon a regular expression that has the property that an operation can only occur when the object is in one particular state, then a simple implementation can be built using semaphores to represent the states.

In this exposition, the implementation for open path expressions is described. An open path expression allows much of the power of a semaphore to be used in a declarative expression and forms the basis for the implementation of "Path Pascal," a version of Pascal that includes concurrency and synchronization [13]. Open path expressions are composed of four constraints: sequence, selection, restriction, and derestriction.

The open path expression notation allows a simple rewriting scheme to translate a path expression synchronization declaration into the prologues and the epilogues of semaphore operations of each of the operations mentioned in the path expression:

[**path**<op_expr>**end**] ⊢
  |α <op_expr>β

Where ⊢ designates a rewrite rule and α^i, β^i are empty strings of semaphore operations forming the prologue and epilogue to be inserted in each of the operations named in op_expr.

The following set of path expression rewrite rules evaluated in the order specified either by parentheses or by following precedence from left to right generate prologues and epilogues that implement the four synchronization constraints: sequence, selection, restriction, and burst:

P

Rule Selection (comma): α <op_expr$_1$>, <op_expr$_2$>$\beta \vdash$

|α <op_expr$_1$> β
|α <op_expr$_2$> β

Rule Sequence (semicolon): α <op_expr$_1$>; <op_expr$_2$> $\beta \vdash$

|α <op_expr$_1$> "S_j.V();"
| "S_j.P();" <op_expr$_2$>β

where Semaphore S_j is initialized to 0

Rule Restriction (colon): α n: (<op_expr$_1$>)$\beta \vdash$

|α || ";S_k.P();" <op_expr$_1$> ";S_k.V();" || β

where Semaphore S_k is initialized to n and || expresses the concatenation of two strings making up a prologue or epilogue.

Rule Derestriction (brace): α{<op_expr$_1$ >} $\beta \vdash$

| "S_k.P(); if count++ == 1 then [" α || "S_k.V();];" <op_expr$_1$> " S_k.P(); if count−− == 1 then [" β || "S_k.V();];"

where Semaphore S_k is initialized to 1, integer count is initialized to 0, the ++ and −− operators denote incrementing or decrementing a counter count by 1, and square parentheses within the strings denote a block of instructions.

The derestriction rule applies the synchronization prologue α if count is incremented through 1 in the prologue and applies the synchronization epilogue β if count is decremented through 1 in the epilogue.

Finally, the prologues and epilogues are complete when <op_expr> contains only a method_name of an operation declared in an object.

Operation: α <method_name > β

| *insert prologue* α; < method_body>; *insert epilogue* β

The following are some examples of the implementations of open path expressions:

1. **path** x, y **end**
   creates empty prologues and epilogues in the methods x and y allowing them to be executed without synchronization constraints.
2. **path** 1: (x) **end**
   rewrites to:
   | S_1.P(); <x_body>; S_1.V();
   Semaphore S_1 = 1;
   The body of x is executed in mutual exclusion.
3. **path** 1: (x, y) **end**
   rewrites to:
   | S_1.P(); <x_body>; S_1.V();
   | S_1.P(); <y_body>; S_1.V();
   Semaphore S_1 = 1;
   and creates prologues and epilogues in the methods x and y constraining their execution to mutual exclusion.
4. **path** 1: (x; y) **end**
   rewrites to:
   | S_1.P(); <x_body>; S_2.V();
   | S_2.P(); <y_body>; S_1.V();
   Semaphore S_1, S_2 = (1, 0);
   and creates prologues and epilogues in the methods x and y constraining their execution to mutual exclusion and a sequence that alternates the execution of x with one of y.
5. **path** 1: (write; {read}) **end**
   rewrites to:
   | S_1.P(); <write_body>; S_2.V();
   | S_3.P(); if count++ == 1 then [S_2.P(); S_3.V();] <read_body> S_3.P(); if count−− == 1 then [S_2.V(); S_3.V();];
   Semaphore S_1, S_2, S_3 = (1, 0, 1); Integer count = 0;
   and allows one write operation to be followed by a burst of read operations.
6. **path** 5: (1: (write); 1: (read)) **end**
   rewrites to:
   | [S_1.P(); S_2.P(); <write_body>; S_2.V(); S_4.V();]
   | [S_4.P(); S_3.P(); <read_body>; S_3.V(); S_1.V();]
   Semaphore S_1, S2, S3, S4 = (5, 1, 1, 0);

The body of write is executed in mutual exclusion as is the body of read. However, there must always be more writes than reads and there may be up to 5 more writes than reads.

Uses of Path Expressions

Path expressions as a declarative form of specifying synchronization have found use in several parallel programming languages [2, 14, 18]. They have also been employed in single-assignment languages [7], real-time control languages [17], distributed objects [16], event systems [12], multimedia systems [11], debugging [3], VLSI design [1], synchronization contracts for web services [15], and workflow languages [10]. Typically,

processes and path expressions are dual approaches to specifying the traces created in a computation; the processes generate sequences of actions which are constrained by path expressions to achieve particular synchronization constraints associated with the operations on abstract data types.

Summary

In practice, synchronization of parallel processes remains a difficult problem that becomes ever more complex with increased concurrency in architecture. Path expressions provide a separate specification of synchronization from the code of processes that, under some circumstances, can simplify parallel programming. Although not widely adopted, their declarative approach to specifying synchronization has been found useful in various parallel applications. Path expressions have been used to synchronize parallel operations on objects in parallel languages, real-time languages, event systems, VSLI, workflow, and debugging.

Bibliographic Notes and Further Reading

The semantics of parallel programs may be described using trace-based semantics. Path expression implementations restrict or recognize the permitted traces of parallel programs as described by Lauer and Campbell [6]. A semantics for path expressions is given by Dinning and Mishra using partially ordered multisets [8] and the authors provide a fully parallel implementation for a path expression language on MIMD shared memory architectures. Path Pascal and open path expressions were used as pedagogical tools for teaching parallel programs [9, 13].

Bibliography

1. Anantharaman TS, Clarke EM, Mishra B (1986) Compiling path expressions into VLSI circuits. Distrib Comput 1:150–166
2. Andler S (1979) Predicate path expressions. In: POPL'79 proceedings of the 6th ACM SIGACT-SIGPLAN symposium on principles of programming languages. ACM Press, New York, pp 226–236
3. Bruegge B, Hibbard P (1983) Generalized path expressions: a high-level debugging mechanism. J Syst Softw 3:265–276 (Elsevier Science Publishing)
4. Campbell RH (1976) Path expressions: a technique for specifying process synchronization. PhD. Thesis, The University of Newcastle Upon Tyne
5. Campbell RH, Habermann AN (1974) The specification of process synchronization by path expressions. In: Gelenbe E, Kaiser C (eds) Operating systems. Lecture notes in computer science, vol 16. Springer, Berlin, pp 89–102
6. Lauer PE, Campbell RH (1979) Formal semantics of a class of high-level primitives for coordinating concurrent processes, Acta Informatica, 5(4):297–332
7. Comte D, Durrieu G, Gelly O, Plas A, Syre JC (1978) Parallelism, control and synchronization expression in a single assignment language. ACM SIGPLAN Not 13(1): 25–33
8. Dinning A, Mishra B (1990) A fully parallel algorithm for implementing path expressions. J Parallel Distrib Comput 10:205–221
9. Dowsing RD, Elliott R (1986) Programming a bounded buffer using the object and path expression constructs of path pascal. Comput J 29(5):423–429
10. Heinlein C (2000) Workflow and process synchronization with interaction expressions and graphs. Ph. D. Thesis (in German), Fakultät für Informatik, Universität Ulm
11. Hoepner P (1992) Synchronizing the presentation of multimedia objects. Comput Commun 15(9):557–564
12. Kidd M-EC (1994) Ensuring critical event sequences in high integrity software by applying path expressions. Sandia Labs, Albuquerque
13. Kolstad RB, Campbell RH (1980) Path Pascal user manual. SIGPLAN Not 15(9):15–24
14. Laure E (1999) ParBlocks – a new methodology for specifying concurrent method executions in opus. In: Amestoy P, Berger P, Dayde M, Ruiz D, Duff I, Fraysse V, Giraud L (eds) Euro-Par'99. Lecture notes in computer science, vol 1685. Springer, Berlin, pp 925–929
15. Preiss O, Shah AP, Wegmann A (2003) Generating synchronization contracts for web services. In: Khosrow-Pour M (ed) Information technology and organizations: trends, issues, challenges & solutions, vol 1. Idea Group Publishing, Hershey, pp 593–596
16. Rees O (1993) Using path expressions as concurrency guards. Technical report, ANSA
17. Schoute AL, Luursema JJ (1990) Realtime system control by means of path expressions. In: Proceedings Euromicro '90 Workshop on Real Time, Horsholm, Denmark, pp 79–86
18. Shaw AC (1978) Software description with flow expressions. IEEE Trans Softw Eng SE-4(3):242–254

PaToH (Partitioning Tool for Hypergraphs)

Ümit Çatalyürek[1], Cevdet Aykanat[2]
[1]The Ohio State University, Columbus, OH, USA
[2]Bilkent University, Ankara, Turkey

Synonyms

Partitioning tool for hypergraphs (PaToH)

P

Definition

PaToH is a sequential, multilevel, hypergraph partitioning tool that can be used to solve various combinatorial scientific computing problems that could be modeled as hypergraph partitioning problem, including sparse matrix partitioning, ordering, and load balancing for parallel processing.

Discussion

Introduction

Hypergraph partitioning has been an important problem widely encountered in VLSI layout design [22]. Recent works since the late 1990s have introduced new application areas, including one-dimensional and two-dimensional partitioning of sparse matrices for parallel sparse-matrix vector multiplication [6–8, 12], sparse matrix reordering [6, 11], permuting sparse rectangular matrices into singly bordered block-diagonal form for parallel solution of LP problems [3], and static and dynamic load balancing for parallel processing [5]. PaToH [9] has been developed to provide fast and high-quality solutions for these motivating applications.

In simple terms, the hypergraph partitioning problem can be defined as the task of dividing a hypergraph into two or more roughly equal sized parts such that a cost function on the hyperedges connecting vertices in different parts is minimized. The hypergraph partitioning problem is known to be NP-hard [22], therefore a wide variety of heuristic algorithms have been developed in the literature to solve this complex problem [1, 15, 21, 23, 25]. Following the success of multilevel partitioning schemes in ordering and graph partitioning [4, 16, 18], PaToH [9] has been developed as one of the first multilevel hypergraph partitioning tools.

Preliminaries

A hypergraph $\mathcal{H}=(\mathcal{V},\mathcal{N})$ is defined as a set of vertices (also called cells) $\mathcal{V}$ and a set of nets (hyperedges) $\mathcal{N}$ among those vertices. Every net $n \in \mathcal{N}$ is a subset of vertices, that is, $n \subseteq \mathcal{V}$. The vertices in a net n are called its *pins* in PaToH. The *size* of a net, $s[n]$, is equal to the number of its pins. The *degree* of a vertex is equal to the number of nets it is connected to. Graph is a special instance of hypergraph such that each net has exactly two pins. Vertices and nets of a hypergraph can be associated with weights. For simplicity in the presentation, net weights are refered as *cost* here and denoted with $c[.]$, whereas $w[.]$ will be used for vertex weights.

$\Pi = \{\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_K\}$ is a *K-way partition* of $\mathcal{H}$ if the following conditions hold:

- Each part $\mathcal{V}_k$ is a nonempty subset of $\mathcal{V}$, that is, $\mathcal{V}_k \subseteq \mathcal{V}$ and $\mathcal{V}_k \neq \emptyset$ for $1 \leq k \leq K$.
- Parts are pairwise disjoint, that is, $\mathcal{V}_k \cap \mathcal{V}_\ell = \emptyset$ for all $1 \leq k < \ell \leq K$.
- Union of K parts is equal to $\mathcal{V}$, that is, $\bigcup_{k=1}^{K} \mathcal{V}_k = \mathcal{V}$.

In a partition Π of $\mathcal{H}$, a net that has at least one pin (vertex) in a part is said to *connect* that part. *Connectivity* λ_n of a net n denotes the number of parts connected by n. A net n is said to be *cut* (*external*) if it connects more than one part (i.e., $\lambda_n > 1$), and *uncut* (*internal*) otherwise (i.e., $\lambda_n = 1$). In a partition Π of $\mathcal{H}$, a vertex is said to be a *boundary* vertex if it is incident to a cut net. A K-way partition is also called a *multiway* partition if $K > 2$ and a *bipartition* if $K = 2$. A partition is said to be balanced if each part $\mathcal{V}_k$ satisfies the *balance criterion*:

$$W_k \leq W_{avg}(1+\varepsilon), \quad \textit{for } k = 1, 2, \ldots, K. \tag{1}$$

In (1), weight W_k of a part $\mathcal{V}_k$ is defined as the sum of the weights of the vertices in that part (i.e., $W_k = \sum_{v \in \mathcal{V}_k} w[v]$), W_{avg} denotes the weight of each part under the perfect load balance condition (i.e., $W_{avg} = (\sum_{v \in \mathcal{V}} w[v])/K$), and ε represents the predetermined maximum imbalance ratio allowed.

The set of external nets of a partition Π is denoted as $\mathcal{N}_E$. There are various [13, 27] *cutsize* definitions for representing the cost $\chi(\Pi)$ of a partition Π. Two relevant definitions are:

$$\begin{aligned} &(a) \quad \chi(\Pi) = \sum_{n \in \mathcal{N}_E} c[n] \quad \textit{and} \\ &(b) \quad \chi(\Pi) = \sum_{n \in \mathcal{N}_E} c[n](\lambda_n - 1). \end{aligned} \tag{2}$$

In (2a), the cutsize is equal to the sum of the costs of the cut nets. In (2b), each cut net n contributes $c[n](\lambda_n - 1)$ to the cutsize. The cutsize metrics given in (2a) and (2b) will be referred to here as *cut-net* and *connectivity* metrics, respectively. The hypergraph partitioning problem can be defined as the task of dividing a hypergraph into two or more parts such that the cutsize is minimized, while a given balance criterion (1) among part weights is maintained.

A recent variant of the above problem is the *multi-constraint hypergraph* partitioning [2, 6, 10, 19, 24] in which each vertex has a vector of weights associated

with it. The partitioning objective is the same as above, and the partitioning constraint is to satisfy a balancing constraint associated with each weight. Let $w[v, i]$ denote the C weights of a vertex v for $i = 1, \ldots, C$. Then balance criterion (1) can be rewritten as:

$$W_{k,i} \leq W_{avg,i}\,(1 + \varepsilon) \text{ for } k = 1, \ldots, K \text{ and } i = 1, \ldots, C\,, \quad (3)$$

where the ith weight $W_{k,i}$ of a part $\mathcal{V}_k$ is defined as the sum of the ith weights of the vertices in that part (i.e., $W_{k,i} = \sum_{v \in \mathcal{V}_k} w[v, i]$), and $W_{avg,i}$ is the average part weight for the ith weight (i.e., $W_{avg,i} = (\sum_{v \in \mathcal{V}} w[v, i])/K$), and ε again represents allowed imbalance ratio.

Another variant is the *hypergraph partitioning with fixed vertices*, in which some of the vertices are fixed in some parts before partitioning. In other words, in this problem, a *fixed-part* function is provided as an input to the problem. A vertex is said to be *free* if it is allowed to be in any part in the final partition, and it is said to be fixed in part k if it is required to be in $\mathcal{V}_k$ in the final partition Π.

Using PaToH

PaToH provides a set of functions to read, write, and partition a given hypergraph, and evaluate the quality of a given partition. In terms of partitioning, PaToH provides a user customizable hypergraph partitioning via multilevel partitioning scheme. In addition, PaToH provides hypergraph partitioning with fixed cells and multi-constraint hypergraph partitioning.

Application developers who would like to use PaToH can either directly use PaToH through a simple, easy-to-use C library interface in their applications, or they can use stand-alone executable.

PaToH Library Interface

PaToH library interface consists of two files: a header file `patoh.h` which contains constants, structure definitions, and functions proto-types, and a library file `libpatoh.a`.

Before starting to discuss the details, it is instructive to have a look at a simple C program that partitions an input hypergraph using PaToH functions. The program is displayed in Fig. 1. The first statement is a function call to read the input hypergraph file which is given by the first command line argument. PaToH partition functions are customizable through a set of parameters. Although the application user can set each of these parameters one by one, it is a good habit to call PaToH function `PaToH_Initialize_Parameters` to set all parameters to one of the three preset default values by specifying `PATOH_SUGPARAM_<preset>`, where `<preset>` is `DEFAULT`, `SPEED`, or `QUALITY`. After this call, the user may prefer to modify the parameters according to his/her need before calling `PaToH_Alloc`. All memory that will be used by PaToH partitioning functions is allocated by `PaToH_Alloc` function, that is, there will be no more dynamic memory allocation inside the partitioning functions. Now, everything is set to partition the hypergraph using PaToH's multilevel hypergraph partitioning functions. A call to `PaToH_Partition` (or `PaToH_MultiConst_Partition`) will partition the hypergraph, and the resulting partition vector, part weights, and cutsize will be returned in the parameters. Here, variable `cut` will hold the cutsize of the computed partition according to cutsize definition (2b) since, this metric is specified by initializing the parameters with constant `PATOH_CONPART`. The user may call partitioning functions as many times as he/she wants before calling function `PaToH_Free`. There is no need to reallocate the memory before each partitioning call, unless either the hypergraph or the desired customization (like changing coarsening algorithm, or number of parts) is changed.

A hypergraph and its representation can be seen in Fig. 2. In the figure, large circles are cells (vertices) of the hypergraph, and small circles are nets. `xpins` and `pins` arrays store the beginning index of pins (cells) connected to each net, and IDs of the pins, respectively. Hence, `xpins` is an array of size equal to the number of nets plus one (11 in this example), and `pins` is an array of size equal to the number of pins in the hypergraph (31 in this example). Cells connected to net n_j are stored in `pins[xpins[j]]` through `pins[xpins[j+1]-1]`.

Stand-Alone Program

Distribution includes a stand-alone program, called `patoh`, for single constraint partitioning (this executable will not work with multiple vertex weights; for multi-constraint partitioning there is an interface and some sample source codes). The program `patoh` gets

its parameters from command line arguments. PaToH can be run from command line as follows:

```
> patoh <hypergraph-file>
  <number-of-parts> [[parameter1]
  [parameter2] ....].
```

Partitioning can be customized by using the optional `[parameter]` arguments. The syntax of these optional parameters is as follows: two-letter abbreviation of a parameter is followed by an equal sign and a value. For example, if the user wishes to change refinement algorithm (abbreviated as "RA") to "Kernighan–Lin with dynamic locking" (sixth algorithm out of 12 implemented in PaToH), the user should specify "RA=6." For a complete example, consider the sample hypergraph displayed in Fig. 2. In order to partition this hypegraph into three parts by using the Kernighan–Lin refinement algorithm with cut-net metric (the default is connectivity metric (Equation (2b)), one has to issue the following command whose output is shown next:

```
> patoh sample.u 3 RA=6 UM=U

+++++++++++++++++++++++++++++++++++++++++++++++++++++++
+++ PaToH v3 (c) Nov 1999-, by  Umit V. Catalyurek
+++ Build # 872 Date: Fri, 09 Oct 2009
+++++++++++++++++++++++++++++++++++++++++++++++++++++++

********************************************************************************
Hypergraph : sample.u   #Cells :     12   #Nets :     11   #Pins :     31
********************************************************************************
 3-way partitioning results of PaToH:

 Cut Cost:     2
 Part Weights    : Min=          4 (0.000) Max=          4 (0.000)
--------------------------------------------------------------------------------
I/O          :       0.000 sec
I.Perm/Cons.H:       0.000 sec  ( 2.9%)
Coarsening   :       0.000 sec  ( 1.1%)
Partitioning :       0.000 sec  (75.8%)
Uncoarsening :       0.000 sec  ( 3.7%)
Total        :       0.001 sec
Total (w I/O):       0.001 sec
--------------------------------------------------------------------------------
```

This output shows that the cutsize (cut cost) according to cut-net metric is 2. Final imbalance ratios (in parentheses) for the least loaded and the most loaded parts are 0% (perfect balance with four vertices in each part), and partitioning only took about 1 ms. The input hypergraph and resulting partition is displayed in Fig. 3. A quick summary of the input file format (the details are provided in the PaToH manual [9]) is as follows: the first non-comment line of the file is a header containing the index base (0 or 1) and the size of the hypergraph, and information for each net (only pins in this case) and cells (none in this example) follows.

All of the PaToH customization parameters that are available through library interface are also available as command line options. PaToH manual [9] contains details of each of those customization parameters.

Customizing PaToH's Hypergraph Partitioning

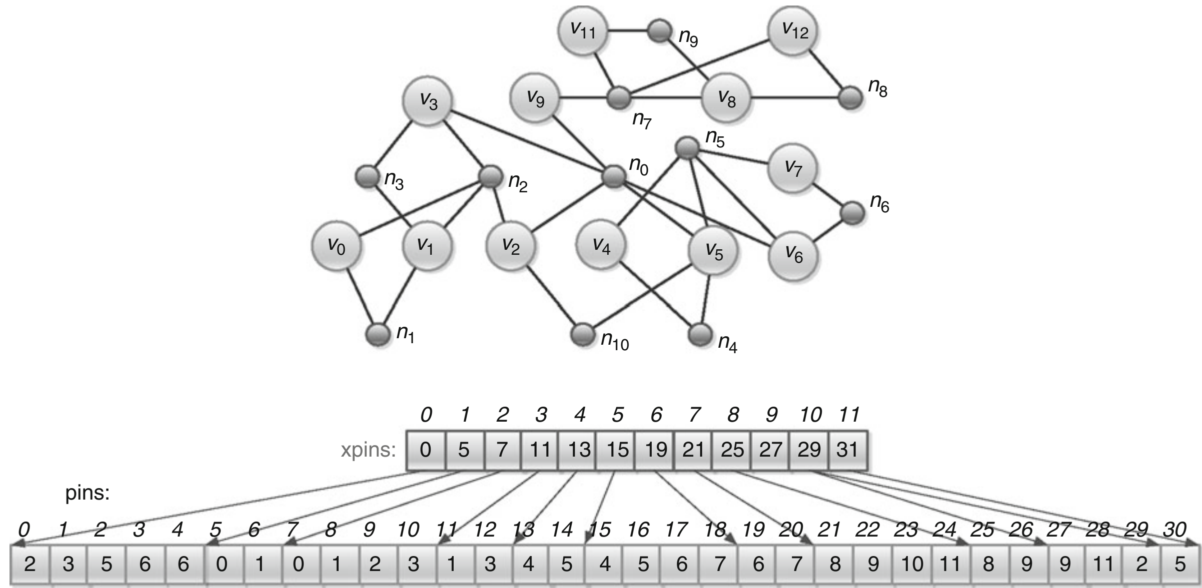
PaToH achieves K-way hypergraph partitioning through recursive bisection (two-way partition), and at each bisection step it uses a multilevel hypergraph bisection

```
#include <stdio.h>
#include "patoh.h"

int main(int argc,char *argv[])
{
PaToH_Parameters args;
int             _c, _n, _nconst, *cwghts, *nwghts,
                *xpins, *pins, *partvec, cut, *partweights;


PaToH_Read_Hypergraph(argv[1], &_c, &n, &_nconst, &cwghts, &nwghts,
                      &xpins, &pins);

printf("Hypergraph %10s -- #Cells=%6d #Nets=%6d #Pins=%8d #Const=%2d\n",
       argv[1], _c, _n, xpins[_n], _nconst);

PaToH_Initialize_Parameters(&args, PATOH_CONPART, PATOH_SUGPARAM_DEFAULT);

args._k = atoi(argv[2]);
partvec = (int *) malloc(_c*sizeof(int));
partweights = (int *) malloc(args._k*sizeof(int));

PaToH_Alloc(&args, _c, _n, _nconst, cwghts, nwghts, xpins, pins);

if (_nconst==1)
    PaToH_Partition(&args, _c, _n, cwghts, nwghts,
                    xpins, pins, partvec, partweights, &cut);
else
    PaToH_MultiConst_Partition(&args, _c, _n, _nconst, cwghts,
                               xpins, pins, partvec, partweights, &cut);

printf("%d-way cutsize is: %d\n", args._k, cut);

free(cwghts);      free(nwghts);
free(xpins);       free(pins);
free(partweights); free(partvec);

PaToH_Free();
return 0;
}
```

PaToH (Partitioning Tool for Hypergraphs). Fig. 1 A simple C program that partitions an input hypergraph using PaToH functions

algorithm. In the recursive bisection, first a bisection of $\mathcal{H}$ is obtained, and then each part of this bipartition is further partitioned recursively. After $\lg_2 K$ steps, hypergraph $\mathcal{H}$ is partitioned into K parts. Please note that, K is not restricted to be a power of 2. For any $K > 1$, one can achieve K-way hypergraph partitioning through recursive bisection by first partitioning $\mathcal{H}$ into two parts with a load ratio of $\lfloor K/2 \rfloor$ to $(K - \lfloor K/2 \rfloor)$, and then recursively partitioning those parts into $\lfloor K/2 \rfloor$ and $(K - \lfloor K/2 \rfloor)$ parts, respectively, using the same approach.

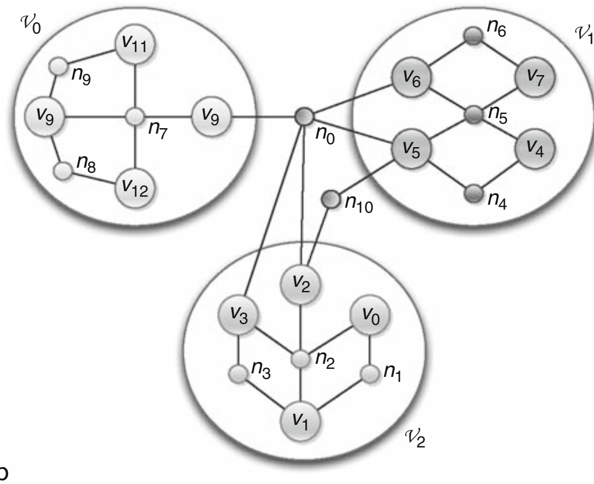
A pseudo-code of the multilevel hypergraph bisection algorithm used in PaToH is displayed in Algorithm 1. Mainly, the algorithm has three phases: *coarsening*, *initial partitioning*, and *uncoarsening*. In the first phase, a bottom-up multilevel clustering is successively applied starting from the original hypergraph until either the number of vertices in the coarsened hypergraph reduces below a predetermined threshold value or clustering fails to reduce the size of the hypergraph significantly. In the second phase, the coarsest



PaToH (Partitioning Tool for Hypergraphs). Fig. 2 A sample hypergraph and its representation

```
% base:(0/1) #cells #nets #pins
0 12 11 31
% pins of each net in the hypergraph
2 3 5 6 9
0 1
0 1 2 3
1 3
4 5
4 5 6 7
6 7
8 9 10 11
8 10
8 11
2 5
```

a



b

PaToH (Partitioning Tool for Hypergraphs). Fig. 3 Text file representation of the sample hypergraph in Fig. 2 and illustration of a partition found by PaToH

hypergraph is bipartitioned using one of the 12 initial partitioning techniques. In the third phase, the partition found in the second phase is successively projected back towards the original hypergraph while it is being improved by one of the iterative refinement heuristics. These three phases are summarized below.

1. Coarsening Phase: In this phase, the given hypergraph $\mathcal{H} = \mathcal{H}_0 = (\mathcal{V}_0, \mathcal{N}_0)$ is coarsened into a sequence of smaller hypergraphs $\mathcal{H}_1 = (\mathcal{V}_1, \mathcal{N}_1)$, $\mathcal{H}_2 = (\mathcal{V}_2, \mathcal{N}_2)$, ..., $\mathcal{H}_\ell = (\mathcal{V}_\ell, \mathcal{N}_\ell)$ satisfying $|\mathcal{V}_0| > |\mathcal{V}_1| > |\mathcal{V}_2| > \ldots > |\mathcal{V}_\ell|$. This coarsening is achieved by coalescing disjoint subsets of vertices of hypergraph $\mathcal{H}_i$ into *clusters* such that each cluster in $\mathcal{H}_i$ forms a single vertex of $\mathcal{H}_{i+1}$. The weight of each vertex of $\mathcal{H}_{i+1}$ becomes equal to the sum of its constituent vertices of the respective cluster in $\mathcal{H}_i$. The net set of each vertex of $\mathcal{H}_{i+1}$ becomes equal to the union of the net sets of the constituent vertices of the respective cluster in $\mathcal{H}_i$. Here, multiple pins of a net $n \in \mathcal{N}_i$ in a cluster of $\mathcal{H}_i$ are contracted to a single pin of the respective net $n' \in \mathcal{N}_{i+1}$ of $\mathcal{H}_{i+1}$. Furthermore, the single-pin

Algorithm 1 Multilevel Bisection.

```
function PaToHMLevelPartition(H = (V, N))
    H_0 ← H
    ℓ ← 0
    /* Coarsening Phase: */
    while |V_ℓ| > CoarseTo do
        find a clustering, C_ℓ, using one of the coarsening
          algorithms
        construct H_{ℓ+1} using C_ℓ
        if (|V_{ℓ+1}| − |V_ℓ|)/|V_ℓ| < CoarsePercent then
            break
        else
            ℓ ← ℓ + 1
        end if
    end while
    /* Initial Partitioning Phase: */
    find an initial partitioning Π_ℓ of H_ℓ
    /* Uncoarsening Phase: */
    while ℓ > 0 do
        refine Π_ℓ using one of the refinement algorithms
        if ℓ > 0 then
            project Π_ℓ to Π_{ℓ−1}
        end if
        ℓ ← ℓ − 1
    end while
    return Π_0
end function
```

nets obtained during this contraction are discarded. The coarsening phase terminates when the number of vertices in the coarsened hypergraph reduces below the predetermined number or clustering fails to reduce the size of the hypergraph significantly.

In PaToH, two types of clusterings are implemented, *matching-based*, where each cluster contains at most of two vertices; and *agglomerative-based*, where clusters can have more than two vertices. The former is simply called *matching* in PaToH, and the latter is called *clustering*.

The matching-based clustering works as follows. Vertices of $\mathcal{H}_i$ are visited in a user-specified order (could be random, degree sorted, etc.). If a vertex $u \in \mathcal{V}_i$ has not been matched yet, one of its unmatched *adjacent* vertices is selected according to a criterion. If such a vertex v exists, the matched pair u and v are merged into a cluster. If there is no unmatched adjacent vertex of u, then vertex u remains unmatched, that is, u remains as a singleton cluster. Here, two vertices u and v are said to be adjacent if they share at least one net, that is, $nets[u] \cap nets[v] \neq \emptyset$.

In the agglomerative clustering schemes, each vertex u is assumed to constitute a singleton cluster $C_u = \{u\}$ at the beginning of each coarsening level. Then, vertices are again visited in a user specified order. If a vertex u has already been clustered (i.e., $|C_u|>1$) it is not considered for being the source of a new clustering. However, an unclustered vertex u can choose to join a multi-vertex cluster as well as a singleton cluster. That is, all adjacent vertices of an unclustered vertex u are considered for selection according to a criterion. The selection of a vertex v adjacent to u corresponds to including vertex u to cluster C_v to grow a new multi-vertex cluster $C_u = C_v = C_v \cup \{u\}$.

PaToH includes a total of 17 coarsening algorithms: eight matchings and nine clustering algorithms, and the default method is a clustering algorithm that uses *absorption* metric. In this method, when selecting the adjacent vertex v to cluster with vertex u, vertex v is selected to maximize $\sum_{n \in nets[u] \cap nets[C_v]} \frac{|C_v \cap n|}{s[n]-1}$, where $nets[C_v] = \cup_{w \in C_v} nets[w]$.

2. Initial Partitioning Phase: The goal in this phase is to find a bipartition on the coarsest hypergraph $\mathcal{H}_\ell$. PaToH includes various random partitioning methods as well as variations of *Greedy Hypergraph Growing (GHG)* algorithm for bisecting $\mathcal{H}_\ell$. In GHG, a cluster is grown around a randomly selected vertex. During the coarse of the algorithm, the selected and unselected vertices induce a bipartition on $\mathcal{H}_\ell$. The unselected vertices connected to the growing cluster are inserted into a priority queue according to their *move-gain* [15], where the gain of an unselected vertex corresponds to the decrease in the cutsize of the current bipartition if the vertex moves to the growing cluster. Then, a vertex with the highest gain is selected from the priority queue. After a vertex moves to the growing cluster, the gains of its unselected adjacent vertices that are currently in the priority queue are updated and those not in the priority queue are inserted. This cluster growing operation continues until a predetermined bipartition balance criterion is reached. The quality of this algorithm is sensitive to the choice of the initial random vertex. Since the coarsest hypergraph $\mathcal{H}_\ell$ is small, initial partitioning heuristics can be run multiple times and select the best bipartition for refinement during the uncoarsening

phase. By default, PaToH runs 11 different initial partitioning algorithms and selects the bipartition with lowest cost.

3. Uncoarsening Phase: At each level i (for $i = \ell, \ell-1, \ldots, 1$), bipartition Π_i found on $\mathcal{H}_i$ is projected back to a bipartition Π_{i-1} on $\mathcal{H}_{i-1}$. The constituent vertices of each cluster in $\mathcal{H}_{i-1}$ is assigned to the part of the respective vertex in $\mathcal{H}_i$. Obviously, Π_{i-1} of $\mathcal{H}_{i-1}$ has the same cutsize with Π_i of $\mathcal{H}_i$. Then, this bipartition is refined by running a KL/FM-based iterative improvement heuristics on $\mathcal{H}_{i-1}$ starting from initial bipartition Π_{i-1}. PaToH provides 12 refinement algorithms that are based on the well-known Kernighan–Lin (KL) [20] and Fiduccia–Mattheyses (FM) [15] algorithms. These iterative algorithms try to improve the given partition by either swapping vertices between parts or moving vertices from one part to other, while not violating the balance criteria. They also provide heuristic mechanisms to avoid local minima. These algorithms operate on passes. In each pass, a sequence of unmoved/unswapped vertices with the highest *gains* are selected for move/swap, one by one. At the end of a pass, the maximum prefix subsequence of moves/swaps with the maximum prefix sum that incurs the maximum decrease in the cutsize is constructed, allowing the method to jump over local minima. The permanent realization of the moves/swaps in this maximum prefix subsequence is efficiently achieved by rolling back the remaining moves at the end of the overall sequence. The overall refinement process in a level terminates if the maximum prefix sum of a pass is not positive.

PaToH includes original KL and FM implementations, hybrid versions, like one pass FM followed by one pass KL, as well as improvements like *multilevel-gain* concept of Krishnamurthy [21] that adds a look-ahead ability, or *dynamic locking* of Hoffman [17], and Dasdan and Aykanat [14] that relaxes vertex moves allowing a vertex to be moved multiple times in the same pass. PaToH also provides heuristic trade-offs, like *early-termination* in a pass of KL/FM algorithms, or *boundary KL/FM*, which only considers vertices that are in the boundary, to speed up the refinement. The default refinement scheme is boundary FM+KL.

Related Entries

►Chaco
►Data Distribution
►Graph Algorithms
►Graph Partitioning
►Hypergraph Partitioning
►Linear Algebra, Numerical
►Preconditioners for Sparse Iterative Methods

Bibliographic Notes and Further Reading

Latest PaToH binary distributions, including recently developed MATLAB interface [26], and related papers can be found on the Web site listed in [9]. The "Hypergraph Partitioning" entry contains some use cases of hypergraph partitioning.

Bibliography

1. Alpert CJ, Kahng AB (1995) Recent directions in netlist partitioning: a survey. VLSI J 19(1–2):1–81
2. Aykanat C, Cambazoglu BB, Uçar B (May 2008) Multi-level direct k-way hypergraphy partitioning with multiple constraints and fixed vertices. J Parallel Distrib Comput 68(5):609–625
3. Aykanat C, Pinar A, Çatalyürek UV (2004) Permuting sparse rectangular matrices into block-diagonal form. SIAM J Sci Comput 26(6):1860–1879
4. Bui TN, Jones C (1993) A heuristic for reducing fill-in sparse matrix factorization. In: Proceedings of the 6th SIAM conference on parallel processing for scientific computing, Norfolk, Virginia, pp 445–452
5. Catalyurek U, Boman E, Devine K, Bozdag D, Heaphy R, Riesen L (Aug 2009) A repartitioning hypergraphy model for dynamic load balancing. J Parallel Distrib Comput 69(8):711–724
6. Çatalyürek UV (1999) Hypergraph models for sparse matrix partitioning and reordering. Ph.D. thesis, Bilkent University, Computer Engineering and Information Science, Nov 1999. http://www.cs.bilkent.edu.tr/tech-reports/1999/ABSTRACTS.1999.html.
7. Çatalyürek UV, Aykanat C (Dec 1995) A hypergraph model for mapping repeated sparse matrix vector product computations onto multicomputers. In: Proceedings of international conference on high performance computing
8. Çatalyürek UV, Aykanat C (1999) Hypergraph-partitioning based decomposition for parallel sparse-matrix vector multiplication. IEEE Trans Parallel Distrib Syst 10(7):673–693
9. Çatalyürek UV, Aykanat C (1999) PaToH: a multilevel hypergraph partitioning tool, version 3.0. Bilkent University, Department of Computer Engineering, Ankara, 06533 Turkey. PaToH. http://bmi.osu.edu/~umit/software.html, 1999 (accessed on November 26, 2010)
10. Çatalyürek UV, Aykanat C (2001) A hypergraph-partitioning approach for coarse-grain decomposition. In: ACM/EEE SC2001, Denver, CO, November 2001
11. Çatalyürek UV, Aykanat C, Kayaaslan E (2009) Hypergraph partitioning-based_ll-reducing ordering. Technical Report

OSUBMI-TR-2009-n02 and BU-CE-0904, The Ohio State University, Department of Biomedical Informatics and Bilkent University, Computer Engineering Department, 2009. submitted for publication
12. Çatalyürek UV, Aykanat C, Ucar B (2010) On two-dimensional sparse matrix partitioning: models, methods, and a recipe. SIAM J Sci Comput 32(2):656–683
13. Cheng C-K, Wei Y-C (1991) An improved two-way partitioning algorithm with stable performance. IEEE Trans Comput Aided Des 10(12):1502–1511
14. Dasdan A, Aykanat C (February 1997) Two novel multiway circuit partitioning algorithms using relaxed locking. IEEE Trans Comput Aided Des 16(2):169–178
15. Fiduccia CM, Mattheyses RM (1982) A linear-time heuristic for improving network partitions. In: Proceedings of the 19th ACM/IEEE design automation conference, pp 175–181
16. Hendrickson B, Leland R (1993) A multilevel algorithm for partitioning graphs. Technical reports, Sandia National Laboratories
17. Hoffmann A (1994) Dynamic locking heuristic – a new graph partitioning algorithm. In: Proceedings of IEEE international symposium on circuits and systems, pp 173–176
18. Karypis G, Kumar V (1998) A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM J Sci Comput 20(1):359–392
19. Karypis G, Kumar V (1998) Multilevel algorithms for multi-constraint graph partitioning. Technical Report 98-019, University of Minnesota, Department of Computer Science/Army HPC Research Center, Minneapolis, MN 55455, May 1998
20. Kernighan BW, Lin S (Feb 1970) An efficient heuristic procedure for partitioning graphs. Bell SystTech J 49(2):291–307
21. Krishnamurthy B (May 1984) An improved min-cut algorithm for partitioning VLSI networks. IEEE Trans Comput 33(5):438–446
22. Lengauer T (1990) Combinatorial algorithms for integrated circuit layout. Willey–Teubner, Chichester, UK
23. Sanchis LA (Jan 1989) Multiple-way network partitioning. IEEE Trans Comput 38(1):62–81
24. Schloegel K, Karypis G, Kumar V (2000) Parallel multilevel algorithms for multi-constraint graph partitioning. In: Euro-Par, pp 296–310
25. Schweikert DG, Kernighan BW (1972) A proper model for the partitioning of electrical circuits. In: Proceedings of the 9th ACM/IEEE design automation conference, pp 57–62
26. Uçar B, Çatalyürek ÜV, Aykanat C (2010) A matrix partitioning interface to PaToH in MATLAB. Parallel Comput 36(5–6):254–272
27. Wei Y-C, Cheng C-K (July 1991) Ratio cut partitioning for hierarchical designs. IEEE Trans Comput Aided Des 10(7):91–921

Partitioning Tool for Hypergraphs (PaToH)

►PaToH (Partitioning Tool for Hypergraphs)

PC Clusters

►Clusters

PCI Express

Jasmin Ajanovic
Intel Corporation, Portland, OR, USA

Synonyms

3GIO; PCI-Express; PCIe; PCI-E

Definition

PCI (Peripheral Component Interconnect) Express is a highly scalable interconnect technology that is the most widely adopted IO interface standard used in the computer and communication industry [2]. By providing scalable speed/width, extendable protocol capabilities, a common configuration/software model, and various mechanical form-factors, PCI Express supports a broad range of applications. It allows implementation of flexible connectivity between a processor/memory complex and an IO subsystems, including peripheral controllers, such as graphics, networking, storage, etc. PCI Express technology development is managed by PCI-SIG (PCI Special Interest Group), an industry association comprising of over 800 member companies.

Discussion

Introduction – A Brief History of PCIe

PCI Express has his roots in Peripheral Component Interconnect (PCI), an open standard specification that was developed by the computing industry in 1992. PCI was a replacement for the ISA bus which was a mainstream PC architecture IO expansion standard at the time. Although there were several alternative solutions, such as MicroChannel, EISA, and VL-bus, that were aiming to replace/supplement ISA, none of them fully addressed the needs of an evolving PC industry. The PCI specification covered both the hardware and software interfaces between PC's CPU/memory complex and add-in cards, such as graphics, network, and disk controllers. One of the most important aspects of PCI was support for the so called "plug-and-play" mechanisms

which allowed operating system software to detect installed hardware components, including both add-in cards and motherboard-down devices, configure system resources required for their operations, including memory address ranges and interrupts, and install appropriate software device drivers. From a hardware perspective, PCI was initially defined as a 32-bit multiplexed address/data bus that was shared among multiple devices attached to it, operating at 5 Volt and at speed of 33 MHz. Over time, faster variants of PCI evolved to support ever increasing performance requirements of CPUs which rose operational frequencies from 66 MHz in 1993 to over 3 GHz by 2003. These variants include:

- 66 MHz 32- and 64-bit PCI operating at 5 V or 3 V.
- AGP (Accelerated Graphic Port), a graphics-optimized interface operating at a common clock speed of 66 MHz and carrying data transfers at 2x, 4x, and 8x of that nominal speed resulting in maximum bandwidth of 0.5, 1, and 2 GB/s over the 32-bit interface.
- PCI-X, a server-optimized solution operating at clock speeds of 66, 100, and 133 MHz and resulting in maximum bandwidth of 0.5, 0.8, and 1 GB/s over the 64-bit interface.

Towards the end of the twentieth century, neither of these solutions proved to be an adequate answer to emerging applications that required higher scalability (performance and protocol capabilities) and a more efficient interface (pins, bandwidth, and power). In 2001, Intel Corporation, together with several other industry leaders, spearheaded the development of the next generation IO architecture. Initially called 3GIO [1], this architecture was later endorsed by the PCI-SIG as PCI Express (PCIe).

Appearing in systems starting in 2003, PCI Express was rapidly adopted by the PC/PCI ecosystem replacing entirely AGP and PCI-X, and in some instances PCI, within 1 or 2 product generations. The key to the PCI Express success was in leveraging of PCI base architecture by supporting full backwards compatibility at the software level while providing a more optimized interconnect solution. The PCI Express physical interface is based on width- and speed-scalable, point-to-point, differential serial signaling technology operating initially at 2.5 GT/s (Giga Transfers/second). As fast as the new PCI Express standard was, processor architectures have continued to accelerate to meet the demands of emerging applications, and PCI Express needed to keep up. For an example, increasing bandwidth was needed to improve the performance of data-intensive graphics workloads as well as server-class storage and network solutions. PCI Express continued to improve by scaling-up speed as well adding architectural/protocol extensions. Figure 1 shows the evolution of PCI Express technology.

Technology Overview

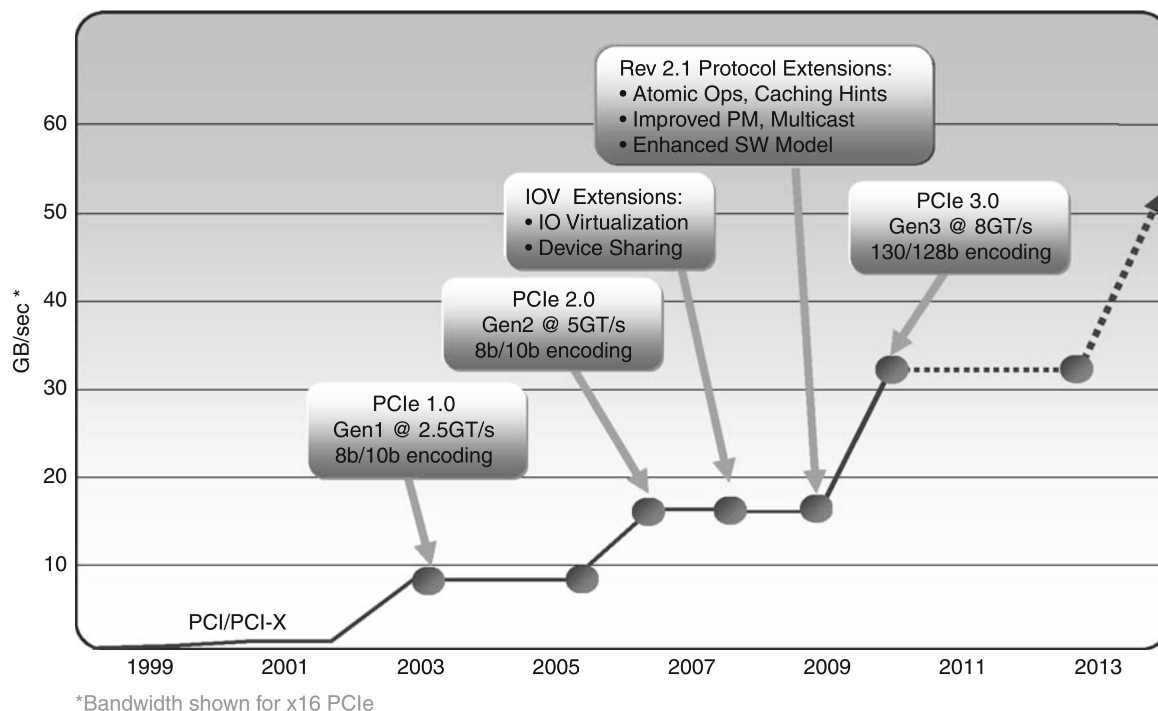
Basic Elements and Concepts

Link and Lane

PCI Express is a point-to-point interconnect technology where Link represents a fundamental element of PCIe -based interconnect fabric. A basic Link is a dual-simplex communications channel between two components that represents a single Lane. Lane consists of two low-voltage, differentially driven signal pairs: a Transmit pair and a Receive pair as shown in Fig. 2. To aggregate the bandwidth of a PCIe Link, multiple Lanes can be grouped together to provide a "wider" Link. In PCIe terminology, Link width is expressed using xN ("by N") denotation, where N is the number of Lanes that form the Link. PCIe architecture specification defines operations for x1, x2, x4, x8, x12, x16, and x32 Link widths. Companion form-factor specifications (add-in card and connector) define operations for a subset of operational widths (x1, x4, x8, and x16).

Signaling, Speed, and Bandwidth

To carry communication over the Link, PCIe uses Low-Voltage Differential Signaling (LVDS) with embedded clocking. Embedded clocking is a mechanism where clock information is embedded within the transmitted data by providing a guaranteed number of transitions between "1"s and "0"s that are required to correctly extract the clock on the receiver side. Revision 1.0 (Gen1) and Revision 2.0 (Gen2) of PCIe Specification use standard 8b/10b encoding scheme [3] which is a common mechanism for a number of industry standard interfaces based on serial signaling technology (e.g., Infiniband, Fibre Channel, SATA, etc.). This scheme



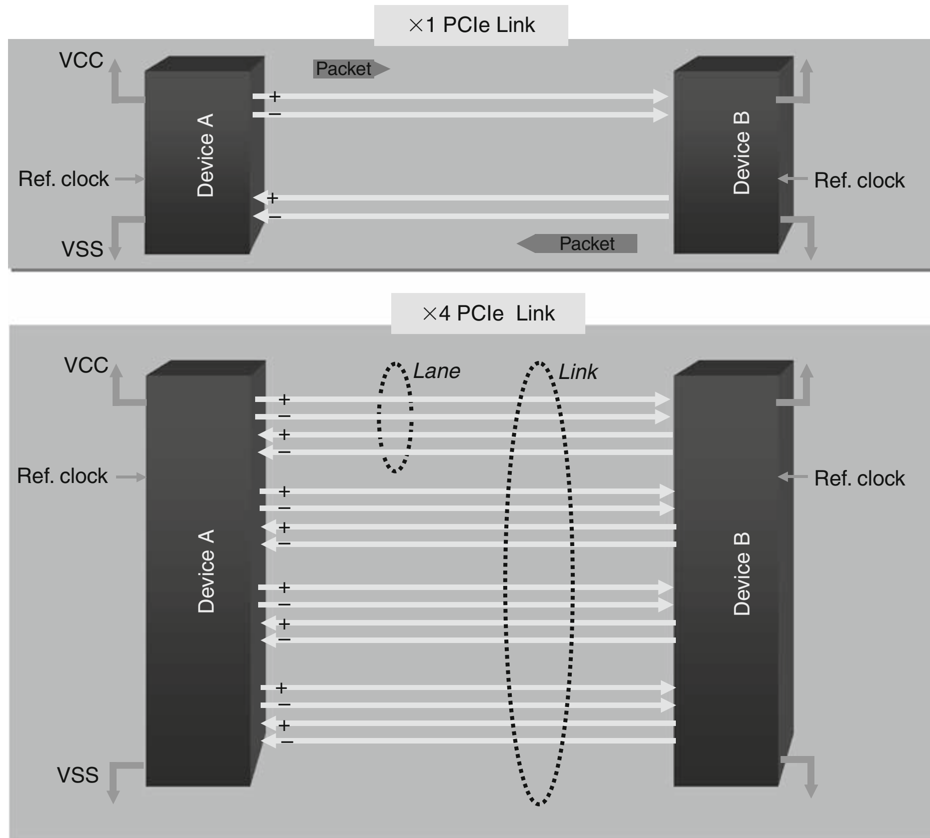
PCI Express. Fig. 1 PCI express technology roadmap

comes with an overhead of 20% because it uses 10bits to carry 8bits of actual information. Since it is relatively simple and very robust, 8b/10b was used by PCIe as a reasonable tradeoff between efficiency and complexity allowing PCIe Gen1 and Gen2 to hit target effective bandwidths while operating at moderately high speeds of 2.5 GT/s and 5 GT/s respectively. However, continuing with 2x increase of operational speed proved to be difficult from a technology enabling/adoptability and ecosystem perspective. For PCIe 3.0 (Gen 3), the PCI-SIG defined a new more efficient 128b/130b encoding scheme, which with ~1% overhead effectively doubles the bandwidth of 5 GT/s Gen2 while operating only at 8 GT/s instead of 10 GT/s (Note that 20% overhead of 8b/10b brings effective data transfers of Gen2 from 5 GT/s down to 4 GT/s.) The following Table 1 shows raw and effective speeds of all three generations of PCIe, including a total bandwidth based on an example of x16 PCIe link.

To support backwards compatibility at the system and component/add-in card level, PCI Express Specification requires newer generation devices that operate at higher speeds to support operations at lower speed. Examples: Gen2 PCIe device operating at nominal 5 GT/s must support operations at 2.5 GT/s, Gen3 device must support operations at both 5 GT/s and 2.5 GT/s.

Link Configuration

PCI Express architecture allows devices that support different Link widths and speeds to be configured for proper operation. The negotiation of width and speed is done as a part of Link initialization process where two devices exchange information about their capabilities using a lowest common denominator method to determine the operational width and speed. For an example, if Device A that supports x8 width at 5.0 GT/s is connected to Device B that supports x4 width at 2.5 GT/s, the Link that connects them will be initialized for x4 width operating at 2.5 GT/s. Note that PCIe allows only configurations of the devices with symmmetric Link width. Link width/speed connfiguration occurs at the Link hardware level without any involvement of software.



PCI Express. Fig. 2 PCI express link examples - x1 and x4

PCI Express. Table 1 PCI express speeds and bandwidths

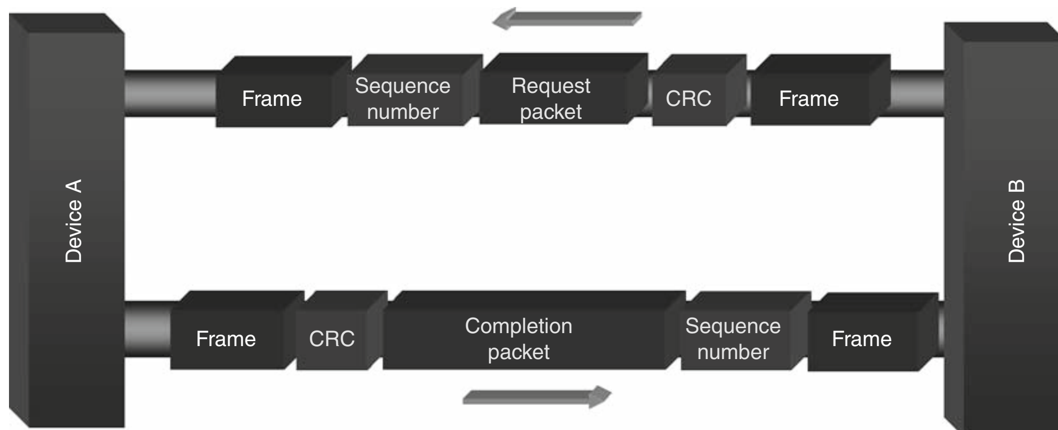
PCIe Generation	Raw bit rate	Effective bit rate	Bandwidth per lane Per direction	Total bandwidth* For x16 link
PCIe 1.x	2.5 GT/s	2 Gb/s	~250 MB/s	~8 GB/s
PCIe 2.0	5.0 GT/s	4 Gb/s	~500 MB/s	~16 GB/s
PCIe 3.0	8.0 GT/s	8 Gb/s	~1 GB/s	~32 GB/s

*Total bandwidth represents the aggregate interconnect bandwidth in both directions

Once Link is initialized and configured for proper width/speed operations, it may be re-configured during the run-time as a result of a RAS (Reliability, Serviceability, Availability) event due to, for example, data integrity problems. Speed and/or width can be reduced in an attempt to correct the problem and keep the system running with reduced performance/functionality. Note that the PCIe Specification defines a mechanism where software, through aceess to configuration/-control registers, can overide the established hardware configuration of the Link and force Link to operate at lower speed than nominally capable.

Packet-Based Protocol

Instead of using dedicated signals for address, control, and data (such as its predecessor PCI), PCI Express uses packets to communicate information between components connected to the PCIe fabric. Packets contain all of the information related to transactions such as: source and target identification/address, type (e.g., Memory Read, Memory Write, Message), attributes (e.g., Isochronous, No Snoop), and data. In addition to this information, Link hardware logic inserts additional information required for correct packet transfer across the Link such as framing, sequencing and packet



PCI Express. Fig. 3 Packet-based protocol

integrity protection (CRC). Packets can be differentiated as Request and Response packets. To complete the entire transaction (e.g., Memory Read), a single Request packet may cause one or multiple Response packets. Note that some request transaction types (e.g., Message) do not require responses (Fig. 3).

PCI Express Layering Overview

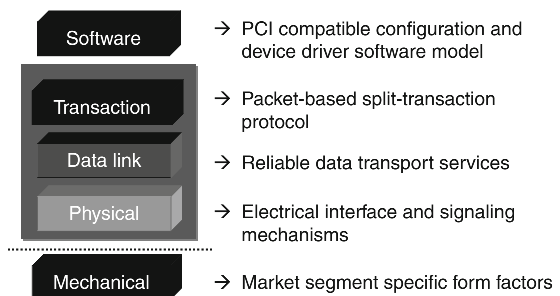
The PCIe interface stack is defined using a layered approach with PCIe Transaction, Data Link, and Physical Layers being formally defined in PCIe Base Specification and the rest of the stack related to Software and Mechanical being covered in companion documents. The following simplified Fig. 4 shows PCIe interface stack and highlights major aspects of each layer.

Transaction Layer

The Transaction Layer is responsible for assembly (for transmit) and disassembly (upon receive) of Transaction Layer Packets (TLPs). PCIe supports load-store as well as message-based transactions, where TLPs are used to communicate transactions such as reads, writes, messages, and events using different type of semantics (Memory, I/O, Configuration, and Message), address types/formats (32-bit/64-bit, device ID), and attributes (No Snoop, Relaxed Ordering, and ID-Based Ordering).

The Transaction Layer manages flow of packets between transmitting and receiving devices using credit-based flow control scheme.

Instead of using physical pins/wires to support sideband signals, such as interrupts, power-management requests, etc., PCIe uses Messages as "virtual wires" that carry information in-band. This improves overall efficiency of the interface by eliminating large number of pins/wires.



PCI Express. Fig. 4 PCI express interface stack

Data Link Layer

This layer provides Link management and data integrity, including error detection and error correction. On the transmit side the Data Link Layer calculates and applies a CRC (Cyclic Redundancy Check) data protection code on the TLP submitted by Transaction Layer. It also adds a TLP sequence number, and passes entire packet to Physical Layer for transmission across the Link. On the receive side the Data Link Layer checks the integrity of received TLPs before passing them to the Transaction Layer for further processing. In the case if an error is detected, this Layer requests retransmission of TLPs until information is correctly received, or the Link is determined to have failed.

Physical Layer

The Physical Layer converts information received from the Data Link Layer into an appropriate serialized format and transmits it across the Link. This Layer consists of Electrical and Logical functional blocks. Electrical block includes all circuitry required for interface operation: output and input buffers, parallel-to-serial and serial-to-parallel conversion, PLL(s), and transmitter/receiver signal conditioning circuitry. Logical block supports functions required for interface initialization and maintenance, including configuration of Link speed and width.

Packet Flow Through the Layers

The TLPs are formed in the Transaction Layer to carry the information from the transmitting component to the receiving component. As the transmitted packets flow through the other layers, they are extended with additional information necessary to carry out transfers. At the receiving side the reverse process occurs and packets get transformed from their Physical Layer representation to the Data Link Layer representation and finally to the form that can be processed by the Transaction Layer of the receiving device. Figure 5 shows the conceptual flow of transaction -level packet information through the layers.

Note that for the purpose of Link management, a simpler form of packet communication is supported between two Data Link Layers that are connected to the same Link. These packets are referred to as Data Link Layer Packets (DLLP).

PCI Express Platform Examples

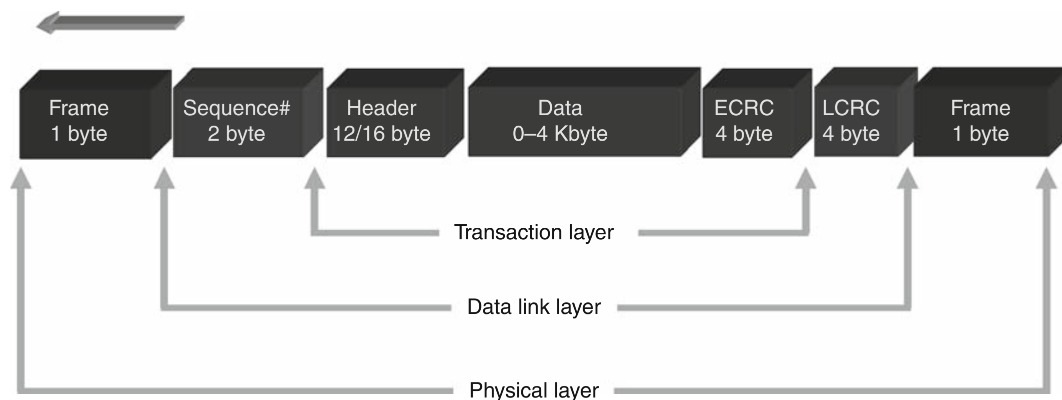
The PCIe architecture supports a variety of platform configurations by allowing the mixing and matching of PCIe link speeds and widths to support different applications. Figure 6 shows examples of PCIe -based client and server computer platforms.

To support high-performance graphics applications, client/workstation computer platforms typically utilize x16 PCIe. Lower bandwidth functions such as network adapters typically use x1 PCIe. Various additional functions (e.g., enhanced audio) can be provided using add-in card slots. Note that an IO Bridge shown on a client platform in Fig. 6 also provides support for some legacy functions, such as old 5 V/33 MHz PCI bus that is required until the technology transition completes.

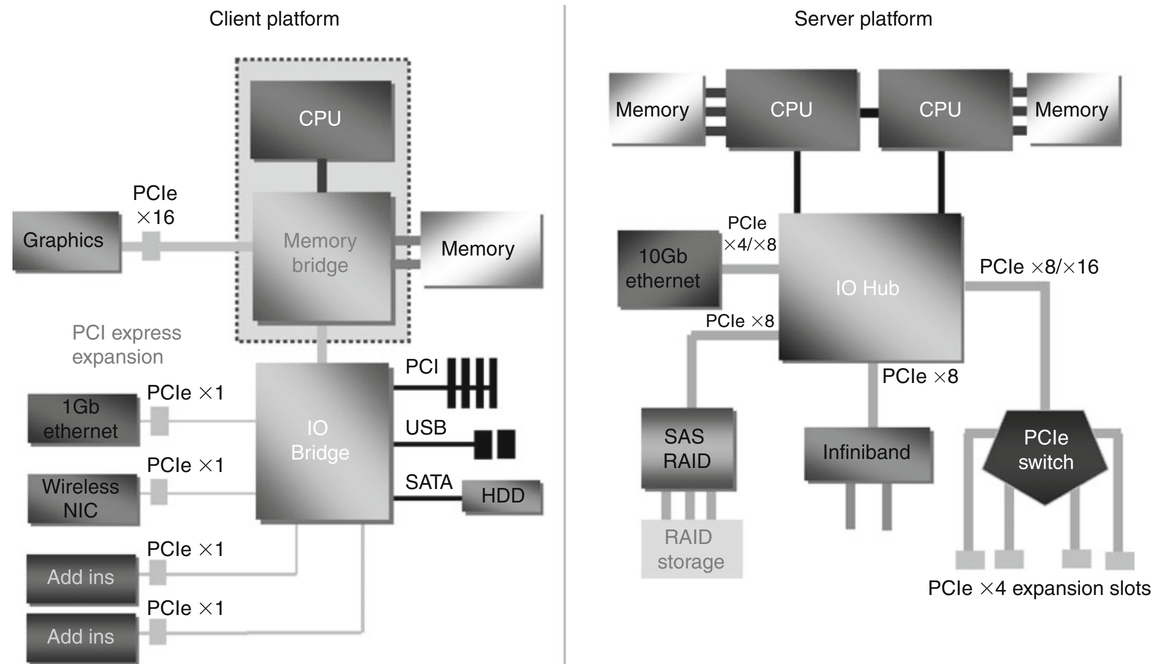
Server platforms typically use x4 and x8 PCIe ports that are required by the high-bandwidth application such as high-performance network (10GigE) and storage (SAS) adapters, Infiniband, Fibre Channel, etc. An important building block for servers is PCIe Switch which is architecturally supported by the PCIe Specification. PCIe Switch is typically used for expansion of IO subsystem by providing additional PCIe interface ports.

Architecture Features

PCI Express is a scalable architecture that provides a highly flexible and expandable set of capabilities. This section highlights essential features of PCIe.



PCI Express. Fig. 5 Packet formation



PCI Express. Fig. 6 Examples of PCIe based platforms

Scalable Protocol

PCIe uses a fully packetized split-transaction protocol as described under the Technology Overview section. In addition to load-store semantics, it provides messaging support that is used by the Virtual Wire mechanism to eliminate side-band signals by converting them into in-band messages. PCIe uses a credit-based scheme to control the flow of packets within the PCIe fabric. This scheme applies per Link, and it is used to prevent buffer overflow. To minimize the dependency between different traffic flows in a system, PCI Express provides Virtual Channel (VC) mechanism and transaction ordering relaxations. These mechanisms mitigate the overhead of a strict producer-consumer ordering model (that may create head-of-line blocking conditions) and allow support for differentiated Quality of Service at the platform level.

For future scalability of supported packet formats, PCIe defines Transaction Layer Packet Prefix, a mechanism for extending TLP headers.

PCI Compatible Software Model

PCIe leverages standard mechanisms defined in the PCI Plug-and-Play specification for device initialization, enumeration, and configuration. This allows legacy software operating systems to boot without modifications on a PCIe -based system. It also preserves compatibility with device driver software model, enabling existing API, and application software to execute unchanged. To remove platform overhead and limitations associated with legacy PCI software configuration, PCIe provides Enhanced Configuration mechanism. This mechanism needs to be supported by newer operating systems to fully take the advantage of PCIe advanced capabilities (e.g., Advanced Error Logging/Reporting, Virtual Channel Mechanism). Enhanced Configuration also allows new capabilities to be added in the future.

Scalable Performance

Primary factor in PCIe performance scaling as measured by the interface bandwidth is function of interface width and operational frequency and can be expressed using the following formula:

Total Link Bandwidth [GB/s]
= ***Link_Width[Number of lanes]***
× **Effective_Signaling_Rate**[**Gb/s**]
× **2**[**directions**]/**8**[**bits**]

GB/s = Giga Byte per second, Gb/s = Giga bit per second, GT/s = Giga Transfer per second
for Gen1 and Gen2: Effective_Signaling_Rate[Gb/s] = Raw_ Signaling_Rate[GT/s] × 0.8
for Gen3: Effective_Signaling_Rate [Gb/s] ≈ Raw_ Signaling_Rate [GT/s]

Secondary factor of PCIe performance is related to the fact that PCIe is point-to-point interconnect with two unidirectional signaling paths which are contention-free i.e., do not require arbitration. This means that in systems that contain multiple PCIe Links, traffic on each Link can occur independently and simultaneously which improves total system throughput. An additional contributor to performance is related to PCIe -pin/bandwidth efficiency. By improving this aspect, PCIe allows a tighter IO integration within the platform, e.g., direct PCIe attach to high-integration CPUs. This can result in lower IO system latencies in an optimized system.

Note that the above formula shows only theoretical bandwidths, but that actual performance as seen at the application level depends on number of factors. These factors include PCIe -interface and system implementation aspects such as: supported data payload size, interface-level data buffering/queuing and effectiveness of flow-control, arbitration mechanisms throughout the platform (e.g., competing for host memory access), power -management mechanisms and policies, software device driver and API overheads, etc.

Advanced Power Management

PCIe defines Link level power -management scheme including the active-state power -management (ASPM) protocol. The ASPM manages power -state transitions on the Link transparently to run-time software by detecting idle conditions on the Link, i.e., when no actual data is being communicated over the Link. Note that in signaling schemes that use embedded clocking, data needs to be transmitted continuously to maintain synchronization between transmitter and receiver. To reduce the power consumption during "idle," PCIe defines low-power link states. These states save power but transitions into and out of them require recovery time to resynchronize the transmitter and receiver which potentially may impact the Link latency and affect the overall system performance. In addition to Link level, power-management PCIe defines related system level power-management mechanisms such as:

- Latency Tolerance Reporting – reduces platform power based on PCIe device service requirements.
- Opportunistic Buffer Flush and Fill – aligns PCIe -device activity with platform power-management events to further reduce platform power.
- Dynamic Power Allocation–dynamic control of power/thermal budget per PCIe device.

Reliability, Availabilty, Serviceability (RAS) Support

RAS capabilities are defined to: ensure data integrity, provide ability to identify/manage errors, and allow installation/removal of components in the running system without requiring operating system shutdown.

- Data Integrity – As a basic/required mechanism, PCIe defines a data integrity scheme where 32-bit CRC (Cyclic Redundancy Check) is used to protect packets transmitted over the single Link. For platforms that use more complex topologies (i.e., with PCIe Switches) and require end-to-end data integrity, PCIe defines an optional end-to-end 32-bit CRC. This CRC code is used in addition to Link local 32-bit CRC to protect the data in high -reliability server applications.
- Hot Plug and Hot Swap – PCIe defines native support for hot plugging or hot swapping of IO cards/modules. Solutions based on PCIe hot plug/swap address requirements of both server and portable computer platforms and support industry standard software stacks for platform management and configuration.
- Advanced Error Reporting/Handling – PCIe defines support for error logging/reporting to improve system -fault isolation and enable recovery solutions.

Differentiated Quality of Service (Qos) Support

Using Virtual Channel (VC) mechanism as a foundation, PCIe supports eight levels of QoS differentiated traffic including isochronous traffic type. PCIe specification defines VC-system configuration and programmable-VC arbitration mechanism necessary to support an end-to-end solution designed for applications, such as isochronous that require real-time delivery of voice and video related data.

IO Virtualization and Device Sharing Support

PCIe defines an IO interface-level mechanism for support of platform virtualization. In virtualized platforms, a single instance of platform hardware is mapped in to a multiple independent Virtual Machines, each running their own operating system and application software stacks. In addition to PCI Express Base Specification, the PCI-SIG maintains a set of specifications that cover PCIe support for:

- Address Translation Services: allow PCIe devices to obtain copies of translated system addresses to mitigate address translation latencies in IO Virtualized platform.
- Single-Root IO Virtualization and Device Sharing: improves system performance scaling by allowing a single PCIe device to be presented to the software as multiple independent/pseudo-independent hardware devices.
- Multi-Root IO Virtualization: allows multiple independent PCIe-based platforms (such as in blade-server systems) to overlap/share system resources.

Note that IO Virtualization capabilities require new software, both at the PCIe fabric configuration/management level as well as at the system level.

Support for Heterogeneous Processing and Application Acceleration

There is a set of capabilities that are provided to support high-performance applications which require efficient co-processing and data sharing, such as GP-GPU, computational accelerators, and network/storage accelerators. These include:

- Transaction Layer Packet Processing Hints – Request hints from PCIe device to enable optimized processing within host memory/cache hierarchy.
- Atomic Read-Modify-Write Transactions – Reduce synchronization overhead for shared data structures.
- Address Based Multicast – Allows significant gains in efficiency compared to multiple unicast transactions by carrying transaction between the single source and multiple destinations.

Form-Factors

In addition to chip-chip connectivity usage, PCI Express supports a variety of system board and add-in card form-factors. To address diversity of applications across many segments (desktop and mobile PCs, servers, embedded/communications, etc.), the following add-in form-factors are defined:

- PCIe Card Electromechanical (CEM) used for desktops, workstations, and servers.
- PCIe Mini Card used for portable computers.
- PCIe Module also known as Server IO Module (SIOM) optimized for server/communication systems.
- ExpressCard used for portable computers and small form-factor desktops.
- PCIe External Cabling Spec used for embedded/communication.

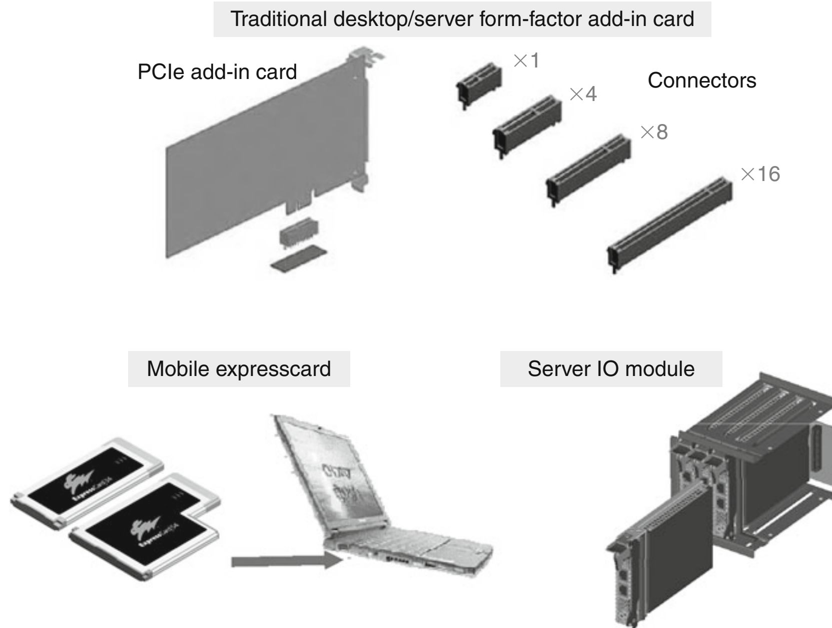
Figure 7 shows a sample of form-factors including the most common PCIe CEM add-in card specified by the PCI-SIG. This card is defined to fit into the traditional desktop PC and server systems and comes with different connector widths (x1, x4, x8, and x16). It supports power usage from baseline 25 W up to 300 W in x16 variant used for high-end graphics and similar applications.

In addition to the PCI-SIG, there are several other industry associations, such as PCMCIA and PICMG that have been involved in development of PCI Express based form-factors.

PCI Express Today

Products based on first generation (2.5 GT/s) and second generation (5 GT/s) of PCIe technologies are deployed broadly across computer and communication/embedded industries. When you consider this in light of the PCI-SIG's 800 members, many of them very active, the PCIe product development and users community represents a large ecosystem. According to a report from an industry analyst organization, In-Stat [2], the size of market of components that include PCIe interface will exceed 440 million units/year by 2010.

PCI-SIG is marching towards the milestone of releasing a third generation of PCI Express spec, i.e., PCIe Rev 3.0 in second half of 2010. In addition to 2.5 and 5 GT/s, PCIe 3.0 supports 8 GT/s operational speed by using a new, more efficient signaling encoding scheme which will allow doubling of bandwidth compared to the prior generation. In addition, the new generation of PCIe products based on Rev 3.0 will support enhanced power management as well as features for



PCI Express. Fig. 7 Example of PCIe form-factors

application performance acceleration through: atomic read-modify-write, data reuse/caching hints, multicast operations, etc.

Future Directions

There are two very significant trends within the mainstream computer and communication industry that will influence the evolution of PCI Express technology:

- Increasing level of integrated functionality resulting in very high-integration single die devices, known as SoC (System on a Chip), or high-integration multi-die components/modules, known as SiP (System in a Package).
- Evolution of Internet with ever increasing need for improved computational, storage, and communication performance of the main building blocks that provide the service.

First trend is mainly driven by the applications that dictate small form-factors, low-power, and low-cost. They range from smart-phones and entertainment devices to the components used in embedded control applications such as home appliances, cars, printers, etc. For some of these products, levels of integration will result in removal of external interfaces such as PCIe and for the others, external interfaces will be still needed, but with a requirement for significantly improved power efficiency. This may dictate evolution of PCIe technology in two directions:

- On-die interconnects that are architecturally equivalent to PCIe and allow integration of PCIe devices in the form of IP (Intellectual Property) building blocks.
- Lower power and lower speed variants of PCIe.

Note that significant value of PCIe ecosystem comes from the ability to support standard operating system software, device drivers, and applications. On-die interconnects that are an architectural equivalent (not necessarily physical/electrical equivalent) of PCIe, will enable integration of hardware functionality without the requirement to change the software stack. This may result in huge R&D savings and faster time-to-market for SoC/SiP designs that use PCIe IP building blocks.

Second trend, evolution and expansion of Internet, will translate into a requirement to the computer industry to provide new generations of communication

servers and data centers that are capable of handling increasing and diverse workloads. This will drive a need for hardware and software optimizations for virtualization and cloud computing. Evolution of traditional rack-servers and blade-servers may open room for a new interconnect backbone/fabric based on PCIe technology. This will drive the following enhancements to the PCIe technology:

- Enhanced routing and configuration to support more elaborate topologies (besides PCI/PCIe hierarchical tree) to allow system scalability through aggregation of large number of discrete devices as well as high-integration devices (that consist of multiple logical devices).
- Capability for tunneling and mapping other protocols (e.g., for storage, networking, system management, etc.) which will allow consolidation of interconnect technologies within the rack/blade and perhaps even a data center.
- Doubling/multiplying bandwidth needed to support next generation of Internet, HPC (High-Performance Computing), and cloud computing. This will eventually result in emergence of PCIe Optical connectivity starting with discrete solutions at speeds ranging from 16GT/s to 25 GT/s and evolving to 50 GT/s silicon-photonics-based integrated solution.

Evolution of PCI Express will likely track the future path of the computer industry. New IO technologies will continue to emerge in the future, using some elements of PCI Express as its foundation.

Related Entries

►Bandwidth-Latency Models (BSP, LoGP)
►Benchmarks
►Buses and Crossbars
►Collective Communication
►Computer Graphics
►Data Centers
►Deadlocks
►Ethernet
►Flow Control
►InfiniBand
►Interconnection Networks
►Network Interfaces
►Routing (Including Deadlock Avoidance)
►Switch Architecture
►Switching Techniques
►Synchronization

Bibliographic Notes and Further Reading

As mentioned in the introduction, development of PCI Express technology as an industry standard is being coordinated through PCI Special Interest Group (PCI-SIG). This organization maintains the website with officially published specification documents and other collaterals that are part of PCI Express standard. For detailed information go to:

- PCI-SIG: www.pcisig.com.

Note that, although some documents are available to a general audience, membership with the PCI-SIG is required for non-restricted access to all documents and specifications managed by the SIG.

In addition to PCI-SIG, there are several other industry organizations, such as:

- PCMCIA: www.pcmcia.org
- PICMG: www.picmg.org

that manage development of complementary industry standards based on PCIe Base Specification. These organizations enhance PCI Express technology by defining additional form-factors and design collaterals that allow even broader adoption of PCI Express component-level hardware products as well as companion software.

Among the component and platform vendors that provide significant additional support for PCI Express technology is Intel Corporation with: Intel® Developer Network for PCI Express* Architecture. This forum provides members with technical data, marketing support and industry connections needed to accelerate the innovation and marketing of PCI Express based solutions. For more details see:

- PCIe DevNet: http://www.intel.com/technology/pciexpress/devnet/

There are several technical books that provide overviews of PCI Express technology as well as detailed implementation guidelines. See references [4–6].

Bibliography

1. Intel white paper. Creating a Third Generation I/O Interconnect. www.intel.com/technology/pciexpress/devnet/docs/WhatisPCIExpress.pdf
2. In-Stat – In Depth Analysis (www.in-stat.com): I/O, I/O, Changing the Status Quo: Chip-to-Chip Interconnects, February 2, 2007, http://www.instat.com/newmk.asp?ID=1909
3. Widmer AX, Franaszek PA (1983) A DC-balanced, partitioned-block 8B/10B transmission code. IBM J Res Dev 27(5):440–451
4. Budruk R, Anderson D, Shanley T (2003) PCI express system architecture. Mindshare, Colorado
5. Wilen A, Schade JP, Thornburg R (2003) Introduction to PCI express: a hardware and software developer's guide Intel Hillsboro
6. Solari Ed, Congdon B, Clark D (2003) The complete PCI express reference: design implications for hardware and software developers (Engineer to Engineer series). Intel, Hillsboro
7. OpenSystems Media – Articles: PCI Express. http://www.opensystems-publishing.com/articles/search/?topic=PCI+Express
8. Compact PCI Systems. http://www.compactpci-systems.com
9. DSP-FPGA.com – Articles, Videos and White Papers: PCI Express. http://www.dsp-fpga.com/articles/search/index.php?mag=&max=10&op=ew&q=pcie&skip=10
10. Embedded Systems. www.embedded.com
11. PCI Express Architecture Frequently Asked Questions, PCI-SIG. http://www.pcisig.com/news_room/faqs/faq_express/
12. PCI Express External Cabling 1.0 Specification. http://www.pcisig.com/specifications/pciexpress/pcie_cabling1.0/
13. PCI-SIG Announces PCI Express 3.0 Bit Rate For Products In 2011 And Beyond. 8 Aug 2007. http://www.pcisig.com/news_room/08_08_07/
14. PHY Interface for the PCI Express Architecture, version 2.00 (PDF). http://download.intel.com/technology/pciexpress/devnet/docs/pipe2_00.pdf
15. PCI Express 3.0 Frequently Asked Questions, PCI-SIG. http://www.pcisig.com/news_room/faqs/pcie3.0_faq/

PCIe

►PCI Express

PCI-E

►PCI Express

PCI-Express

►PCI Express

Peer-to-Peer

Stefan Schmid[1], Roger Wattenhofer[2]
[1]Telekom Laboratories/TU Berlin, Berlin, Germany
[2]ETH Zürich, Zurich, Switzerland

Synonyms

Distributed hash table (DHT); Overlay network; Decentralization; Open distributed systems; Consistent hashing

Definition

The term *peer-to-peer* (p2p) is ambiguous, and is used in a variety of different contexts, such as:

- In popular media coverage, p2p is often synonymous to software or protocols that allow users to "share" files (music, software, books, movies, etc.). p2p file sharing is very popular and a large fraction of the total Internet traffic is due to p2p.
- In academia, the term p2p is used mostly in two ways. A narrow view essentially defines p2p as the "theory behind file-sharing protocols." In other words, how do Internet hosts need to be organized in order to deliver a search engine to find (share) content (files) efficiently? A popular term is "distributed hash table" (DHT), a distributed data structure that implements such a content search engine. A DHT should support at least a search (for a key) and an insert(key, object) operation. A DHT has many applications beyond file sharing, e.g., the Internet domain name system (DNS).
- A broader view generalizes p2p beyond file sharing: Indeed, there is a growing number of applications operating outside the juridical gray area, e.g., p2p Internet telephony à la Skype, p2p mass player games, p2p live audio and video streaming as in PPLive, StreamForge or Zattoo, or p2p social storage and cloud computing systems such as Wuala. Trying to account for the new applications beyond file sharing, one might define p2p as a large-scale distributed system that operates without a central server bottleneck. However, with this definition almost "everything decentralized" is p2p!
- From a different viewpoint, the term p2p may also be synonymous for privacy protection, as various

p2p systems such as Freenet allow publishers of information to remain anonymous and uncensored.

In other words, there is no single well-fitting definition of p2p, as some definitions in use today are even contradictory. In the following, an academic viewpoint is assumed (second and third definition above).

Discussion

The Paradigm

At the heart of p2p computing lies the idea that each network participant serves both as a producer ("server") and consumer ("client") of services. Depending on the application, the shared resources can be data (files), CPU power, disk storage, or network bandwidth. Often p2p systems have an open clientele, and do not rely on the availability of specific individual machines; rather they can deal with dynamic resources and do not exhibit single points of failure or bottlenecks.

Compared to centralized solutions, the p2p paradigm features a better scalability because the amount of resources grows with the network size, availability (avoiding a single point of failure), reliability, fairness, cooperation incentives, privacy, and security – just about everything researchers expect from a future Internet architecture. As such, it is not surprising that new "clean slate" Internet architecture proposals often revolve around p2p concepts.

One might naively assume that for instance scalability is not an issue in today's Internet, as even most popular web pages are generally highly available. However, this is not necessarily due to our well-designed Internet architecture, but rather due to the help of so-called overlay networks: The Google Web site for instance manages to respond so reliably and quickly because Google maintains a large distributed infrastructure, essentially a p2p system. Similarly, companies like Akamai sell "p2p functionality" to their customers to make today's user experience possible in the first place. Quite possibly today's p2p applications are just testbeds for tomorrow's Internet architecture.

Implications

p2p networks are often highly dynamic in nature. While traditional computer systems are typically based on fixed infrastructures and are under a single administrative domain (e.g., owned and maintained by a single company or corporation), the participating machines in p2p networks are under the control of individual (and to some extent: anonymous) users who can join and leave at any time and concurrently. In p2p parlor, such membership changes are called *churn*.

A second implication of the autonomy of the machines in p2p networks is that the network consists of different stakeholders. Users can have various reasons for joining the network. For instance, an (anonymous) user may not voluntarily contribute his or her bandwidth, disk space, or CPU cycles to the system, but prefer to *free ride*. This adds a socioeconomic aspect to p2p computing. As the p2p paradigm relies on the contributions of the participating machines, effective incentive mechanisms have to be designed, which foster cooperation and punish free riders.

Another source of inequality in p2p systems apart from selfishness is *heterogeneity*: Due to the open membership, different machines run different operating systems, have different Internet connections, and so on.

Applications

The best-known representatives of p2p technology are probably the numerous file-sharing applications such as Napster, Gnutella, KaZaA, eMule, or BitTorrent. Also, the Internet telephony tool Skype is very popular and used by millions everyday. Zattoo, PPLive, and StreamForge, among many others, use p2p principles to stream video or audio content. The cloud computing service Wuala offers free online storage by exploiting the participants' disks and Internet connections to improve performance. Recently, the power and anonymity of decentralized Internet working has gained the attention of operators of botnets in order to attack certain infrastructure components by a denial-of-service attack. Finally, p2p technology is used for large-scale computer games.

Architecture Variants

Several p2p architectures are known:

- Client/Server goes p2p: Even though Napster is known to the be first p2p system (1999), by today's standards its architecture would not deserve the label p2p anymore. Napster clients accessed a central server that managed all the information of the shared files, i.e., which file was to be found on

which client. Only the downloading process itself was between clients ("peers") directly, hence p2p. In the early days of Napster the load of the server was relatively small, so the simple Napster architecture was sufficient. Over time, it turned out that the server may become a bottleneck – and an attractive target for an attack. Indeed, eventually a judge ruled the server to be shut down (a "juridical denial of service attack"). However, it remains to note that many popular P2P networks today still include centralized components, e.g., KaZaA or the eDonkey network accessed by the eMule client. Also, the peer swarms downloading the same file in the BitTorrent network are organized by a so-called tracker whose functionality today is still centralized (although initiatives exist to build distributed trackers).

- Unstructured p2p: The Gnutella protocol is the antithesis of Napster, as it is a fully decentralized system, with no single entity having a global picture. Instead each peer connects to a random sample of other peers, constantly changing the neighbors of this virtual overlay network by exchanging neighbors with neighbors of neighbors. (Any unstructured system also needs to solve the so-called bootstrap *problem*, namely how to discover a first neighbor in a decentralized manner. A popular solution is the use of well-known peer lists.) The fact that users often turn off their clients once they downloaded their content implies high levels of churn (peers joining and leaving at high rates), and hence selecting the right "random" neighbors is an interesting research problem. The Achilles' heal of unstructured p2p architectures such as Gnutella is the cost of searching. A search request is typically flooded in the network and each search operation will cost m messages, m being the number of virtual edges in the architecture. In other words, such an unstructured p2p architecture will not scale. Indeed, when Napster was unplugged, Gnutella broke down as well soon afterward due to the inrush of former Napster users.
- Hybrid p2p: The synthesis of client/server architectures such as Napster and unstructured architectures such as Gnutella are hybrid architectures. Some powerful peers are promoted to so-called superpeers (or, similarly, trackers). The set of superpeers may change over time, and taking down a fraction of superpeers will not harm the system. Search requests are handled on the superpeer level, resulting in much less messages than in flat/homogeneous unstructured systems. Essentially, the superpeers together provide a more fault-tolerant version of the Napster server, as all regular peers connect to a superpeer. As of today, almost all popular p2p systems have such a hybrid architecture, carefully trading off reliability and efficiency.
- Structured p2p: Inspired by the early success of Napster, the academic world started to look into the question of efficient file sharing. Indeed, even earlier, in 1997, Plaxton et al. [34] proposed a hypercubic architecture for p2p systems. This was a blueprint for many so-called structured p2p architecture proposals, such as Chord [46], CAN [36], Pastry [37], Tapestry [50], Viceroy [26], Kademlia [27], Koorde [15], SkipGraph [3], and SkipNet [11]. Maybe surprisingly, in practice, structured p2p architectures did not take off yet, apart from certain exceptions such as the Kad architecture (from Kademlia [27]), which is accessible with the eMule client.

Scientific Origins

The scientific foundations of p2p computing were laid many years before the most simple "real" p2p systems like Napster emerged. As already mentioned, in 1997, a blueprint for structured systems has been proposed in [34]. Indeed, also the [34] paper was standing on the shoulders of giants. Some of its eminent precursors are the following:

- Research on linear and consistent hashing, e.g., [16].
- Research on locating shared objects, e.g., [4] or [5].
- Research on so-called compact routing: The idea is to construct routing tables such that there is a trade-off between memory (size of routing tables) and stretch (quality of routes), e.g., [31] or [49].
- Even earlier, hypercubic networks, see below.

Hypercubic Overlays and Consistent Hashing

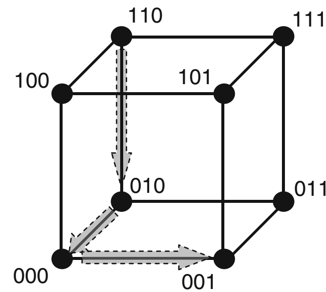
Every application run on multiple machines needs a mechanism that allows the machines to exchange information. A naive solution is to store at each machine the domain name or IP address of every other machine.

While this may work well for a small number of machines, large-scale distributed applications such as file sharing, grid computing, cloud computing, or data center networking systems need a different, more scalable approach: instead of forming a clique (where everybody knows everybody else), each machine should only be required to know some small subset of other machines. This graph of knowledge can be seen as a logical network interconnecting the machines; it is also known as an *overlay network*. A prerequisite for an overlay network to be useful is that it has good topological properties. Among the most important are small peer degree, small network diameter, robustness to churn, or absence of congestion bottlenecks.

The most basic network topologies used in practice are trees, rings, grids, or tori. Many other suggested networks are simply combinations or derivatives of these. The advantage of trees is that the routing is very easy: for every source-destination pair there is only one possible path. However, the root of a tree can be a severe bottleneck. An exception is a p2p streaming system where the single content provider forms the network root. However, trees are also highly vulnerable, e.g., with respect to membership changes.

Essentially all state-of-the-art p2p networks today have some kind of hypercubic topology (e.g., Chord, Pastry, Kademlia). Hypercube graphs have many interesting properties, e.g., they allow for efficient routing: although each peer only needs to store a logarithmic number of other peers in the system (the peers' neighbors), by a simple routing scheme, a peer can reach each other peer in a logarithmic number of steps (or "hops"). In a nutshell, this is achieved by assigning each peer a unique d-bit identifier. A peer is connected to all d peers that differ from its identifier at exactly one bit position. In the resulting hypercube network, routing is done by adjusting the bits in which the source and the destination peers differ – one at a time (at most d many). Thus, if the source and the destination differ by k bits, there are $k!$ routes with k hops. Figure 1 gives an example.

Given a hypercubic topology, it is then simple to construct a distributed hash table (DHT): Assume there are $n = 2^d$ peers that are connected in a hypercube topology as described above. Now a globally known hash function f is used, mapping file names to long bit strings. Let f_d denote the first d bits (prefix) of the bitstring produced by f. If a peer is searching for file name X, it routes a request message $f(X)$ to peer $f_d(X)$. Clearly, peer $f_d(X)$ can only answer this request if all files with hash prefix $f_d(X)$ have been previously registered at peer $f_d(X)$.



Peer-to-Peer. Fig. 1 A simplified p2p topology: a three-dimensional hypercube. Each peer has a three-bit identifier. For example, peer 110 is connected to the three peers 010, 100, 111 whose identifiers differ at exactly one position. In order to route a message from peer 110 to say peer 001, one bit is fixed after the other. One possible routing path is depicted in the figure: 110 → 010 → 000 → 001. An alternative path could be 110 → 111 → 101 → 001

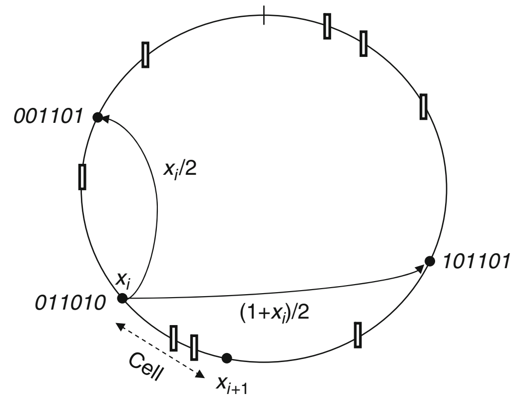
There are some additional issues to be addressed in order to design a DHT from a hypercubic topology, in particular how to allow peers to join and leave without notice. To deal with churn the system needs some level of replication, i.e., a number of peers, which are responsible for each prefix such that failure of some peers will not compromise the system. In addition, there are security and efficiency issues that can be addressed to improve the system.

There are many hypercubic networks that are derived from the hypercube: among these are the butterfly, the cube-connected-cycles, the shuffle-exchange, and the de Bruijn graph. For example, the butterfly graph is basically a "rolled out" hypercube (hence directly providing replication!) of constant degree. Another important class of hypercubic topologies are skip graphs [3, 11].

A simple, interesting way to design dynamic p2p systems is the *continuous–discrete approach* described by Naor and Wieder [29]. This approach is based on a "think continuously, act discretely" strategy, and can be used to design a variety of hypercubic topologies. The continuous–discrete approach gives a unified method

for performing join/leave operations and for dealing with the scalability issue, thus separating it from the actual network. The idea is as follows: Let I be a Euclidean space, e.g., a (cyclic) one-dimensional space. Let G_c be a graph where the vertex set is the continuous set I. Each point in I is connected to some other points. The actual network then is a discretization of this continuous graph based on a dynamic decomposition of the underlying space I into cells where each "server" is responsible for a cell. Two cells are connected if they contain adjacent points in the continuous graph. Clearly, the partition of the space into cells should be maintained in a distributed manner. When a join operation is performed, an existing cell splits, when a leave operation is performed two cells are merged into one. The task of designing a dynamic and scalable network follows these design rules: (1) Choose a proper continuous graph G_c over the continuous space I. Design the algorithms in the continuous setting, which is often simpler (also in terms of analysis) than in the discrete case. (2) Find an efficient way to discretize the continuous graph in a distributed manner, such that the algorithms designed for the continuous graph would perform well in the discrete graph. The discretization is done via a decomposition of I into the cells. If the cells that compose I are allowed to overlap, then the resulting graph would be fault tolerant.

To give an example, in order to build a dynamic *de Bruijn* network (a so-called Distance Halving DHT), a peer at position $x \in [0,1)$ (in binary form $b_1 b_2 \dots$ such that $x = \sum_{i=1}^{\infty} 2^{-b_i}$) connects to positions $l(x) := x/2 \in [0,1)$ and $r(x) := (1+x)/2 \in [0,1)$ in G_c (out-degree two per peer). Observe that if position x is written in binary form, then $l(x)$ effectively shifts in a "0" from the left and $r(x)$ shifts in a "1" from the left. Thus, routing is straightforward: based solely on the current position and the destination (without the overhead of maintaining routing tables), a message can be forwarded by a peer by fixing one bit per hop. The set of peers in the cyclic $[0,1)$ space then define the p2p network: Let x_i denote the position of the ith peer (ordered in increasing order with respect to position). Peer i is responsible for the cell $[x_i, x_{i+1})$, computed in a modulo manner, i.e., this peer is responsible to store the data mapped to this cell plus for the establishment of the corresponding connections defined in G_c. Figure 2 gives an example.



Peer-to-Peer. Fig. 2 The continuous–discrete approach for the dynamic de Bruijn graph. Peers are indicated using circles, files using rectangles. In the continuous setting, the peer at position $x_i = 0.011010$ (in binary notation) is connected to positions $x_i/2$ and $(1+x_i)/2$. In the discrete setting, it is responsible for the cell (i.e., the connections and files that are mapped there) between positions x_i and x_{i+1}

Dealing with Churn

A distinguishing property of p2p systems are the frequent membership changes. Measuring the churn levels of existing p2p systems is challenging and one has to be careful when generalizing a given measurement to entire application classes (e.g., [10]). Nevertheless, several insightful measurement studies have been conducted. For instance, [9, 38] reported on the dynamic nature of early p2p networks such as Napster and Gnutella, and [41] analyzed low-level data of a large Internet Service Provider (ISP) to estimate churn. Also the Kad DHT has been subject to measurement studies, and the reader is referred to the results in [47] and [43].

It is widely believed that hypercubic structures are a good basis for churn-resilient p2p systems. As written earlier, a DHT is essentially a hypercubic structure with peers having identifiers such that they span the ID space of the objects to be stored. A simple approach to map the ID space onto the peers has already been described for the hypercube. To give another example, in the butterfly network, we may use its layers for replication, i.e., all peers with the same ID redundantly store the data of the same hash prefix. Other hypercubic DHTs can be more difficult to design, e.g., networks based on the pancake graph [19].

For many well-known systems, theoretic analyses exist showing that the networks remain well-structured after some joins, leaves, or failures occur. In order to evaluate the robustness formally, metrics such as the network *expansion* (for deterministic failures) or the *span* [6] (for randomized failures) are used. Unfortunately, the span is difficult to compute, and the span value is known only for the most simple topologies.

The continuous–discrete approach [29] already mentioned constitutes the basis of several dynamic systems. For example, the SHELL system [40] is robust to certain attacks by connecting older or more reliable peers in a core network where access can be controlled; SHELL also allows to organize heterogeneous peers in an efficient topology.

Many systems proposed in the literature offer a high robustness in the average case, i.e., they provide probabilistic guarantees that hold with high probability. Robustness under attacks or worst-case dynamics is less well understood. In [19], a system is developed that achieves an optimal worst-case robustness in the sense that there is no alternative system that can tolerate higher churn rates without disconnecting. The basic idea is to simulate a hypercube: each peer is part of a distinct hypercube *node*; each hypercube node consists of a logarithmic number of peers. Peers have connections to other peers of their hypercube node and to peers of the neighboring hypercube nodes. After a number of joins and leaves, some peers may have to change to another hypercube node such that up to constant factors, all hypercube nodes have the same cardinality at all times. If the total number of peers grows or shrinks above or below a certain threshold, the dimension of the hypercube is increased or decreased by one, respectively. The balancing of peers among the nodes can be seen as a *dynamic token distribution problem* on the hypercube: Each node of a graph (hypercube) has a certain number of tokens, and the goal is to distribute the tokens along the edges of the graph such that all nodes end up with the same or almost the same number of tokens. Thus, the system builds on two basic components: (1) an algorithm, which performs the described dynamic token distribution and (2) an information aggregation algorithm, which is used to estimate the number of peers in the system and to adapt the hypercube's dimension accordingly. These techniques also work for alternative graphs, like pancake graphs [19].

An appealing notion of robustness is *topological self-stabilization*: A p2p topology is called self-stabilizing if it is guaranteed that from any weakly connected initial state (e.g., after an attack), it will quickly converge to a desirable network in the absence of further membership changes. In contrast to the worst-case churn considered in [19], self-stabilization focuses on the convergence time in periods without membership changes, but allows for general initial system states. While until recently, self-stabilizing algorithms with guaranteed runtime have only been known for simple one-dimensional or two-dimensional linearization problems [14], recently a construction for a variation of skip graphs, namely SKIP+ graphs [13], has been proposed. Single joins and leaves in SKIP+ can be handled locally, and require logarithmic time and polylogarithmic work only. However, there remains the important open question of how to provide degree guarantees during convergence from arbitrary states.

Fostering Cooperation

The appeal of p2p computing arises from the collaboration of the system's constituent parts, the peers. If all the participating peers contribute some of their resources, highly scalable decentralized systems can be built. However, in reality, peers may act selfishly and strive for maximizing their own utility by benefitting from the system without contributing much themselves [42]. Hence, the performance of a p2p system crucially depends on its capability of dealing with selfishness.

Already in 2000, Adar and Huberman [1] noticed that there exists a large fraction of free riders in the file-sharing network Gnutella. The problem of selfish behavior in p2p systems has been a hot topic in p2p research ever since, and many mechanisms to encourage cooperation have been proposed [30]. Perhaps the simplest fairness mechanism is to directly incorporate contribution monitoring into the client software. For instance, in the file-sharing system KaZaA, the client records the contribution of its user. However, such a solution can simply be bypassed by implementing a different client that hard-wires the contribution level to the maximum, as it was the case with *KaZaA Lite*. Inspired by real economies, some researchers have also proposed the introduction of some form of virtual money, which is used for the transactions.

BitTorrent has incorporated a fairness mechanism from the beginning and has hence been subject to intensive research (e.g., [21, 22, 35]). Although this mechanism has similarities to the well-known tit-for-tat scheme, the strategy employed in BitTorrent distinguishes itself from the classic mechanism in many respects. For instance, it is possible for peers to obtain parts of a file "for free," i.e., without reciprocating. While this may be a useful property for bootstrapping newly joined peers, it has been shown that the BitTorrent mechanism can be exploited: the *BitThief* BitTorrent client [24] allows to download entire files fast without uploading any data. It has also been demonstrated in [24] that sharing communities are particularly vulnerable to such exploits. BitThief is not the only client cheating BitTorrent. Piatek et al. [32] presented *BitTyrant*. BitTyrant's strategy is to exploit the BitTorrent protocol in order to maximize download rates. For instance, BitTyrant uses a smart neighbor selection strategy and connects to those peers with the best reciprocation ratios. In contrast to BitThief, BitTyrant does not free ride. BitTyrant seeks to provide the minimal necessary contribution, and also increases the active neighbor set if this is beneficial to the download rate. The authors claim that their client provides a median 70% performance gain in certain environments.

There can be many other forms of strategic behavior in open distributed systems. One subject that has recently gained attention, especially by the game-theoretic research community, is *neighbor selection* in unstructured p2p networks (e.g., [28]). There may be several reasons for a peer to prefer connecting to some peers rather than others. For instance, a peer may want to connect to peers with high bandwidths, peers storing many interesting files, or peers having large degrees and hence provide quick access to many other peers. At the same time, a selfish peer itself may not be eager to store and maintain too many neighbors itself.

Current Trends and Outlook

One can argue that today, p2p computing is already a relatively mature (research) field; nevertheless, there are still many active discussions and developments, also in the context of the future Internet design. Moreover, there exists a discrepancy between the technology of the systems in use and what is actually known in theory. For example, the Kad network is still vulnerable to quite simple attacks [44].

If employed by the wrong people, the flexibility and robustness of p2p technology also constitutes a threat. Denial-of-service attacks are arguably one of the most cumbersome problems in today's Internet, and it is appealing to coordinate botnets in a p2p fashion. A DHT can be used by the bots, e.g., to download new instructions. For instance, it was estimated that in 2007, the DHT-based *Storm botnet* [20] ran on several million computers. Apart from mechanisms to detect or prevent attacks even before they take place, a smart redundancy management may improve availability during the attack itself (see, e.g., the Chameleon system [7]).

In terms of cooperation, there is a tension between the goal of providing incentive compatible mechanisms that exclude free riders and the goal of designing heterogeneous p2p systems that also tolerate (and make use of!) weak participants. Moreover, in addition to design mechanisms dealing with pure selfishness, there is a trend toward p2p systems that are also resilient to malicious behavior (see, e.g., [23] or [39]).

Another active discussion regards the interface between p2p systems and ISPs. The large amount of p2p traffic raises the question of how ISPs should deal with p2p, e.g., by caching contents. p2p networks often employ inefficient overlay-to-ISP mappings as the logical overlay network is typically not aware of the underlying "real" networks and constraints, and much overhead can be avoided by improving the interface between p2p networks and ISPs, e.g., by an oracle [2]. For a critical point of view on the subject, the reader is referred to [33].

It seems that while a few years ago the lion's share of Internet traffic was due to p2p, the proportion seems to be declining [12] now. Especially web services and server-based solutions such as the popular YouTube and RapidShare are catching up. The measured data traces should be interpreted with care however, as they do not take into account what happens behind the scenes of big corporations. Indeed, it is believed that there is a paradigm shift in p2p computing: While p2p retreats (relatively to other applications) from public Internet traffic, today p2p technology plays a crucial role in the coordination and management of large data centers and server farms of corporations such as Akamai or Google.

Related Entries

►Hypercubes and Meshes

Bibliographic Notes and Further Reading

Beyond the specific literature pointed to directly in the text, there are several recommendable introductory books on p2p computing. In particular, the reader is referred to the classic books [8, 45, 48] and two more recent issues [8, 17]. The theoretically more inclined reader may also be interested in [25], which provides an overview of compact routing solutions, and [18] which discusses trade-offs in local algorithms that achieve global goals based on local information only and without centralized entities whatsoever. Regarding the challenges of distributed cooperation, the recent book [30] gives a thorough and up-to-date survey of current (game-theoretic) trends, and also includes a chapter on p2p specific questions.

Bibliography

1. Adar E, Huberman B (2000) Free riding on gnutella. First Monday 5(10):1–22
2. Aggarwal V, Feldmann A, Scheideler C (2007) Can ISPs and p2p users cooperate for improved performance? ACM Comput Commun Rev 37(3):29–40
3. Aspnes J, Shah G (2003) Skip graphs. In: Proceedings of the 14th annual ACM-SIAM symposium on discrete algorithms (SODA), Baltimore, 2003
4. Awerbuch B, Peleg D (1990) Sparse partitions. In: Proceedings of the 31st annual symposium on foundations of computer science (SFCS), vol 2, pp 503–513, Washington, 1990
5. Awerbuch B, Peleg D (1995) Online tracking of mobile users. JACM 42(5):1021–1058
6. Bagchi A, Bhargava A, Chaudhary A, Eppstein D, Scheideler C (2004) The effect of faults on network expansion. In: Proceedings of the 16th annual ACM symposium on parallelism in algorithms and architectures (SPAA), Barcelona, 2004
7. Baumgart M, Scheideler C, Schmid S. A DoSresilient information system for dynamic data management. In: Proceedings of the 21st ACM symposium on parallelism in algorithms and architectures (SPAA), Calgary, Alberta, 2009
8. Buford J, Yu H, Lua EK (2008) P2P networking and applications. Morgan Kaufmann, San Francisco, 2008
9. Gummadi K, Dunn R, Saroiu S, Gribble SD, Levy HM, Zahorjan J (2003) Measurement, modeling, and analysis of a peer-to-peer file-sharing workload. In: Proceedings of the 19th ACM symposium on operating systems principles (SOSP), Bolton Landing, 2003
10. Haeberlen A, Mislove A, Post A, Druschel P (2006) Fallacies in evaluating decentralized systems. In: Proceedings of the 5th international workshop on peer-to-peer systems (IPTPS), Santa Barbara, 2006
11. Harvey NJA, Jones MB, Saroiu S, Theimer M, Wolman A. Skipnet: a scalable overlay network with practical locality properties. In: Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS), Seattle, 2003
12. IPOQUE (2009) Internet study 2008/2009. http://www.ipoque.com/resources/internet-studies/internet-study-2008-2009, pp 704–713 (accessed on October 31, 2010)
13. Jacob R, Richa A, Scheideler C, Schmid S, Täubig H (2009) A distributed polylogarithmic time algorithm for self-stabilizing skip graphs. In: Proceedings of the ACM symposium on principles of cistributed computing (PODC), New York, 2009
14. Jacob R, Ritscher S, Scheideler C, Schmid S (2009) A self-stabilizing and local delaunay graph construction. In: Proceedings of the 20th international symposium on algorithms and computation (ISAAC), Hawaii, 2009
15. Kaashoek F, Karger DR (2003) Koorde: a simple degree-optimal distributed hash table. In: Proceedings of the international workshop on peer-to-peer systems (IPTPS), Berkeley, 2003
16. Karger D, Lehman E, Leighton T, Panigrahy R, Levine M, Lewin D (1997) Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In: Proceedings of the 29th ACM symposium on theory of computing (STOC), New York, pp 654–663, 1997
17. Khan J, Wierzbicki A (2008) Foundation of peer-to-peer computing. Elsevier Computer Communication, 2008
18. Kuhn F, Moscibroda T, Wattenhofer R (2006) The price of being near-sighted. In: Proceedings of the 17th ACM-SIAM symposium on discrete algorithms (SODA), Miami, 2006
19. Kuhn F, Schmid S, Wattenhofer R (2010) Towards worstcase churn resistant peer-to-peer systems. J Distrib Comput (DIST) 22(4):249–267
20. Larkin E (2007) Storm worm's virulence may change tactics. British Computer Society (accessed on August 03, 2007)
21. Legout A, Urvoy-Keller G, Michiardi P (2006) Rarest first and choke algorithms are enough. In: Proceedings of the 6th ACM SIGCOMM conference on internet measurement (IMC), pp 203–216, Rio de Janeriro, 2006
22. Levin D, LaCurts K, Spring N, Bhattacharjee B (2008) Bittorrent is an auction: analyzing and improving bittorrent's incentives. SIGCOMM Comput Commun Rev 38(4):243–254
23. Li H, Clement A, Marchetti M, Kapritsos M, Robinson L, Alvisi L, Dahlin M (2008) Flightpath: obedience vs choice in cooperative services. In: Proceedings of the symposium on operating systems design and implementation (OSDI), San Diego, 2008
24. Locher T, Moor P, Schmid S, Wattenhofer R (2006) Free riding in bittorrent is cheap. In: Proceedings of the 5th Workshop on Hot Topics in Networks (HotNets), Irvine, 2006
25. Malkhi D (2004) Locality-aware network solutions. Technical Report, The Hebrew University of Jerusalem, HUJI-CSE-LTR-2004-6

26. Malkhi D, Naor M, Ratajczak D (2002) Viceroy: a scalable and dynamic emulation of the buttery. In: Proceedings of the 21st annual symposium on principles of distributed computing (PODC), Monterey, 2002
27. Maymounkov P, Mazières D (2002) Kademlia: a peer-to-peer information system based on the xor metric. In: Proceedings of the 1st international workshop on peer-to-peer systems (IPTPS), Cambridge, 2002
28. Moscibroda T, Schmid S, Wattenhofer R (2006) On the topologies formed by selfish peers. In: Proceedings of the 25th annual symposium on principles of distributed computing (PODC), Denver, 2006
29. Naor M, Wieder U (2003) Novel architectures for p2p applications: the continuous-discrete approach. In: Proceedings of the 15th annual ACM symposium on parallel algorithms and architectures (SPAA), pp 50–59, San Diego, 2003
30. Nisan N, Roughgarden T, Tardos E, Vazirani VV (2007) Algorithmic game theory. Cambridge University Press, Cambridge
31. Peleg D, Upfal E (1988) A tradeoff between space and efficiency for routing tables. In: Proceedings of the 20th annual ACM symposium on theory of computing (STOC), pp 43–52, Chicago, 1988
32. Piatek M, Isdal T, Anderson T, Krishnamurthy A, Venkataramani A (2007) Do incentives build robustness in bittorrent? In: Proceedings of the 4th USENIX symposium on networked systems design and implementation, Cambridge, 2007
33. Piatek M, Madhyastha HV, John JP, Krishnamurthy A, Anderson T (2009) Pitfalls for ISP-friendly p2p design. In: Proceedings of the hotnets, New York, 2009
34. Plaxton C, Rajaraman R, Richa AW (1997) Accessing nearby copies of replicated objects in a distributed environment. In: Proceedings of the 9th ACM symposium on parallel algorithms and architectures (SPAA), pp 311, 320, Newport, 1997
35. Qiu D, Srikant R (2004) Modeling and performance analysis of bittorrent-like peer-to-peer networks. In: Proceedings of the conference on applications, technologies, architectures, and protocols for computer communications (SIGCOMM), pp 367–378, New York, 2004
36. Ratnasamy S, Francis P, Handley M, Karp R, Schenker S (2001) A scalable content-addressable network. In: Proceedings of the ACM SIG-COMM conference on applications, technologies, architectures, and protocols for computer communications, pp 161–172, New York, 2001
37. Rowstron AIT, Druschel P (2001) Pastry: scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In: Proceedings of the IFIP/ACM international conference on distributed systems platforms (middleware), pp 329–350, Heibelberg, 2001
38. Saroiu S, Gummadi PK, Gribble SD (2002) A measurement study of peer-to-peer file sharing systems. In: Proceedings of the multimedia computing and networking (MMCN), San Jose, 2002
39. Scheideler C (2005) How to spread adversarial nodes?: rotate! In: Proceedings of the 37th Annual ACM symposium on theory of computing (STOC), pp 704–713, Baltimore, 2005
40. Scheideler C, Schmid S (2009) A distributed and oblivious heap. In: Proceedings of the 36th international colloquium on automata, languages and programming (ICALP), Rhodes, 2009
41. Sen S, Wang J (2004) Analyzing peer-to-peer traffic across large networks. IEEE/ACM Trans Netw 12(2):219–232
42. Shneidman J, Parkes DC (2003) Rationality and self-interest in peer to peer networks. In: Proceedings of the 2nd international workshop on peer-to-peer systems (IPTPS), Berkeley, 2003
43. Steiner M, Biersack EW, Ennajjary T (2007) Actively monitoring peers in KAD. In: Proceedings of the 6th international workshop on peer-to-peer systems (IPTPS), Bellevue, 2007
44. Steiner M, En-Najjary T, Biersack EW (2007) Exploiting KAD: possible uses and misuses. Comput Commu Rev 37(5):65–69
45. Steinmetz R, Wehrle K (2005) Peer-to-peer systems and applications. Springer, Heidelberg
46. Stoica I, Morris R, Karger D, Kaashoek F, Balakrishnan H (2001) Chord: a scalable peer-to-peer lookup service for internet applications. In: Proceedings of the ACM SIGCOMM conference on applications, technologies, architectures, and protocols for computer communications, San Diego, 2001
47. Stutzbach D, Rejaie R (2006) Understanding churn in peer-to-peer networks. In: Proceedings of the 6th internet measurement conference (IMC), New York, 2006
48. Subramanian R, Goodman B (2005) Peer-to-peer computing: the evolution of a disruptive technology. IGI, Hershey
49. Thorup M, Zwick U (2001) Compact routing schemes. In: Proceedings of the annual ACM symposium on parallel algorithms and architectures (SPAA), pp 1–10, Crete, Greece, 2001
50. Zhao BY, Huang L, Stribling J, Rhea SC, Joseph AD, Kubiatowicz J (2004) Tapestry: a resilient global-scale overlay for Service deployment. IEEE journal on selected areas in commonuincations vol 22, No. 1

Pentium

►Intel Core Microarchitecture, x86 Processor Family

PERCS System Architecture

E. N. (Mootaz) Elnozahy[1], Evan W. Speight[1], Jian Li[1], Ram Rajamony[1], Lixin Zhang[1], Baba Arimilli[2]
[1]IBM Research, Austin, TX, USA
[2]IBM Systems and Technology Group, Austin, TX, USA

Definition

In 2002, IBM started to develop a pioneering supercomputer, codenamed PERCS (Productive, Easy-to-use, Reliable Computing System), with support from the

Defense Advanced Research Project Agency (DARPA). The project was part of DARPA's High Productivity Computing Systems (HPCS)[3] initiative, a 10-year research program that sought to change the landscape of high-end computing by shifting the focus away from just floating point performance toward overall system productivity. Enhancing system productivity required unprecedented innovation in the system design to simplify system usage and programming tasks, all while maintaining the system cost within the requirements of a realistic commercial offering. The system's productivity is orders of magnitude better than previous supercomputers as expressed by the High Productivity Challenge benchmark suite [7].

Discussion

Introduction

High-end computing went through a generational transformation during the 1990s, which saw emerging clusters of commodity processors and networks gradually replace traditional supercomputer systems based on vector and large shared-memory systems. Commodity clusters offered advantages in cost and scalability compared with the technologies they replaced. The simultaneous emergence of the Message Passing Interface (MPI) provided programmers with a portable, standard programming environment that exploited these clusters well. Interest in high-performance computing gained momentum, and a semiannual "top 500 supercomputers" contest was created based on the Linpack benchmark [4] to assess the available supercomputers in the market and to provide an arena for vendors to compete.

With the Linpack benchmark's emphasis on floating-point performance, designers focused on improving processor performance, and the resulting systems were showing great leaps in Linpack performance. These improvements however did not benefit mainstream applications that required a system design that balances floating-point operations with commensurate memory and communication bandwidths. For example, streaming data applications, message-intensive applications exemplified by the GUPS benchmark [7], and digital-signal processing applications based on Fast Fourier Transform [5] are sensitive to network performance and memory bandwidth. Programmers were forced to adapt their algorithms to reduce data transfers and harness the increased computational capability of commodity clusters. The approach proved futile as programmers generally failed to achieve performance gains to show for the added code complexity and programming efforts. A programmer and system productivity crises were clearly at hand.

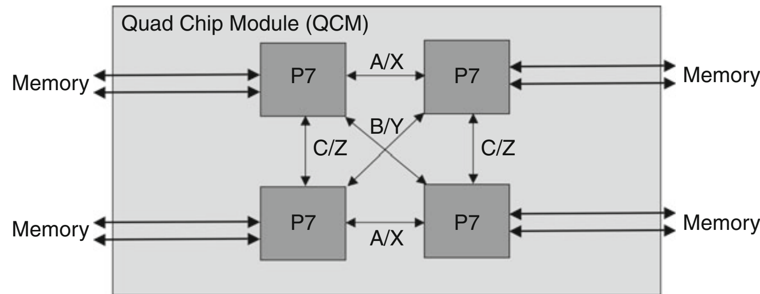
In 2001, visionaries at DARPA and various U.S. defense agencies took the initiative to confront the situation and started the High Productivity Computing System (HPCS) program. HPCS took the form of three competitive stages, the first two being devoted to pure research with each round culminating in the elimination of proposals that were deemed noncompetitive or noninnovative. The third stage was devoted to the commercialization of the research results of the first two stages into a mainstream product. Five companies competed in the initial stage, namely Cray, HP, IBM, SGI, and Sun, and by 2007, only Cray and IBM were supported to proceed with the third stage of commercialization. The HPCS vision called for balanced system attributes and high-level computing abstractions that alleviate programmers from the chores of adapting code to improve performance. The program aimed to foster research that would result in systems that are amenable to productive use without too much effort on the part of the programmer. This article provides a short summary of IBM's PERCS project. Compared to state-of-the-art high-performance computing (HPC) systems, PERCS achieves high performance and productivity goals through tight integration of computing, networking, storage, and software. PERCS focuses on scaling all parts of the system while easing the programming complexity.

Key Elements of the PERCS Design

Compute Node Design

The building block for the PERCS design is the compute node as shown in Fig. 1.

There are four POWER7 chips [6] in a node with a single operating system image that controls resource allocation. Each chip features eight cores, four threads per core, and two memory controllers that can connect to 128GB of DRAM memory. Applications executing on a single compute node can thus utilize 32 cores, 128 SMT threads, eight memory controllers, up



PERCS System Architecture. Fig. 1 PERCS compute node architecture

to 512 GB of memory capacity, one teraflop of compute power, and over 512 GB/s of memory bandwidth. The four POWER7 chips are cache coherent and are tightly coupled using three pairs of buses. Each processor has six fabric bus interfaces to connect to three other processors in a multi-chip module (MCM). These 1-teraflop tightly coupled shared-memory nodes deliver 192 GB/sec of bandwidth (0.2 Byte/flop) to a specialized hub chip used for I/O, messaging, and switching. The POWER7 chip represents a large leap forward in single-chip performance, available both in terms of computational ability and achievable memory bandwidth over other processor chip offerings.

PERCS Interconnect

The challenge of building a highly productive system required the entire system infrastructure to scale along with the microprocessor's capabilities. Previous solutions such as fat-tree Interconnect suffered from substantial communication latency due to message copying, protocol processing, multiple data conversions between optical and electrical signaling, and switching overheads. Other approaches such as toroidal networks mitigated some of these issues at the expense of handing the programmer very complex communication topologies.

Such limitations have forced programmers traditionally to consolidate data communications into large messages that amortize the communications overhead. Applications that required frequent exchange of short messages [7] either had to settle for poor performance or had to be modified in nontrivial ways to exploit large messages. To solve this problem, we had to design an Interconnect that obeyed the following design principles in a cost-effective product:

(a) Minimum data copying
(b) Only one conversion from electrical to optical domain was allowed for any message
(c) Simple communications protocol with limited processing overhead
(d) Maintain a simple topology that simplifies programming

We approached the problem in an unorthodox way by building on the shared-memory protocol that IBM uses to build highly-scalable shared-memory machines. The communication is treated at three levels:

(a) Within a node, communications take place over the shared-memory bus.
(b) A second level consists of a *supernode*, which is a group of 32 nodes connected via noncoherent shared-memory buses. Physically, two supernodes fit within a single rack.
(c) A third level consists of communications between supernodes. Supernodes are connected using a fully connected graph of optical cables. Thus, each supernode has an equal distance to other supernodes, simplifying the topology and the programming task.

A communication coprocessor [1] resides on the memory bus within each node, and thus data can be streamed from source to destination using the shared-memory protocol, using a 128-byte payload. This eliminates the data copying common in previous solutions where data had to be copied from the memory to the I/O bus.

The communication coprocessor, also called the hub-chip [1], is directly connected to the POWER7 MCM at 192 GB/s and provides 336 GB/s to seven other nodes in the same drawer on copper connections; 240 GB/s to 24 nodes in the same supernode (composed of four drawers, or 32 compute nodes) on optical connections; 320 GB/s to other supernodes on optical connections; and 40 GB/s for general I/O, for a total of 1,128 GB/s peak bandwidth per hub chip (see Fig. 2).

Two key design goals for PERCS were to dramatically improve bisection bandwidth (over other topologies such as fat-tree interconnects) and to eliminate the need for external switches. With these goals in mind, the hub chip was designed to support a large number of links that connect it to other hub chips. These links are classified into two categories "L," and "D," which permit the system to be organized into a two-level direct-connect topology. Every hub chip has thirty-one L links that connect to 31 other hub chips. Within this group of thirty-two hub chips, every chip has a direct communication link to every other chip. The hub chip implementation further divides the L links into two categories: seven electrical LL links with a combined bandwidth of 336 GB/s and 24 optical LR links with a combined bandwidth of 240 GB/s. The L links bind thirty-two compute nodes into a supernode.

Every hub chip also has 16 D links that are used to connect to other supernodes with a combined bandwidth of 320 GB/s. The topology maintains at least one D link between every pair of supernodes in the system, although smaller systems can employ multiple D links between supernode pairs.

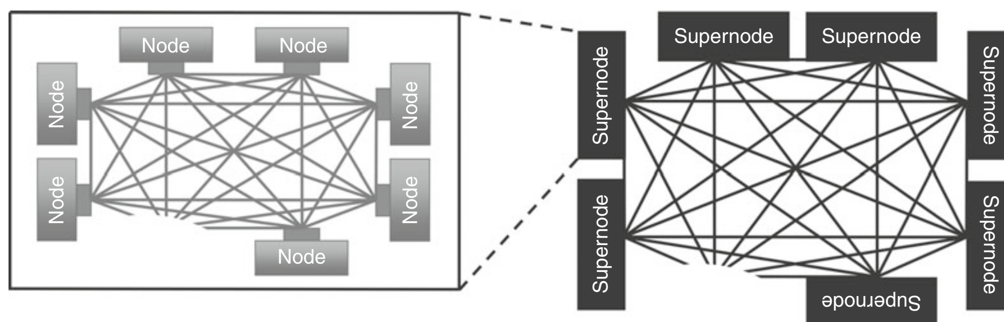
The hub chip is connected to the POWER7 chips in the compute node at a bandwidth of 192 GB/s and has 40 GB/s of bandwidth for general I/O. The peak switching bandwidth of the hub chip exceeds 1.1 GB/s. When all links are populated and operate at peak, the injection bandwidth to network bandwidth ratio is 1:4.6. Note though that by performing the dual roles of data routing and interconnect gateway, the majority of traffic through the hub chip will typically be destined for other compute nodes. The low injection to network bandwidth ratio means that plenty of bandwidth is available for that other traffic.

The topology used by PERCS permits routes to be made up of very small numbers of hops. Within a supernode, any compute node can communicate with any other compute node using a distinct L link. Across supernodes, a compute node has to employ at most one L hop to get to the "right" compute node within its supernode that is connected to the destination supernode (recall that every supernode pair is connected by at least one D link). At the destination supernode, at most one L hop is again sufficient to reach the destination compute node.

Routing Between Nodes

The above-described principles form the basis for direct routing in the PERCS system. A direct route employs a shortest path between any two compute nodes in the system. Since a pair of supernodes can be connected together by more than one D link, there can be multiple shortest paths between a given set of compute nodes. With only two levels in the topology, the longest direct route L-D-L can have at most three hops made up of no more than two L hops and at most one D hop.

PERCS also supports indirect routes to guard against potential interconnect hot spots. An indirect route is one that has an intermediate compute node in



PERCS System Architecture. Fig. 2 Diagram of PERCS interconnect

the route that resides on a different supernode from that of the source and destination compute nodes. An indirect route must employ a shortest path from the source compute node to the intermediate one, and a shortest path from the intermediate compute node to the destination compute node. The longest indirect route L-D-L-D-L can have at most five hops made up of no more than three L hops and at most two D hops. Figure 3 illustrates direct and indirect routing within the PERCS system.

A specific route can be selected in three ways when multiple routes exist between a source-destination pair. First, software can specify the intermediate supernode but let the hardware determine how to route to and then from the intermediate supernode. Second, hardware can select amongst the multiple routes in a round robin manner for both direct and indirect routes. Finally, the hub chip also provides support for route randomization, whereby the hardware can pseudo-randomly pick one of the many possible routes between a source-destination pair.

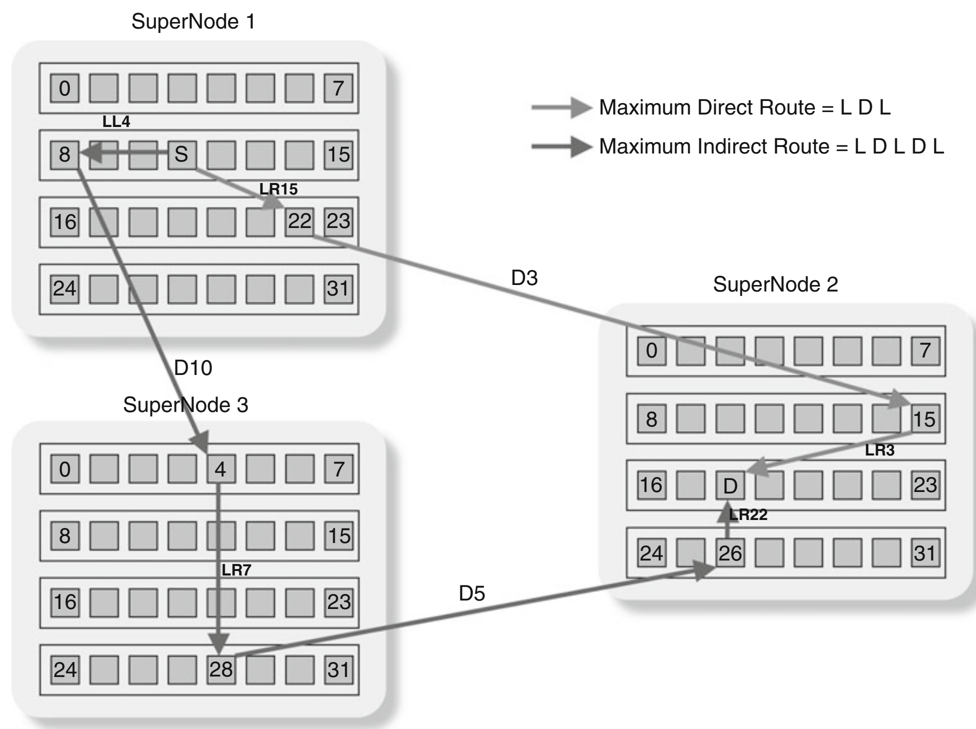
The minimum direct-routed message latency in the system is approximately 1 μ second. The peak unidirectional link bandwidths are as follows: 24 GB/s from each POWER7 chip in an SMP to its hub/switch, 24 GB/s from a given hub/switch to each of the other seven hub/switches in the same drawer (336 GB/s total), 5 GB/s from a given hub/switch to each of the other 24 hub/switches in different drawers in the same supernode (240 GB/s total), and 10 GB/s from a given hub/switch in one supernode to a hub/switch in another supernode to which that particular hub/switch is directly connected (320 GB/s total).

Novel Features of the PERCS Hub Chip

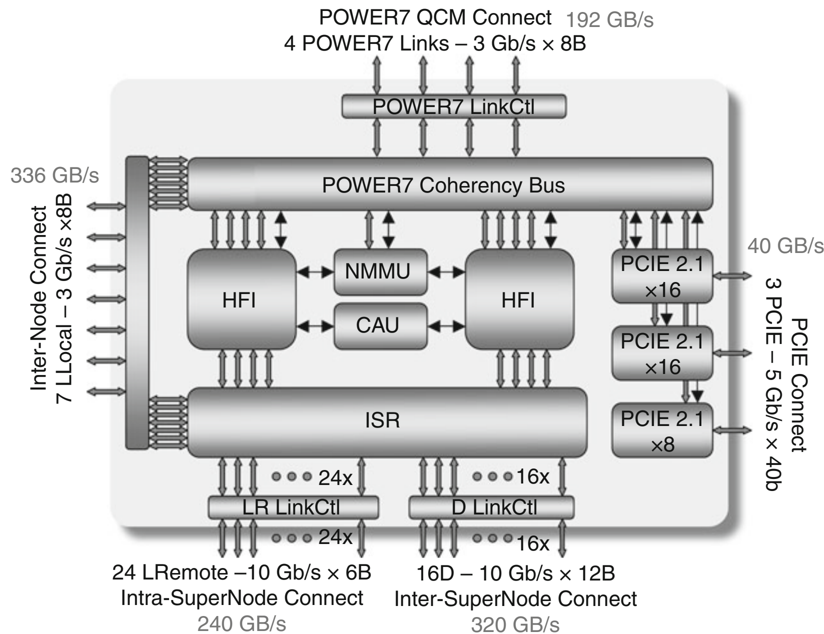
The PERCS Hub Chip, detailed in Fig. 4, incorporates many features to enable the high-performance targets set for the system on HPC applications.

Collective Accelerator Unit (CAU)

Many HPC applications perform collective operations that involve all or a large subset of the nodes



PERCS System Architecture. Fig. 3 Direct and indirect routing in PERCS



PERCS System Architecture. Fig. 4 Details of PERCS Hub chip

participating in the computation. Forward progress can only be realized when each node has completed the collective operation and results have been returned to all participating nodes. The PERCS hub chip provides specialized hardware to accelerate frequently used collective operations such as multicast, barriers, and reduction operations. For reductions, a dedicated arithmetic logic unit (ALU) within the CAU supports the following operations and data types:

- Fixed point: NOP, SUM, MIN, MAX, OR, AND, XOR (signed and unsigned)
- Floating point: MIN, MAX, SUM, PROD (single and double precision)

Software organizes the CAUs in the system into collective trees, firing when data on all of its inputs are available with the result being fed to the next "upstream" CAU in a data-flow manner. Each hub chip contains a single CAU. A multiple-entry content addressable memory (CAM) structure per CAU supports multiple independent trees that can be concurrently used by different applications, for different collective patterns within the same application, or some combination thereof.

Power Bus Interface

The on-chip interconnect for the POWER7 processor is the newly-designed PowerBus architecture. Each hub chip contains a PowerBus interface that enables it to participate in the coherency operations taking place between the four POWER7 chips in the compute node, providing the hub chip visibility to coherence transactions taking place in the node.

Host Fabric Interface

The two Host Fabric Interface (HFI) units in the hub chip manage communication to and from the PERCS interconnect. The HFI was designed to provide user-level access to applications. The basic construct provided by the HFI to applications for delineating different communication contexts is the "window." The HFI supports many hundreds of such windows each with its associated hardware state.

An application invokes the operating system to reserve a window for its use. The reservation procedure maps certain structures of the HFI into the application's address space with window control being possible from that point onward through user-level reads and write to the HFI-mapped structures.

The HFI supports three APIs for communication: a general packet transfer mechanism that can be used for composing either reliable or unreliable protocols upon which to build MPI or other messaging layers; a protocol for global address space operations that allow user-level codes to directly manipulate memory of a task residing on a different compute node; and direct internet protocol transfer ability.

The HFI can extract data that needs to be communicated over the interconnect from either the POWER7 memory or directly from the POWER7 caches. The choice of source is transparent, and data is automatically sourced from the faster location (caches can typically source data faster than memory). In addition to writing network data to memory, the HFI can also inject network data directly into a processor's L3 cache, lowering the data access latency for code executing on that processor [8].

Integrated Switch Router (ISR)

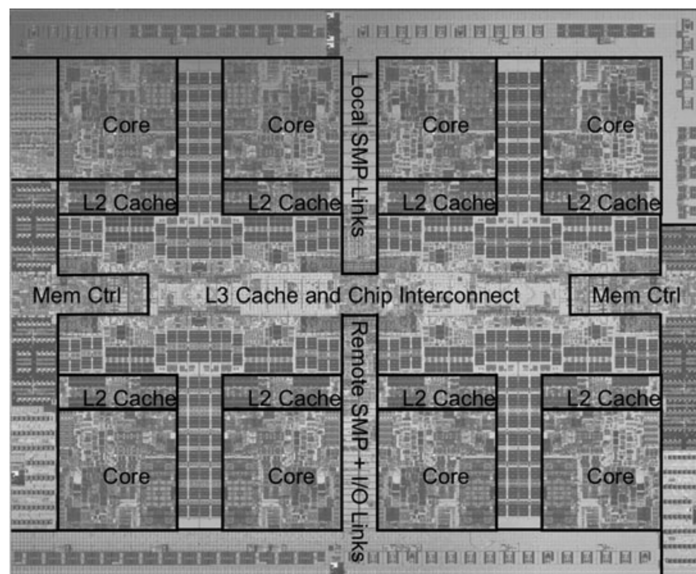
The ISR implements the two-tiered full-graph network utilized in the PERCS system. It is organized as a 56 × 56 full crossbar that operates at up to 3 GHz. In addition to the 47 L and D ports described previously, the ISR also has eight ports to the two local Host Fabric Interfaces, and one service port.

The ISR uses both input and output buffering with a packet replay mechanism to tolerate transient link errors. This feature is especially important since the D links can be several tens of meters in length. The ISR operates in units of 128-byte FLITs with a maximum packet size of 2,048 bytes. Messages are composed of multiple packets with the packets making up a message being potentially delivered out of order.

High-performance computing applications benefit from having access to a single global clock across the entire system. The ISR implements a global clock feature, whereby a clock onboard is globally distributed across the interconnect and kept consistent with the clocks on other Hub chips. Deadlock prevention is achieved through virtual channels, each corresponding to a hop in the L-D-L-D-L worst case route.

POWER7 Processor Overview

PERCS systems use a version of the new POWER7 processor chip fabricated in IBM's 45nm SOI CMOS technology, a die photo of which is shown in Fig. 5. POWER7 incorporates eight high-performance processor cores, L1, L2, and L3 caches per core, on-chip interconnect, multiple I/O controllers, memory controllers, and the SMP fabric controller. The processor design is closely linked with the technology node development, which is co-developed with the high-performance server processors to produce a high-performance server



PERCS System Architecture. Fig. 5 Die photo of the POWER7 processor chip

processor as well as a world class CMOS technology node.

The version of the POWER7 processor chip used in PERCS is fabricated utilizing IBM's 45nm technology at frequencies ranging up to 4 GHz. The chip size is 567 mm^2 and contains eight multithreaded cores. The processor design offers an excellent balance between floating and fixed-point operations, and between computation and memory access. Each core has four double precision floating-point units (*FPU*) implemented as two 2-way SIMD engines (scalar floating point instructions can only use two FPUs), two fixed-point units (*FXU*), two load-store units (LSU), a vector (VMX) unit, a decimal floating point unit, a branch unit, and a condition register. Each FPU is capable of performing one multiply-add instruction (*FMA*) in one cycle, and each LSU can execute simple integer operations. Thus the chip is capable of 256GF/s and 128GOp/s at 4Ghz. The chip also features two memory controllers serving up to eight memory channels, with an aggregate bandwidth of 128GB/s. This offers about 0.5 B/F or 1B/Op greater memory bandwidth balance.

Processor Core

POWER7 implements the 64-bit Power PC-AS architecture and is backward compatible with previous POWER chips, ensuring that existing binaries will run on the new architecture without the need to recompile. IBM's continued focus on single-thread performance in POWER7 remains an important component in its viability in the HPC marketplace. The core uses an extremely power-efficient, high-frequency design with a superscalar, out-of-order execution engine with highly-accurate branch prediction mechanism. Up to eight instructions may be executed in a given cycle.

POWER7 also provides excellent performance for throughput-oriented workloads. Simultaneous multithreading consisting of four hardware contexts per core provides up to 32 different instruction streams per chip for use by software. The multithreaded design ensures that a thread does not block the flow of instructions for other threads through the instruction pipeline, the cache or memory hierarchy, while keeping the area and power overhead low.

Processor Cache Hierarchy

The POWER7 processor chip has three levels of cache, all sharing a common line size of 128 bytes. ECC is used extensively to ensure reliability and data integrity across all cache levels. While the Level 2 (L2) and Level 3 (L3) caches contain both instruction and data, the Level 1 (L1) cache is split into separate instruction and data caches. The L1 instruction cache is 32KB, 4-way set-associative and provides a maximum bandwidth of 64B per processor cycle. The L1 data cache is 32KB, 8-way set-associative and can be accessed in every processor cycle to deliver two 16B load operations, and one 16B store operation.

Each core has a private, 8-way set-associative, 256KB L2 cache. The L2 cache access latency is about seven processor cycles, and in every processor cycle, the processor core can load 32B of data or instructions from L2 and store 16B of data.

Each core has a private 16-way set-associative L3 cache of size 4MB constructed from on-chip embedded DRAM (eDRAM), which offers low power, high density, and excellent access times. The L3 cache access latency is about 22 processor cycles, and in every processor cycle, it can supply 16B of data to the L2 cache and receive 8B of data from the L2 cache. The L3 cache serves as a victim cache of the L2 cache. All L3 caches on a chip may be virtually aggregated into a quasi-shared cache by causing L3s to write back dirty lines to each other before writing them off-chip to memory. A sophisticated multi-tier LRU algorithm maintains balance among this confederation of L3 caches. The L3 cache arrays can also be reconfigured so that two adjacent banks are combined into a larger L3 cache of size 8MB shared between the two corresponding cores, which is useful for commercial workloads. This is another instance of the configurability of the architecture, and how this configurability is used to meet different workload characteristics and enhance the commercial viability of the system as outlined in the vision set forth by DARPA for the HPCS program.

On-chip Integrated Fabric and Chip Interconnect

Another innovative aspect of the POWER7 design is the use of existing system buses to both implement shared-memory traffic and integrate the network switching functionality. Effectively, the POWER7 implements a

distributed switch function within the computer rack. The POWER7 Fabric Bus Controller (FBC) is the building block that implements this distributed switch functionality.

The FBC is integrated on the POWER7 chip and acts as the central point of cache-coherent data traffic. It is responsible for implementing cache-to-cache data transfers, bus arbitration, address routing, etc. It maintains simple and configurable routing tables and implements all necessary flow control to ensure freedom from deadlocks and livelocks. The FBC acts as the hub of all coherent and noncoherent communications among all internal chip units (eight processor cores, two memory controllers, L2 and L3 caches, Non-Cacheable Unit, Gigabit Ethernet, and PCI-Express controller) to other internal units on or outside of the chip. The FBC provides all of the interfaces, buffering, and sequencing of address and data operations within the coherent memory subsystem.

Physically, the Fabric bus is an 8-Byte wide, split-transaction, multiplexed address and data bus. Data packets are four beats each carrying 32 bytes of data. It takes four data packets to transfer the entire 128 Byte cache line. Configuration switches assign node identification for each chip and also for data routing tables. Data transactions within the node are always sent along a unique point-to-point path. A tag travels with the data to help make routing decisions along the way. The Fabric busses provide fully connected topology to reduce latency.

Memory Subsystem

Special care has been taken in the design of the POWER7 chipmemory subsystem to ensure the maximum bandwidth within reasonable constraints of cost and power. The design is flexible and configurable, offering different memory access modes, each optimized to a certain application profile, ranging from sequential access of memory to random access of memory. The memory subsystem also can perform partial cache line loads, improving the efficiency and bandwidth for applications with poor spatial locality such as sparse matrix computations.

The PERCS memory subsystem has two memory controllers with support for up to eight DIMM sockets and uses custom DDR3 DRAM chips running at 1.333 GHz. These DIMMs include specialized buffer chips to increase the amount of memory and bandwidth to memory per chip.

Peak memory bandwidth (read + write) of a single channel is rated at 16GB/s, two thirds of which is for read bandwidth and the remaining one third for write bandwidth. The address space is split across the multiple fully buffered DIMM arrays in a manner that allows for an evenly distributed access pattern across them. This access pattern results in the highest bandwidth available at the processor interface and an evenly distributed thermal pattern.

Blue Waters: The First PERCS Installation

The PERCS system design will first be implemented in the Blue Waters supercomputer to be placed at the National Center for Supercomputing Applications (NCSA) at the University of Illinois [2]. Blue Waters will be comprised of 304 supernodes with nearly 312,000 POWER7 cores, more than 1 PB memory, more than 10 PB disk storage, more than 0.5 EB archival storage, and will achieve close to 10 PF/s peak performance. Acquisition of the Blue Waters system is supported by the National Science Foundation and leverages the investment made by the Defense Advanced Research Projects Agency's High Productivity Computing Systems (HPCS) program.

IBM, the University of Illinois, and the NCSA will work together throughout Blue Waters' lifespan to enhance IBM's high-performance computing environment, ensuring that applications can take full advantage of Blue Waters and achieve performance on a variety of real-world applications. The enhanced high-performance computing environment will also increase the productivity of applications developers, system administrators, and researchers by providing an integrated toolkit for using, analyzing, monitoring, and controlling Blue Waters.

Blue Waters is expected to be one of the most powerful supercomputers in the world when it comes online. It will have a peak performance close to 10 petaflops (10 quadrillion calculations every second) and will achieve sustained performance of 1 petaflop per second running a range of science and engineering codes. Scientists will create breakthroughs in nearly all fields of science using Blue Waters, including predicting the behavior of complex biological systems; understanding how the cosmos

evolved after the Big Bang; designing new materials at the atomic level; predicting the behavior of hurricanes and tornadoes; and simulating complex engineered systems like the power distribution system, airplanes, and automobiles.

Bibliography

1. Arimilli B, Arimilli R, Chung V, Clark S, Denzel W, Drerup B, Hoefler T, Joyner J, Lewis J, Li J, Ni N, Rajamony R (2010) The PERCS high-performance interconnect. Proceedings of Hot Interconnects, Mountain View, CA, August 2010
2. Blue waters project at the national center for supercomputing applications. http://www.ncsa.illinois.edu/BlueWaters/. Accessed January 2010
3. DARPA high productivity computer systems. http://www.highproductivity.org/
4. Dongarra J, Luszczek P, Petitet A (2003) The LINPACK benchmark: past, present, and future. J Concur Comput Pract Exp 15:803–820
5. Duhameland P, Vetterli M (1990) Fast Fourier transforms: a tutorial review and a state of the art. Signal Process 19:259–299
6. Kalla R, Sinharoy B (2009) POWER7: IBM's next generation server processor. IEEE symposium on high-performance chips (Hot chips 21), Stanford, August 2009
7. Luszczek P, Dongarra J, Koester D, Rabenseifner R, Lucas B, Kepner J, Mccalpin J, Bailey D, Takahashi D (2005) Introduction to the HPC challenge Benchmark Suite. http://icl.cs.utk.edu/projectsfiles/hpcc/pubs/hpcc-challenge-benchmark05.pdf. March 2005
8. Milenkovic A, Milutinovic V (2000) Cache injection: a novel technique for tolerating memory latency in bus-based SMPs. EURO-PAR 2000 parallel processing. Lecture notes in computer science, vol 1900/2000, Munich, pp 558–566

Perfect Benchmarks

The Perfect Benchmarks [1] were created in the late 1980s under the leadership of the University of Illinois' Center for Supercomputing Research and Development (CSRD) with the participation of several other institutions (the Perfect Club). The name Perfect is an acronym for PERFormance Evaluation by Cost-effective Transformations. The Benchmarks came from real applications in contrast to the Livermore Loops and other simple codes used in the 1980s to evaluate machines. Thirteen Benchmarks were included: ADM, solves the hydrodynamics equations to simulate air pollution, contributed by IBM Kingston; ARC2D, a finite difference fluid dynamics code developed at NASA Ames; BDNA, a molecular dynamics code for nucleic acid simulation contributed by IBM Kingston; DYFESM, a structural dynamics finite element code contributed by NASA Langley Research Center; FLO52Q, a computation fluid dynamics code contributed by Princeton University; MDG, which uses molecular dynamics to simulate liquid water; MG3D, a signal processing code contributed by Tel Aviv University; OCEAN, a computational fluid dynamics code contributed by Princeton University; QCD, a quantum chromodynamics code contributed by Caltech; SPEC77, a weather simulation code contributed by CSRD; SPICE, a circuit simulator contributed by UC Berkeley; TRACK, which determines the course of a collection of targets from observations taken at regular intervals, contributed by Caltech; and TRFD, a quantum mechanics computation code contributed by IBM Kingston.

Bibliography

1. Berry M, Chen D, Koss P, Kuck D, Lo S, Pang Y, Pointer L, Roloff R, Sameh A, Clementi E, Chin S, Schneider D, Fox G, Messina P, Walker D, Hsiung C, Schwarzmeier J, Lue K, Orszag S, Seidl F, Johnson O, Goodrum R, Martin J (1989) The perfect club Benchmarks: effective performance evaluation of supercomputers. International Journal of Supercomputing Applications 4(3):9–40

Performance Analysis Tools

Michael Gerndt
Technische Universität München, München, Germany

Synonyms

Performance measurement; Profiling; Tracing

Definition

Performance analysis tools support the application developer in tuning the application's performance for a given architecture. They measure performance data during the execution of the application and provide means to analyze and interpret the provided data and to detect performance bottlenecks.

Discussion

Introduction

The development of high-performance applications requires a careful adaptation of the program to the underlying parallel architecture. Due to the manifold interrelations of the parallel program and the architecture, designing an application with optimal performance on parallel systems is almost impossible. Therefore, the application goes through a tuning cycle which consists of measuring performance, detecting performance bottlenecks, and applying program transformations. The assumption for this tuning approach is that the performance will be the same for different runs with the same resources and the same input data.

Performance analysis tools support the programmer in the first two tasks of the tuning cycle. Performance data are gathered during program execution by monitoring the application's execution. Performance data are either summarized and stored as profile data or all the details are stored in so-called trace files.

In addition to application monitoring, performance analysis tools also provide the means to analyze and interpret the provided performance data and thus to detect performance problems. The following sections present the basis of performance analysis, i.e., the execution event model, the different monitoring techniques, and the most important analysis techniques.

Event Model

The abstraction of the execution used in performance analysis is an event model. Events happen at a specific point in time in a process or thread. Events belong to event classes, such as enter and exit events of a user-level function, start and finish events of a send operation in MPI programs, iteration assignment in work-sharing constructs in OpenMP, and cache misses in sequential execution.

In profiling, information about events of the same class is aggregated during runtime. For example, cache misses are counted, the time spent between the start event and finish event of a send operation is accumulated as the communication time of the call site, and the time for taking a scheduling decision in work-sharing loops is accumulated as parallelization overhead.

In tracing, specific information is recorded for each event in a trace file. The event record written to the file at least contains a time stamp and an identification of the executing process or thread. Frequently, additional information is recorded, such as the message receiver and the message length or the scheduling time for an iteration assignment.

Monitoring

The general technique for collecting information about events is called *Monitoring*. Three different monitoring techniques can be distinguished: hardware monitoring, sampling, and instrumentation. These techniques are explained in the next three paragraphs.

Hardware Monitoring

The monitoring of applications should not influence the program's execution. Therefore, hardware monitoring is required for frequent events, such as events in the processor or the caches. If cache misses, e.g., were to be counted via hardware interrupts, the intrusion would be immense. Therefore, current microprocessors provide event counters. A small number of hardware counters can be used to count a large number of different event types. Careful selection of the right event type is thus required.

The low-level APIs for accessing the system's hardware counters are quite different on different architectures. Therefore, standard APIs have been developed. The most notable one is the *Performance Application Programming Interface* (PAPI) [4, 8].

Sampling

Many profiling tools such as the Unix *prof* utility collect statistical performance information via sampling. The processor is interrupted with a certain frequency, known as the sampling rate. The interrupt routine determines the address of the current instruction from the program counter and adds, e.g., the length of the sampling interval to the accumulated execution time of the currently executed source location. Instead of the execution time, other metrics such as the number of cache misses or of floating point operations can be used.

The most important advantage of this technique is that its intrusion is usually low and can be controlled by setting the sampling rate appropriately. The major drawback is that only statistical information is gathered. More precise information can be obtained via instrumentation.

A similar technique is called *Event-Based Sampling*. Each time a hardware counter passes a threshold, an interrupt is generated and information is accumulated for the current source line. The user can control the intrusion of this method by specifying the overflow threshold. A larger threshold will give less accurate information, but is less intrusive. This technique is used, e.g., in the Digital Continuous Profiling Infrastructure (DCPI) [1].

Instrumentation

In contrast to sampling, precise information about the dynamic application behavior can be obtained via program instrumentation. Here the application is modified to gather information at specific events. For example, a call to a monitoring routine is inserted at the beginning and end of a user function to measure the execution time of the function.

Three important instrumentation techniques can be distinguished: source code instrumentation, object code instrumentation, and library interposition. In *source code instrumentation*, the compiler or a source-to-source transformation tool inserts the monitor calls into the program before compilation. In object code instrumentation, the instrumenter patches the machine code.

While the first approach is very portable, *object code instrumentation* is extremely machine specific. It depends not only on the machine's instruction set but also on the compiler, the OS, and the object format. On the other hand, with object code instrumentation, the source language is unimportant and programs can even be instrumented for which no source code is available.

Object code instrumentation can also be applied at runtime. While the program is already executing, instrumentation is inserted only at the required places. This technique was developed within the Paradyn environment and is supported in DYNINST [3, 5] and DPCL [2, 7].

While object code instrumentation is limited to code regions that can be deduced from the binary, it is typically limited to functions or basic blocks. Source code instrumentation can be used to gather information for arbitrary program regions.

Instrumentation of functions can also be done based on a technique called *library interposition*. This technique is used to instrument MPI functions via the MPI profiling interface (PMPI). It determines for all MPI functions a second name via the PMPI prefix. The implementor of an MPI monitoring library can write wrappers for MPI functions and call the PMPI version inside. Within the wrapper, code can be inserted to collect performance information. The wrapper library is then linked to the application before the original MPI library so that the wrappers are called instead of the original functions.

Analysis

The techniques and tools for performance analysis presented in this entry all assume that the performance behavior of the application is the same over multiple experiments. Based on this assumption, multiple experiments can be performed with different tools or different tool configurations to identify and rank performance problems.

The following aspects are relevant for performance analysis tools:

1. Level of detail
2. Performance aspects
3. Application perturbation
4. Level of automation
5. Scalability

Level of Detail

Three classes of performance analysis tools can be distinguished depending on how detailed the raw performance data are gathered and analyzed, i.e., profiles, profile time series, and traces.

Profiling tools aggregate the raw performance data over possibly multiple dimensions. Most common is the aggregation over time, e.g., the number of cache misses for individual functions in each process. Some tools also aggregate the data over processes and threads or even into a single value for the entire execution, e.g., the number of floating point instructions executed by an application. Furthermore, tools apply aggregation with respect to the program regions, i.e., functions and loops. They either compute a flat profile, in which the data are aggregated for each region, or a call path profile, where the data are aggregated for each call path. Call path profiles are very useful if the behavior of functions is different for individual invocations. This is frequently

the case for library routines such as basic mathematical or MPI routines.

Some profiling tools provide not only aggregated data over the entire execution but also time series of snapshots of profile information. Such time series of profile data allow the analysis of time-dependent variations of the performance behavior.

The most detailed analysis is performed by *tracing tools* that store information about individual events in so-called traces. For each event a trace record with a time stamp is generated. A trace record includes additional data, such as the sender and receiver, the amount of data, and an identification of the message sent by an MPI send operation. The trace records are typically stored in trace files that are subsequently analyzed for performance problems.

While profiling tools have the advantage that the amount of performance data is not proportional to the execution time of the application, tracing tools generate performance data proportional to the execution time and the number of processes and threads. On the other hand, profiling tools are limited in the analyses they can perform.

Performance Aspects

Most performance analysis tools are specialized for specific performance aspects. Tuning an application with respect to all possible performance problems probably requires the use of multiple performance analysis tools. The performance tools are typically specialized in one or multiple of the following aspects:

1. Execution time
2. Instruction execution
3. Memory access behavior
4. Memory usage
5. IO
6. Parallel execution

The most basic and thus the most frequently used performance tools provide the user with a time profile. This allows the user to detect the hot code regions, i.e., those regions where most of the execution time is spent and thus have the highest potential for performance improvement.

The next class of tools supports the programmer in the inspection of performance problems related to the utilization of the resources in an individual core. These tools typically provide information about the instruction mix, highlight expensive operations such as divides or type conversions, point out exception handling, or identify pipeline stalls, e.g., due to branch mispredictions. They are based on measurements performed with the processor's hardware performance counters.

Very critical for the performance of applications is the memory access behavior. The long latency of memory accesses and the limited bandwidth require that most of the data accesses are served from the on-chip caches. Programs must be carefully tuned for data locality so that the caches are most effectively used. Performance tools support the analysis of the access behavior with information about the number of cache hits and misses as well as the number of stall cycles for memory references. More precise information can be obtained from cache simulators which are fed with memory reference traces of the application. Cache simulation tools, e.g., can determine the distribution of misses across miss classes, i.e., compulsory, conflict, capacity, and invalidation misses. They can identify false sharing as well as compute higher-level metrics, such as the reuse distance.

Another memory-related analysis supported by performance analysis tools is the inspection of memory usage. Performance problems might arise from allocating too much memory may be due to the size of data structures or to memory leakages.

High-performance applications tend to work on huge data sets. Thus performance tools need to help in the identification of performance problems with respect to file IO. Information can be obtained from runtime libraries implementing IO operations as well as from the OS.

Many performance analysis tools support the detection of performance problems with respect to the parallel execution. Certainly, the two standard programming interfaces MPI and OpenMP have the best support.

Profiling and tracing tools are available for message passing programs that support the analysis of communication among the processes. More elementary tools provide measurements on the time spent in certain MPI functions and the amount of data transferred between communication partners, while advanced tools provide information on different waiting times that indicate certain performance problems, e.g., the late sender problem where the sender of a message arrives later at the

send than the receiver at the receive statement. Tools for message passing also indicate load balancing problems based on the execution time of collective operations.

Analysis tools for shared memory programming, i.e., OpenMP, focus on load imbalances, synchronization, and management overhead. Currently available tools do not yet fully support the extensions of OpenMP 3.0, most notably the tasking concept.

Perturbation

Performance analysis tools are based on measurements. If those measurements are not fully done by additional hardware, the tool will influence or change the codes performance behavior. The intrusion of tools has multiple sources. First of all the time spent in the monitoring library delays the execution. Another important overhead is flushing performance data to external files which needs to be done by tracing tools since the trace records cannot be stored in main memory. These flushes can either be done at global synchronization points and thus delay the execution of all processes, or they can be done asynchronously in the processes which might lead to artificial load imbalance. The intrusion can also be more subtle because memory accesses of the performance analysis tool might change the cache status, or execution delays might even influence the algorithmic behavior in the case of nondeterministic algorithms.

Performance analysis tools try to work around that problem by either correcting the obtained measurements or by reducing the amount of perturbation. Since the first approach is extremely difficult many current tools focus on the second approach.

While the overhead induced by the measurements can be controlled in the sampling approach by selecting an appropriate sampling rate, instrumentation-based measurements can be optimized by reducing the amount of instrumentation and by tuning the instrumentation functions. Instrumentation of program regions that are frequently called but have only little execution time suffer most from the instrumentation overhead. Performance analysis tools provide the means to guide the instrumentation, such as to instrument only certain region types or to exclude individual program regions.

Of course, perturbation not only results from the instrumentation but also from other time-consuming activities of the analysis tools during the program's execution. Examples are the management of trace records and the computation of call path profiles. Various techniques have been developed to tune those operations.

Automation

The ultimate goal of performance analysis tools is to indicate performance problems and thus potential for performance improvement. Ideally, the tools should perform that task in a fully automatic way. Full automation requires the formalization of the performance analysis specialist's and the application specialist's knowledge. Until now, this has only partially been achieved.

One area of automation is program instrumentation. The techniques here range from manual instrumentation to fully automatic instrumentation. Some tools require the user to insert instrumentation into the program. Other tools provide semiautomatic instrumentation via the compiler or a source-to-source instrumenter. The instrumentation can be configured to control the amount of program perturbation. Some tools assist the user in the configuration by pointing out regions that need not be or should not be instrumented based on a previous analysis run. In the fully automatic approach, the instrumentation is selected by the performance analysis tool. Based on the tool's knowledge of which instrumentation is required to obtain the required performance data, the instrumentation is automatically configured.

Automation is also required in the actual measurement of performance data via the monitoring. The analysis tool decides automatically which data will be measured. However for some tools, especially if they access the hardware performance counters, the measurement configuration is done manually. Depending on the type of performance problem the user is interested in, he or she can configure the analysis to measure certain events with the hardware counters. It might even be necessary to perform multiple runs of the application to gather all the required information.

In addition, the last step of the analysis process can be automated. A performance analysis tool should automatically extract and rank the performance problems such that the user can start by tuning the application to solve the most severe performance problem.

In most tools, this step is not automated. The tools allow the user to manually inspect the performance data either via textual or graphical displays. Textual information, i.e., a sorted list of hot regions, is provided with references to the source code or the source code is actually annotated with information such as the number of cache misses. Graphical diagrams are typically used in tracing tools. The dynamic behavior is visualized via timeline diagrams, statistics are shown as bar charts, and many other chart types are utilized to visually represent the huge data sets. It is the task of the user of those tools to select the right displays to identify performance problems.

Only a few tools automate the last step and automatically identify performance problems and their severity. The tools use a formal description of potential performance problems and verify whether those known potential problems can be found based on the measured data.

Fully automatic tools automate the whole process, covering instrumentation, measurement, and analysis. The search for performance problems is driven based on formalized potential performance problems. Following a certain search strategy, they deduce from the performance problem specifications which information is required, measure the data, verify whether a problem exists, and report the found problems to the tool user.

Fully automatic tools can perform the analysis online. Online tools have also been developed for semiautomatic tools that visualize performance data, but those require that the user performs the analysis while the application is executing. This is not appropriate especially since large-scale application runs are only possible in batch jobs. Therefore, most of those tools are offline tools. They produce the data by monitoring an actual run of the application but the inspection and analysis of the data is done independently of the execution.

Scalability

Especially in the field of high-performance computing, performance analysis tools have to be scalable. Applications are run on thousands of processors and the performance behavior cannot be studied for scaled down runs. Tools have to be scalable with respect to the number of processors as well as to the duration of the application's execution. Typical limitations of the tools are the time spent in the analysis as well as the space requirements to store and process performance data. Scalability is obviously a big issue for tracing tools, but profiling-based tools can also easily run into scalability problems if hundreds of thousands of processors are used as in the current BlueGene systems. This problem can be alleviated by processing the data in a tree analysis network (MRNet [15]). The leaves measure the performance data and while the data travel up the tree, they are aggregated to coarsen the information.

Several techniques for tracing tools have been developed to increase scalability. Most important is to reduce the size of the traces. Therefore, compression techniques as well as "on the fly" detection of recurring patterns in traces are applied. A few tools also parallelize the analysis step. SCALASCA post-processes the trace in parallel on the processors of the application after this has terminated. Vampir uses a parallel analysis server that processes the trace files while the user is working with the analysis tool to inspect the measured performance data. Periscope processes the performance data within a hierarchy of analysis agents while the application is executing.

Representative Tools

This section identifies a few performance analysis tools and gives a short characterization. The mentioned tools are only a very small set of examples from the numerous tools developed.

Gprof is the GNU Profiler tool. It provides a flat profile and a callpath profile for the program's functions. The measurements are done by instrumentation.

Vtune is a performance analysis tool from Intel that provides flat and callpath profiles based on sampling and instrumentation, as well as measurements based on the hardware performance counters. The measurements can be inspected on source and assembler level.

pfmon is a tool for accessing the hardware performance counters developed by HP. It can be used to collect certain events specified via command line arguments. Application-specific as well as system-wide measurements can be performed.

ompP is a profiling tool for OpenMP developed at Technische Universität München and University of Tennessee [9]. It is based on instrumentation with Opari

[13] and determines certain overhead categories of parallel regions.

HPC Toolkit from Rice University uses statistical sampling to collect a performance profile [17]. It relates the measurements back to the original source code based on a static analysis of the binary and presents the data by annotating the sources.

Tau is a flexible performance environment from the University of Oregon [16]. It can generate application profiles as well as traces. While it provides tools to analyze the profiles, traces can be analyzed with Vampir.

Vampir is a commercial trace-based performance analysis tool from the Technische Universität Dresden [14]. It provides a powerful visualization of traces and scales to thousands of processors based on a parallel visualization server.

Paraver is a trace visualization and analysis environment from the Barcelona Supercomputing Center. It provides powerful analysis services such as automatic phase detection [6] and clustering.

Paradyn from the University of Wisconsin was the first automatic online analysis tool [12]. Its performance consultant guided the search for performance bottlenecks while the application was executing.

SCALASCA is an automatic performance analysis tool developed at the Forschungszentrum Jülich [10]. It is based on performance profiles as well as on traces. The automatic trace analysis determines MPI wait time via a parallel trace replay after the application execution on the application's processors.

Periscope currently under development at the Technische Universität München follows the approach of Paradyn and performs an online search for performance problems [11]. It is based on a hierarchy of analysis agents to fulfill scalability requirements of current and future HPC systems.

Related Entries

►Intel® Thread Profiler
►OpenMP Profiling with OmpP
►Parallel Tools Platform
►Periscope
►PMPI Tools
►Scalasca
►Tau
►Vampir

Bibliography

1. Digital continuous profiling infrastructure. www.unix.digital.com/dcpi
2. Dynamic probe class library. oss.software.ibm.com/developerworks/opensource/dpcl/
3. Dyninst api. www.dyninst.org
4. Performance application programming interface. icl.cs.utk.edu/papi
5. Buck B, Hollingsworth JK (2000) An API for runtime code patching. Int J High Perform Comput Appl 14(4):317–329
6. Casas M, Badia RM, Labarta J (2007) Automatic phase detection of MPI applications. In: Bischof C et al (eds) Proceedings of the international conference on parallel computing (ParCo '07), Jülich/Aachen. Advances in Parallel Computing, vol 15, IOS, Amsterdam, pp 129–136
7. DeRose L, Hoover T, Hollingsworth JK (2001) The dynamic probe class library – an infrastructure for developing instrumentation for performance tools. IBM, 2001
8. Dongarra J, London K, Moore S, Mucci P, Terpstra D, You H, Zhou M (2003) Experiences and lessons learned with a portable interface to hardware performance counters. In: IPDPS '03: Proceedings of the 17th international symposium on parallel and distributed processing, Washington, DC, IEEE Computer Society, p 289.2
9. Fürlinger K, Gerndt M, Dongarra J (2007) Scalability analysis of the SPEC OpenMP benchmarks on large-scale shared memory multiprocessors. In: Shi Y, van Albada GD, Dongarra J, Sloot PMA (eds) Computational science – ICCS 2007, Beijing. Lecture notes in computer science, vol 4488. Springer, Berlin, pp 815–822
10. Geimer M, Wolf F, Wylie BJN, Mohr B (2006) Scalable parallel trace-based performance analysis. In: Proceedings of the 13th European PVM/MPI users' group meeting on recent advances in parallel virtual machine and message passing interface (EuroPVM/MPI 2006), Bonn, pp 303–312
11. Gerndt M, Ott M (2010) Automatic performance analysis with Periscope. Concurr Comput Pract Exp 22(6):736–748
12. Miller BP, Callaghan MD, Cargille JM, Hollingsworth JK, Irvin RB, Karavanic KL, Kunchithapadam K, Newhall T (1995) The Paradyn parallel performance measurement tool. IEEE Comput 28(11):37–46
13. Mohr B, Malony AD, Shende SS, Wolf F (2001) Towards a performance tool interface for OpenMP: An approach based on directive rewriting. In: Proceedings of the third workshop on OpenMP (EWOMP'01), Barcelona, September 2001
14. Müller MS, Knüpfer A, Jurenz M, Lieber M, Brunst H, Mix H, Nagel WE (2007) Developing scalable applications with Vampir, VampirServer and VampirTrace. In: Bischof C et al (eds) Proceedings of the international conference on parallel computing (ParCo 07), Jülich/Aachen. Advances in Parallel Computing, vol 15, IOS, Amsterdam, pp 113–120
15. Roth PC, Arnold DC, Miller BP (2003) MRNet: A software-based multicast/reduction network for scalable tools. In: Proceedings of the 2003 conference on supercomputing (SC 2003), Phoenix, November 2003

16. Shende SS, Malony AD (2006) The TAU parallel performance system. Int J High Perform Comput Appl, ACTS Collection Special Issue, SAGE, 20(2):287–311
17. Tallent NR, Mellor-Crummey JM, Adhianto L, Fagan MW, Krentel M (2009) Diagnosing performance bottlenecks in emerging petascale applications. In: SC '09: Proceedings of the conference on high performance computing networking, storage and analysis, Portland, ACM, New York, pp 1–11

Performance Measurement

►Performance Analysis Tools

Performance Metrics

►Metrics

Periscope

Michael Gerndt
Technische Universität München, München, Germany

Definition

Periscope is an automatic performance analysis tool for highly parallel applications written in MPI or OpenMP. It performs an online search for performance properties in a distributed fashion. The properties found by Periscope point to areas of parallel programs that might benefit from further tuning.

Discussion

Introduction

Performance analysis is an important step in the development of parallel applications for high-performance architectures. Due to the complexity of the architecture of these machines, applications can typically not be designed to be most efficient. Experiments are required to understand the performance obtained and to identify possible areas in the code that might profit from further tuning.

Performance analysis tools help the developer in analyzing the application's performance. During an execution of the application with a typical input data set and a typical number of processors, performance data are measured. Afterwards those data are analyzed to detect performance problems of the application. Certain tuning actions can then be selected, e.g., selection of different compiler optimization switches or modifications of the source code, based on the detected performance problems.

Periscope is a performance analysis tool developed at Technische Universität München. It is a representative for a class of automatic performance analysis tools automating the whole analysis procedure. Specific for Periscope is that it is an online tool and it works in a distributed fashion. This means that the analysis is done while the application is executing (online) and by a set of analysis agents that are searching for performance problems in a subset of the application's processes (distributed).

Periscope was inspired by the work of the European American APART working group and especially by Paradyn developed at University of Wisconsin, Madison. Paradyn uses a performance consultant that does dynamic instrumentation and searches for bottlenecks based on summary information during the program's execution.

Automation in Periscope is based on formalized performance properties, e.g., inefficient cache usage or load imbalance. Those properties are formalized in the APART Specification Language (ASL). Based on a repository of performance properties the analysis agents can search for those properties in the program execution under investigation. They automatically determine which properties to search for, which measurements are required, which properties were found, and which are more specific properties to look for in the next step. The low-level performance data are analyzed in a distributed fashion and only high-level performance properties are finally reported to the user, significantly reducing the amount of data to be inspected by the user.

Performance Properties

The basis for the analysis is the formalization of performance properties. Each *performance property* consists of a condition, a severity, and a confidence. The *condition* uses measured performance data and static information to decide whether the property holds and thus is a *performance problem*. The *confidence* defines whether

the property is only a hint for a performance problem or a proof. For example, properties based on static information, e.g., the number of prefetch operations, are typically only hints since the actual execution of those operations depends on runtime values. The *severity* states the importance of the performance problem in comparison to the other found performance problems. Typically, the severity is calculated as the amount of time lost due to the problem.

Search Strategies

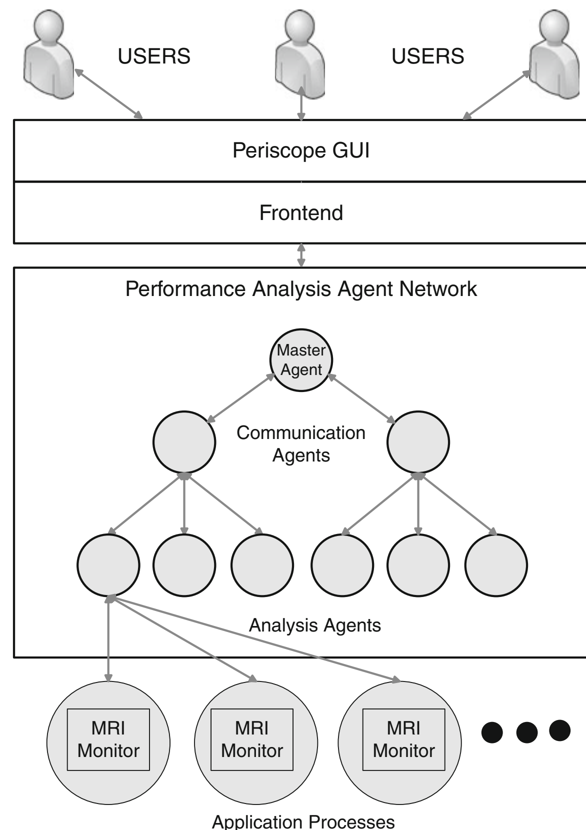
The overall search for performance problems is determined by search strategies. A *search strategy* defines in which order an analysis agent investigates the multidimensional search space of properties, program regions, and processes. Many of Periscope's search strategies are multistep strategies, i.e., they consist of multiple search steps. A very important concept for multistep strategies is a *program phase* which is determined by a certain repetitive program region. A good example for such a phase region is the body of the time loop in scientific simulations where each iteration simulates one time step.

In a first search step, the strategy determines a set of candidate properties, e.g., load imbalance at a barrier in a certain process. The agent then requests certain measurements from the monitor linked into the application. Afterwards the application is released for one execution of the phase. At the end of the phase the execution is stopped again and the measured performance data are returned to the agent. Based on the measurements the agent evaluates the candidate properties and determines the set of found properties. If a refinement is possible, i.e., for the found properties more precise properties are available, a next analysis step is started with a new candidate property set.

Periscope provides search strategies for single-node performance, e.g., searching for inefficient use of the memory hierarchy, MPI, and OpenMP.

Architecture

Periscope consists of an agent hierarchy shown in Fig. 1. The leaves in the hierarchy are the *analysis agents* that perform the actual performance analysis. Each analysis agent is responsible for a subset of the application's processes. It communicates with the monitor linked with the application processes. The MRI monitor serves two purposes: It performs the measurements of performance data requested by the analysis agent and it controls the execution of the process which means, it takes commands from the agent to release or stop the execution.



Periscope. Fig. 1 Architecture of periscope

The root of the hierarchy is the master agent which takes commands from the frontend, propagates the commands to the analysis agents, and provides the list of found performance properties to the frontend. The agents in the middle of the hierarchy are responsible for forwarding information among the analysis agents and the master agent.

The search is controlled by the frontend. It invokes the application and creates the agent hierarchy. How many agents are created depends on the number of application processes and of additional processors provided in the batch job. The processes and agents are mapped to the available processors in a way that communication is local with respect to the physical interconnection network topology.

After the application and the agent hierarchy have been started, the frontend starts the search by propagating a command to all the analysis agents. If an agent requests a new experiment, i.e., an execution of the program phase with measurements to evaluate performance properties, the master agent starts the next experiment via another command. The reason for the global synchronization is that Periscope also supports *automatic restart*. If another experiment is requested but the application terminated, the application can automatically be restarted by the frontend.

On top of the frontend, Periscope provides a graphical user interface based on Eclipse and the Parallel Tools Platform (PTP). The GUI allows the programmer to define a project with all the source files, start a performance analysis via the frontend and, most important, to investigate the performance properties found by Periscope.

Figure 2 illustrates Periscope's GUI. In the lower right a table view visualizes all the properties found in the application. It provides multi-criteria sorting to identify the most severe performance problems, filtering, regular expressions searching, multi-level grouping, as well as clustering. The clustering can be used to identify groups of processes with the same performance problems.

Double clicking on one of the properties will focus on the source code of the respective region in the central source editor. On the right, an outline view of the entire program is provided. It presents an overview of

Periscope. Fig. 2 Inspection of performance properties with the graphical user interface of Periscope

the regions, the nesting of regions, and the number of properties found for a region. Selecting a region in the outline view can be used to enforce that only those properties found for that region are shown in the region view below.

Summary

Periscope is an automatic performance analysis tool for large-scale parallel systems. It performs a distributed online search for performance properties. The properties are formalized and the set of properties can be easily extended. Specialized search strategies determine how the analysis agents walk the multidimensional search space. The result of a search is a set of high-level performance properties that can be inspected via a graphical user interface implemented as an Eclipse plugin. Therefore, the user can follow an integrated approach of program development in Eclipse and performance analysis with Periscope.

Related Entries

►Parallel Tools Platform
►Performance Analysis Tools
►Scalasca
►Tracing

Bibliographic Notes and Further Reading

A recent overview of Periscope can be found in [1]. The steps in applying Periscope to parallel applications as well as results obtained with Periscope are reported in [2]. Details on the graphical user interface can be found in [3]. More information on the formalization of performance properties with the APART Specification Language (ASL) can be found in [4].

Bibliography

1. Gerndt M, Ott M (2010) Automatic performance analysis with Periscope Concurr Comput Pract Exp 22(6):736–748
2. Benedict S, Petkov V, Gerndt M (2009) PERISCOPE: an online-based distributed performance analysis tool. In: 3rd parallel tools workshop, Dresden. Springer, Berlin, pp 1–16, ISBN 978-3-642-11261-4
3. Petkov V, Gerndt M (2010) Integrating parallel application development with, performance analysis in periscope. In: 15th international workshop on high-level progamming models and supportive environments (HIPS 2010), Atlanata, GA, April 19–23. Springer, Berlin, pp 1–8, ISBN 978-1-4244-6533-0
4. Fahringer T, Gerndt M, Riley G, Träff JL (2000) Specification of performance problems in MPI-programs with ASL. In: International conference on parallel processing (ICPP'00), Toronto. IEEE Computer Society, Washington, DC, pp 51–58

Personalized All-to-All Exchange

►All-to-All

Petaflop Barrier

►Roadrunner Project, Los Alamos

Petascale Computer

Petascale computers are those capable of executing at least 10^{15} floating point operations per second. This number of operations per second is known as a petaflop.

Related Entries

►Roadrunner Project, Los Alamos
►Top500

Petri Nets

Jack B. Dennis
Massachusetts Institute of Technology, Cambridge, MA, USA

Synonyms

Place-transition nets

Definition

A *Petri Net* is a graph model for the control behavior of systems exhibiting concurrency in their operation. The graph is bipartite, the two node types being *places* drawn as circles, and *transitions* drawn as bars. The arcs of the graph are directed and run from places to transitions or vice versa. Each place may be empty, or hold a finite number of *tokens*. The state of a Petri net is the distribution of tokens on its places, called a *marking* of the net.

A transition is *enabled* if each of its input places holds at least one token. *Firing* a transition means removing one token from each input place and adding one token to each output place. A *run* of a Petri net is any sequence of firings of enabled transitions; a run defines a sequence of markings. Because many transitions may be enabled in a state, there are often many possible distinct runs of a Petri net. Hence, a Petri net represents a kind of nondeterministic state machine, but in a convenient form for modeling and analyzing concurrent systems. Various extensions and generalizations of Petri nets have been found useful in applications.

Discussion

Introduction

The study of Petri nets began in 1962 with the dissertation of Carl Adam Petri entitled *Communication with Automata*. Petri's work was promoted in the United States by Anatol Holt and Fred Commoner, who introduced the graphical form of Petri nets now widely used. Further development of properties and applications of Petri nets was done at the MIT Laboratory for Computer Science and elsewhere, and promulgated through workshop meetings held in 1970 and 1975. Extensions to Coloured Petri nets, Timed Petri nets, and Continuous Petri nets have been developed for modeling additional properties of concurrent systems. Methods based on Petri nets have become widely used for control aspects of all sorts of concurrent systems, from digital hardware through cooperating computation processes, to diverse applications in areas such as project management and even biological systems.

Definition and Examples

Figure 1 shows a Petri net with three *places* (the circles) and three *transitions* (the bars) interconnected by directed arcs. The graph is bipartite, the two node types being places and transitions. The places may be empty, or may hold any finite number of *tokens*, represented by black dots. The distribution of tokens is called a *marking* of the Petri net and represents its state. Operation of a Petri net produces a sequence of markings by successive applications of the *firing rule*:

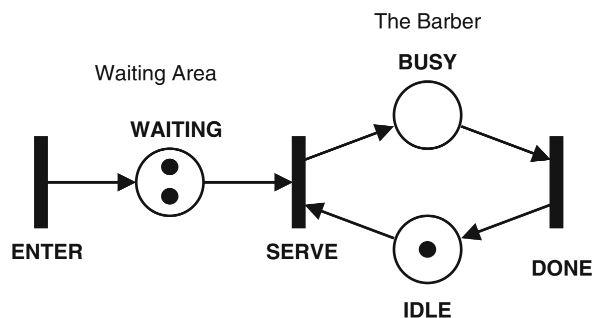
Firing Rule: A transition in a Petri net is *enabled* if each of its input places holds at least one token. *Firing* an enabled transition means removing one token from each of its input places and adding one token to each of its output places.

Note that, because more than one transition may be enabled in a marking of a Petri net, the net is, in general, a nondeterministic system, that is, many distinct firing sequences may exist starting from a given initial marking.

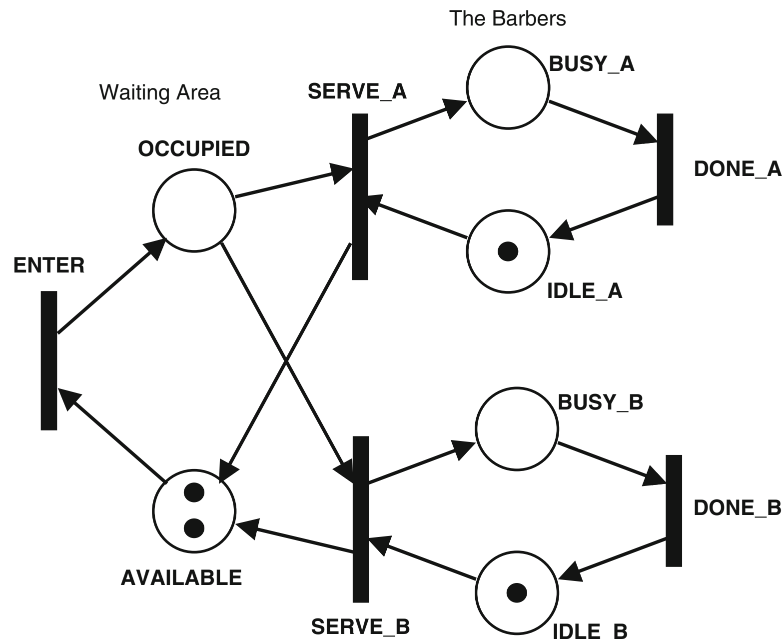
Example: A Barber Shop

Figure 1 is a Petri net model for a barber shop. The net includes a place WAITING that models a waiting area where customers wait for the barber to serve them, and two places that model the barber's status, BUSY or IDLE. The transitions model events that can occur: A firing of transition ENTER corresponds to a customer entering the waiting area of the shop. Transition SERVE corresponds to a customer moving to the barber's chair for service. Transition DONE corresponds to the customer leaving the barber chair after receiving service, with the barber becoming IDLE. This model of the barber shop permits any number of customers to enter the waiting room, but models the constraint that only one customer can occupy the barber chair. This net is an *unbounded* Petri net because the number of tokens in the WAITING place can increase indefinitely.

Figure 2 shows a model of a barber shop with two additional features: there are now two barbers and the waiting area has bounded capacity. Place AVAILABLE models the number of empty spaces for customers in the waiting area and place OCCUPIED models the occupied spaces. This addition turns the model into a *bounded*



Petri Nets. Fig. 1 The barber shop with one barber. Places are shown as circles; transitions are shown as bars. Two tokens represent customers in the waiting area. A third token represents the waiting state of the barber



Petri Nets. Fig. 2 The barber shop with two barbers. Either barber may serve a waiting customer

Petri net. Note that there are now two transitions that model a customer moving to a barber chair. These transitions share an input place and are said to be in *conflict*. This structure represents the nondeterminacy of deciding which barber serves the next customer in the case that both barbers are idle. A Petri net in which there can be no conflict is *determinate*, that is, its behavior is such that all firing sequences are equivalent, even though operation is nondeterministic.

If one wishes more detail about the discipline of the waiting area, it can be modeled as a FIFO queue, as shown in Fig. 3. This net contains as many stages as desired (the capacity of the queue), each stage modeling a single queued item (waiting customer).

The modeling techniques shown by the barber shop example illustrate application of Petri nets to such systems as pipelined processors and production systems for manufacturing or business workflow. Petri nets lack means to model the timing of actions represented by transition firings, but extensions have been developed to remedy this limitation (See below).

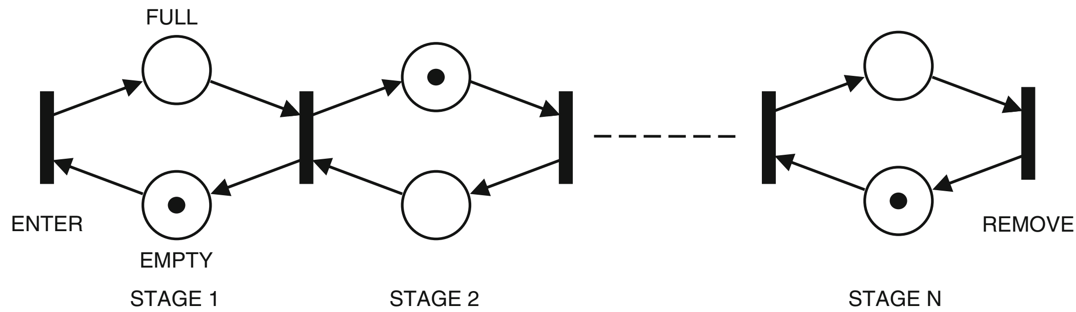
Petri Nets and Finite State Machines

The Finite State Machine (FSM) model, a well-known tool used in the design and analysis of switching circuits and other engineering artifacts, lacks the ability provided by Petri nets to capture the spatial structure of systems in a way that can aid study and analysis. This is illustrated by the FIFO queue of Fig. 3. By means of 2N places and N transitions, a queue of length N may be represented, including the occupancy state of each position in the queue. An equivalent FSM description of the queue would have 2^N states. This difference is a strong argument in favor of using Petri nets in system design and analysis.

Petri nets can exhibit some interesting properties. A Petri net with a marking may have no enabled transitions; no firings are possible and the net is in *deadlock*. Also, as seen in Fig. 1, a Petri net can have firing sequences in which the number of tokens in some places increases without limit. Two definitions restrict Petri nets to subclasses most useful for modeling certain kinds of realistic systems:

Liveness: A Petri net with marking M is *live* if and only if given any marking M′ reachable from the given marking M, and any transition T of the net, there exists a firing sequence starting from M′ that fires transition T.

Safety: A Petri net is *N-safe* for a given marking if and only if no marking reachable from the given marking has more than *N* tokens in any place. A net and marking that is *1-safe* is said to be *safe*.



Petri Nets. Fig. 3 Model of a FIFO queue with stage 2 occupied

Note that the definition of live is formulated to rule out nets containing an initialization sequence that is never executed again in steady-state operation of the net.

A Petri net cannot be both live and *N*-safe for a marking unless the net is connected, that is, the net contains a directed path between any pair of nodes. The nets of Fig. 1 and Fig. 2 are both live as marked, but only Fig. 2 is safe because the WAITING place in Fig. 1 may accumulate tokens without limit. The class of live and safe Petri nets is especially interesting because these properties are characteristic of real systems that continue in operation indefinitely and are implementable with finite hardware.

Conflict and Determinacy

Conflict in a Petri net occurs if, in some reachable marking, two transitions are both enabled and share an input place. A Petri net and marking for which no conflict is possible is *determinate*. The net in Fig. 1 has no conflict and is determinate; the net of Fig. 2 is not determinate because transitions SERVE_A and SERVE_B may be in conflict, representing the choice of barber in the case that both servers are idle.

The Petri Net Hierarchy

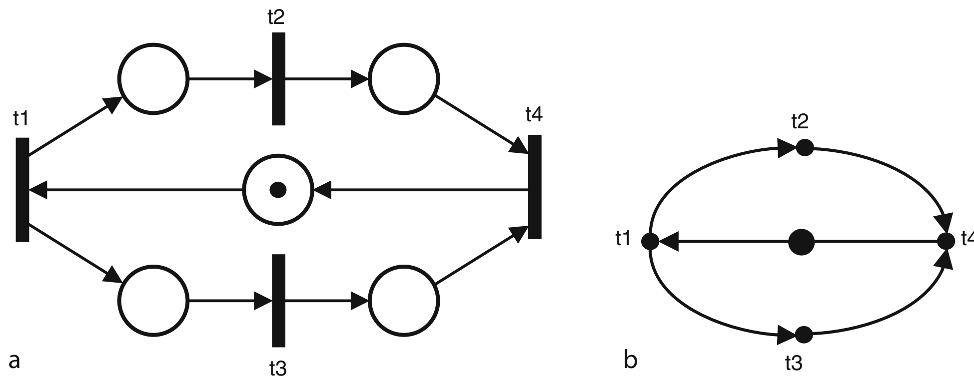
A convenient hierarchy of classes of Petri nets having increasing modeling power may be specified by syntactic constraints on the structure of a net. The simplest of these subclasses is the *marked graphs*, which are Petri nets in which each place is an input place of exactly one transition and an output place of exactly one transition. These nets describe systems whose behavior is an endless repetition of a pattern of events. The name marked graphs is given to these nets because they may be drawn with each place, together with its input arc and output arc, represented as a simple arc, as shown in Fig. 4b for the net of Fig. 4a. Transitions are drawn as dots instead of bars, and a marking is shown by placing tokens on the arcs. A marked graph with an initial marking that is live must be connected, and can be shown to be N-safe for some N.

Another simple class of Petri nets is the *state machines*, which are nets in which each transition has exactly one input and one output place. A state machine with a one-token marking corresponds to an FSM with as many states as the net has places. A Petri net can be both a marked graph and a state machine. Such a net is very simple and consists of a set of disconnected cycles of alternating transitions and places.

Both marked graphs and state machines are determinate because conflict is impossible.

Free choice Petri nets constitute the next interesting level toward greater expressivity. Free choice nets permit a constrained form of conflict to be represented, corresponding to decisions in a digital system, or to choices based on predicates in a system description. A Petri net is a *free choice* net if and only if for each place P that is an input place of two transitions t1 and t2; place P is the only input place of t1 and t2. This ensures that in any marking in which P holds at least one token, it is guaranteed that a free choice is available – both t1 and t2 are enabled and either may fire, disabling the other. Figure 1b is not a free choice net because the decision is conditioned by a barber being available. Free choice Petri nets that have conflict are not determinate for any live marking.

A Petri net is a *simple Petri net* if and only if it has no transition with more than one shared input place. Simple nets can represent systems in which arbitration occurs among several activities competing for shared resources, behavior not representable with free choice nets.



Petri Nets. Fig. 4 A Petri net (**a**) and its representation as a marked graph (**b**)

The subclasses of Petri nets form a hierarchy of strictly increasing expressive power:

Marked Graphs State Machines
Free Choice Nets
Simple Nets
All Basic Petri Nets

Extensions

The basic Petri nets described above have found uses in describing digital systems, modeling process synchronization schemes using Dijkstra's P and V operations, and monitors, and for the control structures underlying dataflow graphs. Methods have been developed for converting Petri nets into logic designs. Outside the realm of computer science, Petri nets have found application in modeling manufacturing systems, among many other uses.

On the other hand, basic Petri nets are cumbersome and limited for application to many important problems in the analysis of concurrent systems such as performance analysis and job/work flow management. To make the formalism more generally useful, several extensions of basic Petri nets have been developed, and a rich body of formal analysis and application for these extensions has evolved. The most important extensions are Coloured Petri Nets, Timed Petri Nets, and Continuous/Hybrid Petri Nets.

Coloured Petri Nets

In a Coloured Petri Net, each token may be labeled with a color from an arbitrary set of colors. The use of colored tokens permits more compact representations of large and complex systems. If transitions are permitted to map the colors of input tokens to new colors of output tokens, a powerful model of arbitrary parallel computations is realized. If the set of colors is finite, then any colored Petri net has an equivalent basic Petri net, albeit possibly very large. Thus, the coloring of tokens does not increase the expressive power of basic nets. However, adding hierarchy to the colored Petri nets permits recursive computation to be directly represented.

Timed Petri Nets

One important extension is the addition of explicit timing. One form of this extension is to associate a fixed time with each transition. The interpretation is that when an enabled transition fires, tokens are removed from its input places; then, at the specified later time, tokens are added to the output places of the transition. In this interpretation, a transition may have several instances of operation progressing simultaneously. This may be seen as a generalization of the Program Evaluation and Review Technique (PERT) of critical path planning for project scheduling. Association of time specifications with places and with arcs has also been studied. Analysis methods have been developed for determining bounds on the overall execution time of timed Petri nets. Another variation associates a probability distribution of operation times with each transition, leading to a generalization of queuing networks. Timed Petri nets have significant applications in performance analysis of computer systems, workflow analysis and manufacturing systems.

Continuous and Hybrid Petri Nets

Another direction in extension of Petri nets is motivated by applications to industrial control and manufacturing

systems. In a *continuous Petri net*, tokens carry values that are nonnegative real numbers and are thought of as comprising an infinite number of *marks*. Transitions are permitted to fire continuously so long as each input place holds a positive, nonzero value. In such a net, a place can model, for example, the amount of liquid in a tank, the number parts in a bin, or the rate of traffic flow on a highway. In a hybrid Petri net, two kinds of places are distinguished: *continuous* places that hold real values, and discrete places that hold a nonnegative, integral number of tokens as in basic Petri nets. Transitions are also of two kinds. By means of a transition that has both a continuous place and a discrete place as inputs, turning a flow on and off can be modeled.

Many variations and extensions of the Petri net model have arisen in the years since the early developments. The principal record of these developments has been the series of conference proceedings published by Springer as *Lecture Notes in Computer Science* (LNCS) under the title Applications and Theory of Petri Nets.

Related Entries

►CSP (Communicating Sequential Processes)
►Data Flow Graphs
►Synchronization

Bibliographic Notes and Further Reading

The study of Petri nets began with the doctoral dissertation of Carl Adam Petri [7]. The graphical expression of Petri nets now widely used was presented by Holt and Commoner [3]. The MIT Project MAC report in which this paper is published includes annotated references to significant prior work in the field. The book by James Peterson [6] was the first exposition of the theory and applications of Petri nets in book form, and includes a good summary of the subject's early development and its literature. The survey paper of Murata [5] is an excellent summary of work on basic Petri nets through the 1980s. Timed Petri nets were first studied by Ramchandani [10] in 1973 and many versions of Petri nets dealing with system timing have been studied since [1]. The three volumes by Jensen provide a comprehensive treatment of "colored" Petri nets [4]. Continuous and hybrid Petri nets are treated by David and Alla [1], where a review of basic net theory and extensive bibliographic notes may be found. The principal current record of advances and applications in Petri nets is the series of Conferences now titled "Applications and Theory of Petri Nets" [11]. An interesting web site is maintained by Carl Adam Petri and Wolfgang Reisig [8], and these two authors are the contributors of the Petri net entry in scholarpedia [9].

Bibliography

1. David R, Alla H (2005) Discrete, continuous, and hybrid Petri Nets. Springer, Berlin
2. Girault C, Valk R (2003) Petri Nets for systems engineering. Springer, New York
3. Holt A, Commoner F (1970) Events and conditions. In: Record of the project MAC conference on concurrent systems and parallel computation. ACM, New York, pp 3–52
4. Jensen K (1995, 1996) Coloured Petri Nets: basic concepts, analysis methods and practical use. Monographs in theoretical computer science, vol 1, 2. Springer, Berlin
5. Murata T (Apr 1989) Petri nets: properties, analysis and applications. Proc IEEE 77(4):541–580
6. Peterson JL (1981) Petri Net theory and the modeling of systems. Prentice-Hall, Englewood Cliffs
7. Petri CA (1962) Kommunikation mit automaten. Schriften des Institutes für Instrumentelle mathematik. Ph.D. dissertation, University of Bonn, Germany
8. Petri CA. http://www.informatik.uni-hamburg.de/TGI/mitarbeiter/profs/petri_eng.html
9. Petri CA, Reisig W (2008) Petri net. Scholarpedia 3(4):6477. http://www.scholarpedia.org/article/Petri_net
10. Ramchandani C (Feb 1974) Analysis of asynchronous concurrent systems by Petri Nets. Technical Report MIT/LCS/TR-120, MIT Laboratory for Computer Science
11. Springer (1980) Application and theory of Petri Nets. In: Informatik fachberichte; (1984–1993) Advances in Petri Nets. Lecture notes in computer science, vol 10; (1992–2009) Applications and theory of Petri Nets. Lecture notes in computer science, vol 18. Springer, Berlin

PETSc (Portable, Extensible Toolkit for Scientific Computation)

BARRY SMITH
Argonne National Laboratory, Argonne, IL, USA

Definition

The Portable, Extensible Toolkit for Scientific computation (PETSc, pronounced PET-see) is a suite of

open source software libraries for the parallel solution of linear and nonlinear algebraic equations. PETSc uses the Message Passing Interface (MPI) for all of its parallelism.

Discussion

The numerical solution of linear systems with sparse matrix representations is at the heart of many numerical simulations, from brain surgery to rocket science. These linear systems arise from the replacement of continuum partial differential equation (PDE) models with suitable discrete models by the use of the finite element, finite volume, finite difference, collocation, or spectral methods and then possibly linearization by a fully implicit strategy such as Newton's method or semi-implicit techniques. The resulting linear systems can range from having a few thousand unknowns to billions of unknowns, thus requiring the largest parallel computers currently available. Large-scale linear systems also arise directly in optimization, economics modeling, and many other non-PDE-based models. The main focus of PETSc is in solving linear systems arising from PDE-based models, though it is applied to other problems as well. PETSc also has limited support for dense matrix computations (through an interface to LAPACK and PLAPACK); but if the computation involves exclusively dense matrices, then PLAPACK or ScaLAPACK are appropriate libraries.

PETSc is a software library intended for use by mathematicians, scientists, and engineers with a solid understanding of programming, some basic understanding of the issues in parallel computing (though they need not have programmed in MPI), a basic understanding of numerical analysis, and an understanding of the basics of linear algebra. It has a higher and deeper learning curve than do software packages such as Matlab. PETSc can be used directly from Fortran 77/90, C/C++, and Python, with bindings that are specialized to each language.

PETSc is used by a variety of parallel PDE solver libraries, including freeCFD, a general-purpose CFD solver; OpenFVM, a finite-volume-based CFD solver; OOFEM, an object-oriented finite element library; libMesh, an adaptive finite element library; and DEAL.II, a sophisticated C++- based finite element simulation package. Magpar is a widely used, parallel micromagnetics package written using PETSc. For large-scale optimization and the scalable computation of eigenvalues, PETSc has two companion packages, TAO and SLEPc, developed by other groups, that use all of the PETSc parallelism and linear solver infrastructure.

The emphasis of the PETSc solvers is on iterative methods for the solution of linear systems, but it provides its own efficient sequential direct (LU and Cholesky factorization-based) solvers as well as interfaces to several parallel direct solvers; see Table 1. PETSc has a unique configuration system that will automatically download and install the multitude of optional packages that it can use. In addition to the direct solvers, it can use several parallel partitioning packages as well as preconditioners in the hypre and TRILINOS solver packages; see Table 2. A key design feature of PETSc is the composibility of its linear solvers. Two or more solvers may be combined in various ways: by splittings, multigrid, and Schur complementing to produce efficient, problem-specific solvers.

The parallelism in PETSc is usually achieved by domain decomposition. The geometry on which the PDE is being solved is divided among the processes,

PETSc (Portable, Extensible Toolkit for Scientific) Computation. Table 1 Partial list of direct solvers available in PETSc

Factorization	Package	Complex numbers support	Parallel support
LU	PETSc	x	
	SuperLU	x	
	SuperLU_Dist	x	x
	MUMPS	x	x
	Spooles	x	x
	PaStiX	x	x
	IBM's ESSL		
	UMFPACK		
	LUSOL		
Cholesky	PETSc	x	
	Spooles	x	x
	MUMPS	x	x
	PaStiX	x	x
	DSCPACK	x	

PETSc (Portable, Extensible Toolkit for Scientific) Computation. Table 2 Partial list of preconditioners available in PETSc

Preconditioner	Package	Complex numbers support	Parallel support
ICC(k)	PETSc	x	
ILU(k)	PETSc	x	
	Euclid/hypre		x
ILUdt	pilut/hypre		x
Jacobi	PETSc	x	x
SOR	PETSc	x	
Block Jacobi	PETSc	x	x
Additive Schwarz	PETSc	x	x
Geometric multigrid	PETSc	x	x
Algebraic multigrid	BoomerAMG/hypre		x
	ML/TRILINOS		x
Approximate inverse	SPAI		x
	Parasails/hypre		x

and each process is assigned the unknowns and matrix elements associated with that domain. The communication required during the solution process is then nearest neighbor ghost (halo) point updates and global reductions (using **MPI_Allreduce()**) over a MPI communicator. PETSc has optimized code based on the inspector-executor model to perform the ghost point updates.

In addition to its broad support for linear solvers, PETSc provides robust implementations of Newton's method for nonlinear systems of equations. These include a variety of line-search and trust-region schemes for globalization. The solvers are extensible, allowing easy provision of user-provided convergence tests, line-search strategies, and damping strategies. Several variants of the Eisenstat–Walker convergence criteria for inexact Newton solves are available. There is also support for grid sequencing to efficiently generate high-quality initial solutions for fine grids. To compute the Jacobians commonly needed for Newton's method, PETSc provides coloring of sparse matrices and efficient computation of the Jacobian entries using the coloring with finite differencing, ADIC (automatic differentiation for C programs), and ADIFOR (automatic differentiation for Fortran 77 programs). All of these run scalably in parallel.

PETSc also provides a family of implicit and explicit ODE integrators, including an extensive suite of explicit Runge–Kutta methods. The implicit methods support all the functionality of the PETSc nonlinear solvers and use of any of the Krylov methods and preconditioners. The more sophisticated adaptive time-stepping ODE integrators of SUNDIALS can also be used with PETSc and allow use of all PETSc preconditioners.

Provided in PETSc is an infrastructure for profiling the parallel performance of the application and the solvers it uses, including floating-point operations done, messages, and sizes of messages sent and received. It provides the results in a table that indicates the percentage of time spent in the various parts of the solver and application.

Development of PETSc was started in 1995 by Bill Gropp, Lois Curfman McInnes, and Barry Smith at Argonne National Laboratory. They were joined shortly later by Satish Balay. Aside from a small amount of National Science Foundation funding in the mid-1990s, the US Department of Energy has provided the funding for PETSc development and support. Since its origin, PETSc has received software contributions from many of its users.

PETSc was the winner of a 2009 R&D 100 award. It also has formed the basis of three Gordon Bell Prize application codes in 1999, 2003, and 2004 as well as several Gordon Bell finalists.

Library Design

PETSc follows the distributed-memory single program multiple data (SPMD) model of MPI, with the flexibility of having different types of computation running on different processes. Specifically PETSc allows users to create their own MPI communicators and designate computations for PETSc to perform on each of these communicators. A typical application code written with PETSc requires very few MPI calls by the developer.

PETSc is written in C using the object-oriented programming techniques of data encapsulation, polymorphism, and inheritance. Opaque objects are defined that contain function tables (using C function pointers) used to call the code appropriate for the underlying

data structures. The six main abstract classes in PETSc are the **Vec** vector class for managing the system solutions, the **Mat** matrix class for managing the sparse matrices, the **KSP** Krylov solver class for managing the iterative accelerators, the **PC** preconditioner class, the **SNES** nonlinear solver class, and the **TS** ordinary differential equations (ODE) integrator class. The **DM** helper class manages transferring information about grids and discretizations into the **Vec** and **Mat** classes. Virtually all of the parallel communication required by PETSc (the MPI message passing and collective calls) takes place within these objects. The constructor for each PETSc object takes an MPI communicator, which determines on what processes the object and its computations will reside. The most common are MPI_COMM_WORLD, in which the object is distributed across all the user's processes (and computations involving the object will require communication within that communicator), and MPI_COMM_SELF, in which the object lives on just that process and no communication is ever required for its computations.

A typical application that requires linear solvers has a structure as depicted in Fig. 1. In this example, the **DA** object, which is an implementation of the **DM** class for structured grids, is used to construct the needed sparse matrix and vectors to contain the solution and right-hand side; it serves as a factory for **Vec** and **Mat** objects. Once the numerical values of the matrix are set, in this case by calls to **MatSetValues()**, the matrix is provided to the linear solver via **KSPSetOperators()**. Since **MatSetValues()** may be called with values that belong to any process, the calls to **MatAssemblyBegin/End()** are used to communicate the values to the process where they belong. Values may be set into vectors either by using **VecSetValues()**, with a concluding **VecAssemblyBegin/End()**, as with matrices, or by accessing the array of values using **VecGetArray()**, **VecGetArrayF90()**, or **DAVecGetArray()** and putting values directly into the array. In this latter case, no communication of off-process values is done by PETSc.

A typical application that requires nonlinear solvers has a structure as depicted in Fig. 2. In addition to serving as a factory for the Jacobian sparse matrix and solution vector (as in the linear case), the **DA** object is used as a factory for the ghosted representation of the solution **xlocal** and performs the ghost point updates with **DAGlobalToLocal()** in the routines **ComputeFunction()** and **ComputeJacobian()**. These call-back routines are registered with the nonlinear solver object **SNES** with the routines **SNESSetFunction()** and **SNESSetJacobian()**. They are called when needed by the solver class.

A typical application that requires ODE integration has a structure as depicted in Fig. 3. This simple example uses the Python interface to **TS** where the entire discretized ODE (in this case using the backward Euler method) is provided directly as the function and Jacobian. It is also possible to provide the function and Jacobian of the right-hand side of the ODE, that is, $u_t = F(u)$, and have the **TS** class manage the ODE discretization, with either an explicit or an implicit scheme.

Each PETSc object has a method **XXXSetFromOptions()** that allows runtime control of almost all of the solver options through the PETSc options database. Command-line arguments (as keyword value pairs) are stored in a simple database. The **XXXSetFromOptions()** routines then search the options database, select any appropriate options, and apply them. For example, the option **-ksp_type gmres** is used by **KSPSetFromOptions()** to call **KSPSetType()** to set the solver type to GMRES. The options database may also be used directly by user code.

Also common to all classes are the **XXXView()** methods. These provide a common interface to printing and saving information about any object to a **PetscView** object, which is an abstract representation of a binary file, a text file (like **stdout**), a graphical window for drawing, or a Unix socket. For example, **MatView(Mat A,PetscViewer v)** will present the matrix in a wide variety of ways depending on the viewer type and its state. Calling the viewer method on a solver class, such as **SNES**, displays the type of solver and all its options; see Fig. 4 for an example. Note that the figure displays both the nonlinear and linear solver options.

For the **Mat** class, PETSc provides several realizations. The most important of these are the following:

- Compressed sparse row (CSR) format
- Point-block version of the CSR where a single index is used for small dense blocks of the matrix
- Symmetric version of the point-block CSR that requires roughly one-half the storage
- User-provided format (via inheritance)

P

```
      program main   !   Solves the linear system  J x = f
#include "finclude/petscalldef.h"
      use petscksp; use petscda
      Vec x,f; Mat J; DA da; KSP ksp; PetscErrorCode ierr
      call PetscInitialize(PETSC_NULL_CHARACTER,ierr)

      call DACreate1d(MPI_COMM_WORLD,DA_NONPERIODIC,8,1,1,PETSC_NULL_INTEGER,da,ierr)
      call DACreateGlobalVector(da,x,ierr); call VecDuplicate(x,f,ierr)
      call DAGetMatrix(da,MATAIJ,J,ierr)

      call ComputeRHS(da,f,ierr)
      call ComputeMatrix(da,J,ierr)

      call KSPCreate(MPI_COMM_WORLD,ksp,ierr)
      call KSPSetOperators(ksp,J,J,SAME_NONZERO_PATTERN,ierr)
      call KSPSetFromOptions(ksp,ierr)
      call KSPSolve(ksp,f,x,ierr)

      call MatDestroy(J,ierr); call VecDestroy(x,ierr); call VecDestroy(f,ierr)
      call KSPDestroy(ksp,ierr); call DADestroy(da,ierr)
      call PetscFinalize(ierr)
      end
      subroutine  ComputeRHS(da,x,ierr)
#include "finclude/petscalldef.h"
      use petscda
      DA da; Vec x;PetscErrorCode ierr;PetscInt xs,xm,i,mx; PetscScalar hx;PetscScalar, pointer::xx(:)
      call DAGetInfo(da,PETSC_NULL_INTEGER,mx,PETSC_NULL_INTEGER,PETSC_NULL_INTEGER,PETSC_NULL_INTEGER,...
      call DAGetCorners(da,xs,PETSC_NULL_INTEGER,PETSC_NULL_INTEGER,xm,PETSC_NULL_INTEGER,...
      hx     = 1.d0/(mx-1)
      call VecGetArrayF90(x,xx,ierr)
      do i=xs,xs+xm-1
         xx(i) = i*hx
      enddo
      call VecRestoreArrayF90(x,xx,ierr)
      return
      end
      subroutine ComputeMatrix(da,J,ierr)
#include "finclude/petscalldef.h"
      use petscda
      Mat J; DA da; PetscErrorCode ierr; PetscInt xs,xm,i,mx; PetscScalar hx
      call DAGetInfo(da,PETSC_NULL_INTEGER,mx,PETSC_NULL_INTEGER,PETSC_NULL_INTEGER,PETSC_NULL_INTEGER,...
      call DAGetCorners(da,xs,PETSC_NULL_INTEGER,PETSC_NULL_INTEGER,xm,PETSC_NULL_INTEGER,...
      hx     = 1.d0/(mx-1)
      do i=xs,xs+xm-1
        if ((i .eq. 0) .or. (i .eq. mx-1)) then
          call MatSetValue(J,i,i,1d0,INSERT_VALUES,ierr)
        else
          call MatSetValue(J,i,i-1,-hx,INSERT_VALUES,ierr)
          call MatSetValue(J,i,i+1,-hx,INSERT_VALUES,ierr)
          call MatSetValue(J,i,i,2*hx,INSERT_VALUES,ierr)
        endif
      enddo
      call MatAssemblyBegin(J,MAT_FINAL_ASSEMBLY,ierr); call MatAssemblyEnd(J,MAT_FINAL_ASSEMBLY,ierr)
      return
      end
```

PETSc (Portable, Extensible Toolkit for Scientific Computation). Fig. 1 Example of linear solver usage in PETSc in Fortran 90

- "Matrix-free" representations, where the matrix entries are not explicitly stored, but instead matrix-vector products are performed by using one of the following:
  - Finite differencing of the function evaluations
  - Automatic differentiation of the function evaluations using either ADIC, for C language code, or ADIFOR, for Fortran 77 language code
  - User-provided C, C++, Fortran, or Python routine

```
static char help[] = "Solves -Laplacian u - exp(u) = 0,  0 < x < 1\n\n";
#include "petscda.h"
#include "petscsnes.h"

int main(int argc,char **argv) {
  SNES snes; Vec x,f; Mat J; DA da;
  PetscInitialize(&argc,&argv,(char *)0,help);

  DACreate1d(PETSC_COMM_WORLD,DA_NONPERIODIC,8,1,1,PETSC_NULL,&da);
  DACreateGlobalVector(da,&x); VecDuplicate(x,&f);
  DAGetMatrix(da,MATAIJ,&J);

  SNESCreate(PETSC_COMM_WORLD,&snes);
  SNESSetFunction(snes,f,ComputeFunction,da);
  SNESSetJacobian(snes,J,J,ComputeJacobian,da);
  SNESSetFromOptions(snes);
  SNESSolve(snes,PETSC_NULL,x);

  MatDestroy(J); VecDestroy(x); VecDestroy(f); SNESDestroy(snes); DADestroy(da);
  PetscFinalize();
  return 0;}
PetscErrorCode ComputeFunction(SNES snes,Vec x,Vec f,void *ctx) {
  PetscInt i,Mx,xs,xm; PetscScalar *xx,*ff,hx; DA da = (DA) ctx; Vec xlocal;
  DAGetInfo(da,PETSC_IGNORE,&Mx,PETSC_IGNORE,PETSC_IGNORE,PETSC_IGNORE,PETSC_IGNORE,PETSC_IGNORE,...
  hx     = 1.0/(PetscReal)(Mx-1);
  DAGetLocalVector(da,&xlocal);DAGlobalToLocalBegin(da,x,INSERT_VALUES,xlocal);DAGlobalToLocalEnd(da,x,...
  DAVecGetArray(da,xlocal,&xx); DAVecGetArray(da,f,&ff);
  DAGetCorners(da,&xs,PETSC_NULL,PETSC_NULL,&xm,PETSC_NULL,PETSC_NULL);

  for (i=xs; i<xs+xm; i++) {
    if (i == 0 || i == Mx-1) ff[i] = xx[i]/hx;
    else  ff[i] =  (2.0*xx[i] - xx[i-1] - xx[i+1])/hx - hx*PetscExpScalar(xx[i]);
  }
  DAVecRestoreArray(da,xlocal,&xx); DARestoreLocalVector(da,&xlocal);DAVecRestoreArray(da,f,&ff);
  return 0;}
PetscErrorCode ComputeJacobian(SNES snes,Vec x,Mat *J,Mat *B,MatStructure *flag,void *ctx){
  DA da = (DA) ctx; PetscInt i,Mx,xm,xs; PetscScalar hx,*xx; Vec xlocal;
  DAGetInfo(da,PETSC_IGNORE,&Mx,PETSC_IGNORE,PETSC_IGNORE,PETSC_IGNORE,PETSC_IGNORE,PETSC_IGNORE,...
  hx = 1.0/(PetscReal)(Mx-1);
  DAGetLocalVector(da,&xlocal);DAGlobalToLocalBegin(da,x,INSERT_VALUES,xlocal);DAGlobalToLocalEnd(da,x,...
  DAVecGetArray(da,xlocal,&xx);
  DAGetCorners(da,&xs,PETSC_NULL,PETSC_NULL,&xm,PETSC_NULL,PETSC_NULL);

  for (i=xs; i<xs+xm; i++) {
    if (i == 0 || i == Mx-1) { MatSetValue(*J,i,i,1.0/hx,INSERT_VALUES);}
    else {
      MatSetValue(*J,i,i-1,-1.0/hx,INSERT_VALUES);
      MatSetValue(*J,i,i,2.0/hx - hx*PetscExpScalar(xx[i]),INSERT_VALUES);
      MatSetValue(*J,i,i+1,-1.0/hx,INSERT_VALUES);
    }
  }
  MatAssemblyBegin(*J,MAT_FINAL_ASSEMBLY);MatAssemblyEnd(*J,MAT_FINAL_ASSEMBLY);*flag = SAME_NONZERO_...
  DAVecRestoreArray(da,xlocal,&xx);DARestoreLocalVector(da,&xlocal);
  return 0;}
```

PETSc (Portable, Extensible Toolkit for Scientific Computation). Fig. 2 Example of nonlinear solver usage in PETSc in C

Because PETSc is focused on PDE problems, row-based storage of the sparse matrices (each process holds a collection of contiguous rows of the matrix) is satisfactory for higher-performance parallel matrix operations. Hence, all of PETSc's built-in sparse matrix implementations use this approach. Custom formats can be provided to handle parallelism for "arrow-head" matrices where row-based distribution does not scale.

PETSc has the point-block-based storage of sparse matrices for faster performance. The speed of sparse matrix computations is essentially always strongly limited by the memory bandwidth of the system, not by the CPU speed. The reason is that sparse matrix

```
import sys, petsc4py
petsc4py.init(sys.argv)
from petsc4py import PETSc
import math

class MyODE:
  def __init__(self,da):
    self.da = da
  def function(self, ts,t,x,f):
    mx = da.getSizes(); mx = mx[0]; hx = 1.0/mx
    (xs,xm) = da.getCorners(); xs = xs[0]; xm = xm[0]
    xx = da.createLocalVector()
    da.globalToLocal(x,xx)
    dt = ts.getTimeStep()
    x0 = ts.getSolution()
    if xs == 0: f[0] = xx[0]/hx; xs = 1;
    if xs+xm >= mx: f[mx-1] = xx[xm-(xs==1)]/hx; xm = xm-(xs==1);
    for i in range(xs,xs+xm-1):
      f[i] = (xx[i-xs+1]-x0[i])/dt + (2.0*xx[i-xs+1]-xx[i-xs]-xx[i-xs+2])/hx - hx*math.exp(xx[i-xs+1])
    f.assemble()
  def jacobian(self,ts,t,x,J,P):
    mx = da.getSizes(); mx = mx[0]; hx = 1.0/mx
    (xs,xm) = da.getCorners(); xs = xs[0]; xm = xm[0]
    xx = da.createLocalVector()
    da.globalToLocal(x,xx)
    x0 = ts.getSolution()
    dt = ts.getTimeStep()
    P.zeroEntries()
    if xs == 0: P.setValues([0],[0],1.0/hx); xs = 1;
    if xs+xm >= mx: P.setValues([mx-1],[mx-1],1.0/hx); xm = xm-(xs==1);
    for i in range(xs,xs+xm-1):
      P.setValues([i],[i-1,i,i+1],[-1.0/hx,1.0/dt+2.0/hx-hx*math.exp(xx[i-xs+1]),-1.0/hx])
    P.assemble()
    return True # same_nz

da = PETSc.DA().create([9],comm=PETSc.COMM_WORLD)
f = da.createGlobalVector()
x = f.duplicate()
J = da.getMatrix(PETSc.MatType.AIJ);

ts = PETSc.TS().create(PETSc.COMM_WORLD)
ts.setProblemType(PETSc.TS.ProblemType.NONLINEAR)
ts.setType('python')

ode = MyODE(da)
ts.setFunction(ode.function, f)
ts.setJacobian(ode.jacobian, J, J)

ts.setTimeStep(0.1)
ts.setDuration(10, 1.0)
ts.setFromOptions()
x.set(1.0)
ts.solve(x)
```

PETSc (Portable, Extensible Toolkit for Scientific Computation). Fig. 3 Example of ODE usage in PETSc in Python

computations involve few operations per matrix entry. For example, for matrix-vector products there are two floating-point operations (a multiply and an addition) for each entry in the matrix. Memory-bandwidth-limited computations are sometimes said to hit the memory wall. In the CSR format, there is a column index for every nonzero entry in the matrix, and the matrix-vector product is coded as $y[i] = \sum_{j=nz_{i-1}}^{j<nz_i} aa[j] * x[aj[j]]$. For each multiply in the computation, a double-precision value of $aa[]$ must be loaded as well as an integer value $aj[]$. Thus, 12 bytes are loaded per multiply. In the point-block CSR format (with block

```
SNES Object:
  type: ls
    line search variant: SNESLineSearchCubic
    alpha=0.0001, maxstep=1e+08, minlambda=1e-12
  maximum iterations=50, maximum function evaluations=10000
  tolerances: relative=1e-08, absolute=1e-50, solution=1e-08
  KSP Object:
    type: fgmres
      GMRES: restart=30, using Classical (unmodified) Gram-Schmidt Orthogonalization
      GMRES: happy breakdown tolerance 1e-30
    maximum iterations=10000, initial guess is zero
    tolerances:  relative=1e-05, absolute=1e-50, divergence=10000
    right preconditioning
    using UNPRECONDITIONED norm type for convergence test
  PC Object:
    type: mg
      MG: type is FULL, levels=2 cycles=v
    Coarse grid solver -- level 0 presmooths=1 postsmooths=1 -----
      KSP Object:(mg_coarse_)
        type: preonly
      PC Object:(mg_coarse_)
        type: lu
          LU: out-of-place factorization
            matrix ordering: nd
          LU: tolerance for zero pivot 1e-12
          LU: factor fill ratio needed 1.875
        Matrix Object:
          type=seqaij, rows=64, cols=64
          total: nonzeros=1024, allocated nonzeros=1024
            using I-node routines: found 16 nodes, limit used is 5
    Down solver (pre-smoother) on level 1 smooths=1 --------------------
      KSP Object:(mg_levels_1_)
        type: gmres
          GMRES: restart=30, using Classical (unmodified) Gram-Schmidt Orthogonalization
          GMRES: happy breakdown tolerance 1e-30
        maximum iterations=1
        tolerances:  relative=1e-05, absolute=1e-50, divergence=10000
        left preconditioning
        using nonzero initial guess
        using PRECONDITIONED norm type for convergence test
      PC Object:(mg_levels_1_)
        type: ilu
          ILU: 0 levels of fill
          ILU: factor fill ratio allocated 1
          ILU: tolerance for zero pivot 1e-12
        Matrix Object:
          type=seqaij, rows=196, cols=196
          total: nonzeros=3472, allocated nonzeros=3472
            using I-node routines: found 49 nodes, limit used is 5
    Matrix Object:
      type=seqaij, rows=196, cols=196
      total: nonzeros=3472, allocated nonzeros=3472
        using I-node routines: found 49 nodes, limit used is 5
```

PETSc (Portable, Extensible Toolkit for Scientific Computation). Fig. 4 Example of output using **SNESView()**

size bs), there is one column index per block, and the matrix-vector product may be coded as $y[bs * i + k] = \sum_{j=nz_{i-1}}^{j<nz_i} \sum_{l=0}^{l<bs} aa[bs * (j + l) + k] * x[aj[j] + l]$. Here, for every $(bs * bs)$ multiplies, $(bs * bs)$ loads of $aa[]$ are needed, but only a single integer $aj[]$. For even moderate block size, this approach reduces the loads per multiply from 12 to less than 8.5 bytes. In addition, the same $x[]$ values are used repeatedly for each k, and a smart unrolling can keep the reused values in registers. Using the block CSR when appropriate, depending on the particular processor, can improve the performance of the sparse matrix operators by a factor of 2 to 3.

The **KSP** Krylov accelerator class provides over a dozen Krylov methods; see Table 3. The data encapsulation and polymorphic design of the **Vec**, **Mat**, and **PC** classes in PETSc allow the immediate use of any of their

PETSc (Portable, Extensible Toolkit for Scientific Computation). Table 3 Partial list of Krylov methods available in PETSc

Richardson (simple) iteration, $x^{n+1} = x^n + B(b - Ax^n)$
Chebychev iteration
Conjugate gradient method
Biconjugate gradient
Biconjugate gradient stabilized (bi-CG-stab)
Conjugate residuals
Conjugate gradient squared
Minimum residuals (MINRES)
Generalized minimal residual (GMRES)
Flexible GMRES (fGMRES)
Transpose-free quasi-minimal residuals (QMR)

implementations with any of the Krylov solvers. When possible, these are implemented to allow left, right, or symmetric preconditioning and the use of various norms of the residual in the convergence tests including the "natural" (energy) norm. Custom convergence tests and monitoring routines can be provided to any of the solvers.

The **PC** preconditioners class contains a variety of both classical and modern preconditioners including incomplete factorizations, domain decomposition methods, and multigrid methods. See Table 2 for a partial list. In addition, several preconditioner classes are designed to allow composition of solvers. These include **PCKSP**, which allows using a Krylov method as a preconditioner; **PCFieldSplit**, which allows constructing solvers by composing solvers for different fields of the solution; **KSPCOMPOSITE**, which allows combining arbitrary solvers; and **PCGALERKIN**, which constructs preconditioners by the Galerkin process (that is, as projections of the error in some appropriate inner product). **PCFieldSplit** preconditioners are often called block preconditioners; for example, when one field is velocity and another pressure, the resulting Stokes solver is often solved with one block for velocity and one for pressure.

Applications

A wide variety of simulation applications have been written by using PETSc. These include fluid flow for aircraft, ship, and automobile design; blood flow simulation for medical device design; porous media flow for oil reservoir simulation for energy development and groundwater contamination modeling; modeling of materials properties; economic modeling; structural mechanics for construction design; combustion modeling; and nuclear fission and fusion for energy development.

PETSc-FUN3d was an early application based on Kyle Anderson's NASA code, FUN3d, that solves the Euler and Navier–Stokes equations including both compressible and incompressible on unstructured grids. PETSc-Fun3d won a Gordon Bell special prize in 1999 running on over 6,000 of the ASCI Red processors. This application, the dissertation work of Dinesh Kaushik, motivated many of the early optimizations of PETSc.

The forward and inverse modeling of earthquakes using the PETSc algebraic solvers, developed by Volkan Akcelik, Jacobo Bielak, George Biros, Ioannis Epanomeritakis, Antonio Fernandez Omar Ghattas, Eui Joong Kim, David O'Hallaron, and Tiankai Tu, resulted in a 2003 Gordon Bell special prize.

The algebraic multigrid solver Prometheus was written by Mark Adams using the PETSc **Vec**, **Mat**, **KSP**, and **PC** classes. It takes advantage of the block CSR sparse format in PETSc to maximize performance. It was to simulate whole-bone micromechanics with over half a billion degrees of freedom, resulting in a 2004 Gordon Bell special prize.

PETSc has been used by several research groups to simulate heart arrhythmias, which are the cause of the majority of sudden cardiac deaths. These applications involve solving the nonlinear bidomain equations, which are two coupled partial differential equations that model the intracellular and extracellular potential of the heart. Numerical solutions to these equations (and more sophisticated models) explain much of the electrical behavior of the heart, including defibrillation.

PFLOTRAN, led by Peter Lichtner of Los Alamos National Laboratory, is a subsurface flow and contaminant transport simulator that uses the PETSc **DM** class to manage the parallelism of its mesh, the **SNES** nonlinear solver class for the solutions needed at each time step, the **Mat** class to contain the sparse Jacobians, and the **Vec** class for its flow and contaminant's solutions. It has been run on up to 64,000 cores of the Cray XT5

and has been used to more accurately model uranium plumes at DOE's Hanford site.

The UNIC neutronics package developed by Mike Smith and Dinesh Kaushik of Argonne National Laboratory has run full reactor core simulations on 290,000 cores of the IBM Blue Gene/P. It supports both the second-order Pn and Sn methods with dozens of energy groups. It parallelizes simultaneously over the geometry by means of domain decomposition and angles using a hierarchy of MPI communicators and PETSc solver objects.

Related Entries

►Algebraic Multigrid
►BLAS (Basic Linear Algebra Subprograms)
►Chaco
►Distributed-Memory Multiprocessor
►Domain Decomposition
►LAPACK
►Memory Wall
►METIS and ParMETIS
►MPI (Message Passing Interface)
►PLAPACK
►Scalability
►ScaLAPACK
►SPAI (SParse Approximate Inverse)
►SPMD Computational Model
►SuperLU

Bibliographic Notes and Further Reading

The PETSc web site is the best location for up-to-date information on PETSc [8]. A complete list of external packages that PETSc can use is given in [5]. More details of the applications developed by using PETSc can be found at [7]. Further details on the design decisions made in PETSc may be found in [2].

Other related parallel solver packages include TRILINOS [9], hypre [10], and SUNDIALS [11]. TRILINOS is a large, general-purpose solver package much in the spirit of PETSc and written largely in C++; it currently has little support for use from Fortran. The hypre package specializes in high-performance preconditioners and includes a scalable algebraic multigrid solver BoomerAMG. SUNDIALS specializes in nonlinear solvers and adaptive ODE integrators; it expects the required linear solver to be provided by the user or another package. Many of the solvers in these other packages can be called through PETSc.

Bibliography

1. Balay S, Buschelman K, Eijkhout V, Gropp WD, Kaushik D, Knepley MG, McInnes LC, Smith BF, Zhang H (2008) PETSc Users Manual, Argonne National Laboratory Technical Report ANL0-95/11 - Revision 3.0.0
2. Balay S, Gropp WD, McInnes LC, Smith BF (1997) Efficient management of parallelism in object oriented numerical software libraries. In: Arge E, Bruaset AM, Langtangen HP (eds) Modern software tools in scientific computing. Birkhauser Press, Boston, pp 163–202
3. Balay S, Gropp WD, McInnes LC, Smith BF (2002) Software for the scalable solution of PDEs. In: Dongarra J, Foster I, Fox G, Gropp B, Kennedy K, Torczon L, White A (eds) CRPC handbook of parallel computing. Morgan Kaufmann Publishers
4. J Dongarra's freely available software for linear algebra. http://www.netlib.org/utk/people/JackDongarra/la-sw.html
5. List of external software packages available from PETSc. http://www.mcs.anl.gov/petsc/petsc-as/miscellaneous/external.html
6. Saad Y (2003) Iterative methods for sparse linear systems, 2nd edn. SIAM
7. Partial list of applications written using PETSc. http://www.mcs.anl.gov/petsc/petsc-as/publications/petscapps.html
8. PETSc's webpage. http://www.mcs.anl.gov/petsc
9. TRILINOS's webpage. http://trilinos.sandia.gov
10. Hypre's webpage. https://computation.llnl.gov/casc/linear_solvers/sls_hypre.html
11. SUNDIAL's webpage. https://computation.llnl.gov/casc/sundials/main.html

PGAS (Partitioned Global Address Space) Languages

George Almasi
IBM, Yorktown Heights, NY, USA

Definition

PGAS (**P**artitioned **G**lobal **A**ddress **S**pace) is a programming model suited for shared and distributed memory parallel machines, e.g., machines consisting of many (up to hundreds of thousands of) CPUs.

P

Shared memory in this context means that the total of the memory space is available to every processor in the system (although access time to different banks of this memory can be different on each processor). Distributed memory is scattered across processors; access to other processors' memory is usually through a network.

A PGAS system, therefore, consists of the following components:

- A set of processors, each with attached local storage. Parts of this local storage can be declared *private* by the programming model, and is not visible to other processors.
- A mechanism by which at least a part of each processor's storage can be *shared* with others. Sharing can be implemented through the network device with system software support, or through hardware shared memory with cache coherence. This, of course, can result in large variations of memory access latency (typically, a few orders of magnitude) depending on the location and the underlying access method to a particular address.
- Every shared memory location has an *affinity* – a processor on which the location is local and therefore access is quick. Affinity is exposed to the programmer in order to facilitate performance and scalability stemming from "owner compute" strategies.

All PGAS programming languages contain the components enumerated above, although the ways in which these are made available to the programmer differ. Every PGAS language allows programmers to distinguish between private and shared memory locations, and to determine the affinity of shared memory locations. Some PGAS languages provide work distribution primitives such as parallel loops based on affinity, or program syntax to allow special handling of remote (and therefore, slow) data accesses. The rest of this entry expands some of these differences between PGAS languages.

Discussion

Introduction

There exist a variety of choices for PGAS languages and implementations. Some of these choices are about the ubiquity of shared memory, the method of accessing remote memory, or the choice of a parent programming language. Consequently there is a wide variety of PGAS-like languages and libraries:

- UPC [9] (Unified Parallel C) is a language descended from C. It extends C arrays and pointers with shared arrays and shared pointers that address into global memory. UPC also features a `forall` loop that distributes iterations based on affinity of array elements.
- Coarray Fortran [5] is a Fortran-based language that extends Fortran arrays with co-dimensions that allow accessing arrays on other processes (called images). A variant of Coarray Fortran is included in the Fortran 2008 standard, making it the only PGAS language with ISO approval.
- Split-C [8] is a C-based PGAS language that acknowledges the latency of remote memory accesses by allowing split-phase, or non-blocking, transactions. This allows overlapping of remote accesses with computation, hiding latency.
- Titanium [11] is a Java-based PGAS language. Titanium features SPMD parallelism, pointers to shared data and an advanced distributed array model.
- ZPL [1] is an array-based language featuring the global view programming model.
- Chapel [2] is Cray Inc's flagship modern programming language. It incorporates elements of ZPL but also features the multiresolution paradigm, allowing users to bore down to performance from an initial high-level program.
- X10 [10] is a PGAS language that provides task parallelism as well as data parallelism. The key feature of X10 is asynchronous task dispatching.
- HPF [3] (High-Performance Fortran) is an early attempt to solidify concepts from global view array programming in a Fortran-based language. It is one of the bases from which the PGAS concepts grew.
- MPI [7] (Message Passing Interface) is the de facto standard for high-performance parallel programming. It does not implement the PGAS programming model, since it does not have the concept of global memory: All inter-processor data exchange is explicit. However, MPI contains many ideas and concepts relevant to PGAS and that makes it worth mentioning in this context.

- OpenMP [6] is a cross-language standard for shared-memory programming used widely in the high-performance computing world. The standard allows loops to be annotated as executed in parallel, and variables as shared or private; the newer standard has task-parallel features as well. OpenMP is in a similar situation to MPI: not a PGAS language, but containing many relevant concepts.
- Global Arrays [4] is a library or parallel array computing. It provides an abstraction of a shared array but is backed by distributed memory. Actual memory operations are implemented by a one-sided messaging library called ARMCI.
- HTAs (Hierarchical Tiled Arrays) are another library-based approach, providing the user with an array abstraction embedded into the multiple levels of a distributed system's memory hierarchy. HTAs can be laid out to reflect this hierarchy: levels of cache, shared memory with affinity to particular processors, and of course nonlocal memory accessed (under the covers) by messaging.

Local Versus Shared Memory

While all PGAS languages distinguish between local, shared local and shared remote memory. However, the default assignment of memory to the local versus shared space greatly varies across the space of languages.

All memory in MPI (the Message Passing Interface standard) is local, and the only way to convey information to another process space is through messages. In contrast all memory in OpenMP (a GAS programming paradigm) is global, and the only way to make memory locations safe from other threads is to explicitly denote it as thread private. In UPC, Coarray Fortran and Split-C memory is declared as private by default, and has to be made global with an explicit declaration modifier. In Titanium program stacks are thread-private, but the heap is shared by default. In the array language ZPL and in the HTA library all arrays are shared by default. In X10, memory is local and only accessible by sending units of work ("asyncs") to the remote locations to execute.

Computation and Address Spaces

Parallelism implies multiple processing units executing a particular program. However, the relationship between executing programs and address spaces differs across programming languages. In UPC and Coarray Fortran address, spaces are tightly bound to computation. Execution units are called threads in UPC; Address affinity is calculated relative to UPC threads. Titanium calls the execution units processes, and locality is bound to these implicitly. By contrast, in Coarray Fortran it is the address spaces themselves that are named – images – and the implication is that each image has computation executing on it. X10 completely separates the notions of address space and computation. Every address space is called a place, and multiple computational threads called activities are allowed to execute simultaneously, subject to the capability of the hardware.

Messaging

The PGAS programming model does not make any representation about the mechanics of accessing data in nonlocal address spaces. On distributed-memory hardware data exchange is done by exchanging messages across any network devices are available on the hardware in question; PGAS programming models are implemented on top of a messaging system.

The preferred messaging system for PGAS implementations is *one sided*: That is, one of the participants is active and is responsible for specifying all parameters of the exchange (identities of sender, receiver, addresses on both ends, amount of data), while the other participant is passive and contributes nothing but the data itself.

Active messages are also used preferentially by PGAS languages. Active messages vary from one-sided messages in that the passive participant is called upon to execute user code as part of receiving the message.

Every PGAS language makes a choice as to what extent language syntax hides the underlying messaging system. In Split-C messages look like assignments, and provisions are made to hide the large latency of such messages. In UPC, local and shared assignments have the same syntax, making the indistinguishable; however, the programmer is allowed to write explicit one-sided messages into the program. Even third-party messages are allowed (e.g., UPC thread A specifying a data transfer between threads B and C). Coarray Fortran and Chapel do not allow explicit messaging. X10 exposes messaging to the programmer in the form of asyncs which are very close in concept to active messages.

References to Remote Memory

Just as in the C language arrays and pointers are two sides of the same coin, in PGAS languages there is a close relationship between arrays and references in global address space especially in those languages rooted in C syntax, like Split-C and UPC. The syntax and semantics of references to remote memory, including pointer arithmetic, tends to follow that of normal pointers. The unique features of remote pointer access revolve around hiding of access latency. Remote accesses tend to be orders of magnitude slower than local ones. The increased latency can be partially mitigated by posting remote operations as soon as the initial conditions are met, e.g., both source data and destination buffers are ready for transfer. However, the operation need not be complete until the data is actually needed on the destination end. To implement this, Split-C features the split assignment operator, and the Berkeley UPC extensions (not part of the UPC standard) allow non-blocking remote memory operations.

Array Programming and Implicit Parallelism

Array programming is a generic term describing a programming environment suitable for the processing of n-dimensional arrays. In these environments arrays are first-class citizens, allowing compact declaration and operators (unlike in conventional imperative programming languages where arrays are handled by loops). Some examples of array programming languages/environments are APL, Fortran 90, MATLAB, and R.

The attraction of array languages is their ability to express operations on large amounts of data with few instructions. This has many benefits, including efficient programming of vector processors (e.g., Intel SSE3, IBM Altivec) and graphics processors (NVIDIA GPUs), but array languages also lend themselves to explicit SPMD parallelism with the PGAS programming model. The programmer specifies the layout of array elements in distributed memory. The compiler and/or the runtime optimize array operations by staying as close as possible to the owner compute rule, i.e., scheduling computation on the CPUs closest to each array element. The execution model of pure array languages is SIMD; conceptually there is a single thread of control acting on a large amount of data. PGAS languages have borrowed heavily from the array processing paradigm. The Fortran D and High-Performance Fortran (HPF) languages allow users to specify data layouts with the TEMPLATE and DISTRIBUTE commands. An HPF template declares a processor layout (and hence the structure of the partitioned address space). Global arrays are distributed across this template. A large set of intrinsic operators allow the concise expression of operations like shifting/transposing/summing up array slices.

In Coarray Fortran, array data distribution takes the form of a co-dimension. Vector indexing in the Fortran 90 style is permitted. The Chapel and ZPL languages offer a refinement of the Fortran 90/Matlab vector syntax by means of regions, or named subsets/slices of arrays: shifts, reductions, dimensional floods (i.e., broadcasts), boundary exchanges can be expressed this way. The partitioned global address space is set up by means of distributions, an analogue of HPF templates. The Titanium programming language also follows this approach.

Less conventional runtime-only approaches include the Hierarchical Tiled Arrays (HTAs) library a pure runtime solution that provides multiple levels of data decomposition, one for each level of non-locality in a modern computer architecture. The Global Arrays toolkit also allows programmers to specify and optimiz their own array layouts. The Matlab Parallel Toolbox uses the `spmd` keyword and specialized array distribution syntax to control data parallel execution.

There is a natural affinity between array processing and parallelism. By putting arrays into global memory one transcends the memory limitations of any single CPU, while still allowing for quick access to the array from anywhere. Array operations are generally floating-point intensive, and therefore natural candidates for parallelization. The large number of operations causes more granular computation, resulting in less parallelism overhead and therefore fewer losses to Amdahl's law.

Well-known parallel algorithms exist for many array operators. Some of these algorithms have good scaling properties (i.e., low cross-CPU communication requirements, good load balance) and can be coded into the supporting runtime system or even the compiler, allowing the programmer instant access to high-performance parallel array operations.

Parallel Loops and Explicit Data Parallelism

The parallel loop construct is an established way of expressing explicit parallelism; Fortran's `DOALL` statement is one of the oldest such constructs. The essence of the construct is to divide the iteration space of a loop nest among processors, either statically or dynamically. OpenMP in particular is known for a wide variety of parallel loop options.

Several PGAS languages have their own versions of parallel loops. Perhaps the most prominent of these is the UPC `forall` construct which ties execution of particular iterations to an affinity expression that can depend on the induction variable of the loop. ZPL, Chapel, X10, and Titanium allow parallel loops to be run on affinity sets which implicitly determine which CPU executes what iteration.

Collectives, Teams, and Synchronization

Collective operations in parallel programming languages denote operations that potentially involve more than two participants. Collective communication concepts were popularized by MPI, although basic ideas like parallel prefix are considerably older.

Collectives are important in the context of parallel programming models for two major reasons. First, collective communication primitives succinctly express complex data movement operations, contributing to brevity and clarity in parallel programs. Second, because of their relatively simple and well-studied semantics, collectives are good optimization targets, resulting in improved performance and scalability.

Collective operations are either pure data exchange protocols (such as broadcast, scatter, and all-to-all exchanges), or computational collectives (like reductions, where data are interpreted and recomputed during the collective).

Another way to describe collectives is based on whether they have synchronizing properties. For example, the `Alltoall` collective causes synchronization between every pair of tasks involved, since completion of the collective involves bidirectional data dependencies on every pair. Other collectives, like `Scatter`, create much fewer data dependencies and therefore do not cause global synchronization. Finally, `Barrier` is an example of a collective that exchanges no data at all; its only purpose is to effect a synchronization.

Collective communication is further categorized by the number of participants. The simplest case is that of every task in a job participating in a collective. However, arbitrary teams of tasks (called communicators in MPI) can be set up for collective communication.

There are several intriguing aspects that cause the mapping of collective communication to be nonobvious in a PGAS context. The most immediate of these involves data integrity. A natural way to think about data integrity in collective communication is as follows: Data buffers passed from the caller to a collective cannot be touched (either read or written) by the user until the collective completes. In other words, data buffers' ownership changes when the collective is invoked and when it terminates.

However, on a system with shared memory and one-sided data access the invocation boundary is fuzzy. The collective may be entered at different times on different processors. For example, in the presence of one-sided communication the calling process is unable to provide a strong guarantee to the collective that the data buffer will not be touched – since other processes may not yet have entered the collective and may be in the middle of a remote update to the very buffer being processed by the collective.

UPC attempts to deal with this problem by allowing the programmer to state pre- and post-conditions on the boundaries of a collective operation with regard to shared data.

Just like the PGAS model extends point-to-point communication with non-blocking and split-phase transactions, a similar extension can be envisaged for collective communication. The evident advantage of non-blocking collective communication is the ability to overlap it with computation or other communication. There is a case to be made for one-sided collectives. Far from a contradiction in terms, one-sided collective communication involves the address spaces of multiple tasks, but possibly not every task participates actively in the collective. An example would be a one-sided broadcast similar to a one-sided write operation but targeting multiple address spaces.

Some PGAS languages, like Coarray Fortran, feature synchronization operation on teams of tasks designated on an ad hoc basis. This extends the MPI notion in which MPI communicators are predefined

and relatively heavyweight objects. Collective communication exists in some of the PGAS languages today. Titanium features teams and exchange, broadcast, reduction collectives. UPC has a usual complement of collectives but no teams. Coarray Fortran features ad hoc teams in its synchronization operations. Many array languages feature array operations that are essentially "syntax sugar" for collective operations.

Memory Consistency

The memory consistency model of PGAS programs deals with the effect of writing to remote memory; i.e., under what conditions does a remote write become visible by the source, destination, or third parties. The gold standard for memory consistency is sequential consistency: In this model, the memory behaves as if it were written by a single processor at a time.

Sequential consistency is expensive to implement in a distributed memory system because performance can be gated by the slowest write. Therefore, most PGAS languages implement a weak consistency model. For example, Coarray Fortran's consistency model is designed to avoid conflicts and allows compiler optimization. Ordering of memory accesses made to remote locations is done explicitly by the programmer by breaking the program into ordered segments. Conflicting writes in the same segment are disallowed: The basic constraint is that if a variable is defined in a segment, it cannot be read or written by any other image in the same segment. UPC has two memory consistency modes, strict and relaxed, where strict consistency is understood to be sequential consistency. In Titanium, local dependencies are observed. Shared reads and writes performed in critical sections cannot appear to have executed outside.

Future Trends

The future of the Partitioned Global Address Space programming model is difficult to predict. A variety of programming languages based on the model have been proposed, none meeting with universal approval. The state of the art in parallel programming continues to be MPI and OpenMP programming; it is safe to say that the programming model has not yet fulfilled its promise.

The face of parallel computing is continuously changing. While single processor performance has stopped following Moore's law, peak performance on the Top500 website continues to track an exponential growth curve. This growth is achieved by machines with hybrid (shared and distributed memory) architectures, forcing a change in programming technology. Also, an increasing share of high-performance programming also targets hybrid architectures of another kind: dedicated accelerators based on, e.g., GPU compute engines. OpenMP and MPI face some difficulty in coping with these challenges, and may leave the field open for new software technology.

Recognizing that PGAS languages are unlikely to replace MPI, the current trend is to enhance interoperability, allowing coexistence of multiple languages in the same executable. The challenge is both conceptual and practical, and includes reconciliation of the execution models, data representations, and execution semantics of different programming models. On a practical level, the trend is toward shared infrastructure with MPI and an expression of PGAS functionality through library calls to enable a multiplicity of language implementations.

Related Entries

►Array Languages
►Chapel (Cray Inc. HPCS Language)
►Coarray Fortran
►Collective Communication
►Fortress (Sun HPCS Language)
►Global Arrays Parallel Programming Toolkit
►HPF (High Performance Fortran)
►Memory Models
►MPI (Message Passing Interface)
►OpenMP
►SPMD Computational Model
►Titanium
►UPC
►ZPL

Bibliography

1. Chamberlain BL, Choi S-E, Christopher Lewis E, Lin C, Snyder L, Weathersby D (2000) ZPL: a machine independent programming language for parallel computers. Softw Eng 26(3):197–211
2. The cascade high productivity language. HIPS, 00:52–60, 2004
3. High Performance Fortran Forum (1993). High performance Fortran language specification, version 1.0. Technical report CRPC-TR92225, Houston
4. Nieplocha J, Palmer B, Tipparaju V, Krishnan M, Trease H, Apra E (2006). Advances, applications and performance of the global arrays shared memory programming toolkit. Int J High Perform Comput Appl 20:203–231

5. Numrich RW, Reid J (1998) Co-array fortran for parallel programming. SIGPLAN Fortran Forum 17(2):1–31
6. Open MP (2000) Simple, portable, scalable SMP programming. http://www.openmp.org/
7. Snir M, Otto S, Huss-Lederman S, Walker D, Dongarra J. MPI-the complete reference. The MPI Core, vol 1. MIT Press, Cambridge, MA
8. Split-C website. http://www.eecs.berkeley.edu/Research/Projects/CS/parallel/castle/split-c/
9. UPC language Specification, V1.2, May 2005
10. The X10 programming language. http://x10.sourceforge.net, 2004
11. Yelick K, Semenzato L, Pike G, Miyamoto C, Liblit B, Krishnamurthy A, Hilfinger P, Graham S, Gay D, Colella P, Aiken A (1998). Titanium: a high-performance Java dialect. Concurrency Pract Experience 10(11–13):825–836

Phylogenetic Inference

►Phylogenetics

Phylogenetics

Alexandros Stamatakis
Heidelberg Institute for Theoretical Studies, Heidelberg, Germany

Synonyms

Molecular evolution; Phylogenetic inference; Reconstruction of evolutionary trees

Definition

Phylogenetics, or phylogenetic inference (bioinformatics discipline), deals with models and algorithms for reconstruction of the evolutionary history – mostly in form of a (binary) evolutionary tree – for a set of living biological organisms based upon their molecular (DNA) or morphological (morphological traits) sequence data.

Discussion

Introduction

The reconstruction of phylogenetic (evolutionary) trees from molecular or morphological sequence data is a comparatively old bioinformatics discipline, given that likelihood-based statistical models for phylogenetic inference were introduced in the early 1980s, while discrete criteria that rely on counting changes in the sequence data date back to the late 1960s and early 1970s.

Computationally, likelihood-based phylogenetic inference approaches represent a major challenge, because of high memory footprints and of floating point intensive computations.

The goal of phylogenetic inference consists in reconstructing the evolutionary history of a set of n present-day organisms for which molecular sequence data can be obtained. In some cases it is also possible to extract ancient DNA or establish the morphological properties (traits) of fossil records.

Input

The input for a phylogenetic analysis is a list of organism names and their associated DNA or protein sequence data. Note that the DNA sequences for distinct organisms will typically have different lengths. In modern phylogenetics, instead of using the raw sequence data, a so-called multiple sequence alignment (MSA) of the molecular data of the organisms is used as input. Multiple sequence alignment is an important – generally NP-hard – bioinformatics problem. The key goal of MSA is to infer homology, that is, determine which nucleotide characters in the sequence data share a common evolutionary history. Because insertions and/or deletions of nucleotides may have occurred during the evolutionary history of the organisms (represented by their DNA sequences), such events are denoted by inserting the gap symbol - into the sequences during the MSA process. After the alignment step, all n sequences will have the same length m, that is, the MSA has m alignment columns (also called: characters, sites, positions). A simple example for an MSA of DNA data for the Human, the Mouse, the Cow, and the Chicken with $n = 4$ species and $m = 27$ sites is provided below:

```
Cow       ATGGCATATCCCA-ACAACTAGGATTC
Chicken   ATGGCCAACCACTCCCAACTAGGCTTA
Human     ATGGCACAT---GCGCAAGTAGGTCTA
Mouse     ATGG----CCCATTCCAACTTGGTCTA
```

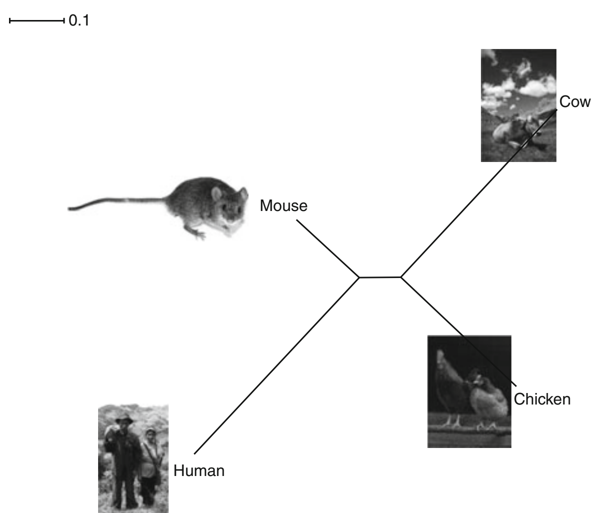
Output

The output of a phylogenetic analysis is mostly an *unrooted* binary tree topology. The present-day organisms under study (for which DNA data can be extracted) are assigned to the leaves (tips) of such a tree, whereas the inner nodes represent common extinct ancestors. The branch lengths of the tree represent the relative

evolutionary time between two nodes in the tree. A likelihood-based phylogenetic tree for the Human, the Mouse, the Cow, and the Chicken using the above MSA is provided in Fig. 1. The biological interpretation of this tree is that the Mouse and the Human are more closely related to each other than to the Cow and the Chicken.

Combinatorial Optimization

In order to reconstruct a phylogenetic tree, criteria are required to assess how well a specific tree topology explains (fits) the underlying molecular sequence data. One may think of this as an abstract function $f()$ that scores alternative tree topologies for a given, fixed MSA. Thus, the goal of a phylogenetic tree reconstruction algorithm is to find the tree topology with the best score according to $f()$, that is, phylogenetic inference is a combinatorial optimization problem. The algorithmic problem in phylogenetics is characterized by the number of possible distinct unrooted binary tree topologies for n organisms that is given by: $\prod_{i=3}^{n}(2i-5)$. For $n = 50$, there already exist approximately 10^{80} possible tree topologies; this number corresponds to the number of atoms in the universe. Because of the size of the tree search space, phylogenetic inference under commonly used scoring criteria $f()$ such as maximum likelihood or maximum parsimony is NP-hard. Therefore, a significant amount of research effort in phylogenetics has focused on the design of efficient heuristic search strategies. Moreover, the optimization of the scoring function $f()$ by algorithmic and technical means also represents an important research objective in phylogenetics.

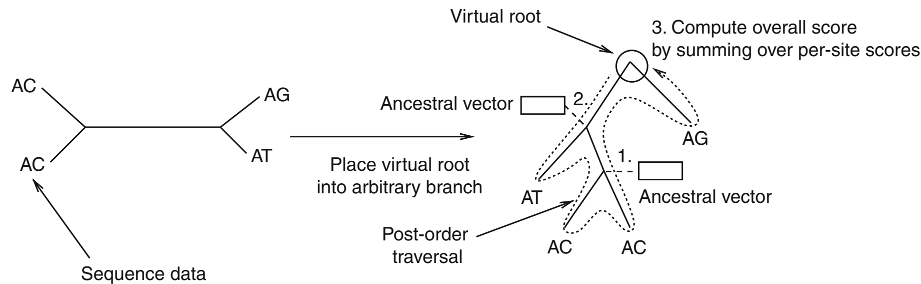


Phylogenetics. Fig. 1 Likelihood-based tree for the Cow, the Chicken, the Mouse, and the Human

Optimality Criteria

The most straightforward approach to phylogenetic inference is to use distance-based methods as opposed to character-based methods (see below). Distance-based methods rely on initially building a symmetric $n \times n$ matrix D of pair-wise distances between the organisms under consideration, which is subsequently used to infer a tree. The optimality of a tree with given branch lengths is determined via a least squares method that is deployed to quantify the difference between the distances given by D and the distances induced by the tree topology (also called patristic distances). Thus, least squares optimization strives to find the tree that minimizes the difference between the pair-wise distances induced by the tree and the corresponding distances in D and is known to be NP-hard. Commonly used heuristics for distance-based tree reconstruction are the Unweighted Pair Group Method with Arithmetic mean (UPGMA) and Neighbor Joining (NJ) methods.

Character-based methods (parsimony and likelihood) directly operate on the sequences of the MSA. The sequences are assigned to the leaves of the tree and an overall score for the tree is computed via a post-order tree traversal with respect to a virtual root. One of the main properties of the likelihood and parsimony criteria is that the respective scores (function $f()$) are invariant to the placement of such a virtual root, that is, the scores will be identical, irrespective of where the virtual root is placed. Parsimony and likelihood criteria are characterized by two additional properties. (1) They assume that MSA columns have evolved independently, that is, given a fixed tree topology, one can simultaneously compute a parsimony or likelihood score for each column of the MSA. To obtain the overall tree score, the sum over all m, where m is number of columns in the MSA, per-column likelihood or parsimony scores at the virtual root is computed. (2) Likelihood and parsimony scores are computed via a post-order tree traversal that proceeds from the tips toward the virtual root and computes ancestral sequence or ancestral probability vectors of length m at each inner node that is visited (see Fig. 2).



Phylogenetics. Fig. 2 Virtual rooting and post-order traversal of a phylogenetic tree. During the post-order traversal, ancestral state vectors are computed. The per-column parsimony or likelihood scores are summed up at the root to obtain the overall tree score

The parsimony criterion intends to minimize the number of nucleotide changes on a tree, while maximum likelihood strives to maximize the fit between the tree and the data (the MSA) using an explicit statistical model of sequence evolution. Bayesian approaches that integrate over the tree (parameter) space using (Metropolis-Coupled) Markov-Chain Monte-Carlo approaches have become popular since the late 1990s. The underlying computational problems are similar, because, as for maximum likelihood, execution times are dominated (85–95%) by evaluations of the phylogenetic likelihood function.

Finally, there exist methods that do not directly use molecular sequence data for phylogeny reconstruction. Instead, these methods use gene order data as input, that is, they strive to infer evolutionary relationships based on distinct arrangements of corresponding genes along the chromosome(s) of the organisms under study. In this context, two organisms are more closely related to each other, if the order of their corresponding genes has not substantially changed, that is, if their chromosomes have not been rearranged to a large extent in the course of evolution. The overall goal can be formulated as inferring a tree that explains the evolutionary history in a parsimonious way, that is, with a minimum number of gene rearrangement events. Even simple versions of this problem are NP-hard [13].

Vectorization

Both likelihood and parsimony computations can be vectorized at a low level.

Parsimony operations for counting the number of changes in a tree can be represented as operations on bit vectors, since ancestral parsimony vectors require 4 bits per alignment column to denote the presence or absence of one of the DNA characters `A, C, G, T`. The bit-level operations that are required are: bit-wise and, bit-wise or, bit-wise nand, and population count (popcount; counting the number of bits that are set to 1 in a data word) operations. Using Streaming Single Instruction Multiple Data (SIMD) Extensions 3 (SSE3) vector instructions on x86 architectures, which are 128 bits wide, 32 ancestral states (128 divided by 4) can be computed during a single CPU cycle.

Likelihood computations can also be vectorized, since the computation of the ancestral state at a position c, where $c = 1 \ldots m$, of the alignment entails computing the probabilities $P(A), P(C), P(G), P(T)$ of observing a nucleotide `A, C, G`, or `T` at this position. At an abstract level, the phylogenetic likelihood computations for DNA data are dominated by a dense matrix–matrix multiplication of a 4×4 floating point matrix (nucleotide substitution matrix) with a $4 \times m$ floating point matrix (ancestral probability vector).

The open-source phylogenetic inference program Randomized Axelerated Maximum Likelihood (RAxML) by Stamatakis [53] provides SSE3-vectorized implementations of the likelihood and parsimony functions for DNA data. Vector instructions for the likelihood function have also been deployed on the IBM CELL architecture by Blagojevic et al. [8]. In general, both optimality criteria allow for vectorization using wider (e.g., 512-bit) vector lengths. Auto-vectorization is not always possible because of code complexity or because the codes need to be redesigned to take advantage of vector instructions. In RAxML, for instance, intrinsic SSE3 functions have been used for explicit vectorization.

Distance-based methods can in principle also be vectorized, but the specific strategy depends on the function used to compute the pair-wise distances between sequences and also on the heuristic search strategy that is deployed.

Fine-Grain Parallelization

As outlined in Fig. 2, the computations of per-site parsimony or likelihood scores are completely independent of each other until the virtual root is reached. Given an MSA with $m = 1000$ sites, this means that all per-site scores can be computed simultaneously and in parallel. The only limitation is that, to obtain the overall score for the tree, the per-site scores need to be accumulated when the virtual root is reached via a respective parallel reduction operation.

The parallel efficiency of this approach depends on the speed of reduction operations in a parallel system and on the number of sites m in the MSA. Generally, scalability increases with m since a large m will yield a more favorable computation to synchronization (via a reduction operation) ratio. Both vectorization and fine-grain parallelization approaches for the likelihood and parsimony criteria are independent of the search strategy used.

To date, fine-grain parallelism has mainly been adopted by likelihood-based programs, since the likelihood function has significantly higher memory and computational requirements than parsimony.

In terms of parallel programming paradigms, Open Multi-Processing (OpenMP), POSIX threads (Pthreads), the Message Passing Interface (MPI), and the Compute Unified Device Architecture (CUDA) have been deployed for exploring fine-grain parallelism. The following types of parallel computer architectures or accelerator devices have been used for phylogenetic likelihood computations to date: Field Programmable Gate Arrays (FPGAs), Graphics Processing Units (GPUs), the IBM Playstation 3 and the IBM CELL processor, Symmetric Multi-Processors (SMPs), multi-core architectures, clusters of SMPs with InfiniBand and Gigabit-Ethernet interconnects, the massively parallel IBM BlueGene/L, and a shared-memory Silicon Graphics Instruments (SGI) Altix 4700 supercomputer.

Medium-Grain Parallelization

Medium-grain parallelization refers to parallelizing the search algorithms of parsimony- or likelihood-based methods and is also called inference parallelism. However, because of their diversity and complexity, the parallelization of the steps of heuristic search algorithms is a nontrivial task. Moreover, every parallelization will need to be highly algorithm-specific and thereby not be generally applicable to other phylogenetic inference programs. An additional problem is that many modern search algorithms, as implemented for instances by Ronquist and Huelsenbeck in MrBayes [50], by Zwickl in GARLI [68], by Guindon and Gascuel in PHYML [28], by Stamatakis in RAxML [53], or by Goloboff in TNT [26], are characterized by hard-to-resolve sequential dependencies. However, other search algorithms like IQPNNI by Minh et al. [38] and TreePuzzle by Strimmer et al. [62] are straightforward to parallelize at this level.

Coarse-Grain Parallelization

Likelihood-based and distance-based phylogenetic inference algorithms exhibit relatively easy-to-exploit sources of coarse-grain/embarrassing parallelism, which are discussed below.

Coarse-Grain Parallelism in Distance-Based Analyses

Parallelization of distance-based analyses is straightforward. The execution times of distance-based analyses are dominated by the computation of the $n \times n$ symmetric distance matrix D that contains the pair-wise distances between all n organisms. All n^2 entries of D are independent of each other and can hence be computed in parallel.

Coarse-Grain Parallelism in Maximum Likelihood Analyses

There are two sources of coarse-grain parallelism in ML analyses. One can conduct a number of completely independent tree searches, starting from different reasonable, that is, nonrandom, starting trees. Such reasonable starting trees can, for instance, be obtained by using simpler (and less computationally intensive)

methods such as neighbor joining or maximum parsimony. Given a collection of distinct, nonrandom, starting trees, individual ML tree searches can be conducted to find the best-scoring (remember that ML optimization is NP-hard) tree. Thereby, one may find a tree with a better likelihood score than by conducting a single search. The same technique can also be applied to parsimony searches for finding the most parsimonious tree.

The second source of embarrassing parallelism in ML phylogenetic analyses is the phylogenetic bootstrapping procedure that was proposed by Felsenstein [20]. Phylogenetic bootstrapping serves as a mechanism for inferring support values, that is, for assigning confidence values to inner branches of a phylogenetic tree. Those confidence values at the inner branches are usually interpreted as the certainty that a particular evolutionary split of the set of organisms has been correctly inferred. Cutting the tree into two parts at an inner branch generates a split (also called bipartition) of the organisms into two disjoint sets. Therefore, the goal consists in obtaining support values for all possible splits/bipartitions induced by the internal branches of a tree.

The phylogenetic bootstrap procedure works as follows: Initially, the input alignment is perturbed by drawing columns/sites (with replacement) at random to assemble a bootstrap replicate alignment of length m. Thus, a bootstrapped alignment has the same number m of sites/columns as the original alignment, but exhibits a different site composition. This re-sampling process is repeated 100–1,000 times, that is, 100–1,000 bootstrap replicate alignments are generated. When those 100–1,000 bootstrapped alignments have been generated, one applies the phylogenetic inference method of choice to infer a tree for each of the replicates. Thereby, as many trees as there are bootstrap replicates are generated. This set of bootstrap trees is then used to answer the question: How stable is the tree topology under slight alterations of the input data?

This collection of bootstrapped trees is then either used to compute a consensus tree, that is, compute the "average" tree topology or for drawing support values on the best-known ML tree obtained on the original – non-bootstrapped – alignment. In the latter case, one just needs to count how frequently each bipartition of the best-known ML tree occurs in the set of bootstrapped trees.

The bootstrapping procedure is embarrassingly parallel because tree searches on individual bootstrap replicates are completely independent from each other and can be parallelized by executing the 100–1,000 bootstrap inferences on a cluster, GRID, or cloud.

A question that naturally arises in this context is: How many bootstrap replicates are required to obtain reliable support values? Hedges [29] proposed a theoretical upper bound for phylogenetic bootstrapping. The number of required bootstrap replicates also appears to depend on the input data. Therefore, adaptive criteria as proposed by Pattengale et al. [45] may be well-suited to determine a sufficient number of bootstrap replicates.

Coarse-Grain Parallelism in Bayesian Analyses

Bayesian analyses exhibit a source of coarse grain parallelism at the level of executing multiple chains in parallel, that is, using the Metropolis-Coupled Markov-Chain Monte-Carlo (MC^3) approach (see Metropolis et al. [37]). In a typical program run of the widely used MrBayes code by Ronquist and Huelsenbeck [50] for Bayesian phylogenetic inference, the program will use three heated Markov chains that accept more radical moves in parameter space (this entails topological moves as well as moves to sample other parameters of the likelihood model) and a cold chain that only accepts more conservative moves. The cold chain operates on the tree with the currently best likelihood. However, one of the heated chains may at some point encounter a tree with a higher likelihood than the cold chain. In this case, the cold chain and the heated chain with the better tree need to exchange states, that is, the cold chain will become a heated chain and the heated chain will become the cold chain. Therefore, while those four chains (three heated chains and one cold chain) may be started as independent Message Passing Interface (MPI) processes, the chains will need to be synchronized after a certain number of, for instance 100, generations (proposals) to assess if states need to be exchanged between chains. Thus, parallel load balance is a critical issue, because the chains will need to run at similar speeds in order to minimize synchronization delays. On a homogeneous cluster (equipped with a single CPU type) this

is not problematic, because the average execution times for 100 or 1,000 proposals are expected to be very similar. This good expected load balance is due to the fact that the same average number of proposal types (tree proposal, model parameter proposal) will be executed in each chain. Finally, completely independent runs can be executed in an embarrassingly parallel manner.

Phylogenetics Today

Phylogenetics currently face two main challenges. One major challenge is the significant advances in wet-lab sequencing technologies that have led to an unprecedented molecular data "flood." In fact, the amount of publicly available molecular data increases at a significantly higher speed than the number of transistors according to Moore's law (see Goldman and Yang [25] for a respective plot). To this end, researchers today do not use sequence data from a single gene or just a couple of genes to reconstruct trees for the organisms they study. Instead, they use data from several hundreds or even thousands of genes (e.g., Hejnol et al. [30]). Thus, the number of sites m in alignments has increased from 1,000–10,000 to over 1,000,000. This transition from single-/few-gene phylogenies to many-gene phylogenies is also reflected by the increased use of the term phylogenomics, that is, phylogenetic inference at (almost) the whole-genome level.

In current phylogenomic analyses the number of organisms n ranges between 50 and 500, but given the constant innovations in wet-lab sequencing, this number may soon increase by one order of magnitude. Because of sequencing innovations, input datasets are also growing with respect to n, that is, analyses using up to ten genes for 10,000–70,000 organisms are becoming more common. To date, the largest published likelihood-based tree contained 13,000 organisms (see Smith and Donoghue [52]) while the largest published parsimony-based tree contained 73,000 organisms (see Goloboff et al. [27]). The general growth in dataset sizes poses problems with respect to the memory requirements of phylogenetic analyses, in particular with respect to likelihood-based approaches. Memory footprints for computing the likelihood on a single tree, exceeding 50GB are not uncommon any more. The highest reported memory footprint the author is aware of was 180GB for a likelihood-based analysis of 35 mammalian genomes (Ziheng Yang, personal communication, April 2010). Memory-related problems also affect distance-based methods (for large n) because of the space requirements of the $n \times n$ distance matrix (Wheeler [65] and Price et al. [48] discuss some technical solutions to this problem). Apart from memory-related problems, the computation of trees with more than 10,000 organisms also poses novel algorithmic challenges. Finally, the increasing complexity of the statistical models that are used for phylogenetic inference, such as, mixture models (e.g., Lartillot and Philippe [31]), further increase the computational complexity of likelihood-based approaches.

The second major challenge is the emergence of multi-core systems at the desktop level and of accelerator architectures such as GPUs (Graphics Processing Units) or the IBM Cell. Those new parallel architectures pose challenges regarding the parallelization of the likelihood or parsimony functions. Given the aforementioned high memory footprints, a parallelization at a fine-grain level, that is, multiple processors/cores working together to compute the score on a single tree, is required. In order to be of value and to be used by the biological user community, such parallelizations need to be readily available at the production level and also need to be easy to install and use. Developers of phylogenetics software, which has come off age by now, are also increasingly facing software engineering issues, because the codes have become more complex over recent years. Programs such as RAxML or MrBayes provide a plethora of substitution models and search algorithms. They also allow for analyzing different data types, for instance, binary, morphological, DNA, RNA secondary structure (see Savill et al. [51] for a discussion of RNA secondary structure evolution models), or protein data. Therefore, the phyloinformatics and HPC communities are facing an unprecedented challenge in trying to keep pace with data accumulation and parallel architecture innovations to provide ever more scalable and powerful analysis tools. HPC in phylogenetics has thus become a key to the success of the field.

Future Directions

One of the major future challenges in phylogenetics consist of efficiently exploiting parallel architectures to compute the likelihood or parsimony scoring

functions at the production level. There exists an ongoing effort to implement an open-source likelihood function library (http://code.google.com/p/beagle-lib/) that can be executed on GPUs and multi-core architectures (including a vectorization using SSE3 intrinsics and an OpenMP-based fine-grain parallelization). Fine-grain parallelizations of those functions are required to be able to accommodate and distribute the growing memory space requirements across several cores or – potentially hybrid – multi-core nodes. Nonetheless, alternative directions for handling memory consumption should be explored. While there exist suggestions for reducing memory requirements via algorithmic means (see Stamatakis and Ott [57] and Stamatakis and Alachiotis [54]), trade-offs between using single and double precision arithmetics as well as their impact on numerical stability need to be further explored. Out-of-core execution may provide a solution for executing large-scale analyses on the desktop. Analyses of trees with over 10,000 organisms require a – potentially difficult – parallelization at the algorithmic level, because scalability of fine-grain parallelism is limited to 8 or 16 cores due to the relatively small number of sites in such alignments. Finally, the simultaneous development of parallelization and algorithmic strategies appears to be the most promising approach to address current and future challenges.

Related Entries

►Bioinformatics
►Cell Broadband Engine Processor
►Clusters
►Collective Communication
►Distributed-Memory Multiprocessor
►Ethernet
►Genome Assembly
►Hybrid Programming With SIMPLE
►IBM Blue Gene Supercomputer
►InfiniBand
►Load Balancing, Distributed Memory
►Loops, Parallel
►MPI (Message Passing Interface)
►Multi-Threaded Processors
►NVIDIA GPU
►OpenMP
►Parallelization, Automatic
►POSIX Threads (Pthreads)
►Reconfigurable Computers

Bibliographic Notes and Further Reading

The text-books by Felsenstein [21] and Yang [66] provide a detailed introduction to the field of phylogenetic inference and the underlying models of evolution. The phylogenetic likelihood function was introduced by Felsenstein in 1981 [19]. One of the early papers dealing with parsimony was published by Fitch and Margoliash [23]. NP-hardness was demonstrated by Day [17] for the least squares approach, by Day et al. for parsimony [18], and by Chor and Tuller for likelihood [16] (also see Roch [49] for a shorter proof). Morrison [42] provides an overview and discussion of current heuristic search strategies. The vectorization of the likelihood function as well as numerical aspects with respect to single and double precision implementations of the likelihood function are addressed by Stamatakis and Berger [7]. FPGA implementations are covered by Alachiotis et al. [3, 4], Mak and Lam [34–36], Zierke and Bakos [67], and Bakos [5, 6]; GPU implementations by Charalambous et al. [15], Suchard and Rambaut [63], while Pratas et al. describe a performance comparison between GPUs, the IBM CELL, and general purpose multi-core architectures [47]. Blagojevic et al. have published a series of papers on porting RAxML to the IBM CELL [8–10, 55]. Pthreads and OpenMP-based parallelization of the parsimony and likelihood functions have been described by Stamatakis et al. [57, 60]. Ott et al. [43, 44] and Feng et al. [22] describe fine-grain parallelizations with MPI on distributed memory machines. Stamatakis and Ott also address load-balance issues [59] and assess performance of Pthreads versus MPI versus OpenMP for fine-grained parallelism in the likelihood function [58]. Medium-grain parallelizations are covered by Minh et al. [38], Stamatakis et al. [56], Stewart et al. [61], and Ceron et al. [14]. Hybrid parallelizations have been described by Minh et al. [39], Feng et al. [22], and Pfeiffer and Stamatakis [46]. The two post-analysis steps for analyzing bootstrapped trees (consensus tree building and drawing bipartitions on the best-known tree) have also been parallelized (see Aberer et al. [1, 2]; the papers also

contain a detailed description of the discrete algorithms for consensus tree building and drawing bipartitions on trees). A related discrete problem on trees, that of reconstructing a species tree from (potentially incongruent) per-gene trees by the minimum possible number of gene duplication events, has been parallelized by Wehe et al. [64] on an IBM BlueGene/L supercomputer. The review by Maddison [33] provides an overview of the gene tree species tree problem. Heuristic algorithms for gene order phylogeny reconstruction have been implemented and optimized by Moret et al. [40, 41] in a tool called GRAPPA. GRAPPA has also been parallelized using a coarse-grain approach to simultaneously enumerate and evaluate all possible trees for 13 organisms on a cluster with 512 cores. The original heuristic algorithm has been proposed by Blanchette et al. [11]. Finally, the papers by Fleissner et al. [24], Loytynoja and Goldman [32], or Bradley et al. [12], for instance, deal with the more challenging and advanced problem of simultaneous alignment (MSA) and tree building.

Bibliography

1. Aberer A, Pattengale N, Stamatakis A (2010) Parallel computation of phylogenetic consensus trees. Procedia Comput Sci 1(1): 1059–1067
2. Aberer A, Pattengale N, Stamatakis A (2010) Parallelized phylogenetic post-analysis on multi-core architectures. J Comput Sci 1(2):107–114
3. Alachiotis N, Sotiriades E, Dollas A, Stamatakis A (2009) Exploring FPGAs for accelerating the phylogenetic likelihood function. In: IEEE international symposium on parallel & distributed processing, 2009. IPDPS 2009, pp 1–8. IEEE
4. Alachiotis N, Stamatakis A, Sotiriades E, Dollas A (2009) A reconfigurable architecture for the Phylogenetic Likelihood Function. In: International Conference on Field Programmable Logic and Applications, 2009. FPL 2009, pp 674–678. IEEE, 2009
5. Bakos J (2007) FPGA acceleration of gene rearrangement analysis. In: Proceedings of 15th annual IEEE symposium on field-programmable custom computing machines. IEEE, Napa, CA, pp 85–94
6. Bakos J, Elenis P, Tang J (2007) FPGA acceleration of phylogeny reconstruction for whole genome data. In: Proceedings of the 7th IEEE international conference on bioinformatics and bio engineering. IEEE, Boston, MA, pp 888–895
7. Berger S, Stamatakis A (2010) Accuracy and performance of single versus double precision arithmetics for maximum likelihood phylogeny reconstruction. Lecture notes in computer science, vol 6068. Springer, pp 270–279
8. Blagojevic F, Nikolopoulos D, Stamatakis A, Antonopoulos C (2007) Dynamic multigrain parallelization on the cell broadband engine. In: Proceedings of PPoPP 2007, San Jose, CA, March 2007, pp 90–100
9. Blagojevic F, Nikolopoulos D, Stamatakis A, Antonopoulos C, Curtis-Maury M (2007) Runtime scheduling of dynamic parallelism on accelerator-based multi-core systems. Parallel Comput 33:700–719
10. Blagojevic F, Nikolopoulos DS, Stamatakis A, Antonopoulos CD (2007) RAxML-Cell: Parallel phylogenetic tree inference on the cell broadband engine. In: Proceedings of international parallel and distributed processing symposium (IPDPS2007), 2007
11. Blanchette M, Bourque G, Sankoff D (1997) Breakpoint phylogenies. In: Miyano S, Takagi T (eds) Workshop on genome informatics, vol 8. Univ. Academy Press, pp 25–34
12. Bradley R, Roberts A, Smoot M, Juvekar S, Do J, Dewey C, Holmes I, Pachter L (2009) Fast statistical alignment. PLoS Comput Biol 5(5):e1000392
13. Bryant D (1998) The complexity of the breakpoint median problem. Technical report, University of Montreal, Canada
14. Ceron C, Dopazo J, Zapata E, Carazo J, Trelles O (1998) Parallel implementation of DNAml program on message-passing architectures. Parallel Comput 24(5–6):701–716
15. Charalambous M, Trancoso P, Stamatakis A (2005) Initial experiences porting a bioinformatics application to a graphics processor. Lecture notes in computer science, vol 3746. Springer, New York, pp 415–425
16. Chor B, Tuller T (2005) Maximum likelihood of evolutionary trees: hardness and approximation. Bioinformatics 21(1):97–106
17. Day W (1987) Computational complexity of inferring phylogenies from dissimilarity matrices. Bulletin of Mathematical Biology 49(4):461–467
18. Day W, Johnson D, Sankoff D (1986) The computational complexity of inferring rooted phylogenies by parsimony. Mathematical biosciences 81(33–42):299
19. Felsenstein J (1981) Evolutionary trees from DNA sequences: a maximum likelihood approach. J Mol Evol 17:368–376
20. Felsenstein J (1985) Confidence limits on phylogenies: an approach using the bootstrap. Evolution 39(4):783–791
21. Felsenstein J (2004) Inferring phylogenies. Sinauer Associates, Sunderland
22. Feng X, Cameron K, Sosa C, Smith B (2007) Building the tree of life on terascale systems. In: Proceedings of international parallel and distributed processing symposium (IPDPS2007), 2007
23. Fitch W, Margoliash E (1967) Construction of phylogenetic trees. Science 155(3760):279–284
24. Fleissner R, Metzler D, Haeseler A (2005) Simultaneous statistical multiple alignment and phylogeny reconstruction. Syst Biol 54:548–561
25. Goldman N, Yang Z (2008) Introduction. statistical and computational challenges in molecular phylogenetics and evolution. Philos Trans R Soc B Biol Sci 363(1512):3889
26. Goloboff P (1999) Analyzing large data sets in reasonable times: solution for composite optima. Cladistics 15:415–428
27. Goloboff PA, Catalano SA, Mirande JM, Szumik CA, Arias JS, Källersjö M, Farris JS (2009) Phylogenetic analysis of 73060 taxa corroborates major eukaryotic groups. Cladistics 25:1–20
28. Guindon S, Gascuel O (2003) A simple, fast, and accurate algorithm to estimate large phylogenies by maximum likelihood. Syst Biol 52(5):696–704

29. Hedges S (1992) The number of replications needed for accurate estimation of the bootstrap P value in phylogenetic studies. Mol Biol Evolution 9(2):366–369
30. Hejnol A, Obst M, Stamatakis A, Ott M, Rouse G, Edgecombe G, Martinez P, Baguna J, Bailly X, Jondelius U, Wiens M, Müller W, Seaver E, Wheeler W, Martindale M, Giribet G, Dunn C (2009) Rooting the bilaterian tree with scalable phylogenomic and supercomputing tools. Proc R Soc B 276:4261–4270
31. Lartillot N, Philippe H (2004) A Bayesian mixture model for across-site heterogeneities in the amino-acid replacement process.Mol Biol Evol 21(6):1095–1109
32. Loytynoja A, Goldman N (2008) Phylogeny-aware gap placement prevents errors in sequence alignment and evolutionary analysis. Science 320(5883):1632
33. Maddison W (1997) Gene trees in species trees. Syst Biol 46(3):523
34. Mak T, Lam K (2003) High speed GAML-based phylogenetic tree reconstruction using HW/SW codesign. In: Bioinformatics Conference, 2003. CSB 2003. Proceedings of the 2003 IEEE, pp 470–473
35. Mak T, Lam K (2004) Embedded computation of maximum-likelihood phylogeny inference using platform FPGA. In: Proceedings of IEEE Computational Systems Bioinformatics Conference (CSB 04), pp 512–514
36. Mak T, Lam K (2004) FPGA-Based Computation for Maximum Likelihood Phylogenetic Tree Evaluation. In: Lecture notes in computer science, pp 1076–1079
37. Metropolis N, Rosenbluth A, Rosenbluth M, Teller A, Teller E et al (1953) Equation of state calculations by fast computing machines. J Chem Phys 21(6):1087
38. Minh B, Vinh L, Haeseler A, Schmidt H (2005) pIQPNNI: parallel reconstruction of large maximum likelihood phylogenies. Bioinformatics 21(19):3794–3796
39. Minh B, Vinh L, Schmidt H, Haeseler A (2006) Large maximum likelihood trees. In: Proceedings of the NIC Symposium 2006, pp 357–365
40. Moret B, Tang J, Wang L, Warnow T (2002) Steps toward accurate reconstructions of phylogenies from gene-order data* 1. J Comput Syst Sci 65(3):508–525
41. Moret B, Wyman S, Bader D, Warnow T, Yan M (2001) A new implementation and detailed study of breakpoint analysis. In: Pacific symposium on biocomputing 6:583–594
42. Morrison D (2007) Increasing the efficiency of searches for the maximum likelihood tree in a phylogenetic analysis of up to 150 nucleotide sequences. Syst Biol 56(6):988–1010
43. Ott M, Zola J, Aluru S, Johnson A, Janies D, Stamatakis A (2008) Large-scale phylogenetic analysis on current HPC architectures. Scientific Programming 16(2–3):255–270
44. Ott M, Zola J, Aluru S, Stamatakis A (2007) Large-scale maximum likelihood-based phylogenetic analysis on the IBM BlueGene/L. In: Proceedings of IEEE/ACM Supercomputing Conference 2007 (SC2007), IEEE, Reno, Nevada
45. Pattengale N, Alipour M, Bininda-Emonds O, Moret B, Stamatakis A (2010) How many bootstrap replicates are necessary? J Comput Biol 17(3):337–354
46. Pfeiffer W, Stamatakis A (2010) Hybrid MPI/Pthreads parallelization of the RAxML phylogenetics code. In: IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010, IEEE, Atlanta, Georgia, pp 1–8
47. Pratas F, Trancoso P, Stamatakis A, Sousa L (2009) Fine-grain Parallelism using multi-core, Cell/BE, and GPU systems: accelerating the phylogenetic likelihood function. In: International conference on parallel processing, 2009. ICPP'09, IEEE, Vienna, pp 9–17
48. Price M, Dehal P, Arkin A (2010) FastTree 2–approximately maximumlikelihood trees for large alignments. PLoS One 5(3):e9490
49. Roch S (2006) A short proof that phylogenetic tree reconstruction by maximum likelihood is hard. IEEE/ACM transactions on Computational Biology and Bioinformatics, pp 92–94
50. Ronquist F, Huelsenbeck J (2003) MrBayes 3: Bayesian phylogenetic inference under mixed models. Bioinformatics 19(12): 1572–1574
51. Savill NJ, Hoyle DC, Higgs PG (2001) Rna sequence evolution with secondary structure constraints: comparison of substitution rate models using maximum-likelihood methods. Genetics 157:399–411
52. Smith S, Donoghue M (2008) Rates of molecular evolution are linked to life history in flowering plants. Science 322(5898):86–89
53. Stamatakis A (2006) RAxML-VI-HPC: maximum likelihood-based phylogenetic analyses with thousands of taxa and mixed models. Bioinformatics 22(21):2688–2690
54. Stamatakis A, Alachiotis N (2010) Time and memory efficient likelihoodbased tree searches on phylogenomic alignments with missing data. Bioinformatics 26(12):i132
55. Stamatakis A, Blagojevic F, Antonopoulos CD, Nikolopoulos DS (2007) Exploring new search algorithms and hardware for phylogenetics: RAxML meets the IBM Cell. J VLSI Sig Proc Syst 48(3):271–286
56. Stamatakis A, Ludwig T, Meier H (2004) Parallel inference of a 10.000-taxon phylogeny with maximum likelihood. In: Proceedings of Euro-Par 2004, September 2004, IEEE, Pisa Italy, pp 997–1004
57. Stamatakis A, Ott M (2008) Efficient computation of the phylogenetic likelihood function on multi-gene alignments and multi-core architectures. Philos Trans R Soc B, Biol Sci 363:3977–3984
58. Stamatakis A, Ott M (2008) Exploiting fine-grained parallelism in the phylogenetic likelihood function with MPI, Pthreads, and OpenMP: a performance study. In: Chetty M, Ngom A, Ahmad S (eds) PRIB, Lecture notes in computer science, vol 5265. Springer, Heidelberg, pp 424–435
59. Stamatakis A, Ott M (2009) Load balance in the phylogenetic likelihood kernel. In: International conference on parallel processing, 2009. ICPP'09, IEEE, Vienna, Austria, pp 348–355
60. Stamatakis A, Ott M, Ludwig T (2005) RAxML-OMP: an efficient program for phylogenetic inference on SMPs. Lecture notes in computer science, vol 3606. Springer, Berlin, Heidelberg, pp 288–302
61. Stewart C, Hart D, Berry D, Olsen G, Wernert E, Fischer W (2001) Parallel implementation and performance of fastDNAml – a program for maximum likelihood phylogenetic inference. In: Supercomputing, ACM/IEEE 2001 conference, ACM/IEEE, Denver, Colorado, pp 32–32

62. Strimmer K, Haeseler A (1996) Quartet puzzling: a quartet maximum likelihood method for reconstructing tree topologies. Mol Biol Evol 13:964–969
63. Suchard M, Rambaut A (2009) Many-core algorithms for statistical phylogenetics. Bioinformatics 25(11):1370
64. Wehe A, Chang W, Eulenstein O, Aluru S (2010) A scalable parallelization of the gene duplication problem. J Parallel Distr Comput 70(3):237–244
65. Wheeler T (2009) Large-scale neighbor-joining with ninja. Lecture notes in computer science, vol 5724. Springer, Berlin, pp 375–389
66. Yang Z (2006) Computational molecular evolution. Oxford University Press, USA
67. Zierke S, Bakos J (2010) FPGA acceleration of the phylogenetic likelihood function for Bayesian MCMC inference methods. BMC Bioinformatics 11(1):184
68. Zwickl D (2006) Genetic algorithm approaches for the phylogenetic analysis of large biological sequence datasets under the maximum likelihood criterion. PhD thesis, University of Texas at Austin, April 2006

Pi-Calculus

Davide Sangiorgi
Universita' di Bologna, Bologna, Italy

Synonyms

Calculus of mobile processes

Definition

The π-calculus is a process calculus that models mobile systems, i.e., systems with a dynamically changing communication topology. It refines the constructs of the calculus of communicating systems (CCS) by allowing the exchange of communication links. Ideas from the λ-calculus have also been influential.

Discussion

Introduction

A widely recognized practice for understanding programming languages, be they sequential or concurrent, is to distill small "core languages," or "calculi," that embody the essential ingredients of the languages. This is useful to develop the theory of the programming language (e.g., techniques for static analysis, behavioral specification, and verification), to study implementations, to devise new or better programming language constructs.

A well-known calculus in the realm of sequential languages is the λ-calculus. Invented by Church in the 1930s, it is a pure calculus of functions. Everything in the λ-calculus is a function, and computation is function application. The λ-calculus has been very influential in programming languages. It has been the basis for the so-called functional programming languages. Even more important, it has been essential in understanding concepts such as procedural abstraction and binding, parameter passing (e.g., call-by-name vs. call-by-value), and continuations. All these concepts have then made their way into mainstream programming languages. The λ-calculus can be regarded as a canonical model for sequential computations with functions: on the one hand, it describes the essential features of functions (function definition and function application) in a simple and intuitive way; on the other hand, all the known models of sequential computation have been proved to have the same expressive power as the λ-calculus. In other words, the λ-calculus describes all computable functions (a statement referred to as "Church's thesis").

For concurrent languages (possibly including distribution), a similar canonical model does not exist. This can be explained with the varieties of such systems. In concurrency, the central computational unit is a process. However, there is no universally accepted definition of what a process is. For instance, the ways of conceiving interaction among processes may be very diverse: via synchronous signals, via shared variables, via broadcasting, via asynchronous message passing, via rendezvous. Therefore, in concurrency one does not find a single calculus, but, rather, a variety of calculi, referred to as *process calculi*, or *process algebras* when one wishes to emphasize that both the analysis and the description of a system are carried out in an algebraic setting. The best known such calculi are calculus of communicating systems (CCS), communicating sequential processes (CSP), algebra of communicating processes (ACP), the π-calculus. Among these, the π-calculus is the closest to the λ-calculus. Many fundamental ideas from the λ-calculus can be carried over to the π-calculus. For instance, abstraction and type systems are central concepts in both calculi. Moreover as the λ-calculus underlies functional programming languages, so the π-calculus, or variants of it, are taken as the basis for new programming languages. The process concept of

the π-calculus is roughly that of structured entities interacting by means of message-passing.

The π-calculus has two aspects. First, it is a theory of mobile systems, with a rich blend of techniques for reasoning about the behavior of systems. Second, it is a general model of computation, in which everything is a process and computation is process interaction. Syntactically, the main difference between the π-calculus and its principal process calculus ancestor, CCS, is that communication links are first-class values, and may therefore be passed around in communications. Thus, the set of communication links used by a term may change dynamically, as computation evolves. Such systems are called *mobile*, and the π-calculus is hence called a *calculus of mobile processes*. In CCS, in contrast, the set of communication links for a term is fixed by its initial syntax, and only very limited forms of mobility can be described. Mobility gives the π-calculus an amazing expressive power. For instance, functions and the λ-calculus can be elegantly modeled in it, reducing function application to special forms of process interactions.

Mobility

A *mobile* concurrent system has a communication topology that can dynamically change, as the system evolves. Mobility can be found in many areas of computer science, such as operating systems, distributed computing, higher-order concurrent programming, and object-oriented programming. Two kinds of mobility can be broadly distinguished. In one kind, *links* move in an abstract space of *linked processes*. For example, hypertext links can be created, can be passed around, and can disappear; the connections between cellular telephones and a network of base stations can change as the telephones are carried around; and references can be passed as arguments of method invocations in object-oriented systems. In the second kind of mobility, *processes* move in an abstract space of linked processes. For instance, code can be sent over a network and put to work at its destination; mobile devices can acquire new functionality; an active laptop computer can be moved from one location to another, and made to interact with resources in a different environment. Languages in which terms of the language itself, such as processes, are first-class values, are called *higher-order* (in this sense, the λ-calculus is higher-order too).

The π-calculus models the first kind of mobility: it directly expresses movement of links in a space of linked processes. There are two kinds of basic entity in the (untyped) π-calculus: names and processes. Names specify links. Processes can interact by using names that they share. The crux is the data that processes communicate in interactions are themselves names, and a name received in one interaction can be used to participate in another. By receiving a name, a process can acquire a capability to interact with processes that were previously unknown to it. Thus, the structure of a system – the connections among its component processes – can change over time, in arbitrary ways. The source of strength in the π-calculusis how it treats scoping of names and extrusion of names from their scopes.

There are various reasons for having only mobility of names in the π-calculus. First, naming and name-passing are ubiquitous: think of addresses, identifiers, links, pointers, and references. Second, as will be shown later, higher-order constructs can often be modeled within the π-calculus. Further, by passing a name, one can pass partial access to a process, an ability to interact with it only in a certain way. Similarly, with name-passing one can easily model sharing, for instance of a resource that can be used by different sets of clients at different times. It can be complicated to model these things when processes are the only transmissible values. Thirdly, the π-calculus has a rich and tractable theory. The theory of process-passing is harder, and important parts of it are not yet well understood.

Syntax

As the λ-calculus, so the language of the π-calculus consists of a small set of primitive constructs. In the λ-calculus, they are constructs for building functions. In the π-calculus, they are constructs for building processes, similar to those one finds in CCS. The syntax is as follows. Capital letters range over processes, and small letters over names; the set of all names is infinite.

$$P ::= x(y).P \mid \overline{x}\langle y\rangle.P \mid \tau.P \mid \mathbf{0} \mid P+P' \mid P \mid P' \\ \mid \nu x\, P \mid !P \mid [x=y]P$$

The *input prefix* $x(z).P$ can receive any name via x and continue as P with the received name substituted for z. The substitution of the all occurrences of z in

P with y is written $P\{y/z\}$ (some care is needed here, to properly respect the bindings of a term). An output $\overline{x}\langle y\rangle.P$ emits name y at x and then continues as P. The *unobservable prefix* $\tau.\ P$ can autonomously evolve to P, without the help of the external environment. As in CCS, τ can be thought of as expressing an internal action of a process. The *inactive* process $\mathbf{0}$ can do nothing. For instance, $x(z).\ \overline{z}\langle y\rangle.\ \mathbf{0}$ can receive any name via x, send y via the name received, and become inactive. A *choice* (or *sum*) process $P + P'$ may evolve either as P or as P'; if one of the processes exercises one of its capabilities, the other process is discarded. In the *composition* $P \mid P'$, the components P and P' can proceed independently and can interact via shared names. For instance, $(x(z).\ \overline{z}\langle y\rangle.\ \mathbf{0} + \overline{w}\langle v\rangle.\ \mathbf{0}) \mid \overline{x}\langle u\rangle.\ \mathbf{0}$ has four capabilities: to receive a name via x, to send v via w, to send u via x, and to evolve invisibly as an effect of an interaction between its components via the shared name x. In the *restriction* $\boldsymbol{\nu}z\ P$, the scope of the name z is restricted to P. Components of P can use z to interact with one another but not with other processes. For instance, $\boldsymbol{\nu}x\ ((x(z).\ \overline{z}\langle y\rangle.\ \mathbf{0} + \overline{w}\langle v\rangle.\ \mathbf{0}) \mid \overline{x}\langle u\rangle.\ \mathbf{0})$ has only two capabilities: to send v via w, and to evolve invisibly as an effect of an interaction between its components via x. The scope of a restriction may change as a result of interaction between processes. This important feature of the calculus will be explained later. (The π-calculus notation for the restriction operator is different from that of CCS; indeed, exchange of names makes restriction in the π-calculus semantically quite different from restriction in CCS.) The *replication* $!P$ can be thought of as an infinite composition $P \mid P \mid \cdots$ or, equivalently, a process satisfying the equation $!P = P \mid !P$. Replication is the operator that makes it possible to express infinite behaviors. For example, $!x(z).\ !\overline{y}\langle z\rangle.\ \mathbf{0}$ can receive names via x repeatedly, and can repeatedly send via y any name it does receive. In certain presentations, *recursive process definitions* are used in place of replication: the expressiveness injected into the calculus by these two constructs is the same. A *match* process $[x = y]P$ behaves as P if names x and y are equal, otherwise it does nothing. For instance, $x(z).\ [z = y]\overline{z}\langle w\rangle.\ \mathbf{0}$, on receiving a name via x, can send w via that name just if that name is y; if it is not, the process can do nothing further.

This is the syntax of the pure, untyped, π-calculus. The calculus is monadic, in that only one value at a time can be exchanged. In examples and applications, one often uses the *polyadic* π-calculus, in which the input and output constructs are refined as follows: a polyadic input process $x(y_1, \ldots, y_n).P$ waits for an n-tuple of names $z_1, \ldots, z_n$ at x and then continues as $P\{z_1, \ldots, z_n/y_1, \ldots, y_n\}$ (i.e., P with the y_i's replaced by the z_i's); a polyadic output process $\overline{x}\langle y_1, \ldots, y_n\rangle.P$ emits names $y_1, \ldots, y_n$ at x and then continues as P.

The input construct $x(y).P$ and the restriction $\boldsymbol{\nu}y\ P$ are binders for the free occurrences of y in P, in the same sense as the abstraction construct of the λ-calculus. These binders give rise in the expected way to the definitions of *free* and *bound* names of a process.

When writing processes, parentheses are used to resolve ambiguity, and the conventions are observed that prefixing and restriction bind more tightly than composition and sum. Thus $\overline{x}\langle v\rangle.\ P \mid Q$ is $(\overline{x}\langle v\rangle.\ P) \mid Q$, and $\boldsymbol{\nu}zP \mid Q$ is $(\boldsymbol{\nu}zP) \mid Q$. A process $\overline{x}\langle\rangle.P$ is abbreviated as $\overline{x}.P$ and, similarly, $x().P$ as $x.P$.

Examples

Below some simple examples are given to illustrate the use of the main constructs. The first example is about encoding of data types. In general, processes interact by passing one another data. The only data of the π-calculus are names. However, it may well be convenient to admit other atomic data, such as integers, and structured data, such as tuples and multisets. It is in the spirit of process calculi generally to allow the data language to be tailored to the application at hand, admitting sets, lists, trees, and so on as convenient. And one *should* admit the relevant data when using the π-calculus to reason about systems. Remarkably, however, all such data can be expressed. For instance,

$$T \stackrel{\text{def}}{=} b(x,y).\overline{x}.\mathbf{0} \quad \text{and} \quad F \stackrel{\text{def}}{=} b(x,y).\overline{y}.\mathbf{0}.$$

represent encodings of the boolean values `true` and `false`, located at name b. These processes receive a pair of names via b; then T will respond by signaling on the first of them, whereas F will signal on the second. The following process R reads the value of a boolean located at b and, depending on the value of the boolean, it will behave as P or Q:

$$R \stackrel{\text{def}}{=} \boldsymbol{\nu}t\ \boldsymbol{\nu}f\ \overline{b}\langle t,f\rangle.\ (t.\ P + f.\ Q),$$

It is assumed that t and f are only used to read a boolean, hence they do not appear in P and Q. Now, a system

with both R and T can evolve as follows, using $\longrightarrow$ to indicate a single computation step, i.e., a single interaction; below $\sim$ indicates the application of some simple garbage-collection algebraic laws of the π-calculus that will be discussed later (laws for garbage collecting trailing inactive processes and restrictions on name that are not used anymore).

$$\begin{aligned} T\,|\,R &\longrightarrow \boldsymbol{\nu} t\, \boldsymbol{\nu} f\ (\bar{t}\,|(t.\ P+f.\ Q)) \\ &\longrightarrow \boldsymbol{\nu} t\, \boldsymbol{\nu} f\ (\mathbf{0}\,|\,P) \\ &\sim\ P \end{aligned}$$

In the first step, R sends to T two names, t and f, that were initially private to R; after the interaction, these names are shared between (what remains of) R and T; no other process in the system (in principle, other processes could run in parallel) will now be able to interfere on the following interaction between R and T along t. The capability of creating new names, and sending them around, is at the heart of the expressiveness and the theory of the π-calculus; without it, the π-calculus would not be much different from CCS.

In the second step, the interaction along t selects P, and the other branch of the choice is discarded. In the final line, the garbage-collection laws are applied.

In the example, the act of interrogating a process T or F for its value destroys it. Persistence of data is represented using replication. The encoding of persistent booleans would then be

$$\text{TRUE} \stackrel{\text{def}}{=} !b(x,y).\,\bar{x}.\,\mathbf{0} \quad \text{FALSE} \stackrel{\text{def}}{=} !b(x,y).\,\bar{y}.\,\mathbf{0}$$

Instances of these processes can conduct arbitrarily many dialogues, yielding the same value in each. For example, with R as above, using $\Longrightarrow$ to indicate the transitive closure of $\longrightarrow$, and using $\sim$ as above to indicate application of some cleanup laws, it holds that

$$R\mid R\mid \text{TRUE} \Longrightarrow\sim P\mid P\mid \text{TRUE}$$

and

$$R\mid R\mid \text{FALSE} \Longrightarrow\sim Q\mid Q\mid \text{FALSE}.$$

Our second example is about the encoding of higher-order constructs in the π-calculus. Consider a higher-order language with constructs similar to those of the π-calculus but with the possibility of passing processes. In such a language one could write, for instance,

$$P \stackrel{\text{def}}{=} a(x).\,(x|x)|\bar{a}\langle R\rangle.\,Q$$

On the right-hand side of the composition, a process R is emitted along a; on the left-hand side, an input at a is expecting a process and then two copies of it will be run. Process P evolves as follows:

$$P \Longrightarrow R\mid R\mid Q$$

This behavior can be mimicked in the π-calculus as follows. The communication of the higher-order value R is translated as the communication of a private name that acts as a pointer to (the translation of) R and that the recipient can use to trigger a copy of (the translation of) R; the mapping from the higher-order language into the π-calculus is indicated with $[\![\,.\,]\!]$:

$$[\![P]\!] \stackrel{\text{def}}{=} a(x).\,(\bar{x}.\,\mathbf{0}|\bar{x}.\,\mathbf{0})|\boldsymbol{\nu} y\, \bar{a}\langle y\rangle.\,(Q|!y.\,R)$$

where name y does not occur elsewhere. It holds that

$$\begin{aligned} [\![P]\!] &\longrightarrow \boldsymbol{\nu} y\ (\bar{y}.\,\mathbf{0}|\bar{y}.\,\mathbf{0}|[\![Q]\!]|!y.\,[\![R]\!]) \\ &\longrightarrow\longrightarrow \boldsymbol{\nu} y\ (\mathbf{0}|\mathbf{0}|[\![Q]\!]|[\![R]\!]|[\![R]\!]|!y.\,[\![R]\!]) \\ &\sim\ [\![Q]\!]|[\![R]\!]|[\![R]\!] \end{aligned}$$

where, on the second line, $\longrightarrow\longrightarrow$ represents two consecutive reduction steps. On the third line, the same algebraic laws as above are used, plus a law for garbage-collecting an input-replicated process such as $!y.\,[\![R]\!]$ if the initial name y is restricted and does not occur elsewhere in the system.

The translation separates the acts of *copying* and of *activating* the value R; copying is rendered by the replication, and activation by communications along the pointer y. Here again, it is essential that the pointer y is initially private to the process emitting at a. This ensures that the following outputs at y are not intercepted by processes external to $[\![P]\!]$.

Names

In the π-calculus, names specify links. But what is a link? The calculus is not prescriptive on this point: the notion of a *link* is construed very broadly, and names can be put to very many uses. This point is important and deserves some attention. For example, names can be thought of as channels that processes use to communicate. Also, by syntactic means and using type systems, π-calculus names can be used to represent names of processes or names of objects in the sense of object-oriented programming. Further, although the π-calculus does not mention locations explicitly, often

when describing systems in π-calculus, some names are naturally thought of as locations. Finally, some names can be thought of as encryption keys as done in calculi that apply ideas from the π-calculus to computer security.

Types

A type system is, roughly, a mechanism for classifying the expressions of a program. Type systems are useful for several reasons: to perform optimizations in compilers; to detect simple kinds of programming errors at compilation time; to aid the structure and design of systems; to extract behavioral information that can be used for *reasoning* about programs. In sequential programming languages, type systems are widely used and generally well-understood. In concurrent programming languages, by contrast, the tradition of type systems is much less established.

In the π-calculus world, types have quickly emerged as an important part of its theory and of its applications, and as one of the most important differences with respect to CCS-like languages. The types that have been proposed for the π-calculus are often inspired by well-known type systems of sequential languages, especially λ-calculi. Also, type systems specific to processes have been investigated, for instance, for preventing certain forms of interferences among processes or certain forms of deadlocks.

One of the main reasons for which types are important for reasoning on π-calculus processes is the following. Although well-developed, the theory of the pure π-calculus is often insufficient to prove "expected" properties of processes. This is because a π-calculus programmer normally uses names according to some precise logical discipline (the same happens for the λ-calculus, which is hardly ever used untyped since each variable has usually an "intended" functionality). This discipline on names does not appear anywhere in the terms of the pure calculus, and therefore cannot be taken into account in proofs. Types can bring this structure back to light.

This point is illustrated with an example that has to do with *encapsulation*. Facilities for encapsulation are desirable in both sequential and concurrent languages, allowing one to place constraints on the access to components such as data and resources. The need of encapsulation has led to the development of abstract data types and is a key feature of objects in object-oriented languages. In CCS, encapsulation is given by the *restriction* operator. Restricting a channel x on a process P, written $\boldsymbol{\nu}x\, P$, guarantees that interactions along x between subcomponents of P occur without interference from outside. For instance, suppose one has two one-place buffers, `Buf1` and `Buf2`, the first of which receives values along a channel x and resends them along y, whereas the second receives on y and resends on z. They can be composed into a two-place buffer that receives on x and resends on z; thus: $\boldsymbol{\nu}y(\texttt{Buf1} \mid \texttt{Buf2})$. Here, the restriction ensures us that actions on y from `Buf1` and `Buf2` are not stolen by processes in the external environment. With the formal definitions of `Buf1` and `Buf2` at hand, one can indeed prove that the system $\boldsymbol{\nu}y\ (\texttt{Buf1} \mid \texttt{Buf2})$ is behaviorally equivalent to a two-place buffer.

The restriction operator provides quite a satisfactory level of protection in CCS, where the visibility of channels in processes is fixed. By contrast, restriction alone is often not satisfactory in the π-calculus, where the visibility of channels may change dynamically. Consider the situation in which several client processes cooperate in the use of a shared resource such as a printer. Data are sent for printing by the client processes along a channel p. Clients may also communicate channel p so that new clients can get access to the printer. Suppose that initially there are two clients

$$\begin{aligned} \texttt{C1} &= \overline{p}\langle j_1\rangle.\overline{p}\langle j_2\rangle \ldots \\ \texttt{C2} &= \overline{b}\langle p\rangle \end{aligned}$$

and therefore, writing P for the printer process, the initial system is

$$\boldsymbol{\nu}p\ \ (P \mid \texttt{C1} \mid \texttt{C2}).$$

One might wish to prove that `C1`'s print jobs represented by j_1 and j_2 are eventually received and processed in that order by the printer, possibly under some fairness condition on the printer scheduling policy. Unfortunately this is false: a misbehaving new client `C3` that has obtained p from `C2` can disrupt the protocol expected by P and `C1` just by reading print requests from p and throwing them away:

$$\texttt{C3} = p(j).\,p(j').\,\mathbf{0}.$$

In the example, the protection of a resource (the printer) fails if the access to the resource is transmitted, because no assumptions on the use of that access by a recipient can be made. Simple and powerful encapsulation barriers against the mobility of names can be created using type concepts familiar from the literature of typed λ-calculi. For instance, the misbehaving printer client `C3` can be recognized by distinguishing between the input and the output capabilities of a channel. It suffices to assign the input capability on channel p to the printer and the output capability to the initial clients `C1` and `C2`. In this way, new clients that receive p from existing clients will only receive the output capability on p. The misbehaving `C3` is thus ruled out as ill-typed, as it uses p for input.

The concept of a channel with direction can be formalized by means of type constructs, sometimes called the *input/output types*. They give rise to a natural subtyping relation, similar to those used for reference types in imperative languages. In the case of the π-calculus encodings of the λ-calculus, this subtyping validates the standard subtyping rules for function types. It is also important when modeling object-oriented languages, whose type systems usually incorporate some powerful form of subtyping.

In the λ-calculus, where functions are the unit of interaction, the key type construct is the arrow type. In the π-calculus names are the unit of interaction and therefore the key type construct is the *channel* (or *name*) type $\sharp\, T$. A type assignment $a : \sharp\, T$ means that a can be used as a channel to carry values of type T. As names can carry names, T itself can be a channel type. If one adds a set of *basic types*, such as integer or boolean types, one obtains the analog of the simply-typed λ-calculus, which is therefore called the *simply-typed π-calculus*. Type constructs familiar from sequential languages, such as those for products, unions, records, variants, recursive types, polymorphism, subtyping, and linearity, can be adapted to the π-calculus. Having recursive types, one may avoid basic types as initial elements for defining types. The calculus with channel, product, and recursive types is the *polyadic π-calculus* mentioned earlier on.

Beyond types inherited from the λ-calculus, several other type systems have been put forward that are specific to processes; they formalize common *patterns of interaction* among processes. Types may even serve as a specification of (parts of) the behavior of processes, as in *session types*. The behavioral guarantees that are guaranteed by types are typically *safety properties*, such as the absence of certain communication errors, of interferences, of deadlock, and information leakage in security protocols. Safety is expressed in the subject reduction theorem, a fundamental theorem of type theory stating the invariance of types under reduction. Type systems for *liveness properties* have been studied too, for instance for properties such as termination (the fact that computation in a concurrent system will eventually stop) and responsiveness (the fact that a process will eventually provide an answer along a certain name). However, liveness properties, already difficult enough to prove in the presence of concurrency, can be even harder to prove for mobile processes (in the sense that it may be difficult to design a type system capable of guaranteeing the property of interest while being expressive enough to handle most common programming idioms).

Theory

Fundamental for a theory of a concurrent language is a means of stating what the behavior of a process is, and what does it means for two behaviors to be equal. These notions in concurrency are usually treated via operational semantics. A brief and informal account of how these issues are treated in the π-calculus is given below, referring to [18] for details.

Traditionally, the operational semantics of a process algebra is given in terms of a *labeled transition system* describing the possible evolutions of a process. This contrasts with what happens in *term rewriting systems*, as based on an *unlabeled reduction system*. In the λ-calculus, probably the best known term-rewriting system, what makes a reduction system possible is that two terms having to interact are naturally in contiguous positions. This is not the case in process calculi, where interaction does not depend on physical contiguity. To put this another way, a *redex* of a λ-term is a subterm, while a "redex" in a process calculus is distributed over the term.

To allow a reduction semantics on processes, axioms for a *structural congruence* relation, usually written $\equiv$, are introduced prior to the reduction system, in order to break a rigid, geometrical view of concurrency; then reduction rules can easily be presented in which redexes are indeed subterms again.

The interpretation of the operators of the language emerges neatly with a reduction semantics, due to the compelling naturalness of each structural congruence and reduction rule. This is not quite the case in the labeled semantics, at least for process algebras expressing mobility: the manipulation of names and the side conditions in the rules are nontrivial and this can make it delicate understanding and justifying the choices made. However, if the reduction system is available, the correctness of the labeled transition system can be shown by proving the correspondence between the two systems.

On the other hand, the advantages of a labeled semantics appear later, when reasoning with processes. In a reduction semantics, the behavior of a process is understood relatively to a context in which it is contained and with which it interacts. Instead, with a labeled semantics every possible communication of a process can be determined in a direct way. This allows us to get simple characterizations of behavioral equivalences. Moreover, with a labeled semantics the proofs benefit from the possibility of reasoning in a purely structural way. Another possible problem with a reduction semantics is that it allows one to accommodate only limited forms of the choice operator. The conclusion is that both semantics are useful and that they integrate and support each other.

To see an example of the two semantics, consider the process

$$S \stackrel{\text{def}}{=} x(y).\,P|\overline{z}\langle w\rangle.\,R|\overline{x}\langle v\rangle.\,Q$$

A reduction semantics would only specify that S has a reduction

$$S \longrightarrow P\{{}^{v}\!/\!_{y}\}|\overline{z}\langle w\rangle.\,R|Q$$

To derive this, structural congruence is first used to bring the participant of the interaction into contiguous positions, using monoidal rules for parallel composition such as

$$T_1 \mid T_2 \equiv T_2 \mid T_1$$

$$T_1 \mid (T_2 \mid T_3) \equiv (T_1 \mid T_2) \mid T_3$$

with which the order of the components in parallel compositions can be modified. Then the rewriting rule

$$a(b).\,T|\overline{a}\langle u\rangle.\,T' \longrightarrow T\{{}^{u}\!/\!_{b}\}|T'$$

can be applied.

In contrast, a labeled transition system reveals, beside the above reduction, also the potentials for S to interact with the environment, namely the input and output transitions:

$$S \xrightarrow{x(y)} P|\overline{z}\langle w\rangle.\,R|\overline{x}\langle v\rangle.\,Q \quad S \xrightarrow{\overline{x}\langle v\rangle} x(y).\,P|\overline{z}\langle w\rangle.\,R|Q$$
$$S \xrightarrow{\overline{z}\langle w\rangle} x(y).\,P|R|\overline{x}\langle v\rangle.\,Q$$

(Note that, with a restriction in front of S, name x becomes private to S and the input and output actions along x are forbidden.) The rules for the labeled transition semantics are formalized following the style of Plotkin's Structured Operational Semantics, where the derivation proof of a transition is uniquely determined by the position, in the syntax of the term, of the prefixes consumed in the transition.

The notion of behavioral equivalence normally adopted in the π-calculus is *barbed congruence*. Its definition is couched in terms of a *bisimulation* game on reductions and a simple notion of *observation* on processes (for instance, a predicate revealing whether a process is capable of emitting a signal at some special name). Two π-terms are deemed barbed congruent if no difference can be observed between the processes obtained by placing them into an arbitrary π-context. The notion of observation has the flavor of the report of the successful outcome of an experiment, as in a theory of *testing*.

Barbed congruence has the advantage of being simple and robust, in that it can be applied to different calculi. Moreover, as its definition involves quantification over contexts, it gives a natural notion of equivalence, properly sensitive to the calculus under consideration (this is important when considering, for instance, type systems, as types implicitly limit the class of contexts in which a process may be placed; as a consequence, more equalities among processes hold). The quantification over contexts, however, makes the definition difficult to work with. This fact motivates the study of auxiliary notions of behavioral equivalences that afford tractable techniques. These *labeled* equivalences are based on direct comparison of the actions that processes can perform, rather than on the observation of arbitrary systems containing them. There are in fact several notions of labeled equivalence, each of which has characteristics that make it useful in some way. Examples are

late bisimilarity, *early bisimilarity*, and *open bisimilarity* [13, 18]. The labeled equivalences are not as robust as barbed congruence. For instance, they can be much too discriminating on refinements of the π-calculus with types.

A number of proof techniques for behavioral equalities on π-calculus processes have been developed. For instance, enhancements of the bisimulation proof method, sometimes called "up-to" techniques [17]. There has been also some work on the development of (semi-) automatic tools to assist in reasoning, though this remains an active research area. Other behavioral equivalences for the π-calculus have been studied, too, for instance *testing equivalence* [3].

The π-calculus has also a well-developed *algebraic* theory. Equational reasoning is a central technique in process calculus. In carrying out a calculation, appeal can be made to any axiom or rule sound for the equivalence in question.

In general, the equivalence of two processes is undecidable, indeed it is not even semi-decidable. On finite processes, however, axiomatizations may be possible. By an *axiomatization* of an equivalence on a set of terms, one means some equational axioms that, together with the rules of equational reasoning, suffice for proving all (and only) the valid equations. The rules of equational reasoning are reflexivity, symmetry, transitivity, and congruence rules that make it possible to replace any subterm of a process by an equivalent term. Here are examples of axioms for the π-calculus. These axioms are sound, in that the processes so equated are barbed congruent. See [18] for more details.

$$P + P = P$$

$$\boldsymbol{\nu} x\, \overline{x}\langle v\rangle .\, P = \mathbf{0}$$

$$\boldsymbol{\nu} x\, P = P \qquad \text{if } x \text{ does not occur in } P$$

$$x(y).\,P|\overline{z}\langle v\rangle .\,Q = x(y).\,(P|\overline{z}\langle v\rangle .\,Q) + \overline{z}\langle v\rangle .\,(x(y).\,P|Q) + [x=z]\tau .\,(P\{{}^{v}\!/\!{}_{y}\}|Q)$$

The first law is an idempotence law for choice. The second law shows that an output at a name x preceded by a restricted on x is a blocked process. The third law allows the removal a useless restriction. The final law, called *expansion*, explains the behavior of a parallel composition in terms of choice and matching.

A behavioral equivalence abstracts from internal action is sometimes called a *weak* equivalence, and one that does not a *strong* equivalence. The above laws are valid for strong barbed congruence, hence also for its weak counterpart. Here is an equality that is only valid for weak barbed congruence:

$$R|\boldsymbol{\nu} x\,(x(y).\,P|\overline{x}\langle v\rangle .\,Q) = R|\boldsymbol{\nu} x\,(P\{{}^{v}\!/\!{}_{y}\}|Q)$$

The law shows that an interaction between two processes along private names cannot be disturbed by other processes.

Variants and Extensions

In the π-calculus, communication between processes is synchronous – it is a handshake synchronization between two processes. Variants of the π-calculus have been proposed in which communication may be thought of as being asynchronous. The most distinctive feature of such extensions is that there is no continuation underneath the output prefix (i.e., all outputs are of the form $\overline{x}\langle v\rangle$), and only limited forms of choice are allowed. A major advantage of the asynchronous variants is that their implementation is simpler, and indeed most programming languages, or constructs for programming languages, inspired by π-calculus are asynchronous.

The ordinary π-calculus does not distinguish between channels, or ports, and variables: they are all names. In some variants, such a separation is made. The π-calculus also does not explicitly mention location or distribution of mobile processes. The issue of location and distribution is orthogonal. Several extensions, or variants, of the π-calculus have appeared with the goal of explicitly addressing distribution and all the associated phenomena. Ideas from π-calculus have contributed to the development of their theories. Examples of languages are the Distributed Join Calculus [6], the Distributed π-calculus [8], the Ambient Calculus [5], and Oz [19].

Related Entries

►Actors
►Bisimulation
►CSP (Communicating Sequential Processes)
►Process Algebras

Bibliographic Notes and Further Reading

The first paper on the π-calculus, [12], was written by Milner et al. A book that gives a gentle introduction to both CCS and the π-calculus, with an emphasis on motivating the interest in them, is [11]. A book with an in-depth treatment of the theory of the π-calculus and its main variants is [18]. A book with a focus on a distributed extension of the π-calculus is [9]. These textbooks may be consulted for details on the concepts outlined above, including type systems, variants of the π-calculus, behavioral theory, and comparison with higher-order languages.

Recent developments, not covered in the above books, include session types [20], type systems for deadlock-freedom, and lock-freedom such as [10]. Experimental typed programming languages, or proposals for typed programming languages, inspired by the π-calculus include Pict [14], Join [7], *XLang*, developed at Microsoft as a component of the *.NET* platform. An active research area is the application of concepts from the π-calculus to languages aimed for Web services (see, e.g., [4]), biology (see, e.g., [15, 16]), and security (see, e.g., [1, 2]).

Bibliography

1. Abadi M, Gordon AD (1999) A calculus for cryptographic protocols: the spi calculus. Inf Comput 148(1):1–70
2. Blanchet B, Abadi M, Fournet C (2008) Automated verification of selected equivalences for security protocols. J Log Algebr Program 75(1):3–51
3. Boreale M, De Nicola R (1995) Testing equivalence for mobile processes. Inf Comput 120:279–303
4. Carbone M, Honda K, Yoshida N (2007) Structured communication-centred programming for web services. In: Proceedings of the ESOP 2007, vol 4421, Lecture Notes in Computer Science. Springer, Heidelberg, pp 2–17, 2007
5. Cardelli L, Gordon AD (1998) Mobile ambients. In: Proceedings of the FoSSaCS'98, vol 1378, Lecture Notes in Computer Science. Springer, Heidelberg, pp 140–155, 1998
6. Fournet C, Gonthier G, Lévy J-J, Maranget L, Rémy D (1996) A calculus of mobile agents. In: Proceedings of the CONCUR'96, vol 1119, Lecture Notes in Computer Science. Springer, Heidelberg, pp 406–421, 1996
7. Fournet C, Gonthier G (2002) The join calculus: a language for distributed mobile programming. In: Summer School APPSEM 2000, vol 2395, Lecture Notes in Computer Science. Springer, Heidelberg, pp 268–332
8. Hennessy M, Riely J (1998) Resource access control in systems of mobile agents. In: Proceedings of the HLCL '98: High-Level Concurrent Languages, vol 16.3, ENTCS. Elsevier Science, 1998
9. Hennessy M (2007) A distributed pi-calculus. Cambridge University Press, New York
10. Kobayashi N (2006) A new type system for deadlock-free processes. In: Proceedings of the CONCUR'06, vol 4137, Lecture Notes in Computer Science. Springer, Bonn, pp 233–247, 2006
11. Milner R (1999) Communicating and mobile systems: the $^{1}/_{4}$-Calculus. Cambridge University Press, Cambridge
12. Milner R, Parrow J, Walker D (1993) A calculus of mobile processes, (Parts I and II). Inf Comput 100:1–77
13. Milner R, Parrow J, Walker D (1992) Modal logics for mobile processes. Theor Comput Sci 114:149–171
14. Pierce BC, Turner DN (2000) Pict: a programming language based on the pi-calculus. In: Proof, Language and Interaction: Essays in Honour of Robin Milner. MIT Press, Cambridge
15. Priami C, Quaglia P, Romanel A (2009) Blenx static and dynamic semantics. In: Proceedings of the CONCUR'09, vol 5710, Lecture Notes in Computer Science. Springer, Bologna, pp 37–52, 2009
16. Regev A, Panina EM, Silverman W, Cardelli L, Shapiro EY (2004) Bioambients: an abstraction for biological compartments. Theor Comput Sci 325(1):141–167
17. Sangiorgi D (1995) On the bisimulation proof method. In: Proceedings of the MFCS'95, vol 969, Lecture Notes in Computer Science. Springer, pp 479–488, 1995
18. Sangiorgi D, Walker D (2001) The $^{1}/_{4}$-calculus: a theory of mobile processes. Cambridge University Press, Cambridge
19. Smolka G (1994) The definition of kernel Oz. Research Report RR-94-23, Deutsches Forschungszentrum für Künstliche Intelligenz, Kaiserslautern, Germany
20. Vasconcelos VT (2009) Fundamentals of session types. In: SFM 2009 School, vol 5569, Lecture Notes in Computer Science. Springer, Heidelberg, pp 158–186

Pipelining

Pipelining [1] is a parallel processing strategy in which an operation or a computation is partitioned into disjoint stages. The stages must be executed in a particular order (could be a partial order) for the operation or computation to complete successfully. Each stage is implemented as a component which could be a hardware device or a software thread. When a stage completes, it becomes available to do other work. Parallelism results from the execution of a sequence of operations

or computations so that at any given time several components of the sequence are under execution and each one of these is at a different stage of the pipeline.

Pipelining is pervasive in today's machines. Processor control units and arithmetic units are typically pipelined. Also, programs take advantage of pipelined parallelism by partitioning computations into stages.

Related Entries

►Cray Vector Computers
►Floating Point Systems FPS-120B and Derivatives
►Fujitsu Vector Computers
►Stream Programming Languages

Bibliography

1. Kogge PM (1981) The Architecture of Pipelined Computers. Hemisphere Publishing Corporation, New York

Place-Transition Nets

►Petri Nets

PLAPACK

John A. Gunnels
IBM Corp, Yorktown Heights, NY, USA

Definition

`PLAPACK` is a software library for dense parallel linear algebra computations.

Discussion

Introduction

The PLAPACK (van de Geijn, Robert., *Using PLAPACK*, pp. 1, 3, 4, 6, 43, 59, 85, ©1997 Massachusetts Institute of Technology, by permission of the MIT Press) library is a modern, dense parallel linear algebra library that is extensible, easy to use, and available under an open source license. It is designed to be user-friendly while offering competitive performance when compared to the more traditionally constructed ScaLAPACK library.

The PLAPACK Project

Motivation PLAPACK's design was motivated by the observation that the parallel implementation of most dense linear algebra operations is a relatively well-understood process. Nonetheless, the creation of general-purpose, high-performance parallel dense linear algebra libraries is severely hampered by the fact that translating the sequential algorithms to a parallel code requires careful manipulation of indices and parameters describing the data, its distribution to processors, and/or the communication required. The creators of PLAPACK believe that these details make such parallel programming highly error prone and that this overhead stands in the way of the parallel implementation of more sophisticated algorithms.

An Object-Based Approach The PLAPACK library is constructed so as to allow the user to express their parallel algorithms in a very concise manner. It does this, largely, through the encapsulation of data in objects. To achieve this, PLAPACK adopted an "object-based" (or object-oriented) approach to programming, inspired by efforts including the Message Passing Interface (MPI) library [1], and the PETSc library [2]. In object-based programming, all of the data related to an operand (matrix, vector, etc.) is cohesive, maintained in a data structure that describes the object. Further, the descriptor is interrogated or modified only indirectly, through the use of accessor and modifier functions.

Objects and Communications

Object Types In the PLAPACK library there are several different types of linear algebra objects. These object types have a one-to-one correspondence with the manner in which they are distributed.

The PLAPACK object types include:

- `PLA_Matrix`: This object is distributed as a two-dimensional object, in a particular block cyclic fashion referred to as the Physically Based Matrix Distribution (PBMD).
- `PLA_Mvector`: The multivector is distributed over the compute fabric in blocked fashion, viewing the grid as one dimensional. It is used as a vector operand and as an intermediate form for copying data from one object (form) to another.

A multivector object may consist of one or more "columns" of data.

- `PLA_Pmvector`: The projected multivector object is distributed in the same manner as (a set of) rows or columns of a matrix. This object is often duplicated in one dimension of the computational grid. For example, a column-oriented projected multivector is distributed across rows of the processor mesh and may be duplicated across processor columns. PLAPACK code often uses this object type to create buffers for communication.
- `PLA_Mscalar`: The multiscalar object is most conveniently thought of as a local matrix. It is not distributed, but replicated on all processors.

The Special Role of the Multivector Multivectors can be thought of as a number of vectors, side-by-side, distributed over the entire processor mesh, where that mesh is viewed as a one-dimensional array of processors. The multivector can be used as a first-class object or as an intermediate form in various kinds of communication. An example of the former would be the redistribution of the panel from a blocked LU decomposition. In taking the object from, typically, a column of processors to the entire processor mesh, more computation per unit of time can be applied to the object. The latter use of the multivector appears in the `PLA_Copy` and `PLA_Reduce` routines. Copying data from one object to another or reducing (e.g., summing) data from several objects into another potentially involves complex mapping and communication, but these are encapsulated in the `PLA_Copy` and `PLA_Reduce` routines, respectively. The copy routine is used to seemlessly move data between objects of potentially differing distributions and types while the reduction operator is used to do the same, but is logically limited to using a duplicated object as its source. The central nature of the multivector in the copy operation is illustrated in Fig. 1.

Referencing (Sub)Objects

Views In order to both facilitate concise algorithmic encodings and to eliminate many indexing errors, PLAPACK makes heavy use of linear algebra object (matrix, vector, etc.) "views." In this context, a view is, abstractly, a subset of an object (a submatrix is the most commonly used view) with its own descriptor.

The promotion of the view to a first-class entity in PLAPACK stems from the fact that many common linear algebra algorithms such as parallel BLAS routines, solvers, and factorization routines, can be expressed in terms of windows into matrices and vectors. Thus, the use of the view (implicit or explicit) is common. By formalizing the use of the view and localizing the functionality related to creating and modifying these objects, PLAPACK both removes the need for the user to create this functionality and places it in a well-tested routine. As the routines used for views are heavily tested and the level of abstraction used to create views is high, relative to explicit indexing, this functionality is intended to reduce the opportunity for programmer error and to make such errors, when introduced, easier to locate.

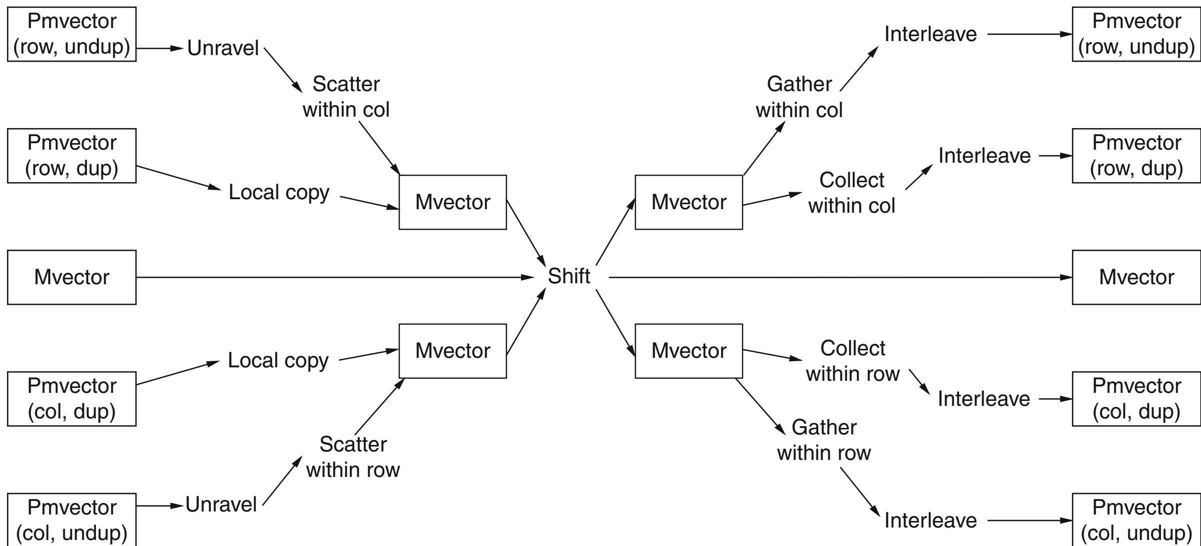
A view is identical to an object created via the `PLA_Obj_create` function except that it references the same data as the parent object or view from which it was derived. Thus, a change to the values in the data that the view translates to a corresponding change in the data that the parent object describes. It is legitimate in PLAPACK to derive views from views.

In PLAPACK, a view of a linear algebra object is created by a call to the `PLA_Obj_view` routine. The user specifies the dimensions of the new view as well as its offset relative to the parent object and this routine creates a new view into an existing object. Routines for shifting (sliding) views as well as specialized subview operators, partitioning matrices into multiple submatrices or coalescing submatrices into a single-view object are also supplied in the PLAPACK library.

Distributing and Interfacing with Parallel Operands

Physically Based Matrix Distribution PLAPACK employs an object distribution called the "Physically Based Matrix Distribution" (PBMD). This distribution scheme is based on the thesis that the elements of vectors are typically associated with data of physical significance, and it is therefore their distribution to nodes that is directly related to the distribution of the problem to be solved. From this point of view, a matrix (discretized operator) merely represents the relation between two vectors (discretized spaces):

$$y = Ax \quad (1)$$



PLAPACK. Fig. 1 A systematic approach to copying (duplicated) (projected) (multi)vectors from and to (duplicated) (projected) (multi)vectors

PLAPACK partitions x and y, and assigns portions of these vectors to nodes. The matrix A should then be distributed to nodes in a fashion consistent with the distribution of the vectors.

To employ PBMD, one must start by describing the distribution of the vectors, here, x and y, to nodes, after which the matrix distribution is induced (derived) by the vector distribution as illustrated in Fig. 2.

The Application Programming Interface PLAPACK was architected under the assumption that applications will employ the library to:

- Create PLAPACK vector, multivector, and matrix objects.
- Fill the entries in these PLAPACK objects.
- Perform a series of parallel linear algebra operations.
- Query elements of the updated PLAPACK objects.

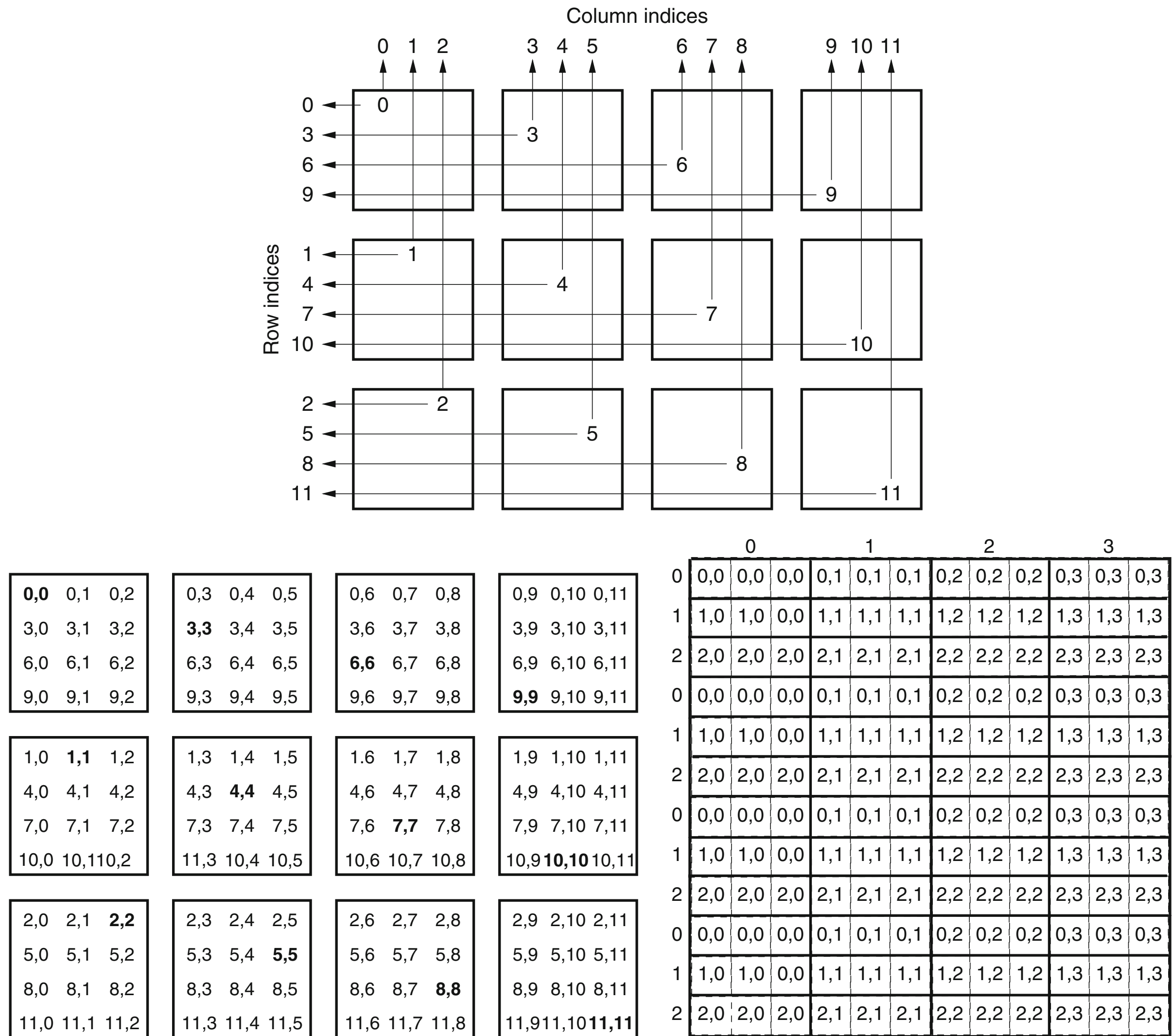
Many applications are inherently set up to generate numerous sub-vector, sub-multivector, or submatrix contributions to global linear algebra objects. The contributions of these applications can be viewed as dimensional "sub-objects." However, frequently they are also partial sums of the global linear algebra objects, in which case the contribution must be added to existing entries. One approach for parallelizing these applications is to partition the numerous sub-object computations among processors. Such a parallelization of the applications' linear algebra object generation phase produces sub-vector, sub-multivector, and submatrix partial sums that may or may not be entirely local with respect to the data distribution of the global linear algebra objects.

In order to fill or retrieve entries in a linear algebra object, an application enters the API-active state of PLAPACK. In this state, objects can be opened in a shared-memory-like mode, which allows an application to either fill or retrieve individual elements, sub-vectors, or subblocks of linear algebra objects. The PLAPACK application interface supports filling global linear algebra objects with sub-object partial sums.

An Illustrative Example

Parallel Cholesky Factorization Fig. 3 is a PLAPACK implementation of a blocked, parallel Cholesky factorization. It utilizes level-3 BLAS routines locally and is a simple, but efficient implementation of this algorithm. The representation is compact and relatively easy to explain to those familiar with the underlying sequential algorithm (see, e.g., the encyclopedia entry on libflame).

The algorithm proceeds by applying (local) Cholesky factorization to the upper left corner of the matrix of interest, updating the remainder of the matrix accordingly, and reducing the active area (the part that will be further updated) of the matrix. The same algorithm



PLAPACK. Fig. 2 Inducing a matrix distribution from vector distributions. *Top*: Here each box represents a node of a 3×4 mesh. The sub-vectors of x and y are assigned to the mesh in column-major order. Thus the number in the box represents the index of the sub-vector assigned to that node. By projecting the indices of y to the left, the distribution of the matrix row-blocks of A is established. By projecting the indices of x to the top, the distribution of the matrix column-blocks of A is determined. *Bottom-left*: The resulting distribution of the subblocks of A is given where the indices refer to the indices of the subblocks of A. *Bottom-right*: The same information, except now from the matrix point of view. The figure shows the matrix partitioned into subblocks, with the indices in the subblocks indicating the node to which the subblock is mapped

is applied to this active part of the matrix until there is no active matrix remaining and the algorithm is complete.

In the implementation in Fig. 3a, the `while` loop continues until the active portion of the matrix no longer exists. That condition is signaled when the top-left (`A_TL`) quadrant of the matrix (the part that has been completely factored) is the entire matrix. Since the matrix is assumed to be (globally) square this condition can be tested by comparing the (global) length of the entire matrix to the factored (sub)section.

The calls to `PLA_Obj_split_size` are used to determine the matrix dimensions of the top-left quadrant of the active matrix that resides on a single

```
int Chol_blk_var3( PLA_Obj A, int nb_alg )
{
  PLA_Obj ATL=NULL,   ATR=NULL,  A00=NULL, A01=NULL, A02=NULL,
          ABL=NULL,   ABR=NULL,  A10=NULL, A11=NULL, A12=NULL,
                                 A20=NULL, A21=NULL, A22=NULL;
  PLA_Obj MINUS_ONE=NULL, ZERO=NULL, ONE=NULL;
  int b;
  /* Create constants -1, 0, 1 of the appropriate type */
  PLA_Create_constants_conf_to( A, &MINUS_ONE, &ZERO, &ONE );
  PLA_Part_2x2( A,    &ATL, &ATR,
                      &ABL, &ABR,     0, 0, PLA_TL );
  while ( PLA_Obj_length( ATL ) < PLA_Obj_length( A ) ){
    /* Determine how big of a block A11 exists within one block */
    PLA_Obj_split_size(ABR, PLA_SIDE_TOP, &size_top, &owner_top );
    PLA_Obj_split_size(ABR, PLA_SIDE_LEFT, &size_left, &owner_left );
    b = min( (min(size_top, size_left), nb_alg );
    PLA_Repart_2x2_to_3x3(ATL, /**/ ATR,   &A00, /**/ &A01, &A02,
                       /* ********** */   /* ***************** */
                                           &A10, /**/ &A11, &A12,
                        ABL, /**/ ABR,     &A20, /**/ &A21, &A22,
                        b, b, PLA_BR );
    /*------------------------------------------------------------*/
    /* Update A_11 <- L_11 = Chol. Fact.( A_11 ) */
    PLA_Local_chol( PLA_LOWER_TRIANGULAR, A11 );
    /* Update A_21 <- L_21 = A_21 inv( L_11' ) */
    PLA_Trsm( PLA_SIDE_RIGHT, PLA_LOWER_TRIANGULAR,
              PLA_TRANSPOSE,  PLA_NONUNIT_DIAG, ONE, A11, A21 );
    /* Update A_22 <- A_22 - L_21 * L_21' */
    PLA_Syrk( PLA_LOWER_TRIANGULAR, MINUS_ONE, A21, ONE, ABR );
    /*------------------------------------------------------------*/
    PLA_Cont_with_3x3_to_2x2(&ATL, /**/ &ATR,    A00, A01, /**/ A02,
                                                 A10, A11, /**/ A12,
                           /* *********** */  /* *************** */
                            &ABL, /**/ &ABR,     A20, A21, /**/ A22,
                            PLA_TL );
  }
  PLA_Obj_free( &ATL ); PLA_Obj_free( &ATR );
  PLA_Obj_free( &ABL ); PLA_Obj_free( &ABR );
  PLA_Obj_free( &A00 ); PLA_Obj_free( &A01 ); PLA_Obj_free( &A02 );
  PLA_Obj_free( &A10 ); PLA_Obj_free( &A11 ); PLA_Obj_free( &A12 );
  PLA_Obj_free( &A20 ); PLA_Obj_free( &A21 ); PLA_Obj_free( &A22 );
  PLA_Obj_free( &MINUS_ONE ); PLA_Obj_free( &ZERO ); PLA_Obj_free( &ONE );
  return PLA_SUCCESS;
}
```

PLAPACK. Fig. 3 An efficient variant of Cholesky factorization implemented in the PLAPACK API

processor. This is done so that an efficient, local (no communication) Cholesky factorization can be performed via `PLA_Local_chol`.

The `PLA_Repart_2x2_to_3x3` and `PLA_Cont_with_3x3_to_2x2` calls create multiple views from a single object and coalesce multiple views into a single object, respectively. The comment bars in the code are visual cues as to the semantics of these functions. In the case of `PLA_Repart_2x2_to_3x3` function, `ABR` is split into four subviews. In order to unambiguously carry out this operation only the size of the `A11` subview needs to be specified (here, that size is `b` by `b`). The `PLA_Cont_with_3x3_to_2x2` function coalesces nine views into four. This is done in order to update the view of the active part of the matrix, shrinking it.

Computational work is performed through the calls to `PLA_Local_chol`, which performs the sequential Cholesky factorization, `PLA_Trsm` whose purpose is to perform a parallel triangular solve with multiple right-hand-sides, and `PLA_Syrk`, a parallel version of the symmetric rank-k update.

Related Entries

►BLAS (Basic Linear Algebra Subprograms)
►LAPACK
►libflame
►ScaLAPACK

Bibliographic Notes and Further Reading

The PLAPACK notation and APIs were used as the basis for those employed in the FLAME [3] project and the libflame library [4].

The first papers that outlined the ideas that PLAPACK is based upon were published in 1997 [5] and a book has been published that describes PLAPACK in detail [6].

Bibliography

1. Gropp W, Lusk E, Skjellum A (1994) Using MPI. The MIT Press, Cambridge, MA
2. Balay S, Gropp W, McInnes LC, Smith B (1996) PETSc 2.0 users manual. Technical report ANL-95/11, Argonne National Laboratory, Cass Avenue Argonne
3. Gunnels JA, Gustavson FG, Henry GM, van de Geijn RA (2001) FLAME: formal linear algebra methods environment. ACM T Math Software 27(4):422–455
4. Van Zee FG (2009) Libflame: the complete reference. http://www.lulu.com/content/5915632/
5. Alpatov P, Baker G, Edwards HC, Gunnels J, Morrow G, Overfelt J, van de Geijn R (1997) PLAPACK: parallel linear algebra package design overview. In: Proceedings of the 1997 ACM/IEEE conference on supercomputing, ACM, New York, pp 29
6. van de Geijn RA (1997) Using PLAPACK. The MIT Press, Cambridge, MA

PLASMA

Jack Dongarra, Piotr, Luszczek
University of Tennessee, Knoxville, TN, USA

Definition

Parallel Linear Algebra Software for Multi-core Architectures (PLASMA) is a free and open-source software library for numerical solution of linear equation systems on shared memory computers with multi-core processors. In particular, PLASMA is designed to give high efficiency on homogeneous multi-core processors and multi-socket systems of multi-core processors. As of today, majority of such systems are on-chip symmetric multiprocessors with classic super-scalar processors as their building blocks augmented with short-vector SIMD extensions (such as SSE and AltiVec). PLASMA is available for download from the PLASMA Web site (To obtain the PLASMA library logon to: http://icl.cs.utk.edu/plasma/).

Discussion

The emergence of multi-core microprocessor designs marked the beginning of a forced march toward an era of computing in which research applications must be able to exploit parallelism at a continuing pace and unprecedented scale [1]. To answer this challenge PLASMA redesigns LAPACK [2] and ScaLAPACK [3] for current and future multi-core processor architectures. To achieve high performance on this type of architectures, PLASMA relies on tile algorithms, which provide fine granularity parallelism. The standard linear algebra algorithms can then be represented as a Directed Acyclic Graph (DAG) [4] where nodes represent tasks, either panel factorization or update of a block-column, and edges represent dependencies among them. Figure 1 shows a small DAG for a tile QR factorization. Tasks of the same type (implemented by the same function) share the same color of nodes.

Moreover, the development of programming models that enforce asynchronous, out of order scheduling of operations is the concept used as the basis for the definition of a scalable yet highly efficient software framework for computational linear algebra applications. In PLASMA, parallelism is no longer hidden inside Basic Linear Algebra Subprograms (BLAS – for more details see http://www.netlib.org/blas/) but is brought to the fore to yield much better performance. Each of the one-sided tile factorizations presents unique challenges to parallel programming. Cholesky factorization is represented by a DAG with relatively little work required on the critical path. LU and QR factorizations have corresponding dependency pattern between the nodes of the DAG. These two factorizations exhibit much more severe scheduling and constraints than the Cholesky factorization. Currently, PLASMA schedules tasks statically while balancing the trade-off between load balancing and data reuse. PLASMA's performance depends

PLASMA. Fig. 1 Task DAG for tile QR factorization

strongly on tunable execution parameters, the outer and inner blocking sizes, that trade off utilization of different system resources. The outer block size (NB) trades off parallelization granularity and scheduling flexibility with single core utilization, while the inner block size (IB) trades off memory load with extra flops due to redundant calculations. Tuning PLASMA consists of finding (NB, IB) pairs that maximize the performance depending on the matrix size and the number of cores.

In terms of numerical calculations, the Cholesky factorization represents one case in which a well-known LAPACK algorithm can be easily reformulated in a tiled fashion. Each operation that defines an atomic step of the LAPACK algorithm can be broken into a sequence of tasks where the same algebraic operation is performed on smaller portions of data, i.e., the tiles. In most of the cases, however, the same approach cannot be applied and novel algorithms must be introduced. For the LU and QR [5, 6] factorizations, algorithms based on the updating factorizations are used to reformulate the algorithms as was done for out-of-core solvers. Updating factorization methods can be used to derive tiled algorithms for LU and QR factorizations that provide very fine granularity of parallelism and the necessary flexibility that is required to exploit dynamic scheduling of the tasks. It is worth noting, however, that, as in any case where such fundamental changes are made, trade-offs have to be taken into account. For instance, in the case of the LU factorization the tiled algorithm replaces partial pivoting with block pairwise pivoting which results in, on average, slightly worse numerical stability. On the positive side, the current analysis of PLASMA's tile algorithms suggests that they are numerically backward stable and the ongoing research aims to determine if they are stable.

PLASMA provides routines to solve dense general systems of linear equations, symmetric positive definite systems of linear equations, and linear least squares problems using LU, Cholesky, QR, and LQ factorizations. Real arithmetic and complex arithmetic are supported in both single precision and double precision.

Related Entries

►LAPACK
►Linear Algebra, Numeric
►ScaLAPACK

Bibliography

1. Herb Sutter A (2005) Fundamental turn toward concurrency in software. Dr. Dobb's J 30(3):202–210
2. Anderson E, Bai Z, Bischof C, Blackford LS, Demmel JW, Dongarra J, Du Croz J, Greenbaum A, Hammarling S, McKenney A, Sorensen D (1992) LAPACK Users' guide. SIAM, Philadelphia
3. Blackford LS, Choi J, Cleary A, D'Azevedo E, Demmel J, Dhillon I, Dongarra J, Hammarling S, Henry G, Petitet A, Stanley K, Walke D, Whaley RC (1997) ScaLAPACK users' guide. SIAM, Philadelphia
4. Christofides N (1975) Graph theory: an algorithmic approach. Academic, New York
5. Buttari A, Langou J, Kurzak J, Dongarra JJ (2006) Parallel tiled QR factorization for multicore architectures. In: PPAM'07: Seventh international conference on parallel processing and applied mathematics, Gdansk, Poland, pp 639–648
6. Buttari A, Langou J, Kurzak J, Dongarra JJ (2007) Lapack working note 191: A class of parallel tiled linear algebra algorithms for multicore architectures. Technical Report UT-CS-07-600, Electrical Engineering and Computer Sciences Department, University of Tennessee

PMPI Tools

Bernd Mohr
Forschungszentrum Jülich GmbH, Jülich, Germany

Synonyms

MPI introspection interface; MPI monitoring interface; MPI profiling interface; PMPI

Definition

MPI is the predominant parallel programming model used in scientific computing which is demonstrated to work on the largest computer systems available. With PMPI there exists a standardized, and therefore portable, MPI monitoring interface. Since it was part of the MPI standard from the beginning, a multitude of tools for MPI programming exist. In the MPI standard, PMPI is called the *Profiling Interface*, which is a little bit misleading as it allows to intercept every MPI call made in a parallel program and can be used for all kind of tools, e.g., tracing or validation tools, and not just profiling ones.

Discussion

Section 8 of the MPI standard [1] describes the so-called *Profiling Interface* of MPI. The objective of this interface is to ensure that it is relatively easy for authors of MPI tools to interface their codes to MPI implementations on different machines. Since MPI is a machine independent standard with many different implementations, it is unreasonable to expect that the authors of profiling tools for MPI will have access to the source code that implements MPI on any particular machine. It was therefore necessary to provide a portable mechanism by which the implementers of such tools can

collect whatever performance information they wish without access to the underlying implementation. This is accomplished by dictating that an implementation of the MPI functions must provide a mechanism through which all of the MPI-defined functions may be accessed with a name shift. Thus, all of the MPI functions (which normally start with the prefix "MPI") should also be accessible with the prefix "PMPI." This can be done by using weak symbols or simply by compiling each MPI function twice but with the different name.

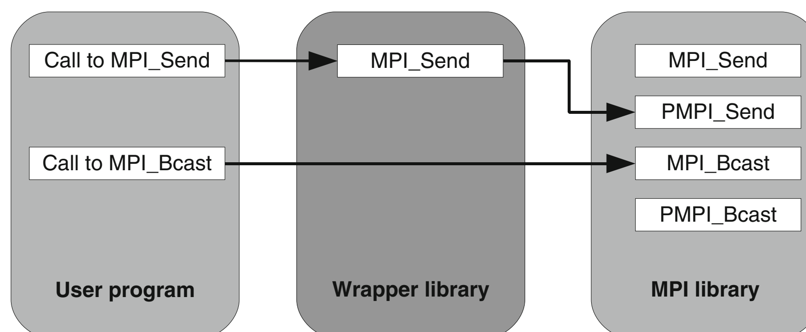
It is now possible for the implementer of an MPI tool to intercept all of the MPI calls that are made by the user program by implementing a library of so-called *wrapper functions*, which implement the same interface as the MPI function they intercept. In the wrapper function one can collect whatever information is required for the tool's task before or after calling the underlying MPI implementation (through its name shifted entry points) to achieve the needed effects. One also has access to all parameter values passed to and returned from the MPI function.

The MPI standard also requires that the standard MPI library be built in such a way that the inclusion of MPI functions can be achieved one at a time. This is necessary so that the authors of the wrapper library need only define those MPI functions which they wish to intercept, references to any others being fulfilled by the normal MPI library. So in order to use an MPI tool, the program is first linked to the tool's wrapper library and only then to the MPI library, as shown in Fig. 1.

Simple Usage Example

The basic structures of any PMPI tool is a collection of wrapper routines that collect the necessary data for each MPI call. At the end of the program execution, e.g., in the MPI_Finalize wrapper, the collected data is potentially aggregated across all processes using MPI communication and then written to disk or presented to the user. Figure 2 shows a very simplistic version of an MPI performance tool that only counts the number of messages sent via MPI_Send.

A more realistic version of this wrapper library example would not only count messages, but much more performance metrics, e.g., the time spent in executing the function. Professional MPI performance tools also determine the distribution of the recorded metrics over the receiver rank or the amount of bytes sent. Of course, to get a complete recording of MPI messages all functions that send messages need to be wrapped, i.e., all variants of MPI_Send (MPI_Bsend, MPI_Ssend, MPI_Rsend, …) and some other functions like MPI_Start. The latter function belongs to the group of functions implementing MPI-persistent communication. To monitor these messages, it is also necessary to track all MPI requests in the tool (implemented by wrapping all MPI calls that create, modify, or delete the opaque MPI request objects). Finally, to support all programming languages the MPI standard supports, both the C and the Fortran version of the wrappers need to be implemented. As the MPI standard defines over 300 functions, this means implementing



PMPI Tools. Fig. 1 Linking of user program, wrapper library, and MPI library. The user program calls MPI_Send and MPI_Bcast. If the user program is first linked to the tool's wrapper library and then to the MPI library, the tool only intercepts MPI_Send calls which uses PMPI_Send to implement the actual sending of the message. Calls to unwrapped functions (here MPI_Bcast) directly call the corresponding function of the MPI library

```
#include <stdio.h>
#include "mpi.h"

static int numsend = 0;

int MPI_Send(void *buf, int count, MPI_Datatype type,
             int dest, int tag, MPI_Comm comm) {
  numsend++;
  return PMPI_Send(buf, count, type, dest, tag, comm);
}

int MPI_Finalize() {
  int me;
  PMPI_Comm_rank(MPI_COMM_WORLD, &me);
  printf("%d sent %d messages.\n", me, numsend);
  return PMPI_Finalize();
```

PMPI Tools. Fig. 2 Simplistic wrapper library example. The wrapper for `MPI_Send` increments a global variable which was initialized to zero. The wrapper for `MPI_Finalize` prints how many messages this particular rank had sent. As `MPI_Finalize` is executed by every rank of the program, each rank is reporting its own result. Inside the wrapper, the rank is determined via `PMPI_Comm_rank` in order to avoid invoking the corresponding wrapper function

over 600 wrapper functions; which is why, many MPI tool projects use wrapper generation tools to simplify the implementation of the wrapper library.

Performance Measurement Tools Based on PMPI

Many MPI performance tools exist and all of them use the PMPI interface to implement the tool. Two widely accepted, portable, and scalable examples are FPMPI-2 and mpiP. They are both open-source, so that the reader can easily download them and study their implementation and usage.

FPMPI-2 is a portable, open-source, very lightweight, and scalable MPI profiling library from Argonne National Laboratory [3]. For each MPI function, it provides the average and the maximum of the sum of metrics over all processes in a single textual output file. The provided metrics are the number of calls and the total execution time of each MPI function. FPMPI-2 has two special features that set it apart from other MPI profiling libraries: For communication functions, it records not only the amount of data transferred but also the distribution of the message sizes (by using an adaptive 32 bins histogram). Secondly, it optionally tries to determine the actual synchronization time within blocking MPI calls by replacing the actual MPI implementation with a logically equivalent implementation using busy-wait on the user level, allowing the blocking time to be estimated.

mpiP is also a portable, more complete, but still scalable MPI profiling library originating from Lawrence Livermore National Laboratory but meanwhile maintained as an open-source project on sourceforge.net [4]. It provides also the number of calls, total execution time, bytes sent for each MPI function, and optionally for each MPI call-site. Captured call-paths are determined by a user-specified traceback level. Also, it provides MPI I/O statistics where applicable. The collected data is not aggregated across the processes but is collected in a scalable manner into one single output file which contains the complete data for all processes.

Besides these two MPI profiling libraries, there are many more performance tools for MPI that utilize the PMPI interface. Some other tools worth mentioning are:

- Integrated Performance Monitoring (**IPM**) [5, 6] is a portable, open-source toolkit with a focus on providing a low-overhead performance profile of the performance aspects and resource utilization in an MPI program. The level of provided detail is selectable at runtime and presented through a variety of text and web reports. Aside from overall performance, reports are available for load balance,

task topology, bottleneck detection, and message size distributions.

- There are also many vendor-specific MPI tools that are commercial products and typically only run on the system sold by the vendor. Examples here are Intel's **Trace Collector** or Cray's **CrayPat**.
- **VampirTrace** [7, 8] is a portable, open-source tool for collecting detailed traces of MPI programs in the *Open Trace Format* (OTF), which can be analyzed and visualized by the commercial Vampir trace browser. The latest version is also distributed as part of OpenMPI.
- The **TAU** Parallel Performance System [9, 10] is a very portable and versatile, open-source toolkit for performance analysis of parallel programs. Among many things, it supports profiling and tracing of MPI programs based on the PMPI interface.
- The **Scalasca** toolset [11, 12] provides call-path profiling and event tracing of MPI, OpenMP, and hybrid MPI/OpenMP programs. Its focus is on extreme scalability: In summer 2009, they reported the first successful tracing experiments done on a 2,94,912 core BlueGene/P system. It is, together with VampirTrace, the only MPI tools that does complete MPI communicator tracking enabling a detailed collective communication analysis. Finally, Scalasca is currently the only tool supporting a detailed analysis of MPI 2.0 RMA functions [13].

Verification Tools Based on PMPI

While most of the PMPI tools measure and analyze the performance of MPI programs, there are certainly other uses for the PMPI interface. For example, it also possible to verify the correct and portable use of the MPI standard. MPI is widely used to write parallel programs, but it does not guarantee full portability between different MPI implementations. When an application runs without any problems on one platform but crashes or gives wrong results on another platform, developers tend to blame the compiler/architecture/MPI implementation. However, in some cases, the problem is a subtle programming error in the application undetected on the first platform. Finding this bug can be a very strenuous and difficult task. The solution is to use an automated tool designed to check the correctness of MPI applications during runtime. Examples of such violations are the introduction of irreproducibility, deadlocks, and incorrect management of resources such as communicators, groups, datatypes, etc., or the use of non-portable constructs.

A PMPI-based correctness checking tool has the advantage that it can be used with any standard-conforming MPI implementation and may thus be deployed on any development platform available to the programmer. Although high-quality MPI implementations detect some of these errors themselves, there are many cases where they do not give any warnings. For example, non-portable implementation-specific behavior is not indicated by the implementation itself, nor are checks performed that would decrease the performance too much, such as consistency checks. What is worse is that MPI implementations tolerate quite a few errors without warnings or crashing, by simply giving wrong results.

MARMOT [14] is one example of a PMPI-based tool designed to detect correctness and portability problems during runtime. For all tasks that require a global view, e.g., deadlock detection or the control of the execution flow, MARMOT uses an additional process, the so-called debug server. Each client registers at the debug server, which in turn gives its clients the permission for execution in a round robin way. In order to ensure that this additional debug process is transparent to the application, MARMOT maps `MPI_COMM_WORLD` to a MARMOT communicator containing only the application processes. Since all other communicators are derived from `MPI_COMM_WORLD` they automatically exclude the debug server process. Everything that can be checked locally, e.g., verification of arguments, such as tags, communicators, or ranks, is performed by the clients. Additionally, the clients and the debug server use MPI internally to transfer information. Unfortunately, this server/client architecture inflicts a bottleneck, thus affecting the scalability and performance of the tool, especially for communication-intensive applications.

UMPIRE [15] is a second example for a PMPI-based MPI correctness-checking tool. Like MARMOT, it performs checks on a rank-local and global level. UMPIRE uses time-out mechanism and dependency graphs to detect deadlocks. Further errors detected by UMPIRE include wrong ordering of collective communication

calls within a communicator, mismatching collective call operations, or errant writes to send buffers before non-blocking sends are completed. UMPIRE does extensive resource tracking. Consequently it is able to unearth resource leaks. For instance, applications can repeatedly create opaque objects without freeing them, leading to memory exhaustion, or there can be lost requests due to overwriting of request handles.

Future Directions

While PMPI certainly has been successfully used for many tools, it also comes with a few limitations: One problem is that it requires users to relink their application. More importantly, the PMPI interface only supports one tool at a time. The user cannot use multiple tools in a single run of their application – a significant limitation since not only application programmers might want to perform multiple performance analyses concurrently, but also tool builders cannot use existing tools, such as an MPI profiler, to evaluate the quality of the implementation of new tools, such as a correctness checker. Further, tool builders cannot easily create tool modules since functionality cannot be split into individual PMPI-accessing layers that could be reused by other tools. Thus, it discourages code reuse during tool development.

To overcome these limitations, researchers at LLNL propose a new infrastructure called **P^{N}MPI** [16, 17]. P^{N}MPI allows users to dynamically load and execute one or more PMPI-based tools concurrently. This is accomplished by linking the P^{N}MPI infrastructure into applications by default. P^{N}MPI then transforms the wrapper libraries included in the PMPI tools into a single tool stack. Once initialized, P^{N}MPI redirects any MPI routine executed by the application into this dynamically created stack and independently calls each tool that contains a wrapper for the routine. This eliminates the need to create a separate executable for each tool and to run each tool separately. Since P^{N}MPI is lightweight by design, it can be included in the default build process thereby removing the need for recompilation to include or remove a tool. P^{N}MPI also provides tool interaction functionality through services in the P^{N}MPI core. Thus, separate modules can now implement common tool functionality to improve code reuse, modularity, and flexibility as well as tool interoperability. Possible usage scenarios for the P^{N}MPI infrastructure are the transparent use of tracing and profiling tools together, or efficient MPI debugging by combining deterministic reply mechanisms with MPI checker libraries like UMPIRE.

Another proposal, the so-called Universal MPI Correctness Interface (**UniMCI**) [18], specifically targets the efficient integration of performance and verification tools. Like P^{N}MPI, it is based on the observation that using multiple tools simultaneously can help pinpoint issues faster, especially the combination of a tracing and a correctness tool can provide the history that leads to a correctness event and add further details to a detected error. Thus, it simplifies the identification of the root cause of an error. Unless P^{N}MPI, it also provides a solution for a generic, portable coupling of any performance *host* tool with a correctness *guest* tool provided they both follow the UniMCI interface.

This interface provides two functions for each MPI call: one to analyze the initial arguments of the MPI call, called *pre* check, and one to analyze the results of the MPI call, called *post* check. All runtime MPI checkers will need an analysis of the initial MPI call arguments to detect errors in the given arguments, which is invoked with the pre check function of UniMCI. The post check is usually needed for MPI calls that create resources, e.g., `MPI_Isend`, which create new requests. MPI correctness tools have to add these new resources to their internal data structures, in order to be aware of all valid handles and their respective state. By splitting the analysis of the MPI call into two parts, it is possible to return the result of the check to the host tool before the actual MPI call is issued, which is important to guarantee that errors are handled before the application might crash. Results of checks are returned with additional interface functions that have to be issued after each check function. A simple correctness message record is used to return problems detected by the guest tool. Future versions of the interface will contain extensions and rules to handle asynchronous correctness checking tools and multi-threaded applications.

Related Entries

►Debugging
►Formal Methods–Based Tools for Race, Deadlock, and Other Errors
►Metrics

▸MPI (Message Passing Interface)
▸Parallel Tools Platform
▸Performance Analysis Tools
▸Periscope
▸Profiling
▸Scalasca
▸TAU
▸Tracing
▸Vampir

Bibliography

1. Louis Turcotte (1994) Message passing interface forum: MPI: a message-passing interface standard. IJSA Special issue on MPI 8(3/4)
2. Accelerated strategic computing initiative: the ASC SMG2000 benchmark code (2001) https://asc.llnl.gov/computing_resources/purple/archive/benchmarks/smg/
3. Argonne national laboratory: the FPMPI-2 MPI profiling library (2007) http://www-unix.mcs.anl.gov/fpmpi/
4. Vetter J, Chambreau C (2007) The mpiP MPI profiling library. http://mpip.sourceforge.net/
5. Skinner D, Wright N, Fuerlinger K, Yelick K (2009) Allan Snavely: integrated performance monitoring. http://ipm-hpc.sourceforge.net/
6. Fürlinger K, Skinner D (August 2009) Capturing and visualizing event flow graphs of MPI applications. In: Workshop on productivity and performance (PROPER 2009) in conjunction with Euro-Par 2009
7. Jurenz M, Knüpfer A, Brendel R, Lieber M, Doleschal J, Mickler H, Hackenberg D, Heyde H, Müller M (2009) VampirTrace. http://www.tu-dresden.de/zih/vampirtrace/
8. Knüpfer A, Brunst H, Doleschal J, Jurenz M, Lieber M, Mickler H, Müller M, Nagel W (2008) The vampir performance analysis tool-set. In: Tools for high performance computing. Springer, Stuttgart, pp 139–155
9. Shende S, Malony A (2009) Tuning and analysis utilities. http://tau.uoregon.edu/
10. Shende S, Malony A (2006) The TAU parallel performance system. Int J High Perform Comput Appl 20:287–331, SAGE Publications
11. Wolf F, Mohr B, Wylie B, Geimer M (2009) Scalable performance analysis of large-scale applications. http://www.scalasca.org/
12. Geimer M, Wolf F, Wylie BJN, Mohr B (2009) A scalable tool architecture for diagnosing wait states in massively parallel applications. Parallel Comput 35:375–388
13. Hermanns M-A, Geimer M, Mohr B, Wolf F (2009) Scalable detection of MPI-2 remote memory access inefficiency patterns. In: Proc. of the 16th European PVM/MPI users' group meeting (EuroPVM/MPI), volume 5759 of Lecture Notes in Computer Science, Springer, Espoo, Finland, pp 31–41
14. Krammer B, Bidmon K, Müller M, Resch M (2003) MARMOT: an MPI analysis and checking tool. In: Procedings of PARCO 2007, volume 13 of Advances in Parallel Computing, Elsevier, pp 493–500
15. Vetter J, de Supinski B (2000) Dynamic software testing of MPI applications with umpire. In: Proceedings of the 2000 ACM/IEEE conference on supercomputing (CDROM), Article No. 51
16. Schulz M, de Supinski B (2007) PnMPI tools: a whole lot greater than the sum of their parts. In: Proceedings of supercomputing, ACM/IEEE 2007 conference
17. Schulz M (2006) A flexible and dynamic infrastructure for MPI tool interoperability. International conference on parallel processing (ICPP)
18. Hilbrich T, Jurenz M, Mix H, Brunst H, Knüpfer A, Müller M, Nagel W (2010) An interface for integrated MPI correctness checking. In: Chapman B et al (eds) Advances in parallel computing. Parallel computing: from multicores and GPU's to petascale, vol 19. IOS Press, pp 693–700

Pnetcdf

▸NetCDF I/O Library, Parallel

Point-to-Point Switch

▸Buses and Crossbars

Polaris

Rudolf Eigenmann
Purdue University, West Lafayette, IN, USA

Synonyms

Parallelization

Definition

Polaris is the name of a parallelizing compiler and research compiler infrastructure. Polaris performs source-to-source translation; programs written in the Fortran language are converted into restructured Fortran programs – typically annotated with directives that express parallelism. Polaris was created in the mid-1990s at the University of Illinois and was one of the most advanced freely available tools of its kind.

Discussion

Introduction

Polaris aimed at pushing the forefront of automatic parallelization (see ▸Parallelization, Automatic) and, at the same time, providing the research community with an infrastructure for exploring program analysis and transformation techniques. Polaris followed several earlier projects with similar goals, a key distinction being that the development of this new compiler was driven by real applications. While the benchmarks that prompted earlier parallelizing compiler research were typically small, challenging program kernels, the Polaris project was preceded and motivated by an effort to manually parallelize the most realistic application program suite available at that time – the Perfect Benchmarks [5]. This effort identified a number of beneficial transformation techniques [3], the automation of which constituted the Polaris project.

Polaris was able to improve the state of the art in automatic parallelization significantly. Earlier autoparallelizers were measured to parallelize to a significant degree only 2 of the 13 Perfect Benchmarks. By contrast, Polaris achieved success in six of them – increasing the parallelization success rate in this class of science and engineering applications from less than 20% to nearly 50%.

Polaris was originally developed at the University of Illinois from 1992 to 1995 [1], with significant extensions made at Purdue University and Texas A&M University, in later project phases. Among the examples and predecessors were several parallelizers developed at the University of Illinois and Rice University. An important contemporary was the Stanford SUIF project [6]. Among the more recent, related compiler infrastructures are Rose [4], LLVM [8], and Cetus [2].

The architecture of Polaris exhibits the classical structure of an autoparallelizer. A number of program analysis and transformation passes detect parallelism and map it to the target machine. The passes are supported by an internal program representation (IR) that represents the source code being transformed and offers a range of program manipulation functions.

Detecting Parallelism

Polaris includes the program analysis and transformation techniques that were found to be most important in a prior manual parallelization project [3]. At the core of any autoparallelizer is a data-dependence detection mechanism. Data dependences prevent parallelism, making dependence-removing techniques essential parts of an autoparallelizer's arsenal. To this end, Polaris includes passes for data privatization, reduction recognition, and induction variable substitution. The compiler focuses on detecting fully parallel loops, which have independent iterations and can thus be executed simultaneously by multiple processors. For the basics of the following techniques, see ▸Parallelization, Automatic.

Data-dependence test. Typical data-dependence tests detect whether or not two accesses to a data array in two different loop iterations could reference the same array element. The detection works well where array subscripts are linear – of the form $a * i + b * j$, where a,b are integer constants and i,j are index variables of enclosing loops. The Polaris project developed new dependence analysis techniques that are able to detect parallelism in the presence of symbolic and nonlinear array subscript expressions. For example, in the above expression, if a is a variable, the subscript is considered symbolic; if the term i^2 appears, the subscript is nonlinear. If the compiler cannot determine the value of a symbolic term, it cannot assume it is linear. Hence, symbolic and nonlinear expressions are related. In real programs it is common for expressions, including array subscripts, to contain symbolic terms other than the loop indices. Through nonlinear, symbolic data-dependence testing, Polaris was able to parallelize several important programs that previous compilers could not.

Privatization. Data privatization [11] is a key enabler of improved parallelism detection. A privatization pattern can be viewed as one where a variable, say t, is being used as a temporary storage during a loop iteration. The compiler recognizes this pattern in that t is first defined (written) before used (read) in the loop iteration. By giving each iteration a separate copy of the storage space for t, accesses to t in multiple iterations do not conflict. Polaris extended the basic technique so that it could detect entire arrays that could be privatized. For example, the following loop can be parallelized after privatizing the array *tmp*. (the notation *tmp(1:m)* means "the array elements from index 1 to m")

```
DO i=1,n
```

```
    tmp(1:m)  = a(1:m)+b(1:m)
    c(1:m)    = tmp(1:m)+sqrt
                (tmp(1:m))
ENDDO
```

Privatization gives each processor a separate instance of *tmp*. Without this transformation, each loop iteration would write to and read from the same *tmp* variable, creating a data dependence and thus inhibiting parallelization.

Implementing array privatization is substantially more complex than the basic scalar privatization technique. The compiler must determine the sections of each array that are being defined and used in a loop. If each element of an array that is being used has previously been defined in the same loop iteration, the array is privatizable. More sophisticated analysis may privatize sections of arrays that fit this pattern. To do so, the compiler analysis must be able to combine array accesses into sections. As array subscript expressions may contain symbolic terms, these operations must be supported by advanced symbolic manipulation functions, which is one of Polaris' strengths.

Reduction recognition. This transformation is another important enabler of parallelization. Similar to the way Polaris extended the privatization technique from scalars to arrays, it extended reduction recognition [9]. The following loop shows an example *array reduction* (sometimes referred to as irregular or histogram reduction). Different loop iterations modify different elements of the *hist* array. The pattern of modification is not important; any two loop iterations may modify the same or different array elements.

```
DO i=1,n
    val = <some computation>
    hist(tab(i)) = hist(tab(i))
                 + val
ENDDO
```

Polaris recognizes array reductions by searching for loops that contain statements with the following pattern: An assignment statement has a right-hand-side expression that is the sum of the left-hand-side plus a term not involving the reduction array. A loop may have several such statements, but the reduction array must not be used in any other statement of the loop.

Because two iterations may access the same element, the compiler has to assume a possible dependence. Reduction parallelization takes advantage of the mathematical property that sum operations can be reordered (even under limited-precision computer arithmetic, reordering is usually valid, although not always). A reduction loop can be executed in parallel like this: Each processor performs the sum operations over the assigned loop iteration space on a private copy of the original reduction array (*hist*, in the above example). At the end of the loop, the local reduction arrays from all participating processors are summed into the original reduction array.

Reduction patterns are common in science and engineering applications. Array reduction parallelization was a key enabler of parallel performance in a number of important loops in the Perfect Benchmarks.

Induction variable substitution. Induction variable substitution eliminates dependences by replacing an arithmetic sequence by a closed-form computation. In the following loop, the induction variable *ind* forms a sequence.

```
ind = ind0
DO i=1,n
    ind = ind + k
    a(ind) = 0
ENDDO
```

In the basic form of an induction variable, the next value in the sequence is generated by adding a constant to the previous value. The term k could be an integer constant or a loop-invariant expression. The closed-form expression for the sequence is $ind0+i^{*}k$. By substituting this expression into the array subscript, $a(ind0+i^{*}k)$, the induction statement can be removed and the dependence on the previous value disappears.

Polaris extended this well-known transformation to a more general form, where k can be nonconstant, such as another induction variable. For example, if the expression k is the loop index ($ind = ind + i$), the closed form becomes $ind0+i^{*}(i+1)/2$.

The closed form of a generalized induction variable usually contains nonlinear terms. Parallelization in the presence of such terms requires the application of nonlinear data-dependence tests. Further complication arises when the induction variable is used after the loop. In this case, the compiler must compute and assign

the last value. In the above example, it would insert the statement *ind = ind0+n*k* after the loop. Such *last-value assignment* will be correct if assignment to the induction variable is guaranteed in all iterations. Polaris makes use of symbolic program analysis techniques to make this guarantee where possible.

Advanced Program Analysis

Symbolic analysis. In addition to powerful transformations, key to Polaris' performance is the availability of advanced symbolic analysis and manipulation utilities. For example, when performing generalized induction variable substitution, the compiler needs to evaluate sum expressions, such as $\sum_1^n j = n(n+1)/2$.

Polaris' symbolic range analysis technique is able to determine symbolic value ranges that program variables may assume during execution. The technique looks at assignment statements, loop statements, and condition statements to determine constraints on the value range of variables. After an assignment, the left-hand-side variable is known to have the newly given value or symbolic expression. Within a loop, the loop variable is guaranteed to be between the loop bounds. In the `then` clause of an `if` statement, the `if` condition is guaranteed to hold (and not hold in the `else` clause). For example, the analysis can determine that inside the following loop the inequality $1 \leq i \leq n$ holds and the loop accesses the array elements from position 2 to $n+1$.

```
DO i=1,n
  a(i+1)=0
ENDDO
```

Polaris created powerful tools for compilation passes to manipulate and reason with symbolic ranges, including comparison, intersection, and union operations. All of the described parallelization techniques depend on the availability of symbolic analysis.

Interprocedural analysis. Because structuring a program into subroutines is an important software engineering principle, the ability of compilers to analyze programs across procedure calls is crucial. Advanced parallelizers, such as Polaris, attempt to find parallelism in outer loops. Larger parallel regions can better amortize the overheads associated with parallel execution, as will be discussed later. However, outer loops tend to encompass subroutine calls, making the application of the described techniques difficult. Furthermore, interprocedural analysis is important even for code sections that do not contain subroutine calls. For many optimization decisions, the compilation passes must collect information from across the program. For example, symbolic analysis will find the value ranges of program variables in the entire program and propagate them to the subroutines where needed.

The needs for interprocedural operation are different for each compiler technique. Creating specialized techniques for each optimization pass can be prohibitively expensive. Instead, Polaris includes a capability to expand subroutines inline. By default, small (by a configurable threshold) subroutines are expanded in place of their call statements. This capability obviates the need for specialized interprocedural analysis in most common cases. The drawback of subroutine inline expansion is code growth. Depending on the subroutine calling structure, a code expansion of an order of magnitude is possible. While Polaris has demonstrated that significant additional parallelism can be found this way, the needed compilation time can grow substantially. To address this issue, Polaris also included an interprocedural data access analysis framework [7] as a basis for unified interprocedural parallelism detection.

Mapping Parallel Computation to the Target Machine

Mapping parallel computation to the target machine has two objectives. First, the parallelism uncovered by an autoparallelizer may not necessarily fit the model of parallel execution by the eventual machine platform. Additional transformations of the parallel execution or the program's data space may be needed. Second, almost all program transformations incur overheads. Privatization uses additional storage; reduction parallelization adds computation (e.g., summing up local reduction arrays); induction variable substitution creates expressions of higher strength (e.g., addition is replaced by multiplication); starting/ending parallel loops incur *fork/join* overheads (e.g., communicating to the participating processors what to do and warming up their cache). Advanced optimizing compilers make use of a performance model to estimate the overheads and to decide which transformations best to apply.

Scheduling parallel execution. Polaris detects loops that are dependence-free and marks them as fully parallel loops. To express parallel execution, it inserts OpenMP (openmp.org) directives, leaving the code generation up to the target platform's OpenMP compiler. A simple parallel loop in OpenMP looks like this.

```
!$OMP PARALLEL DO PRIVATE(t)
   DO i=1,n
      t = a(i)+b(i)
      c(i) = t + t*t
   ENDDO
```

The "OMP" directive expresses that the loop is to be scheduled for parallel execution and the data element *t* must be placed in private storage. By default, the OpenMP compiler assigns the loop iteration space to the available parallel threads (or cores) in chunks. For example, on an eight-core platform, the first core would usually execute the first $n/8$ iterations, etc. Polaris can influence this choice via OpenMP "Schedule" clauses. For example, if the compiler detects that the amount of computation per loop iteration is irregular, it may choose a "dynamic" schedule, which maps iterations to available processors at runtime.

To determine profitability of parallel execution, Polaris applies a simple test. It estimates the size of a loop by considering the number of statements and the number of iterations. If the size can be determined and is below a configurable threshold, the compiler leaves the loop in its original, serial form. While the creation of advanced performance models is an important research area, this simple test worked well in practice.

Data placement. The data model is of further importance. Polaris assumes that all non-private variables are shared. All processors simply refer to the data in the same way the original, serial code does. In the above example, all processors participating in the execution of the parallel loop see the arrays a, b, and c in the same way and have direct access. The variable *t* is stored in processor-private memory. Depending on the architecture, private storage may be actual processor-local memory or it may simply be a processor-private slice of the global address space.

Current multicore architectures implement this shared-memory model in hardware and are thus suitable targets for Polaris. Another important class of parallel computers is not: Many high-performance computer platforms have a distributed memory model. Data need to be partitioned and distributed onto the different memories. Processors do not have direct access to data placed on other processors' memories; explicit communication must be inserted to read and write such data. Autoparallelizers, including Polaris, have not yet been successful in targeting this machine class. Explicit parallel programming by the software engineer is needed.

Internal Organization

Polaris is organized into a number of program analysis and transformation passes, which make use of the functionality for program manipulation offered by the abstract internal program representation (IR). The IR represents the Fortran source program in a way that corresponds to the original code closely. The *Syntax Tree* has a structure that reflects the program's subroutines, statements, and expressions. The compiler passes see the IR in an abstract form – as a C++ class hierarchy; they perform all operations, such as finding properties of a statement or inserting an expression, via access functions.

The objects of the IR class hierarchy are organized in lists, which are traversable with several iteration methods. A typical compiler pass would begin by obtaining a list of Compilation Units (subroutines), traverse them to the desired subroutine, and obtain a reference to the list of statements of that subroutine. Next, the statement list would be traversed to the desired point, where the statement's properties and expressions could be obtained. Various "filters" are available that let pass writers iterate over objects of a specific type, only. For example, for some loop analysis pass, it may be desirable to skip from loop to loop, ignoring the statements in between them. Among the most advanced IR functions are those that allow a pass to traverse the IR until a certain statement or expression pattern is found.

A key design principle of Polaris was that these functions keep the IR consistent at all times. It would not be possible to insert a statement without properly connecting it to the surrounding scope or rename a variable without properly updating the symbol table. These bookkeeping operations are performed internally to the extent possible; they are not visible in the IR's access functions. This design offers the compiler researcher a

convenient, high-level interface. It was one reason for Polaris' popularity as a compiler infrastructure.

The implementation of the compiler consists of approximately 300,000 lines of C++ code. Forty-five percent of the code represents the IR with its access and manipulation functions; 55% implements the compilation passes.

Uses of Polaris

The primary use of Polaris was as an autoparallelizer and compiler infrastructure in the research community. Many contributions to autoparallelization technology have been made using Polaris as an implementation and evaluation test bed. Polaris supported other applications as well. Its ability to perform source-to-source transformations made it a good platform for writing program instrumentation passes. A simple such pass might insert calls to a timing subroutine at the beginning and end of every loop, producing a tool that can create loop profiles. Other uses of Polaris included the creation of functionality that measures the maximum possible parallelism in a program, predicts the parallel performance of application programs, and creates profiles of detected dependences.

Eighteen years after it was first conceived, Polaris continues to be distributed to the research community. The last download was recorded in 2010, in the same month this entry was written.

Challenges and Future Directions

Polaris was able to advance autoparallelization technology to the point where one in two science and engineering programs can be profitably executed in parallel on shared-memory machines. Nonnumerical programs and distributed-memory computer architectures are not yet amenable to this technology and remains an elusive research goal. In pursuing this goal, the increasing complexity of the compiler is a challenge. Learning the underlying theory and realizing its implementation is highly time-consuming for both the researchers exploring new analysis and transformation passes and the engineers developing production-strength compiler products.

One of the severe limitations of Polaris – and autoparallelization in general – is the lack of information that can be gathered from a program at compile time. Both the detection of parallelism and the mapping to the target architecture depend on knowledge of information that may be available only from the program's input data set or the target platform. Therefore, many optimization decisions cannot be made at compile-time or can only be made by the compiler making guesses. Guesses are only legal where they do not affect the correctness of the transformed program. Therefore, compilers often must make *conservative assumptions*, which may limit the degree of optimization. Future compilers will increasingly need to merge with runtime techniques that gather information as the program executes and perform or tune optimizations dynamically. Polaris explored one aspect of this area with *runtime data-dependence techniques.* These techniques detect at runtime whether or not a loop is dependence-free and choose between serial and parallel execution [10].

Related Entries

►Banerjee's Dependence Test
►Code Generation
►Dependence Analysis
►GCD Test
►HPF (High Performance Fortran)
►Loop Nest Parallelization
►Modulo Scheduling and Loop Pipelining
►Omega Test
►Parallelization, Automatic
►Parallelization, Basic Block
►Pipelining
►Speculative Parallelization of Loops
►Run Time Parallelization
►Scheduling Algorithms
►Semantic Independence
►Speculation, Thread-Level
►Trace Scheduling
►Unimodular Transformations

Bibliography

1. Blume W, Doallo R, Eigenmann R, Grout J, Hoeflinger J, Lawrence T, Lee J, Padua D, Paek Y, Pottenger B, Rauchwerger L, Tu P (December 1996) Parallel programming with Polaris. IEEE Comput 29(12):78–82
2. Dave C, Bae H, Min S-J, Lee S, Eigenmann R, Midkiff S (2009) Cetus: a source-to-source compiler infrastructure for multicores. IEEE Comput 42(12):36–42
3. Eigenmann R, Hoeflinger J, Padua D (January 1998) On the automatic parallelization of the perfect benchmarks. IEEE Trans Parallel Distrib Syst 9(1):5–23

4. Quinlan DJ et al Rose compiler project. http://www.rose-compiler.org/
5. Berry M et al (1989) The perfect club benchmarks: effective performance evaluation of supercomputers. Int J Supercomput Appl 3(3):5–40
6. Hall MW, Anderson JM, Amarasinghe SP, Murphy BR, Liao S-W, Bugnion E, Lam MS (December 1996) Maximizing multiprocessor performance with the SUIF compiler. Computer, pp 84–89
7. Hoeflinger J, Paek Y, Yi K (2001) Unified interprocedural parallelism detection. Int J Parallel Program 29(2):185–215
8. Lattner C, Adve V (2004) Llvm: a compilation framework for lifelong program analysis & transformation. In: CGO'04: Proceedings of the international symposium on code generation and optimization. IEEE Computer Society, Washington, DC, p 75
9. Pottenger B, Eigenmann R (1995) Idiom recognition in the polaris parallelizing compiler. In: Proceedings of the 9th ACM international conference on supercomputing, Barcelona
10. Rauchwerger L, Padua D (1995) The LRPD test: speculative run-time parallelization of loops with privatization and reduction parallelization. In: PLDI'95: Proceedings of the ACM SIGPLAN 1995 conference on programming language design and implementation. ACM, New York, pp 218–232
11. Tu P, Padua D (August 1993) Automatic array privatization. In: Proceedings of the 6th workshop on languages and compilers for parallel computing, vol 768, Lecture notes in computer science, pp 500–521

Polyhedra Scanning

►Code Generation

Polyhedron Model

Paul Feautrier[1], Christian Lengauer[2]
[1]CNRS École Normale Supérieure de Lyon, Lyon Cedex 07, France
[2]University of Passau, Passau, Germany

Synonyms

Polytope model

Definition

The polyhedron model (earlier known as the polytope model [21, 37]) is an abstract representation of a loop program as a computation graph in which questions such as program equivalence or the possibility and nature of parallel execution can be answered. The nodes of the computation graph, each of which represents an iteration of a statement, are associated with points of $\mathbb{Z}^n$. These points belong to polyhedra, which are inferred from the bounds of the surrounding loops. In turn, these polyhedra can be analyzed and transformed with the help of linear programming tools. This enables the automatic exploration of the space of equivalent programs; one may even formulate an objective function (such as the minimum number of synchronization points) and ask the linear programming tool for an optimal solution. The polyhedron model has stringent applicability constraints (mainly to FOR loop programs acting on arrays), but extending its limits has been an active field of research. Beyond autoparallelization, the polyhedron model can be useful in many situations which call for a program transformation, such as in memory or performance optimization.

Discussion

The Basic Model

Every compiler must have representations of the source program in various stages of elaboration, as for instance by character strings, abstract syntax trees, control graphs, three-address codes, and many others. The basic component of all these representation is the statement, be it a high-level language statement or a machine instruction. Unfortunately, these representations do not meet the needs of an autoparallelizer simply because parallelism does not occur between statements, but between statement executions or *instances*. Consider:

for $i = 0$ to $n-1$ do
$S: \quad a[i] = 0.0$
od

It makes no sense to ask whether S can be executed in parallel with itself; in this case, parallelism depends both on the way S accesses memory and on the way the loop counter i is updated at each iteration.

A loop program must therefore be represented as a set of instances, its *iteration domain*, here named E. Each instance has a distinct name and consists in the execution of the related statement or instruction, depending on the granularity of the analysis. This set is finite, in the case of a terminating program, or infinite, in the case of a reactive or streaming system.

However, this is not sufficient to specify the object program. One needs to know in which order the

instances are executed; E must be ordered by some relation $\prec$. If $u, v \in E$, $u \prec v$ means that u is executed before v. Since an operation cannot be executed before itself, $\prec$ is a strict order. It is easy to see that the usual control constructs (sequences, loops, conditionals, jumps) are compact ways of defining $\prec$. It is also easy to see that, in a sequential program, two arbitrary instances are always ordered: one says that, in this case, $\prec$ is a total order. Consideration of an elementary parallel program (in OpenMP notation):

```
#pragma omp parallel sections
   S1
#pragma omp section
   S2
#pragma omp end parallel sections
```

shows that S_1 may be executed before or after or simultaneously with S_2, depending on the available resources (processors) and the overall state of the target system. In that case, neither $S_1 \prec S_2$ nor $S_2 \prec S_1$ are true: one says that $\prec$ is a partial order. As an extreme case, an embarassingly parallel program, in which instances can be executed in any order, has the empty execution order. Therefore, one may say that parallelization results in replacing the total execution order of a sequential program by a partial one, under the constraint that the outcome of the program is not modified. This in turn raises the following question: *Under which conditions are two programs with the same iteration domain but different execution orders equivalent?*

Since program equivalence is undecidable in general, one must be content with conservative answers, i.e., with sufficient but not necessary equivalence conditions. The usual approach is based on the concept of *dependences* (see also the ►Dependences entry in this encyclopedia). Assuming that, given the name of an instance u, one can characterize the sets (or supersets) of read and written memory cells, $\mathcal{R}(u)$ and $\mathcal{W}(u)$, u and v are in dependence, written $u\,\delta\,v$, if both access some memory cell shared by them and at least one of them modifies it. In symbols: $u\,\delta\,v$ if at least one of the sets $\mathcal{R}(u) \cap \mathcal{W}(v)$, $\mathcal{W}(u) \cap \mathcal{R}(v)$ or $\mathcal{W}(u) \cap \mathcal{W}(v)$ is not empty. The concept of a dependence was first formulated by Bernstein [8]. One can prove that two programs are equivalent if dependent instances are executed in the same order in both.

► **Aside**. Proving equivalence starts by showing that, under Bernstein's conditions, two independent consecutive instances can be interchanged without modifying the final state of memory. In the case of a terminating program, this is done by specifying a succession of interchanges that convert one order into the other without changing the final result. The proof is more complex for nonterminating programs and depends on a fairness hypothesis, namely, that every instance is to be executed eventually. One can then prove that the succession of values assigned to each variable – its history – is the same for both programs. One first shows that the succession of assignments to a given variable is the same for both programs since they are in dependence, and, as a consequence, that the assigned values are the same, provided all instances are deterministic, i.e., return the same value when executed with the same arguments (see also the ►Bernstein's Conditions in this encyclopedia).

To construct a parallel program, one wants to remove all orderings between independent instances, i.e., construct the relation $\delta \cap \prec$, and take its transitive closure. This execution order may be too complex to be represented by the available parallel constructs, like the parallel sections or the parallel loops of OpenMP. In this case, one has to trade some parallelism for a more compact program.

It remains to explain how to name instances, how to specify the index domain of a program and its execution order, and how to compute dependences. There are many possibilities, but most of them ask for the resolution of undecidable problems, which is unsuitable for a compiler. In the polyhedron model, sets are represented as *polyhedra* in $\mathbb{Z}^n$, i.e., sets of (integer) solutions of systems of affine inequalities (inequalities of the form $A\,x \le b$, where A is a constant matrix, x a variable vector, and b a constant vector). It so happens that these sets are the subject of a well-developed theory, (integer) linear programming [43], and that all the necessary tools have efficient implementations.

The crucial observation is that the iterations of a regular loop (a Fortran DO loop, or a Pascal FOR loop, or restricted forms of C, C++ and Java FOR loops) are represented by a segment (which is a one-dimensional polyhedron), and that the iterations of a regular loop nest are represented by a polyhedron

with as many dimensions as the nest has loops. Consider, for instance, the first statement of the loop program in Fig. 1a. It is enclosed in two loops. The instances it generates can be named by stating the values of i and j, and the iteration domain is defined by the constraints:

$$1 \le i \le n, \;\; 1 \le j \le i + m$$

which are affine and therefore define a polyhedron. In the same way, the iteration domain of the second statement is $1 \le i \le n$. For better readability, the loop counters are usually arranged from outside inward in a vector, the *iteration vector* of the instance.

Observe that, in this representation, n and m are parameters, and that the size of the representation is independent of their values. Also, the upper bound of the loop on j is not constant: iteration domains are not limited to parallelepipeds.

The iteration domain of the program is the disjoint union of these two polyhedra, as depicted in Fig. 1b with dependences. (The dependences and the right side of the figure are discussed below.) To distiguish the several components of the union, one can use statement labels, as in:

$$E = \{\langle S_1, i, j\rangle \mid 1 \le i \le n, 1 \le j \le i + m\} \cup \{\langle S_2, i\rangle \mid 1 \le i \le n\}$$

The execution order can be deduced from two observations:

- In a program without control constructs, the execution order is the *textual order*. Let $u <_{\text{txt}} v$ be true if u occurs before v in the program text.

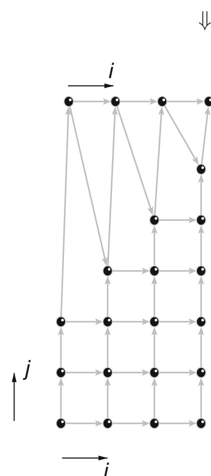
```
for i = 1 to n do
   for j = 1 to i+m do
S1 :  A(i,j) = A(i−1,j) + A(i,j−1)
   od
S2 :  A(i,i+m+1) = A(i−1,i+m) + A(i,i+m)
od
```

a Source loop nest

```
for t = 0 to m+2∗n−1 do
   parfor p = max(0,t−n+1) to min(t,⌈(t+m)/2⌉) do
      if 2∗p = t+m+1 then
   S2 :  A(p−m,p+1) = A(p−m−1,p) + A(p−m,p)
      else
   S1 :  A(t−p+1,p+1) = A(t−p,p+1) + A(t−p+1,p)
      fi
   od
od
```

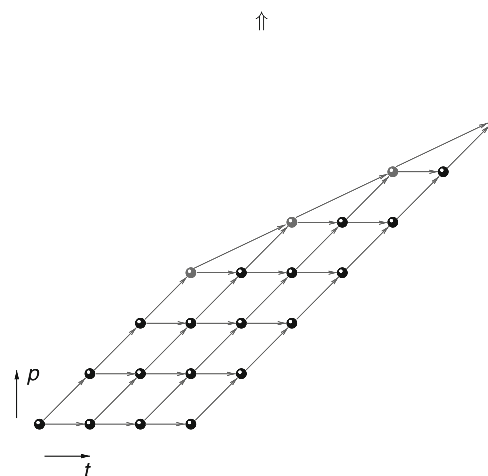
d Target loop nest

⇓ ⇑



b Source iteration domain

⟹



c Target iteration domain

Polyhedron Model. Fig. 1 Loop nest transformation in the basic polyhedron model

- Loop iterations are executed according to the *lexicographic order* of the iteration vectors. Let $x <_{\text{lex}} y$ be true if the vector x is lexicographically less than y.

In more complex cases, these two observations may be combined to give:

$$\langle R, x\rangle \prec \langle S, y\rangle \equiv x[1:N] <_{\text{lex}} y[1:N] \vee (x[1:N] = y[1:N] \wedge R <_{\text{txt}} S),$$

where R and S are two statements, x and y their iteration vectors, N is the number of loops which encloses both R and S, and $x[1:N]$ is the vector x restricted to its N first components. Returning to Fig. 1, one has:

$$\langle S_1, i, j\rangle \prec \langle S_2, i'\rangle \equiv i < i' \vee (i = i' \wedge \mathbf{true}),$$

which simplifies to $\langle S_1, i, j\rangle \prec \langle S_2, i'\rangle \equiv i \leq i'$.

The assumption behind dependence analysis is that the sets $\mathcal{R}(u)$ and $\mathcal{W}(u)$ above only depend on the name of the instance u. This is obviously not true in general. In the polyhedron model, one assumes that all accesses are to scalars and arrays, and that, in the latter case, subscripts are known functions of the surrounding loop counters. One usually also assumes that there is no *aliasing* – two arrays with different names do not overlap – and that subscripts are always within the array bounds. Techniques for detecting and correcting violations of these assumptions are beyond the scope of this entry. With these assumptions, two instances $\langle R, x\rangle$ and $\langle S, y\rangle$ are in dependence if both access the same array A of dimension d_A, and if the *subscript equations*

$$f_R(x) = f_S(y)$$

have solutions within the iteration domains of R and S. Here, f_R and f_S are the respective *subscript functions* of A in R and S. Solving such equations is easy only if each subscript is an affine function of the iteration vector:

$$f_R(x) = F_R\, x + g_R,$$

where F_R is a matrix of dimension $d_A \times d_R$, with d_R being the number of loops surrounding R, and g_R is a vector of dimension d_A. One may associate with each candidate dependence a system of constraints by gathering the subscript equations, the constraints which define the iteration domains of R and S, and the sequencing predicate above. All of these constraints are affine, with the exception of the sequencing predicate which is a disjunction of affine constraints. Each disjunct can be tested for solutions, either by ad hoc conservative methods – see the ►Banerjee's Dependence Test entry in this encyclopedia – or by linear programming algorithms – see the ►Dependences entry.

In summary, a program can be handled in the polyhedron model – and is then called a *regular* or *static control program* – if its only control constructs are (also called regular) loops with affine bounds and its data structures are either scalars or arrays with affine subscripts in the surrounding loop counters. It should be noted that these restrictions must not be taken syntactically but semantically. For instance, in the program:

$$\begin{array}{l} i = 0; k = 0; \\ \text{while } i < n \text{ do} \\ \quad a[k] = 0.0; \\ \quad i = i + 1; \\ \quad k = k + 3 \\ \text{od} \end{array}$$

the loop is in fact regular with counter i, and the subscript of a is really $3\,i$, which is affine. There are many classical techniques – here, induction variable detection – for transforming such constructs into a more "polyhedron-friendly" form.

Regular programs are mainly found in scientific computing, linear algebra, and signal processing, where unbounded iteration domains are frequent. Perhaps more surprisingly, many variants of the Smith and Waterman algorithm [44], which is the basic tool for genetic sequence analysis, are regular and can be optimized with polyhedral tools [30]. Also, while large programs rarely fit in the model, it is often possible to extract regular kernels and to process them in isolation.

Transformations

The main devices for program optimization in the polyhedron model are coordinate transformations of the iteration domain.

An Example

Consider Fig. 1 as an illustration of the use of transformations. Figure 1a presents a sequential source program with two nested loops. The loop nest is *imperfect*: not all statements belong to the innermost loop body.

Figure 1b depicts the iteration domain of the source program, as explained in the previous section. The arrows represent dependences and impose a partial

order on the loop steps. Apart from these ordering constraints, steps can be executed in any order or in parallel.

In the source iteration domain, parallelism is in some sense hidden. The loop on j is sequential since the value stored in $A(i,j)$ at iteration j is used as $A(i,j-1)$ in the next iteration. The same is true for the loop on i. However, parallelism can be made visible by applying a skewing transformation as in Fig. 1c. For a given value of t, there are no dependences between iterations of the p loop, which is therefore parallel.

The required transformation can be viewed as a change of coordinates or a renaming:

$$S_1 : \begin{pmatrix} t \\ p \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} -2 \\ -1 \end{pmatrix}$$

$$S_2 : \begin{pmatrix} t \\ p \end{pmatrix} = \begin{pmatrix} 2 \\ 1 \end{pmatrix} \begin{pmatrix} i \end{pmatrix} + \begin{pmatrix} m-1 \\ m \end{pmatrix}$$

Observe that the transformation for S_1 has the non-singular matrix $\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$ and, hence, is bijective. Furthermore, the determinant of this matrix is 1 (the matrix is *unimodular*), which means that the transformation is bijective in the integers.

A target loop nest which corresponds to the target iteration domain is depicted in Fig. 1d. The issue of target code generation is addressed later. For now, just note that the target loop nest is much more complex than the source loop nest, and that it would be cumbersome and error-prone to derive it manually. On the other hand, the fact that both transformation matrices are unimodular simplifies the target code: both loops have unit stride.

The Search for a Transformation

The fundamental constraint on a transformation in the polyhedron model is *affinity*. As explained before, each row of the transformation matrix corresponds to one axis of the target coordinate system. Each axis represents either a sequential loop or a parallel loop. Iterations of a sequential loop are executed successively; hence, the loop counter can be interpreted as (logical) time. Iterations of a parallel loop are executed simultaneously (available resources permitting) by different processors; the values of their loop counters correspond to processor names. Finding the coefficients for the sequential axes constitutes a problem of *scheduling*, finding the coefficients for the parallel axes one of *placement* or *allocation*. Different methods exist for solving these two problems.

The order in which sequential and parallel loops are nested is important. One can show that it is always possible to move the parallel loops deeper inside the loop nest, which generates lock-step parallelism, suitable for vector or VLIW processors. For less tightly coupled parallelism, suitable for multicores or message-passing architectures, one would like to move the parallel loops farther out, but this is not always possible.

Scheduling

A schedule maps each instance in the iteration domain to a logical date. In contrast to what happens in (▸task graph scheduling), the number of instances is large, or unknown at compile time, or even infinite, so that it is impossible to tabulate this mapping. The schedule must be a closed-form function of the iteration vector; it will become clear presently that its determination is easy only if restricted to affine functions.

Let $\theta_R(i)$ be the schedule of instance $\langle R, i\rangle$. Since the source of a dependence must be executed before its destination, the schedule must satisfy the following *causality constraint*:

$$\forall i,j : \langle R,i\rangle \,\delta\, \langle S,j\rangle \Rightarrow \theta_R(i) < \theta_S(j).$$

There are as many such constraints as there are dependences in the program. The unknowns are the coefficients of θ_R and θ_S. The first step in the solution is the elimination of the quantifiers on i and j. There are general methods of quantifier elimination [38] but, due to the affinity of the constraints in the polyhedron model, more efficient methods can be applied. In fact, the form of the causality constraint above asserts that the affine *delay* $\theta_S(j) - \theta_R(i)$ must be positive inside the *dependence polyhedron* $\{i,j \mid \langle R,i\rangle \,\delta\, \langle S,j\rangle\}$. To this end, it is necessary and sufficient that the delay be positive at the vertices of the dependence polyhedron, or that it be

an affine positive combination of the dependence constraints (Farkas lemma). The result of quantifier elimination is a linear system of inequalities which can be solved by any linear programming tool.

This system of constraints may not be *feasible*, i.e., it may have no solution. This means simply that no linear-time parallel execution exists for the source program. The solution is to construct a multidimensional schedule. In the target loop nest, there will be as many sequential loops as the schedule has dimensions.

More information on scheduling can be found in the ►Scheduling Algorithms entry of this encyclopedia.

Placement

A placement maps each instance to a (virtual) processor number. Again, this mapping must be in the form of a closed affine function. In contrast to scheduling, there is no legality constraint for placements: any placement is valid, but may be inefficient.

For each dependence between instances that are assigned to distinct processors, one must generate a communication or a synchronization, depending on whether the target architecture has distributed or shared memory. These are costly operations, which must be kept at a minimum. Hence, the aim of a placement algorithm is to find a function:

$$\pi : E \to [0, P]$$

where P is the number of processors, such that the size of the set:

$$\mathcal{C} = \{u, v \in E \mid u\,\delta\,v, \pi(u) \neq \pi(v)\}$$

is minimal. Since counting integer points inside a polyhedron is difficult, one usually uses the following heuristics: try to "cut" as many dependences as possible. A dependence from statement R to S can be *cut* if the following constraint holds:

$$\langle R, i\rangle\,\delta\,\langle S, j\rangle \Rightarrow \pi_R(i) = \pi_S(j).$$

This condition can be transformed into a system of homogeneous linear equations for the coefficients of π. The problem is that, in most cases, if one tries to satisfy all the cutting constraints, the only solution is $\pi(u) = 0$, which corresponds to execution on only one processor: this, indeed, results in the minimal number of synchronizations (namely zero)! A possible way out is to solve the cutting constraints one at time, in order of decreasing size of the dependence polyhedron, and to stop just before generating the trivial solution. The uncut dependences induce synchronization operations. If all dependences can be cut, the program has *communication-free parallelism* and can be rewritten with one or more outermost parallel loops.

In the special case of a perfect loop nest with uniform dependences, one may approximate the dependence graph by the translations of the lattice generated by the dependence vectors. If the determinant of this lattice is larger than 1, the program can be split into as many independent parts [17].

Lastly, instead of assigning a processor number to each instance, one may assign all iterations of one statement to the same processor [35, 47]. This results in the construction of a Kahn process network [33].

Code Generation

In the polyhedron model, a transformation of the source iteration domain which optimizes some objective function can be found automatically. The highest execution speed (i.e., the minimum number of steps to be executed in sequence) may be the first thing that comes to mind, but many other functions are possible.

Unfortunately, it is not trivial to generate efficient target code from the optimal solution in the model. There are several factors that can degrade performance seriously. The enumeration of the points in the target iteration domain involves tests for the lower and upper border. If the code is not chosen wisely, these tests will often degrade scalability. For example, in Fig. 1, a maximum and a minimum is involved. The example of Fig. 1 also shows that additional control (the IF statement) may be introduced, which degrades performance. Of course, synchronizations and communications can also degrade performance seriously.

For details on code generation in the polyhedron model, see the ►Code Generation.

Extensions

The following extensions have successively been made to the basic polyhedron model.

WHILE Loops

The presence of a WHILE loop in the loop nest turns the iteration domain from a finite set (a polytope) into an infinite set (a polyhedron). If the control dependence that the termination test of the loop imposes is being respected, the iteration must necessarily be sequential. However, the steps of a WHILE loop in a nest with further (FOR or WHILE) loops may be distributed in space. There have been two approaches to the parallelization of WHILE loops.

The *conservative approach* [22, 25] respects the control dependence. One challenge here is the discovery of global termination. The *speculative approach* [14] does not respect the control dependence. Thus, several loop steps may be executed in parallel if there is no other dependence between them. The price paid is the need for storage of intermediate results, in case a rollback needs to be done when the point of termination has been discovered but further steps have already been executed. In some cases, overshooting the termination point does not jeopardize the correctness of the program and no rollback is needed. Discovering this property is beyond the capability of present compilers.

Conditional Statements

The basic model permits only assignment statements in the loop body. The challenge of conditionals is that a dependence may hold only for certain executions, i.e., not for all branches. A static analysis can only reveal the union of these dependences [13].

Iteration Domain Splitting

In some cases, the schedule can be improved by orders of magnitude if one splits the iteration domain in appropriate places [24]. One example is depicted in Fig. 2. With the best affine schedule of $\lfloor i/2 \rfloor$, each parallel step contains two loop iterations, i.e., the execution is sped up by a factor of 2. (The reason is that the shortest dependence has length 2.) The domain split on the right yields two partitions, each without dependences between its iterations. Thus, all iterations of the upper loop (enumerating the left partition) can be executed in a first parallel step, and the iterations of the lower loop (enumerating the right partition) in a second one, for a speedup of $n/2$.

Tiling

The technique of domain splitting has a further, larger significance. The polyhedron model is prone to yielding very fine-grained parallelism. To coarsen the grain when not enough processors are available, one partitions (parts of) the iteration domain in equally sized and shaped *tiles*. Each tile covers a set of iterations and the points in a tile are enumerated in time rather than in space, i.e., the iteration over a tile is resequentialized.

One can tile the source iteration domain or the target iteration domain. In the latter case, one can tile space and also time. Tiling time corresponds to adding hands to a clock and has the effect of coarsening the grain of processor communications. The habilitation thesis of Martin Griebl [23] offers a comprehensive treatment of this topic and an extensive bibliography. See also the ►Tiling entry of this encyclopedia.

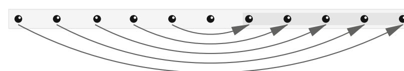
Treatment of Expressions

In the basic model, expressions are considered atomic. There is an extension of the polyhedron model to the parallelization of the evaluation of expressions [18]. It also permits the identification of common subexpressions and provides a means to choose automatically the suitable point in time and the suitable place at which to

for $i = 0$ to $2{*}n - 1$ do
  $A(i) = \ldots\ A(2{*}n{-}i{-}1)$
od

$\Longrightarrow$

for $i = 0$ to $n - 1$ do
  $A(i) = \ldots\ A(2{*}n{-}i{-}1)$
od
for $i = n$ to $2{*}n - 1$ do
  $A(i) = \ldots\ A(2{*}n{-}i{-}1)$
od



Polyhedron Model. Fig. 2 Iteration domain splitting

evaluate it just once. Its value is then communicated to other places.

Relaxations of Affinity

The requirement of affinity enters everywhere in the polyhedron model: in the loop bounds, in the array index expressions, in the transformations. Quickly, after the polyhedron model had been developed, the desire arose to transcend affinity in places. Iteration domain splitting is one example.

Lately, a more encompassing effort has been made to leave affinity behind. One circumstance that breaks the affinity of index expressions is that the so-called structure parameters (e.g., variables n and m in the loops of Figs. 1 and 2) enter multiplicatively as unevaluated variables, not as constants. For example, when a two-dimensional array is linearized, array subscripts are of the form $n\,i + j$, with i, j being the loop iterators. As a consequence, subscript equations are nonlinear in the structure parameters, too. An algorithm for computing the solutions of equation systems with exactly one such structure parameter exists [29].

In transformations and code generation, nonlinear structure parameters, as in expressions $n\,i$, $n^2\,i$, or $n\,m\,i$, can be handled by generalizing existing algorithms (for the case without nonlinear parameters) using quantifier elimination [28]. Code generation can even be generalized to handle nonlinear loop indices, as in $n\,i^2$, $n^2\,i^2$, or $i\,j$. To this end, cylindrical algebraic decomposition (CAD) [27], which corresponds to Fourier–Motzkin elimination in the basic model, is used for computing loops nests which enumerate the points in the transformed domains efficiently. This extends the frontier of code generation to arbitrary polynomial loop bounds.

Applications Other than Loop Parallelization

Array Expansion

It is easy to see that, if a loop modifies a scalar, there is a dependence between any two iterations, and the loop must remain sequential. When the modification occurs early in the loop body, before any use, the dependence can be removed by expanding the scalar to a new array, with the loop counter as its subscript. This idea can be extended to all cases in which a memory cell – be it a scalar or part of an array – is modified more than once. The transformation proceeds in two steps:

- Replace the left side of each assignment by a fresh array, subscripted by the counters of all enclosing loops.
- Inspect all the right sides and replace each reference by its *source* [20].

The source of a use is the latest modification that precedes the use in the sequential execution order. It can be computed by *parametric integer programming*. The result of this transformation is a program in *dynamic single-assignment form*. Each memory cell is written to just once in the course of a program execution. As a consequence, the sets $\mathcal{W}(u) \cap \mathcal{W}(v)$ are always empty: the transformed program has far fewer dependences and, occasionally, much more parallelism than the original.

Array Shrinking

A consequence of the previous transformation is a large increase in the memory footprint of the program. In many cases, the same degree of parallelism can be achieved with less expansion, or the target architecture cannot exploit all parallelism there is, and some of the parallel loops have to be sequentialized. Another situation, in a purely sequential context, is when a careless programmer has used more memory than strictly necessary to implement an algorithm.

The aim of *array shrinking* is to detect these situations and to reduce the memory needs by inserting modulo operators in subscripts. Suppose, for instance, that in the following code:

```
for i = 0 to n−1 do
    a[i] = ... ;
od
```

one replaces $a[i]$ by $a[i \mod 16]$. The dimension of a, which is n in the first version, is reduced to 16 in the second version. Of course, this means that the value stored in $a[i]$ is destroyed after 16 iterations of the loop. This transformation may change the outcome of the program, unless one can prove that the *lifetime* of $a[i]$ does not exceed 16 iterations.

Finding an automatic solution to this problem has been the subject of much work since 1990 (Darte [16] offers a good discussion). The proposed solution is to construct an interference polyhedron for the elements

of a fixed array and to cover it by a maximally tight lattice such that only the lattice origin falls inside the polyhedron. The basis vectors of the lattice are taken as coordinate axes of the reduced array, and their lengths are related to the modulus of the new subscripts.

Communication Generation

When constructing programs for distributed memory architectures, be it with data distribution directives in languages like High-Performance Fortran (HPF) or under the direction of a placement function, one has to generate communication code. It so happens that this is also a problem of polyhedron scanning. It can be solved by the same techniques and the same tools that are used for code generation.

Locality Enhancement

Most modern processors have caches: small but fast memories that retain a copy of recently accessed memory cells. A program has locality if memory accesses are clustered such that there is a high likelihood of finding a copy of the needed information in cache rather than in main memory. Improving the locality of a program is highly beneficial for performance since caches are usually accessed in one cycle while memory latency may range from ten to a hundred cycles.

Since the cache controller returns old copies to memory in order to find room for new ones, locality is enhanced by changing the execution order such that the *reuse distance* between successive accesses to the same cell is minimal. This can be achieved, for instance, by moving all such accesses to the innermost loop of the program [49].

Another approach consists of dividing a program into *chunks* whose memory footprints are smaller than the cache size. Conceptually, the program is executed by filling the cache with the necessary data for one chunk, executing the chunk without any cache miss, and emptying the cache for the next chunk. One can show that the memory traffic will be minimal if each datum belongs to the footprint of only one chunk. The construction of chunks is somewhat similar to scheduling [7]. It is enough to have asymptotic estimates of the footprint sizes. One advantage of this method is that it can be adapted easily to the management of *scratchpad memories*, software-controlled caches as can be found in embedded processors.

Dynamic Optimization

Dynamic optimization resulted from the observation that modern processors and compilers are so complex that building a realistic performance estimator is nearly impossible. The only way of evaluating the quality of a transformed program is to run it and take measurements.

In the polyhedron model, one can define the polyhedron of all legal schedules (see the previous section on ►Scheduling). Usually, one selects one schedule in this polyhedron according to some simple objective function. Another possibility is to generate one program for each legal schedule, measure its performance, and retain the best one. Experience shows that, in many cases, the best program is unexpected, the proof of its legality is not obvious, and the reasons for its efficiency are difficult to fathom. As soon as the source program has more than a few statements, the size of the polyhedron of legal schedules explodes: sophisticated techniques, including genetic algorithms and machine learning are needed to restrict the exploration to "interesting" solutions [39, 40].

Tools

There is a variety of tools which support several phases in the polyhedral parallelization process.

Mathematical Support

PIP [19] is an all integer implementation of the Simplex algorithm, augmented with Gomory cuts for integer programming [43]. The most interesting feature of PIP is that it can solve parametric problems, i.e., find the lexicographically minimal x such that

$$Ax \le By + c$$

as a function of y.

Omega [41] is an extension of the Fourier–Motzkin elimination method to the case of integer variables. It has been extended into a full-fledged tool for the manipulation of Presburger formulas (logical formulas in which the atoms are affine constraints on integer variables).

There are many so-called polyhedral libraries; the oldest one is the PolyLib [11]. The core of these libraries is a tool for converting a system of affine constraints into the vertices of the polyhedron it defines, and back. The PolyLib also includes a tool for counting the number

of integer points inside a parametric polyhedron, the result being an Ehrhart polynomial [10]. More recent implementations of these tools, occasionally using different algorithms, are the Parma Polyhedral Library [2], the Integer Set Library [46], the Barvinok Library [48], and the Polka Library [32]. This list is probably not exhaustive.

Code Generation

CLooG [6] takes as input the description of an iteration domain, in the form of a disjoint union of polyhedra, and generates an efficient loop nest that scans all the points in the iteration domain in the order given by a set of scattering functions, which can be schedules, placements, tiling functions, and more. For a detailed description of CLooG, see the ►Code Generation entry in this encyclopedia.

Full-Fledged Loop Restructurers

LooPo [26] was the first polyhedral loop restructurer. Work on it was started at the University of Passau in 1994, and it was developed in steps over the years and is still being extended. LooPo is meant to be a research platform for trying out and comparing different methods and techniques based on the polyhedron model. It offers a number of schedulers and allocators and generates code for shared-memory and distributed memory architectures. All of the extensions mentioned above have been implemented and almost all are being maintained.

Pluto [9] was developed at Ohio State University. Its main objective is to use placement functions to improve locality and to integrate tiling into the polyhedron model. Its target architectures are multicores and graphical processing units (GPUs).

GRAPHITE [45] is an extension of the GCC compiler suite whose ultimate aim is to apply polyhedral optimization and parallelization techniques, where possible, to run-of-the-mill programs. GRAPHITE looks for static control parts (SCoPs) in the GCC intermediate representation, generates their polyhedral representation, applies transformations, and generates target code using CLooG. At the time of writing, the set of available transformations is still rudimentary, but is supposed to grow.

Related Entries

►Banerjee's Dependence Test
►Code Generation
►Dependence Abstractions
►Dependence Analysis
►HPF (High Performance Fortran)
►Loop Nest Parallelization
►OpenMP
►Scheduling Algorithms
►Speculative Parallelization of Loops
►Task Graph Scheduling
►Tiling

Bibliographic Notes and Further Reading

The development of the polytope model was driven by two nearly disjoint communities. Hardware architects wanted to take a set of recurrence equations, expressing, for instance, a signal transformation, and derive a parallel processor array from it. Compiler designers wanted to take a sequential loop nest and derive parallel loop code from it.

One can view the seed of the model for architecture in the seminal paper by Karp, Miller, and Winograd on analyzing recurrence equations [34] and the seed for software in the seminal paper by Lamport on Fortran DO loop parallelization [36]. Lamport used hyperplanes (the slices in the polyhedron that make up the parallel steps), instead of polyhedra. In the early 1980s, Quinton drafted the components of the polyhedron model [42], still in the hardware context (at that time: systolic arrays).

The two communities met around the end of the 1980s at various workshops and conferences, notably the International Conference on Supercomputing and CONPAR and PARLE, the predecessors of the Euro-Par series. Two developments made the polyhedron model ready for compilers: parametric integer programming, worked out by Feautrier [19], which is used for dependence analysis, scheduling and code generation, and seminal work on code generation by Irigoin et al. [1, 31]. Finally, Lengauer [37] gave the model its name.

The 1990s saw the further development of the theory underlying the model's methods, particularly for scheduling, placement, and tiling. Extensions and

applications other than loop parallelization came mainly in the latter part of the 1990s and in the following decade.

A number of textbooks focus on polyhedral methods. There is the three-part series of Banerjee [3–5], a book on tiling by Xue [50], and a comprehensive book on scheduling by Darte et al. [15]. Collard [12] applies the model to the optimization of loop nests for sequential as well as parallel execution and studies a similar model for recursive programs.

In the past several years, the polyhedron model has become more mainstream. The seed of this development was an advance in code generation methods [6]. With the GCC community taking an interest, it is to be expected that polyhedral methods will increasingly find their way into production compilers.

Bibliography

1. Ancourt C, Irigoin F (1991) Scanning polyhedra with DO loops. In: Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), ACM, pp 39–50
2. Bagnara R, Hill PM, Zaffanella E (2008) The Parma polyhedra library: toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. Sci Comput Program 72(1–2):3–21, http://www.cs.unipr.it/ppl
3. Banerjee U (1993) Loop transformations for restructuring compilers: the foundations. Series on Loop Transformations for Restructuring Compilers. Kluwer, Norwell
4. Banerjee U (1994) Loop parallelization. Series on Loop Transformations for Restructuring Compilers. Kluwer, Norwell
5. Banerjee U (1997) Dependence analysis. Series on Loop Transformations for Restructuring Compilers. Kluwer, Norwell
6. Bastoul C (2004) Code generation in the polyhedral model is easier than you think. In: Proceedings of the 13th International Conference on Parallel Architecture and Compilation Techniques (PACT), IEEE Computer Society Press, pp 7–16, http://www.cloog.org/
7. Bastoul C, Feautrier P (2003) Improving data locality by chunking. In: Compiler Construction (CC), Lecture Notes in Computer Science, vol 2622. Springer, Berlin, pp 320–335
8. Bernstein AJ (1966) Analysis of programs for parallel processing. In: IEEE Transactions on Electronic Computers, EC-15:757–762
9. Bondhugula U, Hartono A, Ramanujam J, Sadayappan P (2008) A practical automatic polyhedral parallelizer and locality optimizer. SIGPLAN Notices, 43(6):101–113, 2008. http://pluto-compiler.sourceforge.net/
10. Clauss P (1996) Counting solutions to linear and non-linear constraints through Ehrhart polynomials. In: Proceedings of the ACM/IEEE Conference on Supercomputing, ACM, pp 278–285
11. Clauss P, Loechner V (1998) Parametric analysis of polyhedral iteration spaces, extended version. J VLSI Signal Process 19(2):179–194, http://icps.u-strasbg.fr/polylib/
12. Collard J-F (2003) Reasoning about program transformations – imperative programming and flow of data. Springer, Berlin
13. Collard J-F, Griebl M (1999) A precise fixpoint reaching definition analysis for arrays. In: Carter L, Ferrante J (eds) Languages and Compilers for Parallel Computing (LCPC), Lecture Notes in Computer Science, vol 1863, Springer, Berlin, pp 286–302
14. Collard J-F (1995) Automatic parallelization of while-loops using speculative execution. Int J Parallel Program 23(2):191–219
15. Darte A, Robert Y, Vivien F (2000) Scheduling and automatic parallelization. Birkhäuser, Boston
16. Darte A, Schreiber R, Villard G (2005) Lattice-based memory allocation. IEEE Transaction on Computers TC-54(10):1242–1257
17. D'Hollander EH (1992) Partitioning and labeling of loops by unimodular transformations. IEEE Trans Parallel Distrib Syst 3(4):465–476
18. Faber P (2007) Code optimization in the polyhedron model – improving the efficiency of parallel loop nests. PhD thesis, Department of Informatics and Mathematics, University of Passau, 2007. http://www.fim.unipassau.de/cl/publications/docs/Faber07.pdf
19. Feautrier P (1988) Parametric integer programming. Oper Res 22(3):243–268, http://www.piplib.org
20. Feautrier P (1991) Dataflow analysis of scalar and array references. Parallel Program 20(1):23–53
21. Feautrier P (1996) Automatic parallelization in the polytope model. In: Perrin G-R, Darte A (eds) The data parallel programming model. Lecture Notes in Computer Science, vol 1132, Springer, Berlin, pp 79–103
22. Griebl M (1997) The mechanical parallelization of loop nests containing WHILE loops. PhD thesis, Department of Mathematics and Informatics, University of Passau, January 1997. http://www.fim.unipassau.de/cl/publications/docs/Gri96.pdf
23. Griebl M (2004) Automatic parallelization of loop programs for distributed memory architectures. Habilitation thesis, Department of Informatics and Mathematics, University of Passau, June 2004. http://www.fim.unipassau.de/cl/publications/docs/Gri04.pdf
24. Griebl M, Feautrier P, Lengauer C (2000) Index set splitting. Int J Parallel Process 28(6):607–631 Special Issue on the International Conference on Parallel Architectures and Compilation Techniques (PACT'99)
25. Griebl M, Lengauer C (1994) On the space-time mapping of WHILE loops. Parallel Process Lett 4(3):221–232
26. Griebl M, Lengauer C (1997) The loop parallelizer LooPo – announcement. In: Sehr D (ed) Languages and Compilers for Parallel Computing (LCPC). Lecture Notes in Computer Science, vol 1239, pp 603–604. Springer, Berlin, http://www.infosun.fim.uni-passau.de/cl/loopo/
27. Größlinger A (2009) The challenges of non-linear parameters and variables in automatic loop parallelisation. PhD thesis, Department of Informatics and Mathematics, University of

P

Passau, December 2009. http://nbnresolving.de/urn:nbn:de:bvb:739-opus-17893
28. Größlinger A, Griebl M, Lengauer C (2006) Quantifier elimination in automatic loop parallelization. J Symbolic Computation 41(11):1206–1221
29. Größlinger A, Schuster S (2008) On computing solutions of linear diophantine equations with one non-linear parameter. In: Proceedings of the 10th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), IEEE Computer Society Press, pp 69–76
30. Guerdoux-Jamet P, Lavenier D (1997) SAMBA: hardware accelerator for biological sequence comparison. Comput Appl Biosci 13(6):609–615
31. Irigoin F, Triolet R (1989) Dependence approximation and global parallel code generation for nested loops. In: Cosnard M, Robert Y, Quinton P, Raynal M (eds) Parallel and distributed algorithms. Bonas, North-Holland, pp 297–308
32. Jeannet B, Miné A (2009) APRON: a library of numerical abstract domains for static analysis. In: Computed Aided Verification (CAV). Lecture Notes in Computer Science, vol 5643, Springer, pp 662–667, http://apron.cri.ensmp.fr/library
33. Kahn G (1974) The semantics of simple language for parallel programming. In: Proceedings of the IFIP Congress, Stockholm, pp 471–475
34. Karp RM, Miller RE, Winograd S (1967) The organization of computations for uniform recurrence equations. J ACM 14(3):563–590
35. Kienhuis B, Rijpkema E, Ed Deprettere F (2000) Compaan: deriving process networks from matlab for embedded signal processing architectures. In: Vahid F, Madsen J (eds) Proceedings of the Eighth International Workshop on Hardware/Software Codesign (CODES 2000), ACM pp 13–17
36. Lamport L (1974) The parallel execution of DO loops. Comm ACM 17(2):83–93
37. Lengauer C (1993) Loop parallelization in the polytope model. In: Best E (ed) CONCUR'93. Lecture Notes in Computer Science, vol 715, pp 398–416, Springer, 1993
38. Loos R, Weispfenning V (1993) Applying linear quantifier elimination. The Computer J 36(5):450–462
39. Pouchet L-N, Bastoul C, Cohen A, Cavazos J (2008) Iterative optimization in the polyhedral model: Part II, multidimensional time. SIGPLAN Notices 43(6):90–100
40. Pouchet L-N, Bastoul C, Cohen A, Vasilache N (2007) Iterative optimization in the polyhedral model: Part I, one-dimensional time. In IEEE/ACM Fifth International Symposium on Code Generation and Optimization (CGO'07), IEEE Computer Society Press, pp 144–156
41. Pugh W (1991) The Omega test: a fast and practical integer programming algorithm for dependence analysis. In: Proceedings of the 5th International Conference on Supercomputing, ACM, pp 4–13. 1991. http://www.cs.umd.edu/projects/omega
42. Quinton P (1983) The systematic design of systolic arrays. In: Soulié FF, Robert Y, Tchuenté M (eds) Automata networks in computer science, chapter 9, pp 229–260. Manchester University Press, 1987. Also: Technical Reports 193 and 216, IRISA (INRIA-Rennes)
43. Schrijver A (1986) Theory of linear and integer programming. Wiley, New York
44. Smith TF, Waterman MS (1981) Identification of common molecular subsequences. J Molecular Biology 147(1):195–197
45. Trifunovic K, Cohen A, Edelsohn D, Feng L, Grosser T, Jagasia H, Ladelsky R, Pop S, Sjödin J, Upadrasta R (2010) GRAPHITE two years after. In: Proceedings of the 2nd International Workshop on GCC Research Opportunities (GROW), pp 4–19, January 2010. http://gcc.gnu.org/wiki/GROW-2010
46. Verdoolaege S (2009) An integer set library for program analysis. In: Advances in the theory of integer linear optimization and its extensions. AMS 2009 Western Section, 2009, http://freshmeat.net/projects/isl/
47. Verdoolaege S, Nikolov H, Todor N, Stefanov P (2006) Improved derivation of process networks. In Proceedings of the 4th International Workshop on Optimization for DSP and Embedded Systems (ODES), 2006, http://www.ece.vill.edu/~deepu/odes/odes-4_digest.pdf
48. Verdoolaege S, Seghir R, Beyls K, Loechner V, Bruynooghe M (2007) Counting integer points in parametric polytopes using Barvinok's rational functions. Algorithmica 48(1):37–66, http://freshmeat.net/projects/barvinok
49. Wolf ME, Lam MS (1991) A data locality optimizing algorithm. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), ACM, pp 30–44
50. Xue J (2000) Loop tiling for parallelism. Kluwer, Boston

Polytope Model

►Polyhedron Model

Position Tree

►Suffix Trees

POSIX Threads (Pthreads)

POSIX Threads [1] are those created, managed, and synchronized following the POSIX standard API. POSIX stands for "**P**ortable **O**perating **S**ystem **I**nterface [for Uni**x**]."

Bibliography

1. Nichols B, Buttlar D, Farrell JP (1996) Pthreads programming. O'Reilly & Associates, Inc., Sebastopol

Power Wall

Pradip Bose
IBM Corp. T.J. Watson Research Center, Yorktown Heights, NY, USA

Definition

The "Power Wall" refers to the difficulty of scaling the performance of computing chips and systems at historical levels, because of fundamental constraints imposed by affordable power delivery and dissipation. The single biggest factor that has led the industry into encountering this wall in the past decade is the significant change in traditional CMOS chip design evolution, which were driven previously by Dennard scaling rules [5, 15].

Discussion

Introduction

Power delivery and dissipation limits have emerged as a key constraint in the design of microprocessors and associated systems even for those targeted for the high end server product space. At the low end of the performance spectrum, power has always dominated over performance as the primary design constraint. However, while battery life expectancies have shown modest increases, the larger demand for increased functionality and speed has increased the severity of the power constraint in the world of handheld and mobile systems. At the high end, where performance was always the primary driver, we are witnessing a trend (dictated primarily by CMOS technology scaling constraints) where increasingly, energy and power limits are dictating the high-level processing paradigms, as well as the lower level issues related to clocking and circuit design. Thus, regardless of the application domain, power consumption constitutes a primary barrier or "wall" when it comes to achieving cost-effective performance growth in future systems. In this entry, we will examine the fundamental causes that have led us up to this *power wall*, and discuss the key solution approaches at hand to circumvent this barrier.

Power Trends

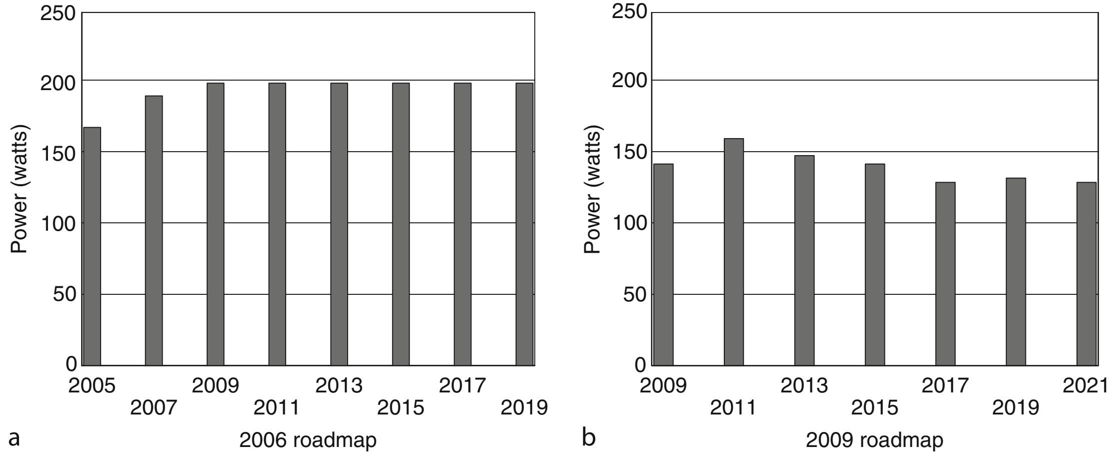
Figure 1 shows the expected maximum chip power (for high performance processors) through the year 2020. The data plotted is based on the 2006 (Fig. 1a) and then updated 2009 projections (Fig. 1b) made by the International Technology Roadmap for Semiconductors (ITRS) [61]. The 2006 projection indicated that beyond the continued growth period (through year 2008) for high-end microprocessors, there will be a saturation in the maximum chip power (with a projected cap of around 200 W from years 2008 all the way through 2020). This is due to thermal/packaging and die size limits that had already started to kick in by 2005. Single-thread performance scaling had actually started to get limited by chip-level power density since the latter part of the 1990s, as depicted in Fig. 2 (adapted from a 2005 keynote speech by David Yen [65], who was then the executive VP of scalable systems at Sun Microsystems). Beyond a certain power (and power density) regime, air cooling is not sufficient to dissipate the heat generated; on the other hand, use of liquid cooling and refrigeration causes a sharp increase of the cost–performance ratio. Thus, power-aware design techniques, methodologies, and tools are of the essence at all levels of design. The 2009 ITRS projection shows that the long-term power cap for high-performance, server-class microprocessors has been revised downward to 130 W. This has to do with factors like the current delivery limits in typical blade server form factors that have dominated the midrange server products since 2006.

CMOS Technology Determinants

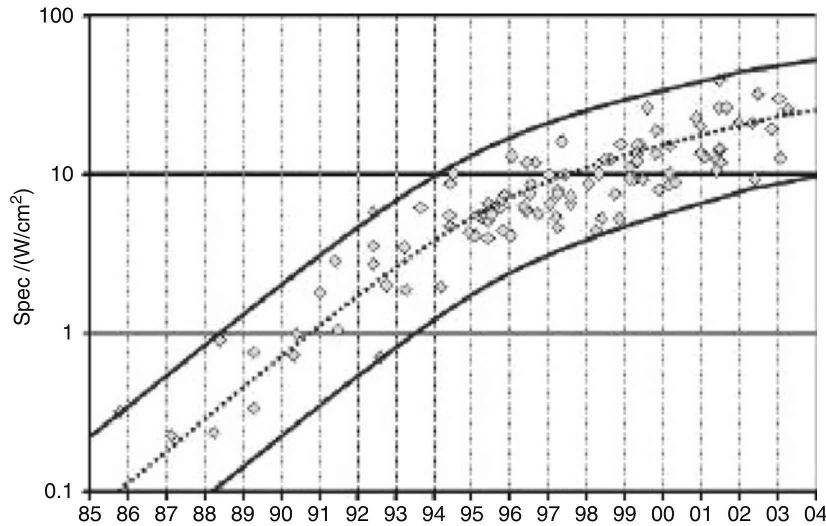
In this subsection, we present a summary of the technological trends that have caused chip-level power and power density to reach the levels that have caused us to identify the *power wall* as a fundamental barrier to achieving cost-effective performance growth in future computing systems.

At the elementary transistor gate (e.g., an inverter) level, total power dissipation can be formulated as the

P



Power Wall. Fig. 1 Maximum chip power projection – high performance with heatsink



Power Wall. Fig. 2 Performance/power-density (SPEC/[W/cm^2]) trends from 1985 to 2004 [65]

sum of three major components: switching loss, leakage and short-circuit loss [6, 12, 19, 21].

$$\text{Power}_{\text{device}} = (1/2)\, C.\, V_{\text{dd}}.\, V_{\text{swing}}.\, a.f + I_{\text{leakage.}}\, V_{\text{dd}} + I_{\text{sc}}.\, V_{\text{dd}} \quad (1)$$

where, C is the output capacitance, V_{dd} is the supply voltage, f is the chip clock frequency, and a is the activity factor ($0 \leqslant a \leqslant 1$) which determines the device switching frequency; V_{swing} is the maximum voltage swing across the output capacitor, which in general can be less than V_{dd}; $I_{leakage}$ is the leakage current and I_{sc} is the short-circuit current. In the literature, V_{swing} is often approximated to be equal to V_{dd} (or simply V for short) making the switching loss ~ $(1/2)C.\, V^2.\, a.f$. Also, as discussed in [19], for a prior generation range of V_{dd} (say 1 V–3 V) switching loss: $(1/2)CV^2af$ was the dominant component, assuming the activity factor to be above a reasonable minimum. So, as a first-order approximation, for chips belonging to the previous generation (e.g., CMOS 180 nm, and before), we may ignore the leakage and short-circuit components in Eq. 1. In other words, in the context of switching power dominated technology generations, we may formulate the power dissipation to be:

$$\text{Power}_{\text{chip}} = (1/2)\left[\sum C_i \,.\, V_i^2 \,.\, a_i \,.\, f_i\right] \quad (2)$$

1. Where, C_i, V_i, a_i, and f_i are unit- or block-specific average values in the most general case; the summation is taken over all blocks or units i, at the microarchitecture level (e.g., icache, dcache, integer unit, floating point unit, load-store unit, register files and buses [if not included in individual units], etc). Also, for the voltage range considered, the operating frequency is roughly proportional to the supply voltage; and the capacitance C remains roughly the same if we keep the same design but scale the voltage. If a single voltage and clock frequency are used for the whole chip, the above reduces to:

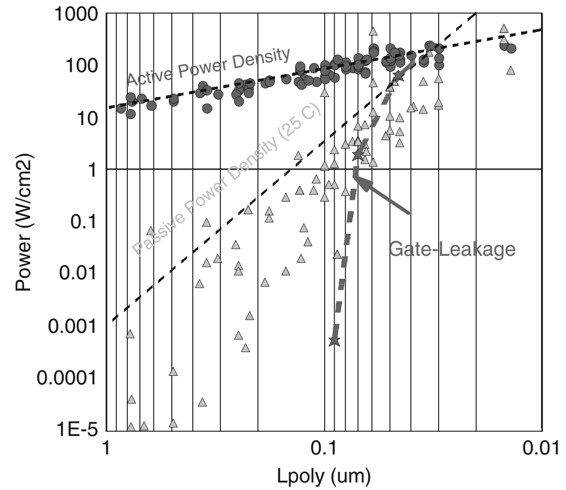
$$\text{Power}_{\text{chip}} = V^3.\left(\sum K_i^v.a_i\right) = f^3.\left(\sum K_i^f.a_i\right) \quad (3)$$

2. If we consider the very worst-case activity factor for each unit i, i.e., if $a_i = 1$ for all i, then, an upper bound on the maximum chip power may be formulated as:

$$\text{MaxPower}_{\text{chip}} = K_V\ .\ V^3 = K_F\ .\ f^3 \quad (4)$$

where K_V and K_F are design-specific constants. Note that an estimation of peak or maximum power is important, for the purposes of determining the packaging and cooling solution required. The larger the maximum power, the more expensive is the net cooling solution. Note that the formulation in Eq. 4 is overly conservative, as stated. In practice, it is possible to estimate the worst-case achievable maximum for the activity factors. This allows the designers to come up with a tighter bound on maximum power before the packaging decision is made.

The last Eq. 4 is what leads to the so-called cube-root rule [19], where redesigning a chip to operate at ½ the voltage (and frequency) results in the power dissipation being lowered to $(1/2)^3$ or 1/8 of the original. This implies the single-most efficient method for reducing power dissipation for a processor that has already been designed to operate at high frequency, namely, reduce the voltage (and hence the frequency). There is a limit, however, of how low V_{dd} can be reduced (for a given technology), which has to do with manufacturability and circuit reliability issues. Thus, a combination of microarchitecture and circuit techniques to reduce power consumption, without necessarily employing multiple or variable supply voltages is of special relevance in the design of robust systems.



Power Wall. Fig. 3 Active and major leakage power component trends [67]

Figure 3 shows the analytically projected trend [67] of escalation in three of the major components of microprocessor power consumption, namely, active or capacitive switching power, subthreshold leakage power and gate leakage power. (The leakage current referred to in Eq. 1 above consists of two major components: subthreshold leakage and gate leakage. There are other components of leakage as well, as discussed below. The short-circuit loss referred to in Eq. 1 is not a technology-dependent component, and so is ignored in this discussion). In post-180 nm technologies, static (i.e., leakage or standby) power has increasingly become a major (if not the dominating) component of chip power. As discussed in [1], the three major types of leakage effects are (a) sub-threshold, (b) gate, and (c) reverse-biased, drain- and source-substrate junction band-to-band tunneling (BTBT). With technology scaling, each of these leakage components tends to increase drastically. For example, as technology scales downward, the supply voltage (V_{dd}) must also scale down to reduce dynamic power and maintain device reliability. However, this requires the scaling down of the threshold voltage (V_{th}) to maintain reasonable gate overdrive (and therefore performance), which is a function of ($V_{dd} - V_{th}$). However, lowering the threshold voltage causes substantial increases in leakage current, and therefore standby power, inspite of the lower V_{dd}. The subthreshold channel leakage current in an MOS

device is governed by an equation that looks like [26]:

$$I_{\text{leakage}} = K_w * W.10^{-Vth/S} \quad (5)$$

where K_w, measured in units of micro-amps per micron (μA/μm) can be thought of as the width-sensitivity coefficient; W is the device width and S is the subthreshold swing (measured in millivolts, like the threshold voltage V_{th}). (In [26], the value of K_w is quoted to be 10). S is a parameter that is defined to characterize the efficiency of a device in turning on or off. It can be shown that the turn-off characteristic of a device is proportional to the thermal voltage (kT/q) and the ratio of junction capacitance (C_j) to oxide capacitance (C_{ox}) [8]. The parameter S can be formulated as:

$$S = 2.3\ (kT/q).\ (1\ +\ C_j/C_{\text{ox}}) \quad (6)$$

This parameter is usually specified in units of millivolts per decade and it defines how many millivolts (mV) the gate voltage must drop before the drain current is reduced by one decade. The thermal voltage kT/q is equal to 26 mV at room temperature. Thus, at room temperature, the minimum value of S is about 60 mV per decade. This means that an ideal device at room temperature would experience a 10X reduction in drain current for every 60 mV reduction of the gate voltage V_{gs} in the subthreshold region. In the deep submicron era, a typical transistor device has an S value in the range of 85–90 mV per decade.

Note also, that the threshold voltage V_{th} (Eq. 5) is itself a function of temperature (T); in fact V_{th} decreases by 2.5 mV/K as temperature increases. Also, K_w itself is a strong function of temperature ($\sim T^2$). Thus, as T increases, leakage current goes up dramatically, both because of its dependence on T and because V_{th} goes down. The delay of an inverter gate is given by the alpha-power model [51] as:

$$T_g \sim \frac{L_{\text{eff}} V_{\text{dd}}}{\mu(T) \cdot (V_{\text{dd}} - V_{\text{th}})^{\alpha}} \quad (7)$$

where, α is typically around 1.3 and μ is the mobility of carriers (which is a function of temperature T, $\mu(T) \sim T^{-1.5}$). As V_{th} decreases, $(V_{dd} - V_{th})$ increases so the inverter becomes faster. As T increases, $(V_{dd} - V_{th})$ increases, but $\mu(T)$ decreases [36]. This latter effect dominates; so, with higher temperatures, the logic gates in a processor generally become slower.

Power-Performance Efficiency Metrics

The most common (and perhaps obvious) metric to characterize the power-performance efficiency of a microprocessor is a simple ratio, like mips/watt. This attempts to quantify the efficiency by projecting the performance achieved or gained (measured in millions of instructions per second) for every watt of power consumed. Clearly, the higher the number, the "better" the machine is. Dimensionally, mips/watt equates to the inverse of the average energy consumed per instruction. This seems a reasonable choice for some domains where battery life is important. However, there are strong arguments against it in many cases, especially when it comes to characterizing higher end processors. Performance has typically been the key driver of such server-class designs and cost or efficiency issues have been of secondary importance. Specifically, a design team may well choose a higher frequency design point (which meets maximum power budget constraints) even if it operates at a much lower mips/watt efficiency compared to one that operates at better efficiency but at a lower performance level. As such, $(\text{mips})^2$/watt or even $(\text{mips})^3$/watt may be the metric of choice at the high end. On the other hand, at the lowest end, where battery-life (or energy consumption) is the primary driver, one may want to put an even greater weight on the power aspect than the simplest mips/watt metric, i.e., one may just be interested in minimizing the watts for a given workload run, irrespective of the execution time performance, provided the latter does not exceed some specified upper limit.

The "mips" metric for performance and the "watts" value for power may refer to average or peak values, derived from the chip specifications. For example, for a 1 GHz (=10^9 cycles/s) processor which can complete up to 4 instructions per cycle, the theoretical peak performance is 4,000 mips. If the average completion rate for a given workload mix is p instructions per cycle, then the average mips would equal 1,000 times p. However, when it comes to workload-driven evaluation and characterization of processors, metrics are often controversial. Apart from the problem of deciding on a "representative" set of benchmark applications, there are fundamental questions which persist about how to boil down "performance" into a single ("average") rating that is meaningful in comparing a set of machines. Since power consumption varies, depending on the

program being executed, the issue of benchmarking is also relevant in assigning an average power rating. In measuring power and performance together for a given program execution, one may use a fused metric like power-delay product (PDP) or energy-delay product (EDP) [21, 49]. In general, the PDP-based formulations are more appropriate for low-power, portable systems, where battery-life is the primary index of energy efficiency. The mips/watt metric is an inverse PDP formulation, where delay refers to average execution time per instruction. The power-delay product, being dimensionally equal to energy, is the natural metric for such systems. For higher end systems (e.g., workstations) the EDP-based formulations are deemed to be more appropriate, since the extra delay factor ensures a greater emphasis on performance. The $(\text{mips})^2$/watt metric is an inverse EDP formulation. For the highest performance, server-class machines, it may be appropriate to weight the "delay" part even more. This would point to the use of $(\text{mips})^3$/watt, which is an inverse ED^2P formulation. Alternatively, one may use $(\text{cpi})^3$.watt as a direct ED^2P metric, applicable on a "per instruction" basis (see [12]).

The energy*$(\text{delay})^2$ metric, or perf^3/power formula is analogous to the cube-root rule [19] which follows from constant voltage scaling arguments (see Section "CMOS Technology Determinants", Eq. 4). Clearly, to formulate a voltage-invariant power-performance characterization metric, we need to think in terms of perf^3/(power). When we are dealing with the SPEC benchmarks, one may therefore evaluate efficiency as $(\text{SPECrating})^x$/watt, or $(\text{SPEC})^x$/watt for short; where the exponent value x (=1, 2, or 3) may depend on the class of processors being compared.

Brooks et al. [12] discuss the power-performance efficiency data for a range of commercial processors of approximately the same generation (circa year 2000). SPEC/watt, SPEC^2/watt, and SPEC^3/watt are used as the alternative metrics, where SPEC stands for the processor's SPEC rating [25]. The data validates our assertion that depending on the metric of choice, and the target market (determined by workload class and/or the power/cost) the conclusion drawn about efficiency can be quite different. For performance-optimized, high-end processors, the SPEC^3/watt metric seems to be fairest. For "power-first" processors, SPEC /watt seems to be the fairest.

Recently, there has been a strong motivation made for energy-proportional computing [4], in the context of future data centers and cloud computing. In this style of measuring system-level energy efficiency, it is not enough to assess the efficiency at the peak utilization levels; rather, how well the system is able to adjust the power level downward, as the workload demand decreases, is also of interest. The recently announced SPEC power benchmark [25] is intended, in part, to measure the degree to which a given server system is energy proportional. The currently used workload in SPECpower is the specjbb application, and the evaluation modality is to measure the power and performance across a range of system-level utilizations, from 100%, down through 0% (11 data points). The benchmark calls for adding up the corresponding power and performance numbers at those 11 data points and then computing the ratio of the summed performance value and the summed power number. The higher the number, the better is the energy proportional characteristic of the measured system.

A Review of Key Ideas in Power-Aware Architectures

In this section, a brief review of power-efficient design concepts will be covered. The motivation, of course, is to examine solution approaches for avoiding the power wall, while preserving performance growth in next generation systems. The initial attention will be on dynamic (also known as "active" or "switching") power governed by the CV^2af formula. Recall that C refers to the switching capacitance, V is the supply voltage, a is the activity factor ($0 < a < 1$), and f is the operating clock frequency. Power reduction ideas must therefore focus on one or more of these basic parameters. Reducing active power generally results in reduction of on-chip temperatures, and this indirectly causes leakage power to go down as well. Similarly, any increase in efficiency directed at lowering the latch count (e.g., by reducing the basic pipeline depth, or by reducing the number of back-end execution pipes within a given functional unit) also results in area and leakage reduction as a side benefit. However, later in this section, we also deal with the problem of mitigating leakage power directly, by providing microarchitectural support to what are primarily circuit-level mechanisms.

Power Efficiency at the Processor Core Level

In this sub-section, we examine the key ideas that have been proposed in terms of (micro)architectural support for power-efficiency, at the level of a single processor core. Early research ideas started being presented since 1998 at a few key conference workshops (e.g., [57–59]); later the field of power-aware microarchitectures matured to the point where all major computer architecture conferences now all have specific sessions devoted to ideas for performance growth in the current power-constrained design era.

The effective (average) value of *C* can be reduced by using: (a) area-efficient designs for various macros; and (b) adaptive structures, that change in effective size, latency, or communication bandwidth depending on the needs of the input workload. The average value of *V* can be reduced via dynamic voltage scaling, i.e., by reducing the voltage as and when required or possible. Microarchitectural support, in this case, is not required, unless the mechanisms to detect "idle" periods or temperature overruns make use of counter-based "proxies," specially architected for this purpose. Note again, however, that since reducing *V* also requires (or results in) reduction of the operating frequency, *f*, net power reduction has a cubic effect; thus, dynamic voltage and frequency scaling (DVFS) is one of the most effective way of power reduction). Deciding when and how to apply DVFS, as a function of the input workload characteristics and overall operating environment, is very much a microarchitectural issue. It is a problem that is increasingly relevant in the era of variability-tolerant, power-efficient multi-core chip design, and we will touch on it briefly in section "►Conclusions."

The average value of the activity factor, *a*, can be reduced by: (a) the use of clock-gating, where the normally free-running, synchronous clock is disabled in selected units or sub-units within the system based on information or predictions about current or future activity in those regions; (b) reducing unnecessary "speculative waste" resulting from executing instructions in mis-speculated branch paths or prefetching useless instructions and data into caches, based on wrong or ill-timed guesses; and (c) the use of data representations and instruction schedules that result in reduced switching.

Microarchitectural support is provided in the form of added mechanisms to: detect, predict, and control the generation of the applied gating signals; or aid in power-efficient data and instruction encodings. Compiler support for generating power-efficient instruction scheduling and data partitioning or special instructions for "nap/doze/sleep" control, if applicable, must also be considered under this category. Power-efficient task scheduling at the system software (i.e., OS and hypervisor) level is also an example of dynamic load balancing that can make the activity distribution uniform over a multi-core processor system, and thereby help reduce power density (and hence temperature and overall power).

While clock-gating helps eliminate (or drastically reduce) active or switching power when a given macro, sub-unit or unit is idle, power-gating can be used to also eliminate the residual leakage power of that idle entity. In this case, as described in detail later on, the power supply voltage *V* is itself gated off from the target circuit block, with the help of a header or footer transistor. Here, the need for microarchitectural support in the form of predictive control of the gating signal is even stronger because of the relatively large performance overheads that would be incurred without such support. There are other techniques, like adaptive body-biasing that are also targeted at leakage power control; and these too require some degree of microarchitectural support. However, these techniques are most relevant to bulk-CMOS designs (as opposed to silicon-on-insulator [SOI]-CMOS technology), and are predominantly device- and circuit-level methods. As such, we do not dwell on them in this entry.

Lastly, the average value of the design frequency, *f*, can be controlled or reduced by using: (a) variable, multiple or locally asynchronous (self-timed) clocks – e.g., in GALS [29] designs; (b) clock-throttling, where the frequency is reduced dynamically in response to power or temperature overrun indicators; (c) reduced pipeline depths in the baseline microarchitecture definition.

Henceforth, the focus of consideration is: power-aware microarchitectural constructs that use *C*, *a*, or *f* as the primary lever for reducing active power; and those that use the supply voltage *V* as the primary lever for reducing leakage power. In any such proposed processor architecture, the efficacy of the particular

power reduction method that is used must be assessed by understanding the net performance impact. Here, depending on the application domain (or market), a PDP, EDP or ED^2P metric for evaluating and comparing power-performance efficiciencies must be used. (See earlier discussion in section "Power-Performance Efficiency Metrics").

Optimal Pipeline Depth

A fundamental question that is asked has to do with pipeline depth. Is a deeply pipelined, high frequency ("speed demon") design better than an IPC-centric lower frequency ("braniac") design? In the context of the topic of this entry, "better" must be judged in terms of power-performance efficiency.

Let us consider, first, a simple, hazard-free, linear pipeline flow process, with k stages. Let the time for the total logic (without latches) to compute one answer be T. Assuming that the k stages into which the logic is partitioned are of equal delay, the time per stage and thus the time per computation becomes (see [39], Chap. 2):

$$t = T/k + D \quad (8)$$

where D is the delay added due to the staging latch. The inverse of t determines the clocking rate or frequency of operation. Similarly, if the energy spent (per cycle, per second or over the duration of the program run) in the logic is W and the corresponding energy spent per level of staging latches is L, then the total energy equation for the k-stage pipelined version is roughly,

$$E = L.k + W \quad (9)$$

The energy equation assumes that the clock is free-running, i.e., on every cycle, each level of staging latches is clocked to enable the advancement of operations along the pipeline. (Later, we shall consider the effect of clock-gating). Equations 8 and 9, when plotted as a function of k, are depicted in Fig. 4a and b respectively.

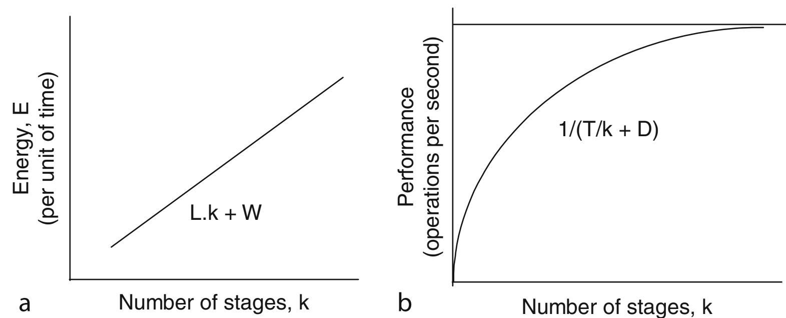
As the number of stages increases, the energy or power consumed increases linearly; while, the performance also increases, but not as fast. In order to consider the PDP-based power-performance efficiency, we compute the ratio:

$$\frac{\text{Power}}{\text{Performance}} = (L.k + W)\,(T/k + D) = L.T + W.D + \left(L.D.k^2 + W.T\right)/k \quad (10)$$

Figure 5 shows the general shape of this curve as a function of k. Differentiating the right hand side expression in Eq. 10 and setting it to zero, one can solve for the optimum value of k for which the power-performance efficiency is maximized, i.e., the minimum of the curve in Fig. 4b can be shown to occur when

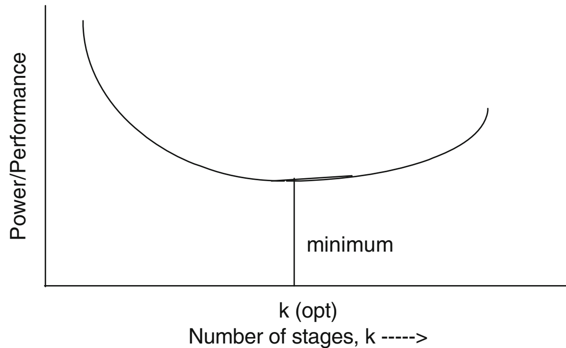
$$k\,(\text{opt.}) = \sqrt{(W.T)/(L.D)} \quad (11)$$

Larson [42] first published the above analysis, albeit from a cost/performance perspective. This analysis shows that, at least for the simplest, hazard-free pipeline flow, the highest frequency operating point achievable in a given technology may not be the most energy-efficient! Rather, the optimal number of stages (and hence operating frequency) is expected to be at a point which increases for greater W or T and decreases for greater L or D. For a prior generation POWER4-class (~0.18 μ) super scalar processor operating at around 1 GHz, [16, 60], the floating point arithmetic unit is estimated to yield values of T = 7.5 ns, D = 0.15 ns,



Power Wall. Fig. 4 Power and performance curves for idealized pipeline flow

P



Power Wall. Fig. 5 Power–performance ratio curve for idealized pipeline flow

$W = 0.15$ W, and $L = 0.1$ W. This yields a k(opt.) ~ 8 (rounded down from 8.67), if we use the idealized formalism (Eq. 11) above.

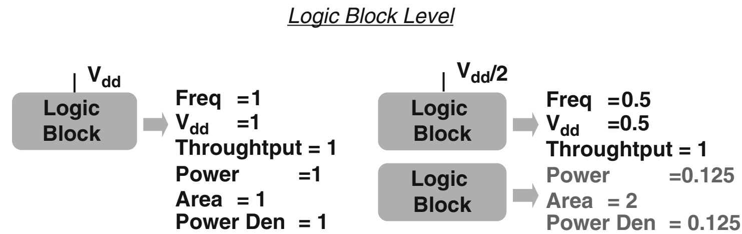
For real super scalar machines, the number of latches in the overall design tends to go up much more sharply with k than the linear assumption in the above model. This tends to make k (opt) even smaller. Also, in real pipeline flow with hazards, e.g., in the presence of branch-related stalls and disruptions, performance actually peaks at a certain value of k before decreasing [17, 24] (instead of the asymptotically increasing behavior shown in Fig. 4a). This effect would also lead to decreasing the effective value of k (opt). (However, k(opt) increases if we use EDP or ED^2P metrics instead of the PDP metric used.). As the number of pipeline stages (k) is increased for a given computation data path, we say that the *depth* of the pipeline increases. This also implies that the levels of combinational logic within each pipeline stage decreases; so, the combinational logic delay per stage decreases as k increases. In detailed simulation-based analysis of a POWER4-class super scalar machine, it has been shown [56, 68] that the optimal pipeline depth using a ED^2P metric like $(BIPS)^3/W$ (where BIPS is the standard performance metric of billions of instructions completed per second) corresponds to around 18 FO4 (Fan-out-of-four (FO4) delay is defined as the delay of one inverter driving four copies of an equally sized inverter. The amount of logic and latch overhead per pipeline stage is often measured in terms of FO4 delay. Decreasing logic FO4 delay per pipeline stage means deeper pipelines (larger values of k) and vice versa.) per pipe stage for SPEC2000 workloads. For commercial workloads like TPC-C, the optimal point is shown to shift to shallower pipelines (25–28 FO4). In contrast, note that if one considered a power-unaware performance-only metric, like BIPS, the optimal pipeline depth for SPEC2000 is around 10 FO4 per stage. For TPC-C, the performance-only optimal point is reported to be [56, 68] pretty flat across the 10–14 FO4 points. For scientific workloads, loop tuning for performance optimization [7] can alter measured power-performance efficiency metrics significantly in some cases. Compiler techniques for super scalar efficiency enhancements are omitted for brevity in this entry.

Exploiting Parallelism to Scale the Power Wall

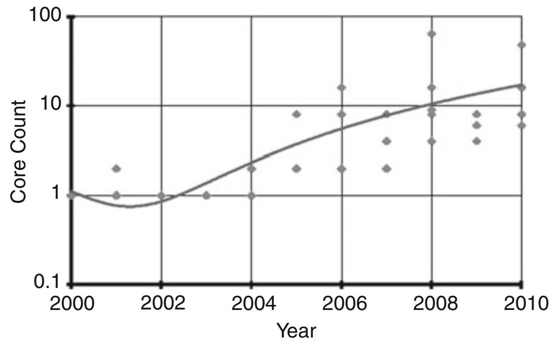
As single-thread frequency and performance growth stalls (driven by technology trends), multi-core parallelism is the new trend. Figure 6 is shown to explain the fundamentals of why parallelism helps power efficiency. If a single task, executed at a given voltage-frequency point, can be split into two independent tasks, each operating at half the voltage and frequency, the net throughput performance remains the same, but the active power density scales down by a factor of 8. A particular application of this concept, even at the level of a single processor core is that of SIMD acceleration as described in the next sub-section. Figure 7 depicts the current trend [28] of growth on number of cores over technology generation in the current regime of power density constrained microprocessor design.

Vector/SIMD Processing and Hybrid Architectures

Vector/SIMD modes of parallelism present in current architectures afford a power-efficient method of extending performance for vectorizable codes. Fundamentally, this is because: for doing the work of fetching and processing a single (vector) instruction, a large amount of data is processed in a parallel or pipelined manner. If we consider a SIMD machine, with p k-stage functional pipelines then looking at the pipelines alone, one sees a p-fold increase of performance, with a p-fold increase in power, assuming full utilization and hazard-free flow, as before. Thus, an SIMD pipeline unit offers the potential of scalable growth in performance, with commensurate growth in power, i.e., at constant power-performance efficiency. If, however, one includes the front-end instruction cache and fetch/dispatch unit



Power Wall. Fig. 6 The classic argument of how parallelism can be exploited to reduce power



Power Wall. Fig. 7 Microprocessor core count over time [28]

that are shared across the *p* SIMD pipelines, then power-performance efficiency actually grows with *p*. This is because, the power dissipation behavior of the instruction cache (memory) and the fetch/decode path remains essentially invariant with *p*, while net performance grows linearly with *p*. In terms of net energy, the front-end consumption actually decreases significantly in SIMD mode, since the number of instructions executed is much less than in scalar mode; and overall, since the execution time decreases, the net savings in leakage energy usually results in a significant net positive benefit for the full machine.

The SIMD extension is actually an example of the general concept of using power-efficient specialized hardware (or accelerators) as part of processor design. Such accelerators can be turned off (e.g., power-gated off) when not in use, while significant pieces of the general purpose ("scalar") core can be switched off when the program encounters a long burst of a code that can be offloaded to an active accelerator. Such accelerators can be fixed function, "table-lookup"-type logic at one extreme, all the way through to programmable sub-cores at the other end of the spectrum. The concept of hybrid or heterogeneous core architectures [41] is an example of the latter extreme. In a super scalar machine with a vector/SIMD (or other accelerator) extension, the overall power-efficiency increase is limited by the fraction of code that runs in vector/SIMD (or other accelerator) mode (per Amdahl's Law).

Clock-Gating and Microarchitectural Support

Clock-gating refers to circuit-level control (e.g., see [23, 63]) for disabling the clock to a given set of latches, a macro, a bus, to a cache or register file access path, or an entire unit, on a particular machine cycle. In current generation server-class microprocessors, about 50–70% of the active (switching) power is consumed by the clock distribution network and its latch load alone. As reported in [9], the major part of the clock power is dissipated close to the leaf nodes of the clock tree that drive latch banks. Since a clock-gated latch keeps its current data value stable, clock gating prevents signal transitions of invalid data from propagating down the pipeline thereby reducing switching power in the combinational logic between latches. In addition to reducing dynamic power, clock gating can also reduce static (leakage) power. As already explained, leakage current in CMOS devices is exponentially dependent on temperature. The temperature reduction brought on by clock-gating can therefore significantly reduce the leakage power as well.

(Micro)architectural support for conventional clock-gating can be provided in at least three ways: (a) dynamic detection of idle modes in various clocked units or regions within a processor or system; (b) static or dynamic prediction of such idle modes; (c) using "data valid" bits within a pipeline flow path to selectively enable/disable the clock applied to the pipeline stage latches. If static prediction is used, the compiler inserts special "nap/doze/sleep/wake" type instructions

where appropriate, to aid the hardware in generating the necessary gating signals. Methods (a) and (b) result in coarse-grain clock-gating, where entire units, macros or regions can be gated off to save power; while, method (c) results in fine-grain clock-gating, where unutilized pipe segments can be gated off during normal execution within a particular unit, like the FPU. The detailed circuit-level implementation of gated-clocks, the potential performance degradation, inductive noise problems, etc., are not discussed in this entry. However, these are very important issues that must be dealt with adequately in today's power-constrained designs.

Referring back to section "A Review of Key Ideas in Power-Aware Architectures" and Fig. 5, note that since (fine-grain) clock-gating effectively causes a fraction of the latches to be "gated off," we may model this by assuming that the effective value of L decreases when such clock-gating is applied. This has the effect of increasing k (opt.), i.e., the operating frequency for the most power-efficient pipeline operation can be increased in the presence of clock-gating. This is an added benefit.

In recently reported work [30], the limits of clock-gating efficiency has been examined and then stretched by adding a couple of new advances: transparent pipeline clock-gating (TCG) [31] and elastic pipeline clock-gating (ECG) [32]. TCG introduces a new way of clock-gating pipelines. In traditional clock-gating, latches are held opaque to avoid data races between adjacent latch stages; thus, N clock pulses are needed to propagate a single data item through an N-stage pipeline, even if at a given clock cycle all other (i.e., $N-1$) stages have invalid input data. In a transparent clock-gated pipeline, latches are held *transparent* by default. TCG is based on the concept of *data separation*. Assume that a pair of data items A and B simultaneously move through a TCG pipeline. A data race between A and B is avoided by separating the two data items by clocking or gating a latch stage opaque, such that the opaque latch stage acts as a barrier separating the two data items from each other. The number of clock pulses required for a data item A to move through an N-stage pipeline is no longer only dependent on N, but also on the number of clock cycles that separate A from the closest upstream data item B. For an N-stage pipeline, where B follows n clock cycles behind A, only floor (N/n) clock pulses have to be generated to move A safely through the pipeline. Elastic pipeline clock gating (ECG) is a different technique that achieves further efficiency by exploiting the inherent storage redundancy afforded by a traditional master–slave latch pair. ECG allows the designer to allow stall signals to propagate backward in pipeline flow logic in a stage-by-stage fashion, without incurring the leakage power and area overhead of explicitly inserted stall buffers. Logic-level details of TCG and ECG are available in the originally published papers [30–32]. As reported there, TCG enables clock power reduction to the tune of 50% over traditional stage-level clock gating under commercial (TPC-C) class workloads. Even under heavy floating point workloads where fewer bubbles are available in the pipeline, the clock power in the floating point pipeline can be reduced by 34%. The significant reduction in dynamic stall power (27%) and leakage power (44%) afforded by ECG in a floating point unit design have also been reported in the published literature.

Predictive Power Gating

As previously indicated, leakage power is a major (if not dominant) component of total power dissipation in current and future CMOS microprocessors. Cutting off the power supply (Vdd) to major circuit blocks to conserve idle power ("sleep mode") is not a new concept, especially for batter-powered mobile systems. However, dynamically effecting such gating, on a unit-by-unit basis as function of input workload demand is not a design technique that has seen widespread usage yet, especially in server-class microprocessors. The main reasons have been the perceived risks or negative effects arising from: (a) performance and area overheads; (b) inductive noise on the power supply grid; and (c) potential design tools and verification concerns. Advances in circuit design have minimized the area and cycle-time delay overhead concerns in recent industrial practices. Microarchitectural predictive techniques (e.g., [18, 27, 38, 44]) have recently advanced to the point where now they equip designers with the tools needed to minimize any architectural performance overheads as well. The inductive noise concerns do persist, but there are known solution approaches in the realm of power distribution networks and package design that will no doubt mature to help mitigate those concerns. The design tools and verification challenge, of course

will prevail as a difficult roadblock – but again, solutions will eventually emerge to get rid of that concern. Per-core power gating, within a multi-core setting has recently gained acceptance within the chip design community (e.g., see Intel's Nehalem processor design [54]). The power-efficiency benefits of per-core power gating have been quantified recently for server and data center class workloads [43, 45].

Variable Bit-Width Operands

One of the techniques proposed for reducing dynamic power consists of exploiting the behavior of data in programs, which is characterized by the frequent presence of small values. Such values can be represented as and operated upon as short bit-vectors. Thus, by using only a small part of the processing datapath, power can be reduced without loss of performance. Brooks and Martonosi [10] analyzed the potential of this approach in the context of 64-bit processor implementations (e.g., the Compaq Alpha^{TM} architecture). Their results show that roughly 50% of the instructions executed had both operands whose length was less than or equal to 16 bits. Brooks and Martonosi proposed an implementation that exploits this by dynamically detecting the presence of narrow-width operands on a cycle-by-cycle basis.

Adaptive Microarchitectures

Another method of reducing power is to adjust the size of various storage resources within a processor or system, with changing needs of the workload. Albonesi [2] proposed a dynamically reconfigurable caching mechanism, that reduces the cache size (and hence power) when the workload is in a phase that exhibits reduced cache footprint. Such downsizing also results in improved latency, which can be exploited (from a performance viewpoint) by increasing the cache cycling frequency on a local clocking or self-timed basis. Maro et al. [47] have suggested the use of adapting the functional unit configuration within a processor in tune with changing workload requirements. Reconfiguration is limited to "shutting down" certain functional pipes or clusters, based on utilization history or IPC performance. In that sense, the work by Maro et al. is not too different from coarse-grain clock-gating support, as discussed earlier. In early work done at IBM Watson, Buyuktosunoglu et al. [13] designed an adaptive issue queue that can result in (up to) 75% power reduction when the queue is sized down to its minimum size. This is achieved with a very small IPC performance hit. Another example is the idea of adaptive register files (e.g., see [14]) where the size and configuration of the active size of the storage is changed via a banked design, or through hierarchical partitioning techniques. A recent tutorial article by Albonesi et al. [3] provides an excellent coverage of advances in the field of adaptive architectures.

Dynamic Thermal Management

Most clock-gating techniques are geared toward the goal of reducing *average* chip power. As such, these methods do not guarantee that the worst-case (maximum) power consumption will not exceed safe limits. The processor's maximum power consumption dictates the choice of its packaging and cooling solution. In fact, as discussed in [23], the net cooling solution cost increases in a piecewise linear manner with respect to the maximum power, and the cost gradient increases rather sharply in the higher power regimes. This necessitates the use of mechanisms to limit the maximum power to a controllable ceiling, one defined by the cost profile of the market for which the processor is targeted. Most recently, in the high performance world, Intel's Pentium 4 processor is reported to use an elaborate on-chip thermal management system to ensure reliable operation [23]. At the lower end, the G3 and G4 PowerPC microprocessors [50, 52] include a Thermal Assist Unit (TAU) to provide dynamic thermal management. In recently reported academic work, Brooks and Martonosi [11] discuss and analyze the potential reduction in "maximum power" ratings without significant loss of performance, by the use of specific dynamic thermal management (DTM) schemes. The use of DTM requires the inclusion of on-chip sensors to monitor actual temperature, or proxies of temperature [11] estimated from on-chip counters of various events and rates.

Dynamic Throttling of Communication Bandwidths

This idea has to do with reducing the width of a communication bus dynamically, in response to reduced needs or in response to temperature overruns. Examples of on-chip buses that can be throttled are:

instruction fetch bandwidth, instruction dispatch/issue bandwidths, register renaming bandwidth, instruction completion bandwidths, memory address bandwidth, etc. In the G3 and G4 PowerPC microprocessors [50, 52], the TAU invokes a form of instruction cache throttling as a means to lower the temperature when a thermal emergency is encountered.

Speculation Control

In current generation, high performance microprocessors, branch mispredictions and mis-speculative prefetches end up wasting a lots of power. Manne et al. [22, 46] and Karkhanis et al. [37] have described means of detecting or anticipating an impending mispredict and using that information to prevent mis-speculated instructions from entering the pipeline. These methods have been shown to reduce energy waste, with minimal impact on performance.

Power-Efficient Microarchitecture Paradigms

Now that we have examined specific microarchitectural constructs that aid power-efficient design, let us briefly examine the inherent power-performance scalability and efficiency of selected paradigms that are currently established or are emerging in the high-end processor roadmap. In particular, we consider: (a) wide-issue, speculative super scalar processors; (b) multi-cluster superscalars; (c) simultaneously multithreaded (SMT) processors; and (d) chip multiprocessors (CMP): those that use single program speculative multithreading, as well as those that are general multi-core SMP or throughput engines.

Single-Core Superscalar Processor Paradigm

One school of thought anticipates a continued progression along the path of wider, aggressively superscalar paradigms. Researchers continue to innovate in an attempt to extract the last "ounce" of IPC-level performance from a single-thread instruction-level parallelism (ILP) model. Value prediction advances (pioneered by Lipasti et al. [62]) promise to break the limits imposed by true data dependencies. Trace caches (Smith et al. [62]) ease the fetch bandwidth bottleneck, which can otherwise impede scalability. However, increasing the superscalar width beyond a certain limit tends to yield diminishing gains in *net* performance. At the same time, the power-performance efficiency metric (e.g., performance per watt or $(\text{performance})^2$/watt, etc) tends to degrade beyond a certain complexity point in the single-core superscalar design paradigm.

The microarchitectural trends beyond the current superscalar regime are effectively targeted toward the goal of extending the power-performance efficiency factors. Whenever we reach a maximum in the power-performance efficiency curve, it is time to invoke the next paradigm shift.

Next, we examine some of the promising new trends in microarchitecture that can serve as the next platform for designing power-performance scalable machines.

Multicluster Superscalar Processors

Zyuban et al. [66, 69] studied the class of multicluster superscalar processors as a means of extending the power-efficient growth of the basic super scalar paradigm. One way to address the energy growth problem at the microarchitectural level is to replace a classical superscalar CPU with a set of clusters, so that all key energy consumers are split among clusters. Then, instead of accessing centralized structures in the traditional superscalar design, instructions scheduled to an individual cluster would access local structures most of the time. The main advantage of accessing a collection of local structures instead of a centralized one is that the number of ports and entries in each local structure is much smaller. This reduces the latency and energy per access. If the non-accessed sub-structures of a resource can be "gated off" (e.g., in terms of the clock), then, the net energy savings can be substantial.

According to the results obtained in Zyuban's work, the energy dissipated per cycle in every unit or sub-unit within a superscalar processor can be modeled to vary (approximately) as $\text{IPC}_{\text{unit}} * (\text{IW})^g$, where IW is the issue width, IPC_{unit} is the average IPC performance at the level of the unit or structure under consideration, and g is the energy growth parameter for that unit. Then, the energy-delay product (EDP) for the particular unit would vary as:

$$\text{EDP}_{\text{unit}} \frac{\text{IPC}_{\text{unit}} * (\text{IW})^g}{\text{IPC}_{\text{overall}}} \tag{12}$$

Zyuban shows that for real machines, where the overall IPC always increases with issue width in a sublinear manner, the overall EDP of the processor can be bounded as:

$$(IPC)^{2g-1} \leq EDP \leq (IPC)^{2g} \quad (13)$$

where g is the energy-growth factor of a given unit and IPC refers to the overall IPC of the processor; and IPC is assumed to vary as $(IW)^{0.5}$. Thus, according to this formulation, superscalar implementations that minimize g for each unit or structure will result in energy-efficient designs. The eliot/elpaso tool does not model the effects of multi-clustering in detail; however, from Zyuban's work, we can infer that a carefully designed multicluster architecture has the potential of extending the power-performance efficiency scaling beyond what is possible using the classical superscalar paradigm. Of course, such extended scalability is achieved at the expense of reduced IPC performance for a given superscalar machine width. This IPC degradation is caused by the added inter-cluster communication delays and other power management overhead in a real design. Some of the IPC loss (if not all) can be offset by a clock frequency boost which may be possible in such a design, due to the reduced resource latencies and bandwidths.

High-performance processors (e.g., the Compaq Alpha 21264 and the IBM POWER4/5) certainly have elements of multi-clustering, especially in terms of duplicated register files and distributed issue queues. Zyuban proposed and modeled a specific multicluster organization in his work. This simulation-based study determined the optimal number of clusters and their configurations, for the EDP metric.

Simultaneous Multithreading (SMT)

Let us examine the SMT paradigm [64] to understand how this may affect our notion of power-performance efficiency. With SMT, assume that we can fetch from two threads (simultaneously, if the icache is dual-ported, or in alternate cycles if the icache remains single-ported). The back-end execution engine is shared across the two threads, although each thread has its own architected register space. This facility allows the utilization factors, and the net throughput performance to go up, without a commensurate increase in the maximum clocked power. This is because, the issue width W is not increased, but the execution and resource stages or slots can be filled up simultaneously from both threads. The added complexity in the front-end, of maintaining two program counters (fetch streams) and the global register space increase adds to the power a bit, but the throughput gain is significantly higher. SMT is clearly a very area-efficient (and hence leakage-efficient) microarchitectural paradigm. IBM's POWER processor family has progressed from 2-way SMT per core in POWER5 [35] to 4-way SMT per core in POWER7 [34].

Seng and Tullsen [53] presented analysis to show that using a suitably architected SMT processor, the per-thread speculative waste can be reduced, while increasing the utilization of the machine resources by executing simultaneously from multiple threads. This was shown to reduce the average energy per instruction by 22%.

Chip Multiprocessing

In a multiscalar-like speculative-multi-threading machine [55], different iterations of a single loop program could be initiated as separate tasks or threads on different core processors on the same chip. Iterations beyond the first one are speculatively executed, assuming no loop-carried dependencies. Register values set in one task are forwarded in sequence to dependent instructions in subsequent tasks. Execution on each processor proceeds speculatively, assuming the absence of load-store address conflicts between tasks; dynamic memory address disambiguation hardware is required to detect violations and restart task executions as needed. In this paradigm, if the performance can be shown to scale well with the number of tasks, and if each processor is designed as a limited-issue, limited-speculation (low complexity) core, it is possible to achieve better overall scalability of performance-power efficiency.

A key real trend in high-end microprocessors is classical chip multiprocessing (CMP), where multiple user programs (or sub-tasks of a single program) execute separately (and non-speculatively) on different processors on the same chip. A commonly used paradigm in this case is that of (shared memory) symmetric multiprocessing (SMP) on a chip (see Hammond et al. in [62]). Larger SMP server nodes can be built from such chips. Server system vendors like IBM have relied on such CMP paradigms as the scalable paradigm for the immediate future. IBM's POWER4 and POWER5 designs [16, 35, 60] were the first examples of this industry-wide trend; later examples being:

Intel's Montecito [48] and Sun's Niagara [40]. Most recent server-class multi-core/multi-threaded processors include Intel's Nehalem [54] and IBM's POWER7 [34]. Such CMP designs offer the potential of convenient coarse-grain clock-gating and "power-down" modes, where one or more processors on the chip may be "turned off" or "slowed down" to save power when needed.

In general, in a design era where technological constraints like power dissipation have caused a significant slowdown in the growth of single-thread execution clock frequency (and performance), parallelism via multiple cores on a die, each operating at lower than achievable single-core frequency (i.e., at larger FO4 per stage than prior generation processors), is the natural paradigm of choice for power-efficient, scalable performance growth through the next several generations of processor design. Heterogeneity in the form of different types of cores, accelerators and mixed signal components [20, 33, 41] or variable per-core frequency support are already prevalent concepts. Such heterogeneity is needed to support the diverse set of workloads that will enable the next generation of computing systems in a power-constrained design era.

Conclusions

In this entry, we first defined the "power wall" and examined the root technological causes behind the onset of the current power-constrained design era. We then discussed issues related to power-performance efficiency and metrics from an architect's viewpoint. We showed that depending on the application domain (or market), it may be necessary to adopt one metric over another in comparing processors within that segment. Next, we described some of the promising new ideas in power-aware microarchitecture design. This discussion included circuit-centric solutions like clock-gating, and power-gating where microarchitectural support is needed to make the right decisions at run time. We then looked at the trends in core- and chip-level microarchitecture design paradigms at a high-level, given the reality of the power wall. We limited our focus to a few key ideas and paradigms of interest in future power-aware processor design. Many new ideas to address various aspects of power reduction have been presented in recent research papers. All of these could not be discussed in this entry; but the interested reader should certainly refer to the cited references and recent conference publications for further detailed study.

Bibliography

1. Agrawal A, Mukhopadhyay S, Raychowdhury A, Roy K, Kim CH (2006) Leakage power analysis and reduction in nanoscale circuits. IEEE Micro 26(2):68–80
2. Albonesi D (1998) Dynamic IPC/clock rate optimization. In: Proceedings of the 25th annual international symposium on computer architecture (ISCA), ACM/IEEE Computer Society, Barcelona, pp 282–292
3. Albonesi DH, Balasubramonian R, Dropsho SG, Dwarkadas S, Friedman EG, Huang MC, Kursun V, Magklis G, Scott ML, Semeraro G, Bose P, Buyuktosunoglu A, Cook PW, Schuster SE (2003) Dynamically tuning processor resources with adaptive processing. IEEE Comput 36(12):49–58, Special Issue on Power-Aware Computing
4. Barroso L, Holzle U (2007) The case for energy proportional computing. IEEE Comput 40(12):33–37
5. Bohr M (2007) A 30 year retrospective on dennard's MOSFET scaling paper. P Solid St Circ Soc 12(1):11–13
6. Borkar S (1999) Design challenges of technology scaling. P IEEE Micro 19(4):23–29
7. Bose P, Kim S, O'Connell FP (1999) Ciarfella WA Bounds modeling and compiler optimizations for superscalar performance tuning. J Syst Architect 45:1111–1137; Elsevier
8. Brews JR (1979) Subthreshold behavior of uniformly and non-uniformly doped long-channel MOSFET. IEEE T Electron Dev 26:1282–1291
9. Brooks D, Bose P, Srinivasan V, Gschwind MK, Emma PG, Rosenfield MG (2003) New methodology for early-stage, microarchitecture-level power-performance analysis of microprocessors. IBM J Res Dev 47(5/6):653–662
10. Brooks D, Martonosi M (1999) Dynamically exploiting narrow width operands to improve processor power and performance. In: Proceedings of the 5th international symposium on high-performance computer architecture (HPCA-5), IEEE Computer Society, Orlando
11. Brooks D, Martonosi M (2001) Dynamic thermal management for high-performance microprocessors. In: Proceedings of the 7th internatinal symposium On high performance computer architecture, IEEE Computer Society, Nuevo Leone, pp 20–24
12. Brooks DM, Bose P, Schuster SE, Jacobson H, Kudva PN, Buyuktosunoglu A, Wellman J-D, Zyuban V, Gupta M, Cook PW (2000) Power-aware microarchitecture: design and modeling challenges for next-generation microprocessors. P IEEE Micro 20(6):26–44
13. Buyuktosunoglu A et al (2000) An adaptive issue queue for reduced power at high performance. In: Proceedings ISCA Workshop on complexity-effective design (WCED), Vancouver
14. Cruz J-L, Gonzalez A, Valero M, Topham NP (2000) Multiple-banked register file architectures. In: Proceedings of the international symposium on computer architecture (ISCA), Vancouver, pp 316–325

15. Dennard R et al (1974) Design of ion-implanted MOSFETs with very small physical dimensions. IEEE J Solid St Circ SC-9(5):256–268
16. Diefendorff K (1999) POWER4 focuses on memory bandwidth. Microprocessor Report 13(13):11–17
17. Dubey PK, Flynn MJ (1990) Optimal pipelining. J Parallel Distr Com 8(1):10–19
18. Flautner K, Kim NS, Martin S, Blaauw D, Mudge T (2002) Drowsy Caches: simple techniques for reducing leakage power. In: Proceedings of the international symposium on computer architecture (ISCA), IEEE Computer Society, Anchorage
19. Flynn MJ, Hung P, Rudd K (1999) Deep-submicron microprocessor design issues. P IEEE Micro 19(4):11–22
20. Gara A et al (2005) Overview of the blue gene/L system architecture. IBM J Res Dev 49(2/3):195–212
21. Gonzalez R, Horowitz M (1996) Energy dissipation in general purpose microprocessors. IEEE J Solid-St Circ 31(9):1277–1284
22. Grunwald D, Klauser A, Manne S, Pleszkun, A (1998) Confidence estimation for speculation control. In: Proceedings 25th annual international symposium on computer architecture (ISCA), ACM/IEEE Computer Society, Barcelona, pp 122–131
23. Gunther SH, Binns F, Carmean DM, Hall JC (2000) Managing the impact of increasing microprocessor power consumption. In: Proceedings of the Intel Technology Journal
24. Hartstein A, Puzak TR (2002) The optimum pipeline depth for a microprocessor. Proceedings of the 29th international symposium on computer architecture (ISCA-29), IEEE Computer Society, Anchorage
25. http://www.specbench.org
26. Hu C (1996) Device and technology impact on low power electronics. In: Rabaey J (ed) Low power design methodologies. Kluwer, Boston, pp 21–35
27. Hu Z, Buyuktosunoglu A, Srinivasan V, Zyuban V, Jacobson H, Bose P (2004) Microarchitectural techniques for power gating of execution units. In: Proceedings of the international symposium on low power electronics and design (ISLPED), ACM, Newport Beach
28. ISSCC (International Solid State Circuits Conference) (2010) Trends Report, http://isscc.org/doc/2010/ISSCC2010_TechTrends.pdf
29. Iyer A, Marculescu D (2002) Power-performance evaluation of globally asynchronous, locally synchronous processors. In: Proceedings of the international symposium on computer architecture (ISCA), IEEE Computer Society, Anchorage
30. Jacobson H, Bose P, Hu Z, Eickemeyer R, Eisen L, Griswell J (2005) Stretching the limits of clock-gating efficiency in server-class processors. In: Proceedngs of the international symposium on high performance computer architecture (HPCA), IEEE Computer Society, San Francisco
31. Jacobson HM (2004) Improved clock-gating through transparent pipelining. In: Proceedings of the international symposium on low Power electronics and design (ISLPED), ACM, California
32. Jacobson HM et al (2002) Synchronous interlocked pipelines. In: Proceedings of the international symposium on advanced research in asynchronous circuits and systems, IEEE Computer Society, Manchester
33. Kahle JA et al (2005) Introduction to the Cell multiprocessor. IBM J Res Dev 49(4/5):589–604
34. Kalla R, Sinharoy B, Starke WJ, Floyd MJ (2010) Power7: IBM's next generation server processor. IEEE Micro 30(2):7–15
35. Kalla R, Sinharoy B, Tendler J (2004) IBM POWER5 chip: a dual-core multithreaded processor, IEEE Micro 24(2):40–47
36. Kanda K et al (2001) Design impact of positive temperature dependence on drain current in sub-1-V CMOS VLSIs. IEEE JSSC, 36(10):1559–1564
37. Karkhanis T, Bose P, Smith J (2002) Saving energy with just in time instruction delivery. In: Proceedings of the intenational symposium on low power electronics and design (ISLPED), ACM, Monterey
38. Kaxiras S, Hu Z, Martonosi M (2001) Cache Decay: exploiting generational behavior to reduce cache leakage power. In: Proceedings of the international symposium on computer architecture (ISCA), Goteborg
39. Kogge PM (1981) The architecture of pipelined computers. Hemisphere Publishing Corporation, New York
40. Kongetira P (2004) A 32-way multithreaded SPARC® processor. Presented at Hot Chips
41. Kumar R, Tullsen D, Jouppi N, Ranganathan P (2005) Heterogeneous chip multiprocessors. IEEE Comput 38(11):32–38
42. Larson AG (1973) Cost-effective processor design with an application to fast fourier transform computers. Digital systems laboratory report SU-SEL-73-037, Stanford University, Stanford; see also, Larson and Davidson (1973) Cost-effective design of special purpse processors: a fast fourier transform case study. In: Proceedings 11th annual allerton conference on circuits and system theory. University of Illinois, Champaihn-Urbana, pp 547–557
43. Leverich J, Monchiero M, Talwar V, Ranganathan P, Kozyrakis C (2009) Power management of datacenter workloads using per-core power gating. IEEE Comput Archit Lett 8(2): 48–51
44. Lungu A, Bose P, Buyuktosunoglu A, Sorin D (2009) Dynamic power gating with quality guarantees. In: Proceedings of the international symposium on low power electronics and design (ISLPED), ACM, New York
45. Madan NS, Buyuktosunoglu A, Bose P, Annavaram M (2011) Guarded power gating in a multi-core setting, presented at ISCA workshop on energy-efficient design (WEED), June 2010; to appear as a Lecture notes on computer science (LNCS) volume in 2010; see also full paper in Proceedings of the 17th international Symposium on high performance computer architecture (HPCA)
46. Manne S, Klauser A, Grunwald D (1998) Pipeline gating: speculation control for energy reduction. In: Proceedings of the 25th annual international symposium on computer architecture (ISCA), ACM/IEEE Computer Society, Barcelona, pp 132–141
47. Maro R, Bai Y, Bahar RI (2000) Dynamically reconfiguring processor resources to reduce power consumption in high-performance processors. In: Proceedings of power aware computer systems (PACS) Workshop, held in conjunction with ASPLOS, Cambridge
48. McNairy C, Bhatia R (2005) Montecito: a dual-core, dual-thread Itanium Processor. IEEE Micro 25(2):10–20 (see also, Hot Chips 2004)

49. Oklobdzija VG (1998) Architectural tradeoffs for low power. In: Proceedings of the ISCA Workshop on Power-Driven Microarchitectures, Barcelona
50. Reed P et al (1997) 250 MHz 5 W RISC microprocessor with on-chip L2 cache controller. Dig Tech Pap IEEE Int Solid St Circ Conf 40:412
51. Sakurai T, Newton R (1990) Alpha-power law MOSFET model and its applications to CMOS inverter delay and other formulas. IEEE JSSC 25(2):584–594
52. Sanchez H et al (1997) Thermal management system for high performance PowerPC microprocessors. In: Digest of papers, IEEE COMPCON, p 325
53. Seng JS, Tullsen DM, Cai G (2000) The power efficiency of multithreaded architectures. In: Invited talk presented at: ISCA Workshop on Complexity-Effective Design (WCED), Vancouver
54. Singhal R (2008) Inside intel core microarchitecture (Nehalem). Presented at Hot Chips-20, Stanford
55. Sohi G, Breach SE, Vijaykumar TN (1995) Multiscalar Processors. In: Proceedings of the 22nd annual international symposium on computer architecture, IEEE CS Press, Los Alamitos, pp 414–425
56. Srinivasan V, Brooks D, Gschwind M, Bose P, Zyuban V, Strenski PN, Emma PG (2002) Optimizing pipelines for power and performance. In: Proceedings of the 35th annual IEEE/ACM symposium on microarchitecture (MICRO-35), ACM/IEEE, Istanbul
57. Talks presented at the ISCA workshops on complexity effective design (WCED-2000 through WCED-2006), http://www.csl.cornell.edu/~albonesi/wced.html
58. Talks presented at the kool chips workshops (1998) http://www.cs.colorado.edu/~grunwald/LowPowerWorkshop
59. Talks presented at the power aware computer systems (PACS) Workshops; e.g. the 2004 offering: http://www.ece.cmu.edu/~pacs04/
60. Tendler JM, Dodson JS, Fields JS Jr, Le H, Sinharoy B (2002) POWER4 system microarchitecture. IBM J Res Dev 46:(1):1–116
61. The International Technology Roadmap for Semiconductors. http://www.itrs.net/reports.html
62. Theme issue (1997) The future of processors. IEEE Comput 30(9):97–93
63. Tiwari V et al (1998) Reducing power in high-performance microprocessors. In: Proceedings of the IEEE/ACM Design Automation Conference. ACM, New York, pp 732–737
64. Tullsen DM, Eggers SJ, Levy HM (1995) Simultaneous Multithreading: Maximizing On-Chip parallelism. In: Proceedings of the 22nd annual internatonal symposium on computer architecture, Santa Margherita Ligure, pp 292–403
65. Yen D (2005) Chip multithreading processors enable reliable high throughput computing. Keynote speech at international symposium on reliability physics (IRPS)
66. Zyuban V (2000) Inherently lower-power high performance super scalar architectures. PhD thesis, Department of Computer Science and Engineering, University of Notre Dame
67. Papers in the special issue of IBM Journal of Research and Development, March/May 2002
68. Zyuban V, Brooks D, Srinivasan V, Gschwind M, Bose P, Strenski P, Emma P (2004) Integrated analysis of power and performance for pipelined microprocessors. IEEE T Comput 53(8):1004–1016
69. Zyuban V, Kogge P (2000) Optimization of high-performance superscalar architectures for energy efficiency. In: Proceedings of the IEEE symposium on low power electronics and design, ACM, New York

PRAM (Parallel Random Access Machines)

Joseph F. JaJa
University of Maryland, College Park, MD, USA

Definition

The Parallel Random Access Machine (PRAM) is an abstract model for parallel computation which assumes that all the processors operate synchronously under a single clock and are able to randomly access a large shared memory. In particular, a processor can execute an arithmetic, logic, or memory access operation within a single clock cycle.

Discussion

Introduction

Parallel Random Access Machines (PRAMs) were introduced in the late 1970s as a natural generalization to parallel computation of the *Random Access Machine* (RAM) model. The RAM model is widely used as the basis for designing and analyzing sequential algorithms. The PRAM model assumes the presence of a number of processors, each identified by a unique id, which have access to a single unbounded shared memory. The processors operate synchronously under a single clock such that each processor can execute an arithmetic or logic operation or a memory access operation within a single clock cycle. In general, each processor under the PRAM model can execute its own program, which can be stored in some type of a private program memory. However, almost all the known PRAM algorithms are of the SPMD (Single Program Multiple Data) type in which a single program is executed by all the processors such that an instruction can refer to the id of the processor, which is responsible for executing the corresponding instruction. A processor may or may not be active during any given clock cycle. In fact, most of the

PRAM algorithms are of the SIMD (Single Instruction Multiple Data) type in which, during each cycle, a processor is either idle or executing the same operation as the remaining active processors.

As a simple example of a PRAM algorithm, consider the computation of the sum S of the elements of an array $A[1:n]$ using an n-processor PRAM, where the processors are indexed by $pid = 1, 2, \ldots, n$. A PRAM algorithm can be organized as a balanced binary tree with n leaves such that each internal node represents the sum computation applied to the values available at the children. The PRAM algorithm proceeds level by level, executing all the computations at each level in a single parallel step. Hence the processors complete the process in $\lceil \log n \rceil$ parallel steps. The algorithm is illustrated in Fig. 1.

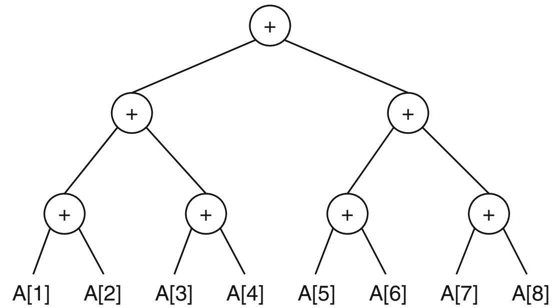
The PRAM algorithm written in the SPMD style is given next where for simplicity $n = 2^k$ for some positive integer k. The array $B[1:n]$ is used to store the intermediate results.

Algorithm 1 PRAM SUM Algorithm

Input: An array A of size $n = 2^k$ stored in shared memory.
Output: The sum of the elements of A.
begin
  $B[pid] = A[pid]$
  $d = n$
  for $h = 1$ to k **do**
    $d = \frac{d}{2}$
    if $pid \le d$ **then**
      $B[pid] = B[2pid - 1] + B[2pid]$
    end if
    end for
  return$(B[1])$
end

Given that the processors can simultaneously access the shared memory within the same clock cycle, several variants of the PRAM model exist depending on the assumptions made for simultaneous access to the **same** memory location.

- **Exclusive Read Exclusive Write (EREW) PRAM**. No simultaneous accesses to the same location are allowed under the EREW PRAM model.



PRAM (Parallel Random Access Machines). Fig. 1 A Balanced Binary Tree for the Sum Computation. The PRAM algorithm proceeds from the leaves to the root while executing the operations at each level within a single clock cycle

- **Concurrent Read Exclusive Write (CREW) PRAM**. Simultaneous read accesses to the same location are allowed but no concurrent write operations to the same location are permitted.
- **Concurrent Read Concurrent Write (CRCW) PRAM**. Simultaneous read or write operations to the same memory location are allowed. This model has several subversions depending on how the concurrent write operations to the same location are resolved. The **Common CRCW PRAM** model assumes that the processors are writing the same value, while the **Arbitrary CRCW PRAM** model assumes that one of the processors attempting a concurrent write into the same location succeeds. The **Priority CRCW** assumes that the indices of the processors are linearly ordered, and allows the processor with the highest priority to succeed in case of a concurrent write.

The different assumptions on the concurrent accesses to a single location lead to parallel algorithms that can differ significantly in performance. Consider, for example, the problem of determining whether all the entries of a binary array $M[1:n]$ are all equal to 1. The output A of this computation is in fact the logical AND of all the entries in M. An n-processor Common CRCW PRAM algorithm can perform this computation as follows. Initially, A is set equal to 1. For all $1 \le pid \le n$, processor pid tests whether the entry $M[pid] = 0$, and in the affirmative, executes the operation $A = 0$. Clearly this algorithm correctly computes the AND of the n elements

P

of M in constant time on the Common CRCW PRAM. However, it can be shown, under a very general model, that a CREW PRAM will require $\Omega(\log n)$ parallel steps regardless of the number of processors available. On the other hand, the computational gap between the various versions of the PRAM model is limited in the sense that the strongest p-processor PRAM model, namely the priority CRCW from our list above, can be simulated by a p-processor EREW (weakest PRAM model) with a slowdown of a factor of $O(\log p)$.

From a theoretical perspective, the design of a PRAM algorithm amounts to the development of a strategy that achieves the highest level of parallelism, that is, the smallest number of parallel steps, using a "reasonable" number of processors. Note that the number of processors can depend on the input size. In particular, the class of "well-parallelizable" problems under the PRAM model can be defined as the class of the problems that can be solved in polylogarithmic number of parallel steps (i.e., $O(\log^k n)$ for some fixed constant k), using a polynomial number of processors. Such a class is referred to as the NC complexity class, which has been related to the circuits model used in traditional computational complexity. See the bibliographic notes for references to related work.

Complexity Measures and Work–Time Framework

A PRAM algorithm can be evaluated by several complexity measures, where a complexity measure is a worst case asymptotic estimate of the performance as a function of the input length n. Given a p-processor PRAM algorithm to solve a problem with input size n, let $T_P(p,n)$ be the number of parallel steps used by the algorithm. The optimal (or often the best known) sequential complexity is denoted by $T_S(n)$. The **speedup** is defined to be

$$\frac{T_S(n)}{T_P(p,n)},$$

which represents the speedup of the parallel algorithm *relative to the best sequential algorithm*. Clearly the best possible speedup is $\Theta(p)$. On the other hand, the **relative speedup** is defined to be

$$\frac{T_P(1,n)}{T_P(p,n)},$$

which refers to the speedup achieved by the algorithm with p processors relative to the same algorithm running with one processor.

Another related measure is the **relative efficiency** of a PRAM algorithm defined as the ratio

$$\frac{T_P(1,n)}{pT_P(p,n)}.$$

The two PRAM algorithms presented earlier assume the presence of n processors, where n is the input size. To derive the complexity measures defined above, these algorithms can be mapped into p-processor ($p < n$) PRAM algorithms by distributing the concurrent operations carried out at each parallel step as evenly among the p processors as possible. In particular, the SUM algorithm can be executed in

$$\left\lceil \frac{n}{p} \right\rceil + \left\lceil \frac{n/2}{p} \right\rceil + \cdots + 1 = O\left(\frac{n}{p} + \log n\right)$$

parallel time on a p-processor PRAM algorithm. Hence the speedup is $\Theta(p)$ whenever $p = O(n/\log n)$. However, writing the corresponding algorithm for each processor (using processor ids) is in general a tedious process that distracts from the simplicity of the PRAM model.

An alternative is to describe a PRAM algorithm in the so-called **Work–Time** or simply **WT** framework, which is also related to the **data parallel** paradigm. The algorithm does not assume any specific number of processors, nor does it refer to any processor index, but rather it is expressed as a sequence of steps, where each step is either a typical sequential operation or a set of concurrent operations that can be executed in parallel. A set of parallel operations is specified by the pseudo-instruction **pardo** or **forall**. For example, the previous PRAM SUM algorithm can be expressed in this framework as shown in Algorithm 2.

Under the WT framework, the complexity of a PRAM algorithm can be measured by two parameters – **work** $W(n)$ and **time** $T_P(n)$. The work $W(n)$ is the *total* number of operations required by the algorithm as a function of n and the time $T_P(n)$ is defined as the number of parallel steps needed by the algorithm assuming an unlimited number of processors. The corresponding functions for the SUM algorithm are $W(n) = O(n)$ and $T_P(n) = O(\log n)$, respectively.

Algorithm 2 Parallel SUM Algorithm in the WT Framework

Input: An array A of size $n = 2^k$ residing in memory.
Output: The sum of the elements of A.
begin
  $d = n$
  for all $1 \le i \le n$ **pardo**
    $B[i] = A[i]$
  end for
  for $h = 1$ to k **do**
    $d = \frac{d}{2}$
    for all $1 \le i \le d$ **pardo**
      $B[i] = B[2i-1] + B[2i]$
    end for
  end for
  return$(B[1])$
end

In general, a WT-parallel algorithm with complexity $W(n)$ and $T_P(n)$ can be simulated on a p-processor PRAM using Brent's scheduling principle (see bibliographic notes) in time

$$T_P(p,n) = O\left(\frac{W(n)}{p} + T_P(n)\right)$$

The algorithm is called **work optimal** if $W(n) = \Theta(T_S(n)$. In this case, the corresponding p-processor PRAM algorithm achieves optimal speedup as long as $p = O\left(\frac{T_S(n)}{T_P(n)}\right)$.

Under the WT framework, a typical goal is to develop an algorithm that achieves the fastest parallel time among the work-optimal algorithms. That is, the main goal to develop a work-optimal algorithm that is as fast as possible.

Basic PRAM Techniques

Basic PRAM techniques are illustrated through the description of PRAM algorithms for the following computations: matrix multiplication, prefix sums or scan, list ranking, fractional independent set, and computing the maximum. The WT framework is used to express these algorithms.

Matrix Multiplication

Given two matrices A and B of dimensions $m \times n$ and $n \times q$, respectively, the product $C = AB$ is a matrix of dimension $m \times q$ such that

$$C[i,j] = \sum_{k=1}^{n} A[i,k]B[k,j],\ 1 \le i \le m,\ 1 \le j \le q.$$

Computing the matrix C on the PRAM model is quite simple since both matrices A and B are initially stored in shared memory, and their entries can each be accessed randomly within a single clock cycle. The algorithm in the WT framework amounts to computing in parallel all the products $A[i,k]B[k,j]$, for $1 \le i \le m$, $1 \le k \le n$, and $1 \le j \le q$. The PRAM SUM algorithm can then be used to compute all the entries $C[i,j]$ in parallel, $1 \le i \le m$ and $1 \le j \le q$. Thus, the parallel complexity of the resulting algorithm is $T_P = O(\log n)$ and the total work is $W = O(nmq)$.

This algorithm is based on the standard sequential algorithm. Should the initial sequential algorithm be one of the faster sequential matrix multiplication algorithm, the corresponding PRAM algorithm will then run in logarithmic parallel time using the same number of operations as the initial algorithm.

Prefix Sums or Scan

Given a set of elements stored in an array $A[1:n]$ and a binary associative operation $\otimes$, the prefix sums of A consist of the n partial sums defined by

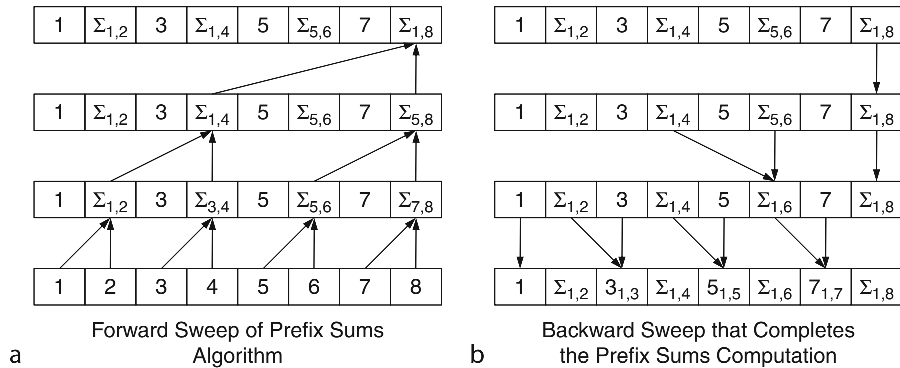
$$PS[i] = A[1] \otimes A[2] \otimes \cdots \otimes A[i], 1 \le i \le n$$

The straightforward sequential algorithm computes $PS[i] = PS[i-1] \otimes A[i]$ for $2 \le i \le n$ starting with $PS[1] = A[1]$, and hence the sequential complexity is $T_S(n) = \Theta(n)$.

A simple fast PRAM algorithm can be designed using a balanced binary tree built on top of the n elements of A. The algorithm consists of a forward sweep through the tree in a way similar to that carried out by the SUM algorithm. A backward sweep will then compute the prefix sums of the set of elements available at each level. A recursive version is described in Algorithm 3.

The algorithm is illustrated in Fig. 2, where the left part illustrates the forward sweep and the right part illustrates the backward sweep.

Since the algorithm involves a forward and a backward sweep through a balanced binary tree of height $O(\log n)$, the parallel complexity of the algorithm satisfies $T_P(n) = O(\log n)$. Also, since the number of



PRAM (Parallel Random Access Machines). Fig. 2 Computation of prefix sums through a forward sweep and a backward sweep of the balanced binary tree algorithm. The notation $\sum_{i,j}$ represents the sum of elements from $A[i]$ through $A[j]$, and at each level an entry with two arrows pointing to it represent a sum operation

---

Algorithm 3 Prefix Sums Algorithm

Input: An array A of size $n = 2^k$ stored in shared memory.
Output: The prefix sums of A stored in the array PS.
begin
  if $n = 1$ **then**
    return($PS[1] = A[1]$)
  end if
  for all $1 \le i \le n/2$ **pardo**
    $Y[i] = A[2i-1] \otimes A[2i]$
end for
  Recursively compute the prefix sums of $Y[1 : n/2]$ in place
  for all $1 \le i \le n$ **pardo**
    i even: set $PS[i] = Y[i/2]$
    $i = 1$: set $PS[1] = A[1]$
    else: set $PS[i] = Y[(i-1)/2] \otimes A[i]$
  end for
end

---

internal nodes of a binary tree on n leaves is $n - 1$, the total work is clearly $W(n) = O(n)$, and hence this algorithm is work optimal. It follows that a p processor PRAM can compute the prefix sums of n elements in time $T_P(p, n) = O\left(\frac{n}{p} + \log n\right)$.

List Ranking

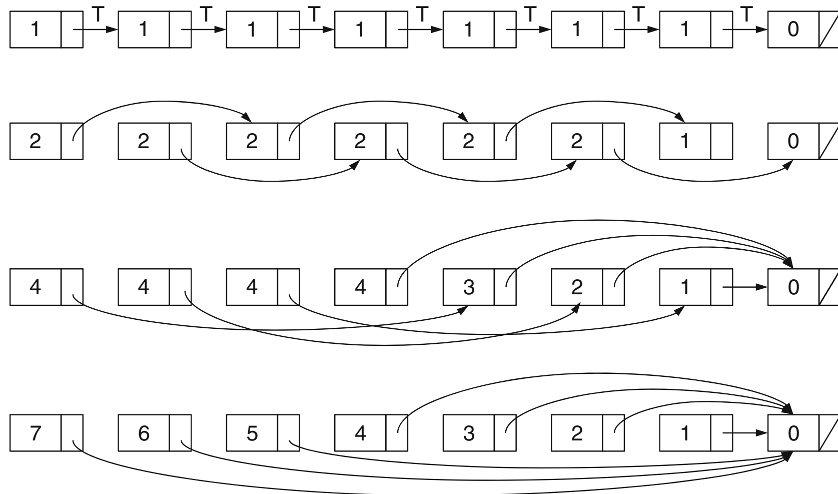
Consider a linked list L of n nodes whose successor relationship is represented by an array $S[1 : n]$ such that $S[i]$ is equal to the index of the successor node of i. For the last node k, its successor is denoted by $S[k] = 0$. The list ranking problem is to determine for each node i the distance $R[i]$ of the node i to the end of the list. While an optimal sequential algorithm is relatively straightforward – start by inverting the list and proceed to compute the ranks incrementally following the predecessor links – the problem may seem at first sight to be quite difficult to parallelize. It turns out that a fast PRAM algorithm can be developed through the introduction of the *pointer jumping* technique as illustrated in Algorithm 4. Note that T is used for the intermediate manipulation of the pointers, while the initial pointers stored in S remain intact.

The list ranking algorithm is illustrated in Fig. 3, where the pointer jumping technique is applied on each node i satisfying $T[i] \ne 0$ and $T[T[i]] \ne 0$.

Since the execution of the last parallel loop involves replacing the successor by its successor, the distance between a node and its successor doubles after each iteration and hence the T pointer of each node will point to the last node after $\lceil \log n \rceil$ iterations. Therefore, the algorithm terminates correctly after $O(\log n)$ iterations, and each iteration involves $O(n)$ operations. The algorithm has parallel complexity $O(\log n)$ using $O(n \log n)$ total number of operations. The work can be made optimal but the corresponding PRAM algorithm is more complex.

Fractional Independent Set

The purpose of introducing the next problem is to illustrate the use of randomization in the design of PRAM



PRAM (Parallel Random Access Machines). Fig. 3 Application of the list ranking algorithm on a list with eight elements

---

Algorithm 4 List Ranking Algorithm

Input: An array S of size n representing the successor relationship of a linked list.
Output: The distance $R[i]$ of node i to the end of the list, $1 \leq i \leq n$.
begin
  for all $1 \leq i \leq n$ **pardo**
  if $S[i] \neq 0$ **then** $R[i] = 1$ **else** $R[i] = 0$
end for
  for all $1 \leq i \leq n$ **pardo**
  $T[i] = S[i]$
end for
  repeat $\lceil \log n \rceil$ **times do**
  {
  for all $1 \leq i \leq n$ **pardo**
  if $T[i] \neq 0$ **and** $T[T[i]] \neq 0$ **then**
  { $R[i] = R[i] + R[T[i]]$; $T[i] = T[T[i]]$ }
end for
  }
end

---

algorithms to break symmetry. Given a directed cycle $C =< v_1, v_2, \ldots v_n >$, a fractional independent set is a subset U of the vertices such that: (i) U is an independent set, that is, no two vertices in U are connected by a directed edge; and (ii) the size of U is a constant fraction of the size of V. The fractional independent set problem is to determine such a set.

It is trivial to develop a sequential algorithm to solve this problem. Starting from any vertex, place every other vertex in U. A simple and fast PRAM algorithm can be designed using randomization as follows. Each vertex, in parallel, is randomly assigned a label 1 or 0 with equal probability. Clearly, the expected number of vertices of each label is $n/2$. However, this does not necessarily solve the problem since there is no guarantee that the vertices with the same label form an independent set. To ensure that this is indeed the case, another parallel step is carried out which involve changing the label of a vertex to 0 if the labels of this vertex and its successor are both equal to 1. The remaining vertices of label 1 are now guaranteed to form an independent set. It can be shown that, with high probability, the size of such an independent set is a constant fraction of the size of the original vertex set V.

On the PRAM model, this algorithm runs in $O(1)$ parallel time using $O(n)$ operations. It is worth noticing that this algorithm can in particular be used to make the list ranking algorithm more efficient (i.e., using a total number of operations which is asymptotically less than $n \log n$).

Superfast Maximum Algorithm

The algorithm presented in this section illustrates an important PRAM technique, called *accelerated cascading*, in combining two strategies. The first amounts to a work optimal algorithm (possibly a sequential

Algorithm 5 Randomized Fractional Independent Set Algorithm

Input: A directed cycle with a vertex set V whose arcs are specified by an array $S[1:n]$, i.e., $< i, S[i] >$ is an arc.
Output: A fractional independent set $U \subset V$.
begin
  for all $v \in V$ **pardo**
  Randomly assign $label(v) = 1$ or 0 with equal probability
  if $label(v) = 1$ and $lavel(S[v]) = 1$ **then** $label(v) = 0$
end for
  return$(U = \{v | label(v) = 1\})$
end

algorithm) to reduce the size of the problem below a certain threshold. The second strategy uses a very fast PRAM algorithm that involves a nonoptimal number of operations. The maximum algorithm will be used to illustrate such a technique and to also demonstrate the extra power of the PRAM model when concurrent write operations are allowed.

The PRAM strategy introduced earlier to compute the sum of n elements can be used to compute the maximum of n elements, resulting in the parallel time complexity $T_P(n) = O(\log n)$ and total work of $W(n) = O(n)$. However, it is possible to develop a *constant time* algorithm on the Common CRCW PRAM model as illustrated in **Algorithm 6**.

Algorithm 6 Constant Time Maximum Algorithm

Input: An array $A[1:n]$ consisting of n distinct elements
Output: A Boolean array $M[1:n]$ such that $M[i] = 1$ if, and only if, $A[i]$ is the maximum element
begin
  for all $1 \le i, j \le n$ **pardo**
  if$A[i] \ge A[j]$ **then** $B[i,j] = 1$ **else** $B[i,j] = 0$
end for
  for all $1 \le i \le n$ **pardo**
  $M[i] = \bigwedge_{j=1}^{n} B[i,j]$
end for
end

The Boolean AND of n binary variables can be performed in $O(1)$ parallel steps using $O(n)$ operations on the Common CRCW PRAM. Therefore, the above algorithm achieves constant parallel time but uses $O(n^2)$ operations, and thus it is extremely inefficient. To remedy this problem, and still achieve faster than $O(\log n)$ parallel time, a doubly logarithmic depth tree is used instead of the balanced binary tree. Essentially, the root of a doubly logarithmic depth tree has $\Omega(\sqrt{n})$ children, and each subtree is defined similarly. It is not hard to see that such a tree has height $O(\log\log n)$, where n is the number of leaves. Using the doubly logarithmic depth tree and the constant time maximum algorithm at each node of the tree, the maximum can be computed in $O(\log\log n)$ time using $O(n \log\log n)$ operations.

The accelerated cascading strategy is now used to turn this fast but nonoptimal work algorithm into a fast and work optimal algorithm as follows:

- Partition the array A into $n/\log\log n$ blocks, such that each block contains approximately $\log\log n$ elements and use the sequential algorithm to compute the maximum element in each block.
- Use the doubly logarithmic depth tree on the maxima computed in the first step.

The resulting algorithm has a parallel time complexity of $O(\log\log n)$ using only $O(n)$ operations, and hence is work optimal. Therefore, the maximum can be computed in $O(\log\log n)$ parallel time using a work optimal strategy on the Common CRCW PRAM.

Bibliographic Notes and Further Reading

The PRAM model was initially introduced through a number of papers, most notably in the papers by Fortune and Wyllie [2], Goldschlager [3], and Ladner and Fisher [9]. The Work–Time framework, which ties Brent's scheduling principle [1] and the PRAM model, was first observed by Shiloach and Vishkin [11] and used extensively in the book by JaJa [6]. Related data parallel algorithms were introduced by Hillis and Steele [5]. Substantial work dealing with the design and analysis of PRAM algorithms and the theoretical underpinnings of the model has been carried out in the 1980s and 1990s. An early survey is the paper by Karp and Ramachandran [7], and an extensive coverage of the topic can be found in JaJa's book [6].

An intriguing relationship exists between the PRAM model and the circuits model used in traditional computational complexity. The NC class was formally introduced by Pippenger [10] for circuits. An important related theoretical direction is based on the notion of P-completness, which tries to shed light on problems that do not seem to be highly parallelizable under the PRAM model with polynomial numbers of processors. Interested reader can consult the reference [4] for a good overview of this topic.

Some recent efforts have been devoted to advocate the PRAM as a practical parallel computation model, both in terms of developing prototype hardware that supports the model and software that enables the writing of PRAM programs. The book of Keller, Kessler, and Traff [8] and the recent paper by Wen and Vishkin [12] are illustrative of such efforts.

Bibliography

1. Brent R (1974) The parallel evaluation of general arithmetic expressions. JACM 21(2):201–208
2. Fortune S, Wyllie J (1978) Parallelism in random access machines. In: Proceedings of the tenth ACM symposium on theory of computing. San Diego, CA, pp 114–118
3. Goldschlager L (1978) A unified approach to models of synchronous parallel machines. In: Proceedings of the tenth ACM symposium on theory of computing. San Diego, CA, pp 89–94
4. Greenlaw R, Hoover HJ, Ruzzo WL (1995) Limits to Parallel Computation: P-Completeness Theory. In: Topics in parallel computation. Oxford University Press, Oxford
5. Hillis WD, Steele GL (1986) Data parallel algorithms. Commun ACM 29(12):1170–1183
6. JaJa J (1992) An introduction to parallel algorithms. Addison Wesley Publishing Co., Reading, MA
7. Karp RM, Ramachandran V (1990) Parallel algorithms for shared-memory machines. In: van Leeuwen J (ed) Handbook of theoretical computer science, North Holland, Amsterdam, The Netherlands, Chapter 17, pp 869–942
8. Keller J, Kessler C, Traff J (2001) Practical PRAM programming. Wiley, New York
9. Ladner R, Fisher M (1980) Parallel prefix computations. JACM 27(4):831–838
10. Pippenger N (1979) On simultaneous resource bounds. In: Proceedings twentieth annual IEEE symposium on foundations of computer science. San Juan, Puerto Rico, pp 307–311
11. Shiloach Y, Vishkin U (1982) An $O(n2 \log n)$ parallel max-flow algorithm. J Algorithms 3(2):128–146
12. Wen X, Vishkin U (2008) FPGA-based prototype of a PRAM-on-chip processor. In: Proceedings of the 2008 ACM conference on computing frontiers. Ischia, Italy, pp 55–66

Preconditioners for Sparse Iterative Methods

Anshul Gupta
IBM T.J. Watson Research Center, Yorktown Heights, NY, USA

Synonyms

Linear equations solvers

Definition

Iterative methods for solving sparse systems of linear equations are potentially less memory and computation intensive than direct methods, but often experience slow convergence or fail to converge at all. The robustness and the speed of Krylov subspace iterative methods is improved, often dramatically, by *preconditioning*. Preconditioning is a technique for transforming the original system of equations into one with an improved distribution (clustering) of eigenvalues so that the transformed system can be solved in fewer iterations. A key step in preconditioning a linear system $Ax = b$ is to find a nonsingular *preconditioner* matrix M such that the inverse of M is as close to the inverse of A as possible and solving a system of the form $Mz = r$ is significantly less expensive than solving $Ax = b$. The system is then solved by solving $(M^{-1}A)x = M^{-1}b$. This particular example shows what is known as *left preconditioning*. There are two other formulations, known as *right preconditioning* and *split preconditioning*. The basic concept, however, is the same. Other practical requirements for successful preconditioning are that the cost of computing M itself must be low and the memory required to compute and apply M must be significantly less than that for solving $Ax = b$ via direct factorization.

Discussion

Preconditioning methods are being actively researched and have been for a number of years. There are several classes of preconditioners; some are more amenable to being computed and applied in parallel than others. This chapter gives an overview of the generation (i.e., computing M in parallel) and application (i.e., solving a system of the form $Mz = r$ in parallel) of the most commonly used parallel preconditioners.

While solving a sparse linear system $Ax = b$ in parallel, the matrix A and the vectors x and b are typically partitioned. The partitions are assigned to tasks that are executed by individual processes or threads in a parallel processing environment. Both the creation and the application of a preconditioner in parallel is affected by the underlying partitioning of the data. A commonly used effective and natural way of partitioning the data involves partitioning the graph, of which the coefficient matrix is an adjacency matrix. Other than partitioning the problem for parallelization, the graph view of the matrix plays a useful role in many aspects of solving sparse linear systems. Figure 1 illustrates a partitioning of the rows of a matrix among four tasks based on a four-way partitioning of its graph.

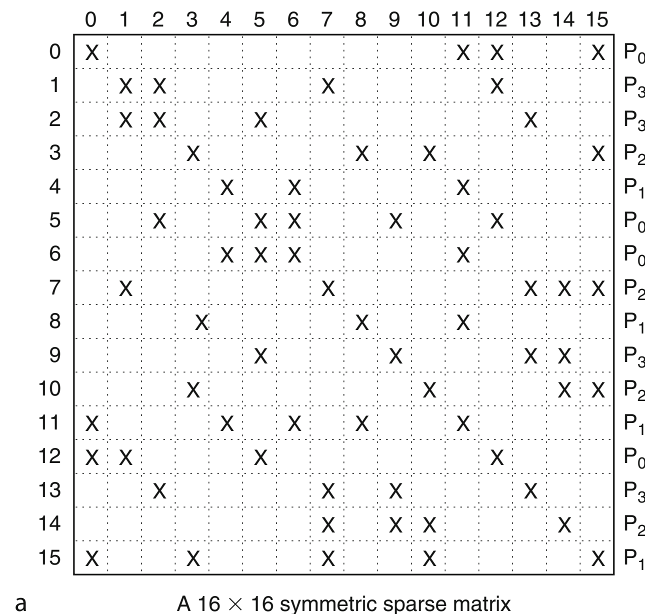
Simple Preconditioners Based on Stationary Methods

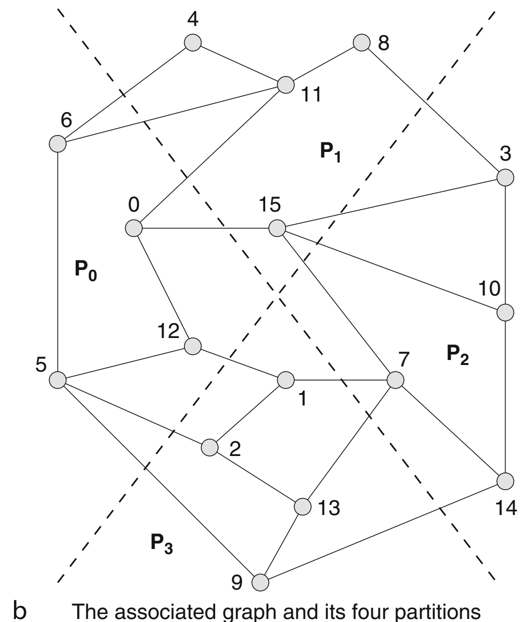
Stationary iterative methods are relatively simple algorithms that start with an initial guess of the solution (like all iterative methods) and attempt to converge toward the actual solution by repeated application of a correction equation. The correction equation uses the current residual and a fixed (stationary) operator or matrix, which is an approximation of the original coefficient matrix. While stationary iterative methods themselves have poor convergence properties, the approximating matrix can serve as a preconditioner for Krylov subspace methods.

Jacobi and Block-Jacobi Preconditioners

One of the simplest preconditioners is the point-Jacobi preconditioner, which is nothing but the diagonal D of the matrix A of coefficients. Applying the preconditioner in parallel is straightforward. It simply involves division with the entries (or multiplication with their inverses) of the part of the diagonal corresponding to the portion of the matrix that each thread or process is responsible for. In fact, scaling the coefficient matrix by the diagonal so that the scaled matrix has all 1's on the diagonal is equivalent to Jacobi preconditioning.

A block-Jacobi preconditioner is made up of nonoverlapping square diagonal blocks of the coefficient matrix. These blocks may be of the same or different sizes. These blocks are usually factored or inverted (independently, in parallel) during the preconditioner construction phase so that the preconditioner can be applied inexpensively during the Krylov solver's iterations.





Preconditioners for Sparse Iterative Methods. Fig. 1 A 16 × 16 sparse matrix with symmetric structure and its associated graph partitioned among four tasks

Gauss–Seidel Preconditioner

Let the coefficient matrix A be represented by a three-way splitting as $L + D + U$, where L is the strictly lower triangular part of A, D is a diagonal matrix that consists of the principal diagonal of A, and U is the strictly upper triangular part of A. The Gauss–Seidel preconditioner is defined by

$$M = (D + L)D^{-1}(D + U). \quad (1)$$

A system $Mz = r$ is then trivially solved as $y = (D+L)^{-1}r$, $w = Dy$, and $z = (D + U)^{-1}w$. Thus, applying the Gauss–Seidel preconditioner in parallel involves solving a lower and an upper triangular system in parallel. In general, equation i of a lower triangular system can be solved for the i-th unknown when equations $1 \dots i - 1$ have been solved. Similarly, equation i of an $N \times N$ upper triangular system can be solved after equations $i + 1 \dots N$ have been solved. Since the matrices L and U in our case are sparse, the i-th equation while computing $y = (D + L)^{-1}r$ depends only on those unknowns that have nonzero coefficients in the i-th row of L. Similarly, the i-th equation while computing $x = (D+U)^{-1}w$ depends only on those unknowns that have nonzero coefficients in the i-th row of U. As a result, while solving both these systems, multiple unknowns may be computed simultaneously – those that do not depend on any unsolved unknowns. This is an obvious source of parallelism. This parallelism can be maximized by reordering the rows and columns of A (and hence those of L and U) in a way that maximizes the number of independent equations at each stage of the solution process.

Figure 2 illustrates one such ordering, known as red-black ordering, that can be used to parallelize the application of the Gauss–Seidel preconditioner for a matrix arising from a finite-difference discretization. The vertices of the graph corresponding to the matrix are assigned colors such that no two neighboring vertices have the same color. All vertices and the corresponding rows and columns of the matrix are numbered first, followed by those of the other color. Assignment of matrix rows to tasks is based on a partitioning of the graph. With red-black ordering, each triangular solve is performed in two phases. During the lower triangular solve, first, all red unknowns are computed in parallel because they are all independent. After this step, all black unknowns can be computed. The order is reversed during the upper triangular solve. On a distributed memory platform, each computation phase is followed by a communication phase. During a communication phase, each process communicates the values of the unknowns corresponding to graph vertices on the partition boundaries with its neighboring processes.

The idea of red-black ordering can be extended to general sparse matrices and their graphs, for which more than two colors may be required to ensure that no neighboring vertices have the same color. The triangular solves are then performed in as many parallel phases as the number of colors. A multicolored ordering with four colors is illustrated in Fig. 3. For improved cache performance, block variants of multicolored ordering can be constructed by assigning colors to clusters of graph vertices and ensuring that no two clusters that have an edge connecting them are assigned the same color.

The reader is cautioned that often the convergence of a Krylov subspace method is sensitive to the ordering of matrix rows and columns. While red-black and multicolor orderings enhance parallelism, they may result in a deterioration of the convergence rate in some cases.

SOR Preconditioner

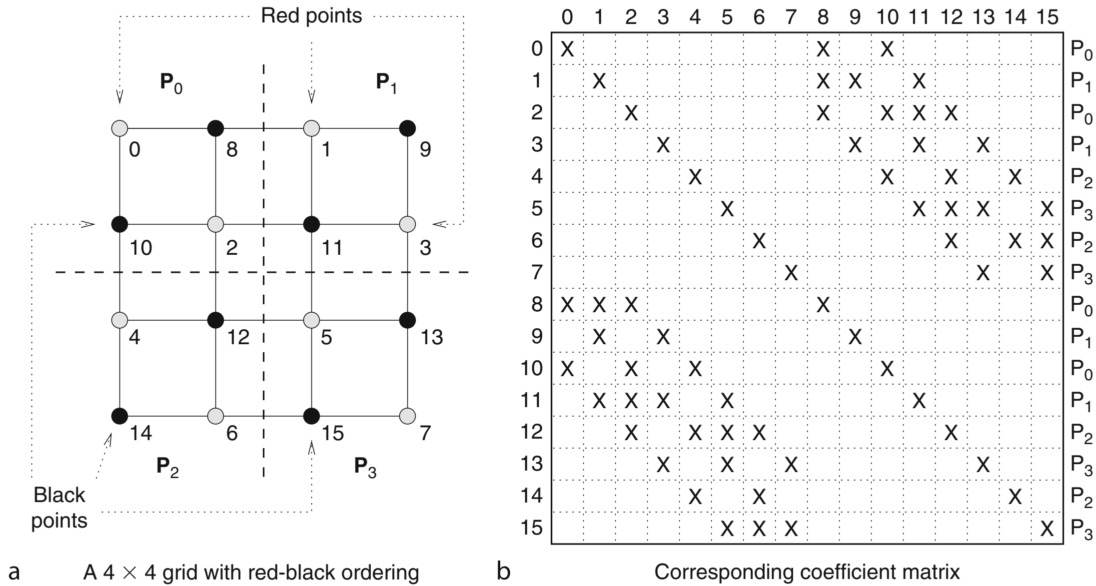
A significant increase in convergence rate may be obtained by a modification of Eq. 1 as follows, with $0 < \omega < 2$:

$$M = \left(\frac{D}{\omega} + L\right)\frac{\omega D^{-1}}{2 - \omega}\left(\frac{D}{\omega} + U\right), \quad (2)$$

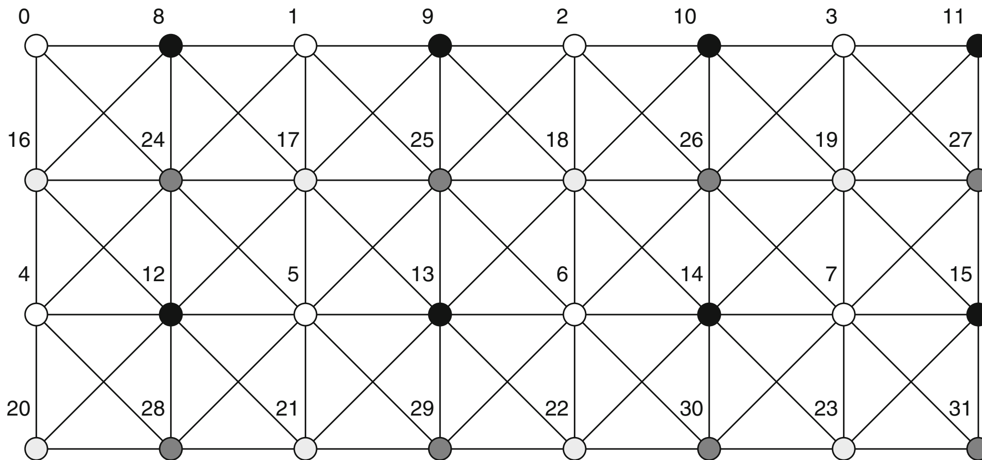
although determining an optimal value of ω can be expensive. The preconditioner specified by Eq. 2 is known as successive overrelaxtion or SOR preconditioner. Its symmetric formulation, when $L = U$, is known as symmetric SOR or SSOR preconditioner. The issues in parallel application of the SOR preconditioner are identical to those for the Gauss–Seidel preconditioner.

Preconditioners Based on Incomplete Factorization

A class of highly effective but conceptually simple preconditioners is based on incomplete factorization methods. Recall that iterative methods are used in applications where a factorization of the form $A = LU$ is not feasible because the triangular factor matrices L and U



Preconditioners for Sparse Iterative Methods. Fig. 2 The sparse matrix corresponding to a 4 × 4 finite-difference grid with *red-black* ordering, partitioned among four parallel tasks



Preconditioners for Sparse Iterative Methods. Fig. 3 Multicolored ordering of a hypothetical finite-element graph using four colors

are much denser than A, and therefore, too expensive to compute and store. The idea behind incomplete factorization is to perform a factorization of A along the lines of a regular Gaussian elimination or Cholesky factorization, but dropping a large proportion of nonzero entries from the triangular factors along the way. Depending on the underlying factorization method, incomplete factorization is referred to as ILU (incomplete LU) or IC (incomplete Cholesky) factorization. Due to dropping, the resulting triangular factors $\widetilde{L}$ and $\widetilde{U}$ are much sparser than L and U and are computed with significantly less computing effort. Entries are chosen for dropping using some criteria that strive to keep the inverse of $M = \widetilde{L}\widetilde{U}$ as close to the inverse of A as possible. Devising effective dropping criteria has been an active area of research. Incomplete factorization methods can be broadly classified as follows based on the dropping criteria.

Static-Pattern Incomplete Factorization

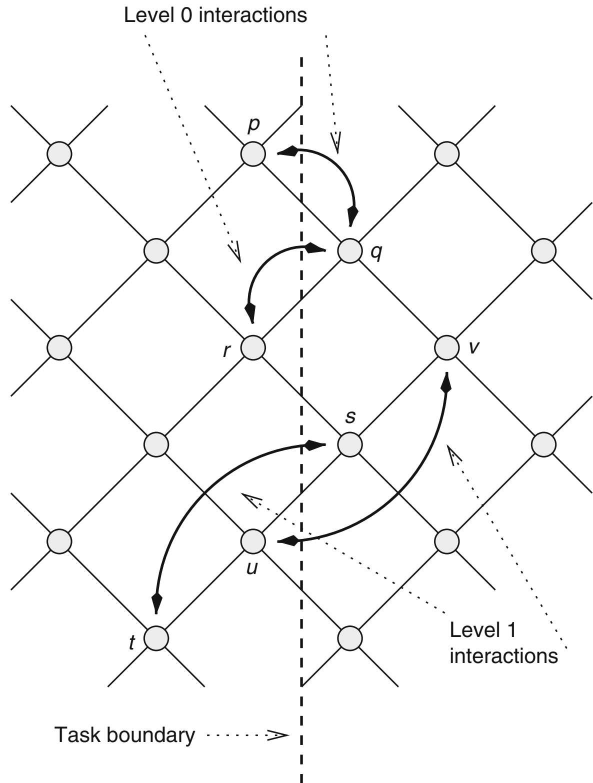
A static-pattern incomplete factorization method is one in which the locations of the entries that are kept in the factors and those that are dropped are determined a priori, based only on the structure of A. The simplest form of incomplete factorizations are ILU(0) and IC(0), in which the structure of $\widetilde{L}+\widetilde{U}$ is identical to structure of A; i.e., only those factor entries whose locations coincide with those of nonzero entries in the original matrix are saved. A generalization of static-pattern incomplete factorization is a level-κ incomplete factorization, referred to as ILU(κ) or IC(κ) factorization in the literature. The structure of a level-κ incomplete factorization is computed symbolically as follows. Initially, $level(i,j) = 0$ if $a_{ij} \neq 0$; otherwise, $level(i,j) = \infty$. This is followed by an emulation of factorization where each numerical update step of the form $a_{ij} = a_{ij} - a_{ik}.a_{kj}$ is replaced by updating $level(i,j) = min(level(i,j), level(i,k) + level(k,j) + 1)$. During the subsequent numerical factorization phases, entries in locations with a level greater than κ are dropped.

In graph terms, upon the completion of symbolic factorization, $level(i,j)$ is the length of the shortest path between vertices i and $j - 1$. During the computation and application of an ILU(κ) or IC(κ) preconditioner, the computation corresponding to a vertex in the graph requires data associated with vertices that are up to $\kappa + 1$ edges away from it. For example, in the graph shown in Fig. 4, data exchange is required among vertex pairs (p,q) and (q,r) for $\kappa = 0$. For $\kappa = 1$, additional exchanges among vertex pairs such as (s,t) and (u,v) are required. The figure also illustrates that in a parallel environment, data associated with $\kappa+1$ layers of vertices adjacent to a partition boundary needs to be exchanged with a neighboring task.

Both the computation per vertex of the graph and the data exchange overhead per task in each iteration of a Krylov solver using a level-κ incomplete factorization preconditioner increase as κ increases. On the other hand, the overall number of iterations typically declines. The optimum value of κ is problem dependent.

Threshold-Based Incomplete Factorization

Although it permits a relatively easy and fast parallel implementation, static-pattern incomplete factorization is robust for a few classes of problems only, including those with diagonally dominant coefficient



Preconditioners for Sparse Iterative Methods. Fig. 4 Illustration of data exchange across a task boundary when level 0 and level 1 fill is permitted in incomplete factorization

matrices. It can, and often does delete fill entries of large magnitudes that do not happen to be located in the predetermined locations. The resulting large error can make the preconditioner ineffective. Threshold-based incomplete factorization rectifies this problem by dropping entries from the factors as they are computed. Regardless of their locations, entries greater in magnitude than a user-defined threshold τ are kept and the others are dropped. Typically, a second threshold γ is also used to limit the factors to a predetermined size. If n_i is the number of nonzeros in row (or column) i of the coefficient matrix, then at most γn_i entries (those with the largest magnitudes) are permitted in row (or column) i of the incomplete factor.

Successful and scalable parallel implementations of threshold-based incomplete factorization preconditioning use graph partitioning and graph coloring for balancing computation and minimizing communication

among parallel tasks. The use of these two techniques in the context of sparse matrix computations has already been discussed earlier. Graph partitioning enables parallel tasks to independently compute and apply (i.e., perform forward and back substitution) the preconditioner independently for matrix rows and columns corresponding to the internal vertices of the graph. An internal vertex and all its neighbors belong to the same partition. Graph coloring permits parallel incomplete factorization and forward and back substitution of matrix rows and columns corresponding to the boundary vertices. In the context of incomplete factorization, coloring is applied to a graph that includes only the boundary vertices (i.e., after the internal vertices have been eliminated) but includes the additional edges (fill-in) created as a result of the elimination of the internal vertices. The reason is that the dependencies among the rows and columns of the matrix are determined by all the nonzeros in the incomplete factors, both original and those created as a result of fill-in.

Incomplete Factorization Based on Inverse-Norm Estimate

This is a relatively new class of incomplete factorization preconditioners in which the dropping criterion takes into account and seeks to minimize the growth of the norm of the inverse of the factors. These preconditioners have been shown to be more robust and effective than incomplete factorization with dropping based solely on the position or absolute value of the entries. The issues in the parallel generation and application of these preconditioners are very similar to those in threshold-based incomplete factorization – in both cases, the location of nonzeros in the factors cannot be determined a priori.

Sparse Approximate Inverse Preconditioners

While the incomplete factorization preconditioners seek to compute $\widetilde{L}$ and $\widetilde{U}$ as approximations of the actual factors L and U of the coefficient matrix A, sparse approximate preconditioners seek to compute M^{-1} as an approximation to its inverse A^{-1}. The problem of computing M^{-1} is framed as the problem of minimizing the norm $\|I - AM\|$ or $\|I - MA\|$. To support parallelism, these minimization problems can be reduced to independent subproblems for computing the rows and columns of M^{-1}. Note that the actual inverse of a sparse A is dense, in general. It is therefore imperative that a number of entries be dropped in order to keep M^{-1} sparse. Just like incomplete factorization, the dropping can be structural (static), or based on values (dynamic), or both. In graph terms, while computing the rows or columns of M^{-1} in parallel, dropping is typically orchestrated in a way that confines the interaction to pairs of vertices that are either immediate neighbors or have short paths connecting them in the graph of A. Graph partitioning is used to facilitate load balance and minimize interaction among parallel tasks – both during the computation and the application of the preconditioner.

There are some important advantages to explicitly using an approximation of A^{-1} for preconditioning, rather than using A's approximate factors. First, the preconditioner computation avoids the kind of breakdowns that are possible in incomplete factorization due to small or zero (or negative, in case of incomplete Cholesky) diagonals. Secondly, the application of the preconditioners involves a straightforward multiplication of a vector with the sparse matrix M^{-1}, which may be simpler and more easily parallelizable than the forward and back substitutions with $\widetilde{L}$ and $\widetilde{U}$. However, just like $\widetilde{L}$ and $\widetilde{U}$, M^{-1} may be denser than A. Therefore, the graph of M^{-1} may have many more edges than the graph of A and multiplying a vector with M^{-1} may require more communication than multiplying a vector with A.

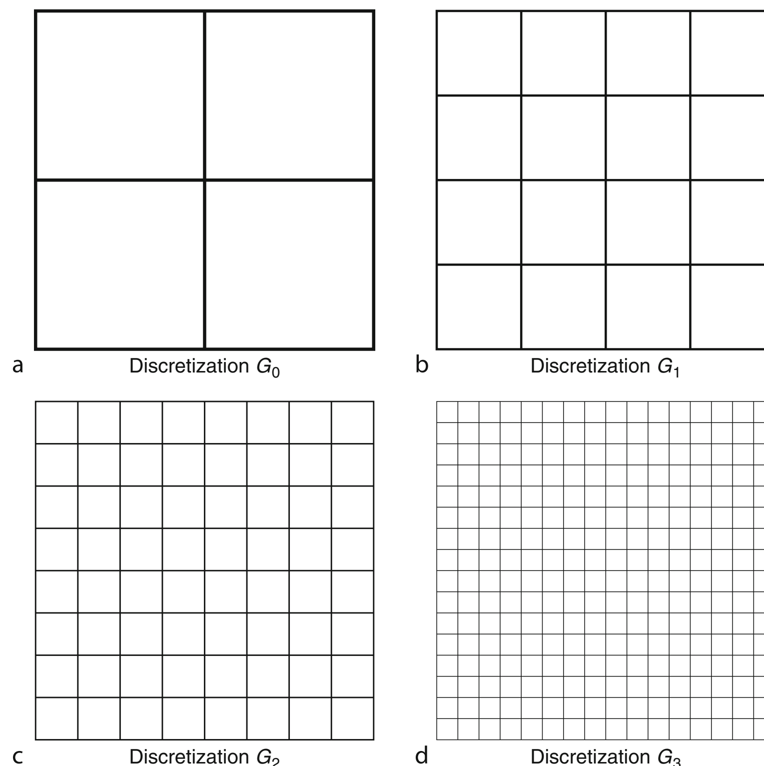
Multigrid Preconditioners

Multigrid methods are a class of iterative algorithms for solving partial differential equations (PDEs) efficiently, often by exploiting more problem-specific information than a typical Krylov subspace method. Each iteration of a multigrid solver is a somewhat complex recursive procedure. In many applications, the effectiveness of a multigrid algorithm can be substantially enhanced by using it to precondition a Krylov subspace method rather than using it as the solver. This is done by replacing the preconditioning step of the Krylov subspace solver with one iteration of the multigrid algorithm; i.e., treating the approximate solution obtained by an iteration of the multigrid algorithm as the solution with respect to a hypothetical preconditioner matrix.

Geometric Multigrid

Multigrid methods were originally designed for solving elliptic PDEs by discretizing them using a hierarchy of regular grids of varying degrees of fineness over the same domain. For example, consider a domain D and a sequence of successively finer discretizations $G_0, G_1, \ldots, G_m$. Here G_0 is the coarsest discretization and G_m is the finest discretization over which the eventual solution to the PDE is desired. Figure 5 shows a square domain with $m = 3$. As the figure shows, the grid points in G_i are a subset of the grid points in G_{i+1}. A simple formulation of multigrid would work as follows:

1. First, the linear system corresponding to discretization G_0 is solved. Since G_0 has a small number of points, the associated linear system is small and can be solved inexpensively by an appropriate direct or iterative method.
2. The solution at G_0 is interpolated to obtain an initial guess of the solution of the system corresponding to G_1, which is four times larger. Among various ways of *interpolation* (also known as *prolongation*), a simple one involves approximating the value of the solution at a point that is in G_1 but not in G_0 by the average of the values of its neighbors.
3. Starting with the initial guess obtained by interpolation, a few steps of *relaxation* (also referred to as *smoothing*) are used to refine the solution at G_1. Often, the relaxation steps are simply a few iterations of a relatively inexpensive stationary method such as Jacobi or Gauss–Siedel.
4. The process of relaxation and interpolation continues from G_i to G_{i+1}, until $i + 1 = m$. After the relaxation at G_m, a first approximation x_0^m to the solution x^m of the linear system $A^m x^m = b^m$ corresponding to G_m is obtained. Successively more accurate approximations $x_1^m, x_2^m, \ldots$ are obtained by $x_{i+1}^m = x_i^m + d_i^m$, where d_i^m is obtained by solving $Ad_i^m = r_i^m \equiv (b^m - Ax_i^m)$.
5. The system $Ad_i^m = r_i^m$ in the i-th multigrid iteration is solved by a recursive process, in which the residual r_i^m corresponding to G_m is projected on to G_{m-1} and so on. The process of *projection* (also known as *restriction*) is the reverse of interpolation. At the end



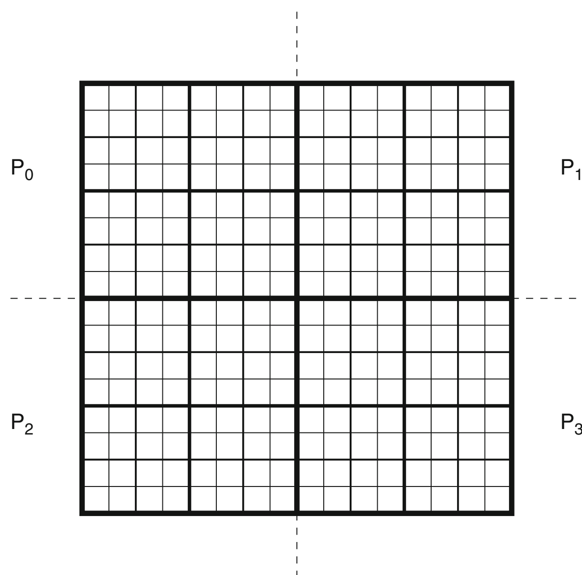
Preconditioners for Sparse Iterative Methods. Fig. 5 Successively finer discretizations of a domain

of the recursive projection steps, a relatively small system of equations $Ad_i^0 = r_i^0$ corresponding to G_0 is obtained, which is readily solved by an appropriate iterative or direct method.

6. The cycle of interpolation and relaxation is then repeated to obtain d_i^m.
7. The process is stopped after k multigrid iterations if r_k^m is smaller than a user-defined threshold.

When the multigrid method is used as a preconditioner, then instead of repeating the multigrid cycle k times to solve the problem, the approximate solution after one cycle is substituted as the solution with respect to the preconditioner inside a Krylov subspace algorithm. Whether multigrid is used as a solver or a preconditioner, the parallelization process is the same.

The first step in implementing a parallel multigrid procedure is to partition the domain among the tasks such that each task is assigned roughly the same number of points of the finest grid. For example, Fig. 6 shows the partitioning of the domain of Fig. 5 and its discretizations G_0–G_3 into four parallel tasks. Unlike Figs. 5 and 6, the domain in a real problem may be irregular and the partitioning may not be trivial. The partitioning of the domain implicitly defines a partitioning of the grids at all levels of discretization. It is easily seen that the parallel interpolation, relaxation, and projection at any grid level require a task to exchange information corresponding to the grid points along the partition boundary with its neighboring tasks. All computations corresponding to each partition's interior points can be performed independently.



Preconditioners for Sparse Iterative Methods. Fig. 6 The domain and discretizations G_0–G_3 of Fig. 5 mapped onto four parallel tasks

Algebraic Multigrid

The algebraic multigrid (AMG) method is a generalization of the hierarchical approach of the geometric multigrid method so that it is not dependent on the availability of the meshes used for discretizing a PDE, but can be used as a black-box solver for a given linear system of equations. In the geometric multigrid method, successively finer meshes are constructed by a geometric refining of the coarser meshes. On the other hand, the starting point for AMG is the final system of equations (analogous to the finest grid), from which successively smaller (coarser) systems are constructed. The coefficients of a coarse system in AMG are only algebraically related to the coefficients of the finer systems, which is in contrast to the geometric relationship between successive grids in geometric multigrid.

Just like geometric multigrid, an AMG method can be used either as a solver or a preconditioner. In either case, the method involves two phases. In the first set-up phase, the hierarchy of coarse systems is constructed from the original linear system and the prolongation and restriction operators are defined. The second solution phase consists of the prolongation, relaxation, and restriction cycles.

Efficient parallelization of AMG is much harder than that of geometric multigrid. As usual, the basis of parallelization is a good partitioning of the graph corresponding to the coefficient matrix and assigning the partitions to individual tasks. The solution phase, in principle, can then be parallelized with computation corresponding to the interior nodes remaining independent and that involving the nodes at or close to the boundary involving exchange of data with neighboring partitions. The boundary communication can be more involved than in the case of geometric multigrid because of the irregularity of the graph and the fact that the sets of interacting boundary nodes and the interaction pattern can be different for prolongation, relaxation, and restriction. However, the main difficulty in parallelizing AMG is in the set-up phase.

Effective parallelization of the set-up phase is essential for the overall scalability of parallel AMG because this phase can account for up to one-fourth of the total execution time. The process of construction of a coarser linear system in the classical AMG approach relies on a notion of the "strength" of dependencies among coefficients. This process is not only sequential in nature, but also involves a highly nonlocal pattern of interaction between the vertices of the graph of the coefficient matrix. Therefore, the algorithm must be adapted for parallelization, which can make the convergence rate and per iteration cost of AMG dependent on the number of parallel tasks. Care must be taken to ensure that parallelization does not adversely affect the convergence rate or the iteration complexity of AMG. Typical approaches to parallelizing coarsening in AMG rely on decoupling the partitions, using parallel independent sets (similar to multicoloring described earlier), or performing subdomain blocking, which starts the coarsening at the partition boundaries and then proceeds to the interior nodes. Note that the coarsening scheme has an impact on the inter-task interaction during the prolongation and restriction steps of the solution phase.

When parallelizing AMG on large parallel machines, the number of parallel tasks may exceed the number of points in some of the grids at the coarsest levels. This situation requires special treatment. A commonly used work-around to this problem is agglomeration, in which neighboring domains are coalesced leaving some tasks idle during the processing of the coarsest levels. Another approach is to stop the coarsening when the number of coefficients per task becomes too small, which makes the behavior of the overall algorithm dependent on the number of parallel tasks.

Stochastic Preconditioners

There has been a fair amount of research on algorithms for approximating the solution of linear systems based on random sampling of the coefficient matrix or on random walks in the graph corresponding to it. Most of these methods have been proven to work on limited classes of linear systems only, such as symmetric diagonally dominant systems. A few practical solvers have been developed recently by using some of these techniques for preconditioning Krylov subspace methods. An attractive property of these methods is that they are usually trivially parallelizable. The quest for scalable massively parallel sparse linear solvers may prompt more active research into statistical techniques for preconditioning.

Matrix-Free Methods and Physics-Based Preconditioners

Note that this chapter discusses preconditioners derived explicitly from the coefficient matrix A of the system $Ax = b$ that needs to be solved. In some applications, A is never constructed explicitly to save time and storage. Instead, it is applied implicitly to compute the matrix-vector products required in the Krylov subspace solver. In some of these cases, preconditioning is also applied implicitly, or the knowledge of the physics of the application is utilized to construct the preconditioner, which cannot be derived from the coefficient matrix in a matrix-free method. Such preconditioners are called physics-based preconditioners. Due to the highly application-specific nature of matrix-free methods and physics-based preconditioners, these topics are not covered in further detail in this chapter.

Related Entries

►Graph Partitioning
►Linear Algebra Software
►Rapid Elliptic Solvers

Bibliographic Notes

The readers are referred to Saad's book [11] and the survey by Benzi [1] for a fairly comprehensive introduction to various preconditioning techniques. These do not cover parallel preconditioners and may not included some of the most recent work in preconditioning. However, these are excellent resources for gaining an insight into the state of the art of preconditioning circa 2002.

Hysom and Pothen [7] and Karypis and Kumar [8] cover the fundamentals of scalable parallelization of incomplete factorization–based preconditioners. The work of Grote and Huckle [6] and Edmond Chow [3, 4] is the basis of modern parallel sparse approximate inverse preconditioners. Chow et al.'s survey [5] should give the readers a good overview of parallelization techniques for geometric and algebraic multigrid methods.

Last, but not the least, almost all parallel preconditioning techniques rely on effective parallel heuristics for two critical combinatorial problems – graph partitioning and graph coloring. The readers are referred

to papers by Karypis and Kumar [9, 10] and Bozdag et al. [2] for an overview of these.

Bibliography

1. Benzi M (2002) Preconditioning techniques for large linear systems: a survey. J Computat Phys 182(2):418–477
2. Bozdag D, Gebremedhin AH, Manne F, Boman EG, Catalyurek UV (2008) A framework for scalable greedy coloring on distributed memory parallel computers. J Parallel Distrib Comput 68(4):515–535
3. Chow E (2000) A priori sparsity patterns for parallel sparse approximate inverse preconditioners. SIAM J Sci Comput 21(5): 1804–1822
4. Chow E (2001) Parallel implementation and practical use of sparse approximate inverse preconditioners with a priori sparsity patterns. Int J High Perform Comput appl 15(1):56–74
5. Chow E, Falgout RD, Hu JJ, Tuminaro RS, Yang UM (2006) A survey of parallelization techniques for multigrid solvers. In Heroux MA, Raghavan P, Simon HD (eds) Parallel processing for scientific computing. SIAM, Philadelphia
6. Grote MJ, Huckle T (1997) Parallel preconditioning with sparse approximate inverses. SIAM J Sci Comput 18(3): 838–853
7. Hysom D, Pothen A (2000) A scalable parallel algorithm for incomplete factor preconditioning. SIAM J Sci Comput 22(6): 2194–2215
8. Karypis G, Kumar V (1996) Parallel threshold-based ILU factorization. Technical report TR 96-061, Department of Computer Science, University of Minnesota, Minnesota
9. Karypis G, Kumar V (1997) ParMETIS: parallel graph partitioning and sparse matrix ordering library. Technical report TR 97-060, Department of Computer Science, University of Minnesota, Minnesota
10. Karypis G, Kumar V (1998) Parallel algorithms for multilevel graph partitioning and sparse matrix ordering. J Parallel Distrib Comput 48:71–95
11. Saad Y (2003) Iterative methods for sparse linear systems, 2nd edn. SIAM, Philadelphia

Prefix

►Reduce and Scan
►Scan for Distributed Memory, Message-Passing Systems

Prefix Reduction

►Reduce and Scan
►Scan for Distributed Memory, Message-Passing Systems

Problem Architectures

►Computational sciences

Process Algebras

Rocco De Nicola
Universita' di Firenze, Firenze, Italy

Synonyms

Process calculi; Process description languages

Definition

Process Algebras are mathematically rigorous languages with well-defined semantics that permit describing and verifying properties of concurrent communicating systems. They can be seen as models of processes, regarded as agents that act and interact continuously with other similar agents and with their common environment. The agents may be real-world objects (even people), or they may be artifacts, embodied perhaps in computer hardware or software systems. Many different approaches (operational, denotational, algebraic) are taken for describing the meaning of processes. However, the operational approach is the reference one. By relying on the so-called Structural Operational Semantics (SOS), labeled transition systems are built and composed by using the different operators of the many different process algebras. Behavioral equivalences are used to abstract from unwanted details and identify those systems that react similarly to external experiments.

Introduction

The goal of software verification is to assure that developed programs fully satisfy all the expected requirements. Providing a formal semantics of programming languages is an essential step toward program verification. This activity has received much attention in the last 40 years. At the beginning the interest was mainly on sequential programs, then it turned also on concurrent program that can lead to subtle errors in very critical activities. Indeed, most computing systems today are concurrent and interactive.

Classically, the semantics of a sequential program has been defined as a function specifying the

induced input–output transformations. This setting becomes, however, much more complex when concurrent programs are considered because they exhibit nondeterministic behaviors. Nondeterminism arises from programs interaction and cannot be avoided. At least, not without sacrificing expressive power. Failures do matter, and choosing the wrong branch might result in an "undesirable situation." Backtracking is usually not applicable because the control might be distributed. Controlling nondeterminism is very important. In sequential programming, it is just a matter of efficiency, in concurrent programming it is a matter of avoiding getting stuck in a wrong situation.

The approach based on process algebras has been very successful in providing formal semantics of concurrent systems and proving their properties. The success is witnessed by the Turing Award given to two of their pioneers and founding fathers: Tony Hoare and Robin Milner. Process algebras are mathematical models of processes, regarded as agents that act and interact continuously with other similar agents and with their common environment. Process algebras provide a number of constructors for system descriptions and are equipped with an operational semantics that describes systems evolution in terms of labeled transitions. Models and semantics are built by taking a compositional approach that permits describing the "meaning" of composite systems in terms of the meaning of their components.

Moreover, process algebras often come equipped with observational mechanisms that permit identifying (through behavioral equivalences) those systems that cannot be taken apart by external observations (*experiments* or *tests*). In some cases, process algebras have also algebraic characterizations in terms of equational axiom systems that exactly capture the relevant identifications induced by the behavioral operational semantics.

The basic component of a process algebra is its syntax as determined by the well-formed combination of operators and more elementary terms. The syntax of a process algebra is the set of rules that define the combinations of symbols that are considered to be correctly structured programs in that language. There are many approaches to providing a rigorous mathematical understanding of the semantics of syntactically correct process terms. The main ones are those also used for describing the semantics of sequential systems, namely, operational, denotational, and algebraic semantics.

An *operational semantics* models a program as a labeled transition system (LTS) that consists of a set of states, a set of transition labels and a transition relation. The states of the transition system are just process algebra terms, while the labels of the transitions between states represent the actions or the interactions that are possible from a given state and the state that is reached after the action is performed by means of visible and invisible actions. The operational semantics, as the name suggests, is relatively close to an abstract machine-based view of computation and might be considered as a mathematical formalization of some implementation strategy.

A *denotational semantics* maps a language to some abstract model such that the meaning/denotation (in the model) of any composite program is determinable directly from the meanings/denotations of its subcomponents. Usually, denotational semantics attempt to distance themselves from any specific implementation strategy, describing the language at a level intended to capture the "essential meaning" of a term.

An *algebraic semantics* is defined by a set of algebraic laws which implicitly capture the intended semantics of the constructs of the language under consideration. Instead of being derived theorems (as they would be in a denotational semantics or operational semantics), the laws are the basic axioms of an equational system, and process equivalence is defined in terms of what equalities can be proved using them. In some ways it is reasonable to regard an algebraic semantics as the most abstract kind of description of the semantics of a language.

There has been a huge amount of research work on process algebras carried out during the last 30 years that started with the introduction of CCS [31, 32], CSP [11], and ACP [6]. In spite of the many conceptual similarities, these process algebras have been developed starting from quite different viewpoints and have given rise to different approaches (for an overview see, e.g., [2]).

CCS takes the operational viewpoint as its cornerstone and abstracts from unwanted details introduced by the operational description by taking advantage of behavioral equivalences that allow one to identify those systems that are indistinguishable according to some observation criteria. The meaning of a CCS term is a labeled transition system factored by a notion of observational equivalence. *CSP* originated as the

theoretical version of a practical language for concurrency and is still based on an operational intuition which, however, is interpreted w.r.t. a more abstract theory of decorated traces that model how systems react to external stimuli. The meaning of a CSP term is the set of possible runs enriched with information about the interactions that could be refused at intermediate steps of each run. *ACP* started from a completely different viewpoint and provided a purely algebraic view of concurrent systems: processes are the solutions of systems of equations (axioms) over the signature of the considered algebra. Operational semantics and behavioral equivalences are seen as possible models over which the algebra can be defined and the axioms can be applied. The meaning of a term is given via a predefined set of equations and is the collection of terms that are provably equal to it.

At first, the different algebras have been developed independently. Slowly, however, their close relationships have been understood and appreciated, and now a general theory can be provided and the different formalisms (CCS, CSP, ACP, etc.) can be seen just as instances of the general approach. In this general approach, the main ingredients of a specific process algebra are:

1. A minimal set of carefully chosen operators capturing the relevant aspect of systems behavior and the way systems are composed in building process terms
2. A transition system associated with each term via structural *operational semantics* to describe the evolution of all processes that can be built from the operators
3. An equivalence notion that allow one to abstract from irrelevant details of systems descriptions

Verification of concurrent system within the process algebraic approach is carried out either by resorting to behavioral equivalences for proving conformance of processes to specifications or by checking that processes enjoy properties described by some temporal logic formulae [14, 28]. In the former case, two descriptions of a given system, one very detailed and close to the actual concurrent implementation, the other more abstract (describing the sequences or trees of relevant actions the system has to perform) are provided and tested for equivalence. In the latter case, concurrent systems are specified as process terms, while properties are specified as temporal logic formulae, and model checking is used to determine whether the transition systems associated with terms enjoy the property specified by the formulae.

In the next section, many of the different operators used in process algebras will be described. By relying on the so-called structural operational semantic (SOS) approach [37], it will be shown how labeled transition systems can be built and composed by using the different operators. Afterward, many behavioral equivalences will be introduced together with a discussion on the induced identifications and distinctions. Next, the three most popular process algebras will be described; for each of them a different approach (operational, denotational, algebraic) will be used. It will, however, be argued that in all cases, the operational semantics plays a central rôle.

Process Operators and Operational Semantics

To define a process calculus, one starts with a set of uninterpreted action names (that might represent communication channels, synchronization actions, etc.) and with a set of basic processes that together with the actions are the building blocks for forming newer processes from existing ones. The operators are used for describing sequential, nondeterministic, or parallel compositions of processes, for abstracting from internal details of process behaviors and, finally, for defining infinite behaviors starting from finite presentations. The operational semantics of the different operators is inductively specified through SOS rules: for each operator, there is a set of rules describing the behavior of a system in terms of the behaviors of its components. As a result, each process term is seen as a component that can interact with other components or with the external environment.

In the rest of this section, most of the operators that have been used in some of the best-known process algebras will be presented with the aim of showing the wealth of choices that one has when deciding how to describe a concurrent system or even when defining one's "personal" process algebra. A new calculus can, indeed, be obtained by a careful selection of the operators while taking into account their interrelationships with respect to the chosen abstract view of process and thus of the behavioral equivalence one has in mind.

A set of operators is the basis for building process terms. A labeled transition system (LTS) is associated to each term by relying on structural induction by providing specific rules in for each operator. Formally speaking, an LTS is a set of nodes (corresponding to process terms) and (for each action a in some set) a relation $\xrightarrow{a}$ between nodes, corresponding to processes transitions. Often LTSs have a distinguished node n_0 from which computations start; when defining the semantics of a process term, the state corresponding to that term is considered as the initial state. To associate an LTS to a process term, inference systems are used, where the collection of transitions is specified by means of a set of syntax-driven inference rules.

Inference systems: An inference system is a set of inference rule of the form:

$$\frac{p_1, \ldots, p_n}{q}$$

where $p_1, \ldots, p_n$ are the *premises* and q is the *conclusion*. Each rule is interpreted as an implication: if all premises are true, then also the conclusion is true. Sometimes, rules are decorated with predicates and/or negative premises that specify when the rule is actually applicable.

A rule with an empty set of premises is called *axiom* and written as:

$$\frac{}{q}$$

Transition rules: In the case of an operational semantics, the premises and the conclusions will be triples of the form (P, α, Q), often rendered as $P \xrightarrow{\alpha} Q$, and thus the rules for each operator op of the process algebras will be of the following form, where $\{i_1, \cdots, i_m\} \subseteq \{1, \cdots, n\}$ and $E'_i = E_i$ when $i \notin \{i_1, \ldots, i_m\}$:

$$\frac{E_{i_1} \xrightarrow{\alpha_1} E'_{i_1} \ \cdots \ E_{i_m} \xrightarrow{\alpha_m} E'_{i_m}}{op(E_1, \ldots, E_n) \xrightarrow{\alpha} C[E'_1, \ldots, E'_n]}$$

In the rule above, the target term $C[\ ]$ indicates the new context in which the new subterms will be operating after the reduction and α represents the action performed by the composite system when some of the components perform actions $\alpha_1, \ldots, \alpha_m$. Sometimes, these rules are enriched with side conditions that determine their applicability. By imposing syntactic constraints on the form of the allowed rules, *rule formats* are obtained that can be used to establish results that hold for all process calculi whose transition rules respect the specific rule format.

A small number of SOS inference rules is sufficient to associate an LTS to each term of any process algebra. The set of rules is fixed once and for all. Given any process, the rules are used to derive its transitions. The transition relation of the LTS is the **least** one satisfying the inference rules. It is worth remarking that *structural induction* allows one to define the LTS of complex systems in terms of the behavior of their components.

Basic actions: An elementary action of a system represents the **atomic** (non-interruptible) abstract step of a computation that is performed by a system to move from one state to the next. Actions represent various activities of concurrent systems, like sending or receiving a message, updating a memory cell, and synchronizing with other processes. In process algebras two main types of atomic actions are considered, namely, *visible* or external actions and invisible or *internal* actions. In the sequel, visible actions will be denoted by Latin letters $a, b, c, \ldots$, invisible actions will be denoted by the Greek letter τ. Generic actions will be denoted by μ or other, possibly indexed, Greek letters. In the following, A will be used to denote the set of visible actions while A_τ will denote the collection of generic actions.

Basic processes: Process algebras generally also include a null process (variously denoted as *nil*, 0, *stop*) which has no transition. It is inactive, and its sole purpose is to act as the inductive anchor on top of which more interesting processes can be generated. The semantics of this process is characterized by the fact that there is no rule to define its transition: it has no transition.

Other basic processes are also used: *stop* denotes a deadlocked state, $\surd$ denotes, instead, successful termination.

$$\frac{}{\surd \xrightarrow{\surd} stop}$$

Sometimes, uninterpreted actions μ are considered basic processes themselves:

$$\frac{}{\mu \xrightarrow{\mu} \surd}$$

Sequential composition: Operators for sequential composition are used to temporally order processes execution and interaction. There are two main operators

for this purpose. The first one is *action prefixing*, $\mu.$-, that denotes a process that executes action μ and then behaves like the following process.

$$\frac{}{\mu.E \xrightarrow{\mu} E}$$

The alternative form of sequential composition is obtained by explicitly requiring *process sequencing* , - ; -, that requires that the first operand process be fully executed before the second one.

$$\frac{E \xrightarrow{\mu} E'}{E;F \xrightarrow{\mu} E';F} \quad (\mu \neq \surd) \qquad \frac{E \xrightarrow{\surd} E' \qquad F \xrightarrow{\mu} F'}{E;F \xrightarrow{\mu} F'}$$

Nondeterministic composition: The operators for nondeterministic choice are used to express alternatives among possible behaviors. This choice can be left to the environment (*external choice*) or performed by the process (*internal choice*) or can be mainly external, but leaving the possibility to the process to perform an internal move to prevent some of the choices by the environment (*mixed choice*).

The rules for mixed choice are the ones below. They offer both visible and invisible actions to the environment; however, only the former kind of actions can be actually controlled.

$$\frac{E \xrightarrow{\mu} E'}{E+F \xrightarrow{\mu} E'} \qquad \frac{F \xrightarrow{\mu} F'}{E+F \xrightarrow{\mu} F'}$$

The rules for internal choice are very simple, they are just two axioms stating that a process $E \oplus F$ can silently evolve into one of its subcomponents.

$$\frac{}{E \oplus F \xrightarrow{\tau} E} \qquad \frac{}{E \oplus F \xrightarrow{\tau} F}$$

The rules for external choice are more articulate. This operator behaves exactly like the mixed choice in case one of the components executes a visible action; however, it does not discard any alternative upon execution of an invisible action.

$$\frac{E \xrightarrow{\alpha} E'}{E \,\square\, F \xrightarrow{\alpha} E'} \; (\alpha \neq \tau) \qquad \frac{F \xrightarrow{\alpha} F'}{E \,\square\, F \xrightarrow{\alpha} F'} \; (\alpha \neq \tau)$$

$$\frac{E \xrightarrow{\tau} E'}{E \,\square\, F \xrightarrow{\tau} E' \,\square\, F} \qquad \frac{F \xrightarrow{\tau} F'}{E \,\square\, F \xrightarrow{\tau} E \,\square\, F'}$$

Parallel composition: Parallel composition of two processes, say E and F, is the key primitive distinguishing process algebras from sequential models of computation. Parallel composition allows computation in E and F to proceed simultaneously and independently. But it also allows interaction, that is synchronization and flow of information between E and F on a shared channel. Channels may be synchronous or asynchronous. In the case of synchronous channels, the agent sending a message waits until another agent has received the message. Asynchronous channels do not force the sender to wait. Here, only synchronous channels will be considered.

The simplest operator for parallel composition is *interleaving*, - ||| -, that aims at modeling the fact that two parallel processes can progress by alternating at any rate the execution of their actions.

$$\frac{E \xrightarrow{\mu} E'}{E \,|||\, F \xrightarrow{\mu} E' \,|||\, F} \qquad \frac{F \xrightarrow{\mu} F'}{E \,|||\, F \xrightarrow{\mu} E \,|||\, F'}$$

Another parallel operator is *binary parallel composition*, - | -, that not only models the interleaved execution of the actions of two parallel processes, but also the possibility that the two partners synchronize whenever they are willing to perform complementary visible actions (below represented as a and $\overline{a}$). In this case, the visible outcome is a τ-action that cannot be seen by other processes that are acting in parallel with the two communication partners. This is the parallel composition used in CCS.

$$\frac{E \xrightarrow{\mu} E'}{E|F \xrightarrow{\mu} E'|F} \qquad \frac{F \xrightarrow{\mu} F'}{E|F \xrightarrow{\mu} E|F'}$$

$$\frac{E \xrightarrow{\alpha} E' \qquad F \xrightarrow{\overline{\alpha}} F'}{E|F \xrightarrow{\tau} E'|F'} (\alpha \neq \tau)$$

Instead of binary synchronization, some process algebras, like CSP, make use of operators that permit *multiparty synchronization*, - $|[L]|$ -. Some actions, those in L, are deemed to be synchronization actions and can be performed by a process only if all its parallel components can execute those actions at the same time.

$$\frac{E \xrightarrow{\mu} E'}{E \,|[L]|\, F \xrightarrow{\mu} E' \,|[L]|\, F} \; (\mu \notin L)$$

$$\frac{F \xrightarrow{\mu} F'}{E \,|[L]|\, F \xrightarrow{\mu} E \,|[L]|\, F'} \; (\mu \notin L)$$

$$\frac{E \xrightarrow{a} E' \qquad F \xrightarrow{a} F'}{E \,|[L]|\, F \xrightarrow{a} E' \,|[L]|\, F'} \ (a \in L)$$

It is worth noting that the result of a synchronization, in this case, yields a visible action, and that by setting the synchronization alphabet to $\emptyset$ the multiparty synchronization operator $|\emptyset|$ can be used to obtain pure interleaving, $|||$.

A more general composition is the *merge* operator, $- \parallel -$ that is used in ACP. It permits executing two process terms in parallel (thus freely interleaving their actions), but also allows for communication between its process arguments according to a *communication function* $\gamma : A \times A \longrightarrow A$, that, for each pair of atomic actions a and b, produces the outcome of their communication $\gamma(a, b)$, a partial function that states which actions can be synchronized and the outcome of such a synchronization.

$$\frac{E \xrightarrow{\mu} E'}{E \parallel F \xrightarrow{\mu} E' \parallel F} \qquad \frac{F \xrightarrow{\mu} F'}{E \parallel F \xrightarrow{\mu} E \parallel F'}$$

$$\frac{E \xrightarrow{a} E' \qquad F \xrightarrow{b} F'}{E \parallel F \xrightarrow{\gamma(a,b)} E' \parallel F'}$$

ACP has also another operator called *left merge*, - $\lfloor\!\lfloor$ -, that is similar to $\parallel$ but requires that the first process to perform an (independent) action be the left operand.

$$\frac{E \xrightarrow{\mu} E'}{E \lfloor\!\lfloor F \xrightarrow{\mu} E' \parallel F}$$

The ACP *communication merge*, - $|_c$ -, requires instead that the first action be a synchronization action.

$$\frac{E \xrightarrow{a} E' \qquad F \xrightarrow{b} F'}{E |_c F \xrightarrow{\gamma(a,b)} E' \parallel F'}$$

Disruption: An operator that is between parallel and nondeterministic composition is the so-called *disabling* operator, $-[> -$, that permits interrupting the evolution of a process. Intuitively, $E [> F$ behaves like E, but can be interrupted at any time by F, once E terminates F is discarded.

$$\frac{E \xrightarrow{\mu} E'}{E [> F \xrightarrow{\mu} E' [> F} \ (\mu \neq \surd) \qquad \frac{E \xrightarrow{\surd} E'}{E [> F \xrightarrow{\tau} E'}$$

$$\frac{F \xrightarrow{\mu} F'}{E [> F \xrightarrow{\mu} F'}$$

Value passing: The above parallel combinators can be generalized to model not only synchronization, but also exchange of values. As an example, below, the generalization of binary communication is presented.

There are complementary rules for sending and receiving values. The first axiom models a process willing to input a value and to base its future evolutions on it. The second axiom models a process that evaluates an expression (via the valuation function $val(e)$) and outputs the result.

$$\frac{}{a(x).E \xrightarrow{a(v)} E\{v/x\}} \ (v \text{ is a value}) \qquad \frac{}{\overline{a}\, e.E \xrightarrow{\overline{a}\ val(e)} E}$$

The next rule, instead, models synchronization between processes. If two processes, one willing to output and the other willing to input, are running in parallel, a synchronization can take place and the perceived action will just be a τ-action.

$$\frac{E \xrightarrow{\overline{a}\ v} E' \quad F \xrightarrow{a(v)} F'}{E|F \xrightarrow{\tau} E'|F'} \qquad \frac{E \xrightarrow{a(v)} E' \quad F \xrightarrow{\overline{a}\ v} F'}{E|F \xrightarrow{\tau} E'|F'}$$

In case the exchanged values are channels, this approach can be used to provide also models for mobile systems.

Abstraction: Processes do not limit the number of connections that can be made at a given interaction point. But interaction points allow interference. For the synthesis of compact, minimal, and compositional systems, the ability to restrict interference is crucial.

The *hiding* operator, $-/L$, hides (i.e., transforms into τ-actions) all actions in L to forbid synchronization on them. However, it allows the system to perform the transitions labeled by hidden actions.

$$\frac{E \xrightarrow{\mu} E'}{E/L \xrightarrow{\mu} E'/L} \ (\mu \notin L) \qquad \frac{E \xrightarrow{\mu} E'}{E/L \xrightarrow{\tau} E'/L} \ (\mu \in L)$$

The *restriction* operator, $-\backslash L$ is a unary operator that restricts the set of visible actions a process can perform. Thus, process $E\backslash L$ can perform only actions not in L. Obviously, invisible actions cannot be restricted.

$$\frac{E \xrightarrow{\mu} E'}{E \backslash L \xrightarrow{\mu} E' \backslash L} \ (\mu, \overline{\mu} \notin L)$$

The operator $[f]$, where f is a *relabeling* function from A to A, can be used to rename some of the actions

a process can perform to make it compatible with new environments.

$$\frac{E \xrightarrow{a} E'}{E[f] \xrightarrow{f(a)} E'[f]}$$

Modeling infinite behaviors: The operations presented so far describe only finite interaction and are consequently insufficient for providing full computational power, in the sense of being able to model all computable functions. In order to reach full power, one certainly needs operators for modeling non-terminating behavior. Many operators have been introduced that allow finite descriptions of infinite behavior. However, it is important to remark that most of them do not fit the formats used so far and cannot be defined by structural induction.

One of the most used is the construct *rec x.* -, well-known from the sequential world. If E is a process that contains the variable x, then *rec x. E* represents the process that behaves like E once all occurrences of x in E are replaced by *rec x. E*. In the rule below, that models the operational behavior of a recursively defined process, the term $E[f/x]$ denotes exactly the above mentioned substitutions.

$$\frac{E[rec\,x.\;E/x] \xrightarrow{\mu} E'}{rec\,x.\;E \xrightarrow{\mu} E'}$$

The notation *rec x. E* for recursion sometimes makes the process expressions more difficult to parse and less pleasant to read. A suitable alternative is to allow for the (recursive) definition of some fixed set of constants, that can then be used as some sort of procedure calls inside processes. Assuming the existence of an environment (a set of process definitions)

$$\Gamma = \{X_1 \triangleq E_1, X_2 \triangleq E_2, \ldots, X_n \triangleq E_n\}$$

the operational semantics rule for a process variable becomes:

$$\frac{X \triangleq E \in \Gamma \quad E \xrightarrow{\mu} E'}{X \xrightarrow{\mu} E'}$$

Another operator used to describe infinite behaviors is the so-called *bang* operator, ! –, or *replication*. Intuitively, $!E$ represents an unlimited number of instances of E running in parallel. Thus, its semantics is rendered by the following inference rule:

$$\frac{E|!\,E \xrightarrow{\mu} E'}{!E \xrightarrow{\mu} E'}$$

Three Process Algebras: CCS, CSP and ACP

A process algebra consists of a set of terms, an operational semantics associating LTS to terms, and an equivalence relation equating terms exhibiting "similar" behavior. The operators for most process algebras are those described above. The equivalences can be traces, testing, bisimulation equivalences, or variants thereof, possibly ignoring invisible actions.

Below, three of the most popular process algebras are presented. First the syntax, i.e., the selected operators, will be introduced, then their semantics will be provided by following the three different approaches outlined before: operational (for CCS), denotational (for CSP), and algebraic (for ACP). For CSP and ACP, the relationships between the proposed semantics and the operational one, to be used as a yardstick, will be mentioned. To denote the LTS associated to a generic CSP or ACP process p via the operational semantics, the notation **LTS**(p) will be used.

Reference will be made to specific behavioral equivalences over LTSs that consider as equivalent those systems that rely on different standing about which states of an LTS have to be considered equivalent. Three main criteria have been used to decide when two systems can be considered equivalent:

1. The two systems perform the same sequences of actions.
2. The two systems perform the same sequences of actions and after each sequence are ready to accept the same sets of actions.
3. The two systems perform the same sequences of actions and after each sequence exhibit, recursively, the same behavior.

These three different criteria lead to three groups of equivalences that are known as *traces* equivalences, *decorated-traces* equivalences (testing and failure equivalence), and *bisimulation-based* equivalences (strong bisimulation, weak bisimulation, branching bisimulation).

CCS: Calculus of Communicating Systems

The Calculus of Communicating Systems (CCS) is a process algebra introduced by Robin Milner around 1980. Its actions model indivisible communications

between exactly two participants, and the set of operators includes primitives for describing parallel composition, choice between actions, and scope restriction. The basic semantics is operational and permits associating an LTS to each CCS term.

The set A of basic actions used in CCS consists of a set Λ, of labels and of a set $\overline{\Lambda}$ of complementary labels. A_τ denotes $A \cup \{\tau\}$. The syntax of CCS, used to generate all terms of the algebra, is the following:

$$P ::= nil \mid x \mid \mu.P \mid P\backslash L \mid P[f] \mid P_1{+}P_2 \mid P_1|P_2 \mid rec\,x.\;P$$

where $\mu \in A_\tau$; $L \subseteq \Lambda$; $f : A_\tau \to A_\tau$; $f(\bar{\alpha}) = \overline{f(\alpha)}$ and $f(\tau) = \tau$. The above operators are taken from those presented in Section 2:

- The atomic process (nil)
- Action prefixing ($\mu.P$)
- Mixed choice ($+$)
- Binary parallel composition ($|$)
- Restriction ($P\backslash L$)
- Relabeling ($P[f]$)
- Recursive definitions ($rec\,x.\;E$)

The operational semantics of the above operators is exactly the same as the one of those operators with the same name described before, and it is thus not repeated here. CCS has been studied with bisimulation and testing semantics that are used to abstract from unnecessary details of the LTS associated to a term. Also denotational and axiomatic semantics for the calculus have been extensively studied. A denotational semantics in terms of so-called *acceptance trees* has been proved to be in full agreement with the operational semantics abstracted according to testing equivalences. Different algebraic semantics have been provided that are based on sound and complete axiomatizations of bisimilarity, testing equivalence, weak bisimilarity, and branching bisimilarity.

CSP: A Theory of Communicating Sequential Processes

The first denotational semantics proposed for CSP associates to each term just the set of the sequences of actions the term could induce. However, while suitable to model the sequences of interactions a process could have with its environment, this semantics is unable to model situations that could lead to deadlock. A new approach, basically denotational but with a strong operational intuition, was proposed next. In this approach, the semantics is given by associating a so-called refusal set to each process. A refusal set is a set of failure pairs $\langle s, F\rangle$, where s is a finite sequence of visible actions in which the process might have been engaged, and F is a set of action the process is able to reject on the next step. The semantics of the various operators is defined by describing the transformation they induce on the domain of refusal sets.

The meaning of processes is then obtained by postulating that two processes are equivalent if and only if they cannot be distinguished when their behaviors are observed and their reactions to a finite number of alternative possible synchronization is considered. Indeed, the association of processes to refusal sets is not one-to-one; the same refusal set can be associated to more than one process. A congruence is then obtained that equates processes with the same denotation.

The set of actions is a finite set of labels, denoted by $\Lambda \cup \{\tau\}$. There is no notion of complementary action. The syntax of CSP is reported below, and for the sake of simplicity, only finite terms (no recursion) are considered:

$$E ::= \textsc{Stop} \mid skip \mid a \to E \mid E_1 \sqcap E_2 \mid E_1 \;\Box\; E_2 \mid E_1 \;|[L]|\; E_2 \mid E/a$$

- Two basic processes: successful termination ($skip$), null process ($\textsc{Stop}$)
- Action prefixing here denoted by $a \to E$
- Internal choice ($\oplus$) here denoted by $\sqcap$ and external choice ($\Box$)
- Parallel composition with synchronization on a fixed alphabet ($|[L]|$, $L \subseteq \Lambda$)
- Hiding ($/a$, an instance of the more general operator $/L$ with $L \subseteq \Lambda$)

Parallel combinators representing pure interleaving and parallelism with synchronization on the full alphabet can be obtained by setting the synchronization alphabet to $\emptyset$ or to Λ, respectively.

The denotational semantics of CSP compositionally associates a set of failure pairs to each CSP term generated by the above syntax. A function $\mathcal{F}[\![-]\!]$ maps each CSP process (say P) to set of pairs (s, F), where s is one of the sequences of actions P may perform, and F represents the set of actions that P can refuse after performing s. As anticipated, there is a strong correspondence between the denotational semantics of CSP and the operational semantics that one could define by

relying on the one presented in the previous section for the specific operators.

- $\mathcal{F}[[P]] = \mathcal{F}[[Q]]$ if and only if $\mathbf{LTS}(P) \simeq_{test} \mathbf{LTS}(Q)$.

ACP: An Algebra of Communicating Processes

The methodological concern of ACP was to present "first a system of axioms for communicating processes … and next study its models" ([6], p. 112). The equations are just a means to realize the real desideratum of abstract algebra, which is to abstract from the nature of the objects under consideration. In the same way as the mathematical theory of rings is about arithmetic without relying on a mathematical definition of number, ACP deals with process theory without relying on a mathematical definition of process.

In ACP, a process algebra is any mathematical structure, consisting of a set of objects and a set of operators, like, e.g., sequential, nondeterministic, or parallel composition, that enjoys a specific set of properties as specified by given axioms.

The set of actions Λ_τ consists of a finite set of labels $\Lambda \cup \{\tau\}$. There is no notion of complementary action. The syntax of ACP is reported below, and for the sake of simplicity, only finite terms (no recursion) are considered:

$$P ::= \surd|\delta|a|P_1 + P_2|P_1{\cdot}P_2|P_1\|P_2|P_1 \lfloor\!\lfloor P_2|P_1|_c P_2|\partial_H(p)$$

- Three basic processes: successful termination ($\surd$), null process, here denoted by δ, and atomic action (a)
- Mixed choice
- Sequential composition (;), here denoted by $\cdot$
- Hiding ($\backslash H$ with $H \subseteq \Lambda$), here denoted by $\partial_H(-)$
- Three parallel combinators: merge ($\|$), left merge ($\lfloor\!\lfloor$) and communication merge ($|_c$)

The system of axioms of ACP is presented as a set of formal equations, and some of the operators, e.g., left merge ($\lfloor\!\lfloor$) have been introduced exactly for providing finite equational presentations. Below, the axioms relative to the terms generated by the above syntax are presented. Within the axioms, x and y denote generic ACP processes.

(A1) $x + y = y + x$ (A2) $(x + y) + z = x + (y + z)$

(A3) $x + x = x$ (A4) $(x + y){\cdot}z = x{\cdot}z + y{\cdot}z$

(A5) $(x{\cdot}y){\cdot}z = x{\cdot}(y{\cdot}z)$ (A6) $x + \delta = x$

(A7) $\delta{\cdot}x = \delta$

The set of axioms considered above induces an *equality relation*, denoted by =. A *model* for an axiomatization is a pair $\langle \mathcal{M}, \phi \rangle$, where $\mathcal{M}$ is a set and ϕ is a function (the unique isomorphism) that associates elements of $\mathcal{M}$ to ACP terms. This leads to the following definitions:

1. A set of equations is *sound* for $\langle \mathcal{M}, \phi \rangle$ if $s = t$ implies $\phi(s) = \phi(t)$.
2. A set of equations is *complete* for $\langle \mathcal{M}, \phi \rangle$ if $\phi(s) = \phi(t)$ implies $s = t$.

Any model of the axioms seen above is an ACP process algebra. The simplest model for ACP has as elements the equivalence classes induced by =, i.e., all ACP terms obtained starting from atomic actions, sequentialization and nondeterministic composition and mapping each term t to its equivalence class $[[t]]$ as determined by =. This model is correct and complete and is known as the *initial model* for the axiomatization.

Different, more complex, models can be obtained by first using the SOS rules to give the operational semantics of the operators, building an LTS in correspondence of each ACP term and then using bisimulation to identify some of them. This construction leads to establishing a strong correspondence between the axiomatic and the operational semantics of ACP. Indeed, if we consider the language with the null process, sequential composition and mixed choice we have:

- Equality = as induced by (A1)-(A7) is *sound* relative to bisimilarity ~, i.e., if $p = q$ then $\mathbf{LTS}(p) \sim \mathbf{LTS}(q)$;
- Equality = as induced by (A1)-(A7) is *complete* relative to bisimilarity ~, i.e., if $\mathbf{LTS}(p) \sim \mathbf{LTS}(q)$ then $p = q$.

Similar results can be obtained when new axioms are added and weak bisimilarity or branching bisimilarity are used to factorize the LTSs.

Future Directions

The theory of process algebra is by now well developed. The reader is referred to [7] to learn about its developments since its inception in the late 1970s to the early 2000. Currently, in parallel with the exploitation of the developed theories in classic areas such as protocol verification and in new ones such as biological systems, there is much work going on concerning:

- Extensions to model mobile, network aware systems
- Theories for assessing quantitative properties
- Techniques for controlling state explosion

In parallel with this, much attention is dedicated to the development of software tools to support specification and verification of very large systems and to the development of techniques that permit controlling the state explosion phenomenon that arise as soon as one considers the possible configurations resulting from the interleaved execution of (even a small number of) processes.

Mobility and network awareness: Much of the ongoing work is relative to the definition of theories and formalisms to naturally deal with richer classes of systems, like, e.g., mobile systems and network aware applications. The π-calculus [33] the successor of CCS, developed by Milner and coworkers with the aim of describing concurrent systems whose configuration may change during the computation has attracted much attention. It has laid the basis for research on process networks whose processes are mobile and the configuration of communication links is dynamic. It has also lead to the development of other calculi to support network aware programming: Ambient [13], Distributed π [25], Join [19], Spi [1], Klaim [9], etc. There is still no unifying theory and the name process calculi is preferred to process algebras because the algebraic theories are not yet well assessed. Richer theories than LTS (Bi-graph [34], Tiles [20], etc.) have been developed and are still under development to deal with the new dimensions considered with the new formalisms.

Quantitative extensions: Formalisms are being enriched to consider not only qualitative properties, like correctness, liveness, or safety, but also properties related to performance and quality of service. There has been much research to extend process algebra to deal with a quantitative notion of time and probabilities and integrated theories have been considered. Actions are enriched with information about their duration, and formalisms extended in this way are used to compare systems relatively to their speed. For a comprehensive description of this approach, the reader is referred to [4]. Extensions have been considered also to deal with systems that in their behavior depend on continuously changing variables other than time (*hybrid systems*). In this case, systems descriptions involve differential algebraic equations, and connections with dynamic control theory are very important. Finally, again with the aim of capturing quantitative properties of systems and of combining functional verification with performance analysis, there have been extensions to enrich actions with rates representing the frequency of specific events and the new theories are being (successfully) used to reason about system performance and system quality.

Tools: To deal with non-toy examples and apply the theory of process algebras to the specification and verification of real systems, tool support is essential. In the development of tools, LTSs play a central role. Process algebra terms are used to obtain LTSs by exploiting operational semantics and these structures are then minimized, tested for equivalence, model checked against formulae of temporal logics, etc. One of the most known tools for process algebras is CADP (Construction and Analysis of Distributed Processes) [21]: together with minimizers and equivalence and model checkers, it offers many others functionalities ranging from step-by-step simulation to massively parallel model checking. CADP has been employed in an impressive number of industrial projects. CWB (Concurrency Workbench) [35] and CWB-NC (Concurrency Workbench New Century) [15] are other tools that are centered on CCS, bisimulation equivalence, and model checking. FDR (Failures/Divergence Refinement) [40] is a commercial tool for CSP that has played a major role in driving the evolution of CSP from a blackboard notation to a concrete language. It allows the checking of a wide range of correctness conditions, including deadlock and livelock freedom as well as general safety and liveness properties. TAPAs (Tool for the Analysis of Process Algebras) [12] is a recently developed software to support teaching of the theory of process algebras; it maintains a consistent double representation as term and as graph of each system.

Moreover, it offers tools for the verification of many behavioral equivalences, possibly with counterexamples, minimization, step-by-step execution, and model checking. TwoTowers is instead a versatile tool for the functional verification, security analysis, and performance evaluation of computer, communication, and software systems modeled with the stochastic process algebra EMPA [4]. μCRL [23] is a toolset that offers an appropriate treatment of data and relies also on theorem proving. Moreover, it make use of interesting techniques for visualizing large LTSs.

Relationships to Other Models of Concurrency

In a private communication, in 2009, Robin Milner, one of the founding fathers of process algebras, wrote:

▸ The concept of process has become increasingly important in computer science in the last three decades and more. Yet we still don't agree on what a process is. We probably agree that it should be an equivalence class of interactive agents, perhaps concurrent, perhaps non-deterministic.

This quote summarizes the debate on possible models of concurrency that has taken place during the last thirty years and has been centered on three main issues:

- Interleaving vs true concurrency
- Linear-time vs branching-time
- Synchrony vs asynchrony

Interleaving vs True Concurrency

The starting point of the theory of process algebras has been automata theory and regular expressions, and the work on the algebraic theory of regular expressions as terms representing finite state automata [16] has significantly influenced its developments. Given this starting point, the underlying models of all process algebras represent possible concurrent executions of different programs in terms of the nondeterministic interleaving of their sequential behaviors. The fact that a system is composed by independently computing agents is ignored and behaviors are modeled in terms of purely sequential patterns of actions. It has been demonstrated that many interesting and important properties of distributed systems may be expressed and proved by relying on interleaving models. However, there are situations in which it is important to keep the information that a system is composed of the independently computing components. This possibility is offered by the so-called non-interleaving or true-concurrency models, with Petri nets [39] as the prime example. These models describe not only temporal ordering of actions, but also their causal dependences. Non-interleaving semantics of process algebras have also been provided, see, e.g., [36].

Linear-time vs Branching-time

Another issue, again ignored in the initial formalization of regular expressions, is how the concept of nondeterminism in computations is captured. Two possible views regarding the nature of nondeterministic choice induce two types of models giving rise to the linear-time and branching-time dichotomy. A linear-time model expresses the full nondeterministic behavior of a system in terms of the set of possible runs; time is treated as if each moment there is a unique possible future. Major examples of structures used to model sets of runs are Hoare traces (captured also by traces equivalence) for interleaving models [26], and Mazurkiewicz traces [30] and Pratt's pomsets [38] for non-interleaving models. The branching-time model is the main one considered in process algebras and considers the set of runs structured as a computation tree. Each moment in time may split into various possible futures, and semantic models are computation trees. For non-interleaving models, event structures [44] are one of the best-known models taking into account both nondeterminism and true concurrency.

Synchrony vs Asynchrony

There are two basic approaches to describing interaction between a sender and a receiver of a message (signal), namely, synchronous and asynchronous interaction. In the former case, before proceeding, the sender has to make sure that a receiver is ready. In the latter case, the sender leaves track of its action but proceeds without any further waiting. The receiver has to wait in both cases. Process algebras are mainly synchronous, but asynchronous variants have been recently proposed and are receiving increasing attention. However, many other successful asynchronous models have been developed. Among these, it is important to mention Esterel

[8], a full-fledged programming language that allows the simple expression of parallelism and preemption and is very well suited for control-dominated model designs; Actors [3], a formalism that does not necessarily records messages in buffers and puts no requirement on the ordering of message delivery; Linda [22], a model of coordination and communication among several parallel processes operating upon objects stored in and retrieved from shared, virtual, associative memory; and, to conclude, Klaim [17], a distributed variant of Linda with a strong process algebraic flavor.

Related Entries

►Actors
►Behavioral Equivalences
►Bisimulation
►CSP (Communicating Sequential Processes)
►Pi-Calculus

Bibliographic Notes and Further Reading

A number of books describing the different process algebras can be consulted to obtain deeper knowledge of the topics sketched here. Unfortunately most of them are concentrating only on one of the formalisms rather than on illustrating the unifying theories.

CCS: The seminal book on CCS is [31], in which sets of operators equipped with an operational semantics and the notion of observational equivalence have been presented for the first time. The, by now, classical text book on CCS and bisimulation is [32]. A very nice, more recent, book on CCS and the associated Hennessy Milner Modal Logic is [29]; it also presents timed variants of process algebras and introduces models and tools for verifying properties also of this new class of systems.

CSP: The seminal book on CSP is [27], where all the basic theory of failure sets is presented together with many operators for processes composition and basic examples. In [41], the theory introduced in [27] is developed in full detail, and a discussion on the different possibilities to deal with anomalous infinite behaviors is considered together with a number of well-thought examples. Moreover the relationships between operational and denotational semantics are fully investigated. Another excellent text book on CSP is [43] that also considers timed extensions of the calculus.

ACP: The first published book on ACP is [5], where the foundations of algebraic theories are presented and the correspondence between families of axioms, and strong and branching bisimulation are thoroughly studied. This is a book intended mainly for researchers and advanced students, a gentle introduction to ACP can be found in [18].

Other approaches: Apart from these books, dealing with the three process algebras presented in these notes, it is also worth mentioning a few more books. LOTOS, a process algebra that was developed and standardized within ISO for specifying and verifying communication protocols, is the central calculus of a recently published book [10] that discusses also the possibility of using different equivalences and finer semantics for the calculus. A very simple and elegant introduction to algebraic, denotational, and operational semantics of processes, which studies in detail the impact of the testing approach on a calculus obtained from a careful selection of operators from CCS and CSP, can be found in [24]. The text [42] is *the* book on the π-calculus. For studying this calculus, the reader is, however, encouraged to consider first reading [33].

P

Bibliography

1. Abadi M, Gordon AD (1999) A calculus for cryptographic protocols: the spi calculus. Inform Comput 148(1):1–70
2. Aceto L, Gordon AD (eds) (2005) Proceedings of the workshop “Essays on algebraic process calculi” (APC 25), Bertinoro, Italy. Electronic notes in theoretical computer science vol 162. Elsevier, Amsterdam
3. Agha G (1986) Actors: a model of concurrent computing in distributed systems. MIT Press, Cambridge
4. Aldini A, Bernardo M, Corradini F (2010) A process algebraic approach to software architecture design. Springer, New York
5. Baeten JCM, Weijland WP (1990) Process algebra. Cambridge University Press, Cambridge
6. Bergstra JA, Klop JW (1984) Process algebra for synchronous communication. Inform Control 60(1–3):109–137
7. Bergstra JA, Ponse A, Smolka SA (eds) (2001) Handbook of process algebra. Elsevier, Amsterdam
8. Berry G, Gonthier G (1992) The esterel synchronous programming language: design, semantics, implementation. Sci Comput Program 19(2):87–152

9. Bettini L, Bono V, Nicola R, Ferrari G, Gorla D, Loreti M, Moggi E, Pugliese R, Tuosto E, Venneri B (2003) The klaim project: theory and practice. In: Global computing: programming environments, languages, security and analysis of systems, Lecture notes in computer science, vol 2874. Springer-Verlag, Heidelberg, pp 88–150
10. Bowman H, Gomez R (2006) Concurrency theory: calculi and automata for modelling untimed and timed concurrent systems. Springer, London
11. Brookes SD, Hoare CAR, Roscoe AW (1984) A theory of communicating sequential processes. J ACM 31(3):560–599
12. Calzolai F, De Nicola R, Loreti M, Tiezzi F (2008) Tapas: a tool for the analysis of process algebras. In: Transactions on Petri nets and other models of concurrency, vol 1, pp 54–70
13. Cardelli L, Gordon AD (2000) Mobile ambients. Theor Comput Sci 240(1):177–213
14. Clarke EM, Emerson EA (1982) Design and synthesis of synchronization skeletons using branching-time temporal logic. In: Proceedings of logic of programs, Lecture notes in computer science, vol 131. Springer-Verlag, Heidelberg, pp 52–71
15. Cleaveland R, Sims S (1996) The ncsu concurrency workbench. In: CAV, Lecture notes in computer science, vol 1102. Springer-Verlag, Heidelberg, pp 394–397
16. Conway JH (1971) Regular algebra and finite machines. Chapman and Hall, London
17. De Nicola R, Ferrari GL, Pugliese R (1998) Klaim: a kernel language for agents interaction and mobility. IEEE Trans Software Eng 24(5):315–330
18. Fokkink W (2000) Introduction to process algebra. Springer-Verlag, Heidelberg
19. Fournet C, Gonthier G (2000) The join calculus: a language for distributed mobile programming. In: Barthe G, Dybjer P, Pinto L, and Saraiva J (eds) APPSEM, Lecture notes in computer science, vol 2395, Springer, Heidelberg, pp 268–332
20. Gadducci F, Montanari U (2000) The tile model. In: Plotkin G, Stirling C, Tofte M (eds) Proof, language and interaction: essays in honour of Robin Milner. MIT Press, Cambridge, pp 133–166
21. Garavel H, Lang F, Mateescu R (2002) An overview of CADP 2001. In: European Association for Software Science and Technology (EASST), vol 4. Newsletter, pp 13–24
22. Gelernter D, Carriero N (1992) Coordination languages and their significance. Commun ACM 35(2):96–107
23. Groote JF, Mathijssen AHJ, Reniers MA, Usenko YS, van Weerdenburg MJ (2009) Analysis of distributed systems with mcrl2. In: Alexander M, Gardner W (eds) Process algebra for parallel and distributed processing. Chapman Hall, Boca Raton, FL, pp 99–128
24. Hennessy M (1988) Algebraic theory of processes. The MIT Press, Cambridge
25. Hennessy M (2007) A distributed pi-calculus. Cambridge University Press, Cambridge
26. Hoare CAR (1981) A calculus of total correctness for communicating processes. Sci Comput Program 1(1–2):49–72
27. Hoare CAR (1985) Communicating sequential processes. Prentice-Hall, Upper Saddle River
28. Kozen D (1983) Results on the propositional -calculus. Theor Comput Sci 27:333–354
29. Larsen KG, Aceto L, Ingolfsdottir A, Srba J (2007) Reactive systems: modelling, specification and verification. Cambridge University Press, Cambridge
30. Mazurkiewicz A (1995) Introduction to trace theory. In: Rozenberg G, Diekert V (ed) The book of traces. World Scientific, Singapore, pp 3–67
31. Milner R (1980) A calculus of communicating systems. Lecture notes in computer science, vol 92. Springer-Verlag, Heidelberg
32. Milner R (1989) Communication and concurrency. Prentice-Hall, Upper Saddle River
33. Milner R (1999) Communicating and mobile systems: the pi-calculus. Cambridge University Press, Cambridge
34. Milner R (2009) The space and motion of communicating agents. Cambridge University Press, Cambridge
35. Moller F, Stevens P (1999) Edinburgh Concurrency Workbench user manual. Available from http://homepages.inf.ed.ac.uk/perdita/cwb/. Accessed January 2011
36. Olderog ER (1991) Nets, terms and formulas. Cambridge University Press, Cambridge
37. Plotkin GD (2004) A structural approach to operational semantics. J Log Algebr Program 60–61:17–139
38. Pratt V (1986) Modeling concurrency with partial orders. Int J Parallel Process 1:3371
39. Reisig W (1985) Petri nets: an introduction. Monographs in theoretical computer science. An EATCS Series, vol 4. Springer-Verlag, Berlin
40. Roscoe AW (1994) Model-checking csp. In: A classical mind: essays in honour of C.A.R. Hoare. Prentice-Hall, Upper Saddle River
41. Roscoe AW (1998) The theory and practice of concurrency. Prentice-Hall, Upper Saddle River
42. Sangiorgi D, Walker D (2001) The π-Calculus: a theory of mobile processes. Cambridge University Press, Cambridge
43. Schneider SA (1999) Concurrent and real time systems: the CSP approach. John Wiley, Chichester
44. Winskel G (1989) An introduction to event structures. In: de Bakker JW, de Roever WP, Rozenberg G (eds) Linear time, branching time and partial order in logics and models for concurrency – Rex workshop, Lecture notes in computer science, vol 354. Springer, Heidelberg, pp 364–397

Process Calculi

►Process Algebras

Process Description Languages

►Process Algebras

Process Synchronization

►Path Expressions
►Synchronization

Processes, Tasks, and Threads

Processes, Tasks, and Threads are programs or parts of programs under execution. A program does not define a process, task, or thread since the same code may underlay different processes, tasks, or threads. At any given time, the state of a process, task, or thread includes the location of the instruction being executed and the value stored in all memory locations accessible to the program. An important characteristic of processes, tasks, and threads is that they must execute their instructions at a speed greater than zero when they are not explicitly blocked by synchronization operations.

Although the notion of process, tasks, and threads differs across systems, typically a process may contain multiple threads, and threads share memory while processes communicate via messages.

Processor Allocation

►Job Scheduling

Processor Arrays

►Systolic Arrays

Processors-in-Memory

Processors-in-memory (PIM) are devices that tightly integrate, for example, on a single chip, both processing logic and memory with the objective of reducing memory latency and increasing memory bandwidth at a low cost in terms of power, complexity, and space.

Bibliography

1. Patterson D, Anderson T, Cardwell N, Fromm R, Keeton K, Kozyrakis C, Thomas R, Yelick K (1997) A case for intelligent RAM. IEEE Micro 17(2):34–44
2. Kogge PM, Brockman JB, Sterling T, Gao G Processing in memory: chips to petaflops. In: Workshop on mixing logic and DRAM: chips that compute and remember at ISCA'97 1997

Profiling

►Intel® Thread Profiler
►OpenMP Profiling with ompP
►Performance Analysis Tools
►Scalasca
►TAU

Profiling with OmpP, OpenMP

►OpenMP Profiling with OmpP

Program Graphs

►Data Flow Graphs

Programmable Interconnect Computer

►Blue CHiP

Programming Languages

►Array Languages
►C*
►Chapel (Cray Inc. HPCS Language)
►Cilk
►CoArray Fortran
►Concurrent ML

►Connection Machine Fortran
►Connection Machine Lisp
►Deterministic Parallel Java
►Fortran 90 and Its Successors
►Fortress (Sun HPCS Language)
►Glasgow Parallel Haskell (GpH)
►HPF (High Performance Fortran)
►Linda
►*Lisp
►Multilisp
►NESL
►OpenMP
►Functional Languages
►Logic Languages
►PGAS (Partitioned Global Address Space) Languages
►Sisal
►Stream Programming Languages
►Titanium
►UPC
►ZPL

Programming Models

►Actors
►BSP (Bulk Synchronous Parallelism)
►Concurrent Collections Programming Model
►CSP (Communicating Sequential Processes)
►SPMD Computational Model

Prolog

►Logic Languages

Prolog Machines

Prolog machines [1, 2] include instructions and devices for the efficient implementation of Prolog programs. During the years of Japan's fifth generation project, several data flow machines were designed to implement KL1, a parallel variant of Prolog.

Related Entries

►Data Flow Computer Architecture
►Logic Languages

Bibliography

1. Warren DH (1982) A view of the fifth generation and its impact. AI Mag 3(4):34–39
2. Holmer BK, Sano B, Carlton M, Van Roy P, Despain AM (1994) Design and analysis of hardware for high performance prolog. J Logic Program 19(20):1–679

Promises

►Futures

Protein Docking

Roger S. Armen[1], Eric R. May[2], Michela Taufer[3]
[1]Thomas Jefferson University, Philadelphia, PA, USA
[2]University of Michigan, Ann Arbor, MI, USA
[3]University of Delaware, Newark, DE, USA

Definition

Predict the final atomic resolution structure of a protein–protein complex starting from the coordinates of the unbound conformation of each component protein.

Discussion

Introduction

Most successful approaches to protein–protein docking use multistage hierarchical methods typically consisting of an initial global rigid body search (or a simplified reduced search based on known information) to identify a number of candidate protein–protein complexes followed by a more sophisticated refinement procedure. After searching, filtering, and refining, the predicted complexes are rescored based on a sophisticated scoring function for selecting the lowest energy complex. The native protein–protein complex is at the global free energy minimum; therefore those models with the lowest (free) energy should be the most similar to the native complex.

Approaches to Conformational Searching

Exhaustive Rigid Body Searching

The first step in docking two known protein structures is to generate configurations of protein–protein complexes. Traditionally, the generation of conformations

consists of large-scale simulations of independent jobs and thus can be performed in parallel on parallel or distributed systems such as clusters, volunteer computing systems, grid computing systems, or cloud computing platforms. Treating the proteins as rigid bodies is the most computationally efficient method and will be discussed in this section (incorporation of increasing levels of protein flexibility will be covered later). There are six degrees of freedom (three rotation and three translation) that must be explored for an exhaustive search of complex geometries.

The interactions that dominate the favorable binding energy of a protein–protein complex are interactions between the surface residues of the proteins. Therefore, one can naively assume that the most favorable complex geometries are those that bury the largest amount of surface area at the interface. This concept leads to the notion of surface complementarity (i.e., bulges on the surface of Protein A align with a valley on the surface of the Protein B), which is a phenomena observed in many experimentally determined protein complexes. To identify those complexes that maximize the interfacial surface area, a method of describing the individual protein surfaces must first be addressed. An effective method is to project each protein onto a 3D grid and identify which grid points lie at the protein surface and which in the protein interior, as was first described by Katchalski-Katzir et al. [1]. The grid can be represented by a function a when Protein A is placed in the grid, and function b when Protein B is placed in the grid, where

$$a_{l,m,n} = \begin{cases} 1 & \text{surface points} \\ << 0 & \text{interior points} \\ 0 & \text{exterior points} \end{cases}$$

and

$$b_{l,m,n} = \begin{cases} 1 & \text{surface points} \\ > 0 & \text{interior points} \\ 0 & \text{exterior points,} \end{cases}$$

where l, m, and n are the grid indices. With these definitions, a correlation function can be calculated between the functions a and b as follows:

$$c_{\alpha,\beta,\gamma} = \sum_l \sum_m \sum_n a_{l,m,n} \cdot b_{l+\alpha,m+\beta,n+\gamma} \tag{1}$$

where $\{\alpha, \beta, \gamma\}$ is a shift vector applied to Protein B. The value of c is zero when there is no contact between the proteins, is negative when there is overlap of protein interior regions, and is positive when the surface regions overlap. Complexes with strong positive values have the largest surface complementarity and are retained for further assessment.

The evaluation of c for a given configuration, requires $O(N^3)$ computations (for a grid with dimensions $N \times N \times N$); searching all translations of Protein B (all shift vectors) requires $O(N^6)$ computations. A discrete Fourier transform (DFT) of Eq. 1 can be performed and the inverse transform allows c to be reexpressed as

$$c_{\alpha,\beta,\gamma} = \frac{1}{N^3} \sum_{o=1}^{N} \sum_{p=1}^{N} \sum_{q=1}^{N} \exp\left[\frac{2\pi i(o\alpha + p\beta + q\gamma)}{N}\right] \cdot C_{o,p,q} \tag{2}$$

where C is the DFT of c, defined by

$$C_{o,p,q} = A^*_{o,p,q} \cdot B_{o,p,q},$$

where B is the DFT of b and A^* is the complex conjugate of the DFT of a. Using a fast Fourier transform algorithm (FFT) reduces the $O(N^6)$ computations to $O(N^3 \bullet \ln[N^3])$. Calculating c for all values of the shift vector on Protein B only accounts for translational degrees of freedom for a given relative rotation. To fully search the 6D space, Protein B must be incrementally rotated through the three Euler angles that define its relative rotation. Therefore the calculation in Eq. 2 is performed $(360 \times 360 \times 180)/D^3$ times, where D is the angle increment.

Another method for finding surface complementarity uses spherical harmonics as the basis functions for describing the protein surfaces, instead of a Cartesian 3D grid, and was introduced by Ritchie and Kemp [2]. In this representation, the natural degrees of freedom are five Euler angles (two for Protein A, three for Protein B) and the center of mass separation of the two proteins. The shape complementarity can be computed via spherical polar Fourier correlations. This method exploits the property of spherical harmonics that they transform among themselves under rotation, which results in a more computationally efficient algorithm than the 3D Cartesian grid FFT method.

The correlation function above only accounts for the surface complementarity and does not account for the chemical nature of the interaction. However, one can use a more complex function that takes into account

electrostatics, van der Waals, and/or desolvation effects, which can be mapped onto the grid as well. The details of these different "scoring functions" are covered in the subsequent sections.

Reduced Search Space Methods

Both the FFT and the spherical polar Fourier method are designed for an exhaustive search of the 6D space defined by the rigid body rotations and translations of one protein about the other fixed protein. In many instances, additional information about the system may be available which can narrow the search space. For example, when the binding site region of one protein is known, a targeted search can be performed about that site, significantly reducing the search space. A rigid body search is still performed in the initial phase, but the reduced space makes methods such as Monte Carlo and genetic algorithms (GA) viable options. In the Monte Carlo methods, an interaction potential is used to calculate an interaction energy; trial moves are accepted or rejected based on the Metropolis criterion. The programs RosettaDock [3] and ICM-DISCO [4] utilize Monte Carlo in their docking procedures. Neither RosettaDock nor ICM-DISCO use a GA, but in other approaches either a traditional GA or variations (e.g., Lamarckian GA) have been used to drive the search for new low-energy conformations in which the "chromosomes" of the GA consist of the relative positions, rotations, and orientations of one or both proteins. For example, GA can be used to move the surface of one protein relative to the other and locate the area of greatest surface complementarity between the two. Search methods using GA have been widely evaluated in several protein–ligand docking programs, e.g., AutoDock and DOCK.

The incorporation of NMR Nuclear Overhauser Effect (NOE) data, which indicates which residues are interacting, makes solving the docking problem much more tractable. Other more ambiguous data, such as chemical shift perturbations, residual dipolar couplings, and mutagenesis (i.e., alanine scanning), can also be incorporated. The program HADDOCK [5] uses these data sets in the form of Ambiguous Interaction Restraints (AIR)s. An AIR restraint is satisfied if an interface residue on one protein interacts with any interface residue on the other protein. The HADDOCK search method generates random orientations and does a rigid body minimization, followed by a torsion angle–simulated annealing protocol that introduces both side chain and backbone flexibility. Cartesian space molecular dynamics are then conducted in explicit solvent. A newer approach toward reducing the search space is the prediction of protein active sites using bioinformatics techniques. While it is still difficult to reliably predict a protein "hot spot," further development of these techniques provides a promising avenue of future study.

Side-Chain Refinement

In some protein–protein complexation events, the interprotein contacts can induce large-scale conformation changes in the individual protein configurations. These complexes are the most difficult to predict through docking methods and require the proteins to be fully flexible, greatly increasing the configurational search space that must be sufficiently sampled, all of which comes at a great computational cost. In most complexes, just small local deviations from the individual protein crystal structures occur. The majority of these small configurational changes occur in the residue side chains, while the protein backbone remains (nearly) rigid. In multistage docking methods, after a favorable binding geometry is determined from rigid docking, the residue side chains can be (re)built and optimized. The side-chain configurations can be limited to a discrete set of states, derived from known protein structures. From this precalculated rotamer library of low-energy states, the global minimum of side-chain configurations is sought. A combinatorial approach can be used to sample all states, or more efficient algorithms can be used. Dead end elimination is one such algorithm, which removes possible rotamer states that are determined as not being present in the global energy minimum state. Full flexibility of the side chains can be introduced after rotamer optimization, via a molecular mechanics method, to sample deviations from the rotamer library configuration.

Incorporation of Protein Flexibility in Protein–Protein Docking

Under physiological solution conditions, it is well known that proteins are dynamic rather than being entirely static and exhibit flexible motions that are thermally accessible within their thermodynamically favorable native-state topology. The range of accessible

motions scales from minor side-chain rearrangements and minor flexible loop backbone motions to more substantial backbone deviations (usually within 1.0–1.5 Å C_a-RMSD) including larger-scale “shear motions” and “hinge motions.” The most significant and complicated rearrangements upon complex formation seem to involve partial unfolding and refolding of flexible segments of the protein (usually termini). Although many protein–protein complexes exhibit minor deviations in the unbound to bound conformation of the component proteins, it appears that a modest number of protein–protein complexes undergo significant rearrangement upon complex formation. Therefore, incorporation of protein flexibility for each protein in a given complex is an important aspect required for accurate predictions of these challenging complexes.

Incorporating protein flexibility into the docking search is a significant computational challenge; many different approaches and algorithms are currently being explored and evaluated. Most successful approaches incorporate some level of backbone flexibility at the level of the global search. This is because even modest backbone rearrangements have been shown to have a significant outcome on the identification of the native protein–protein interface. Therefore, most approaches first generate an ensemble of discrete flexible conformations, use this ensemble for the global search phase, and identify the most likely solutions before moving forward to more detailed model refinements and final scoring of complexes. One way of reducing the search space is to consider only one of the two binding partners to be flexible and only a small ensemble of the more flexible binding partner docking to a representative structure of the other binding partner.

There are numerous approaches to generate an ensemble of structures for ensemble docking that incorporate backbone deviations. The ensemble can be taken from an experimental NMR ensemble or from multiple diverse conformations from X-ray crystallography. In many cases, where there is experimental structural information, there remains insufficient structural diversity in the known structures. If some conformational diversity does exist in experimental structures, it is then possible to use sophisticated tools (e.g., DynDom, HingeFind, FlexProt) to compare the known structure for the identification of flexible loops, as well as shear and hinge motions. It is also possible to use methods like targeted molecular dynamics to generate a continuous trajectory of conformations that connect the known experimental conformations.

In the more generic case, where only one unbound conformational state is known, it is necessary to generate an ensemble of discrete flexible conformations using computational methods that aim to accurately represent the assessable states of the flexible protein. To this end, it is possible to predict flexible segments using molecular framework approach algorithms (FIRST) and hinge-detection algorithms. Following this approach, conformers can then be generated by sampling a limited set of flexible degrees of freedom to reduce the search space. Alternatively, diverse conformations can be generated directly from physical approaches that more fully sample the available degrees of freedom to the entire protein. These approaches include Molecular Dynamics (MD), MD methods that employ advanced conformational sampling techniques, Essential Dynamics (Principle Component Analysis), and variations of Normal Mode Analysis (NMA).

Standard MD techniques are based on describing the all-atom structure of the system with a detailed force field and modeling dynamics by numerical integration of Newton’s equations of motion. MD-simulated annealing conformational searches have been shown to be quite effective in protein–ligand docking (CDOCKER). However, using MD in this way for protein–protein docking is not feasible due to the huge differences in the search-space (geometry of a protein-ligand binding site vs. protein-protein interface) and the number of flexible degrees of freedom. Standard MD methods can explore local minor conformational changes of a protein including side-chain rearrangements and minor backbone relaxations on the picoseconds to nanosecond timescale. However, large-scale changes in protein conformational space occur on longer experimental timescales (microseconds and milliseconds to seconds) and are separated by large energy barriers. Other advanced sampling techniques (simulated annealing, biased methods, torsion angle dynamics, replica exchange) can be coupled with MD to allow for more rapid crossing of energy barriers and sampling alternative low-energy stable conformations. The conformational space sampled by these MD methods can be clustered in Cartesian space to identify low-energy representative conformations.

Alternatively, any given set of generated conformations (such as the conformational space explored by MD) can also be analyzed using Essential Dynamics that is also known as Principle Component Analysis (PCA). In PCA, a square covariance matrix is constructed from the conformational space describing the deviation of each atom coordinate from the average position. When this matrix is diagonalized, the largest eigenvalues represent the "principle components" of the proteins flexibility, where the direction of motion is described by the eigenvector and the amplitude of motion by the eigenvalue. Linear combinations of a subset of the most important eigenvectors can be used to generate an ensemble of "eigenstructures" for docking (CONCORD and Dynamite).

Normal Mode Analysis (NMA) is a method for analytically describing the thermally available deviations from a given equilibrium reference structure within the harmonic approximation. For a given potential energy function describing the system around a minimum energy conformation, a Hessian matrix can be constructed from the mass-weighted second derivatives of the potential energy. When this matrix is diagonalized, the eigenvectors of the matrix are the "normal modes" of the proteins flexibility, where the direction of motion is described by the eigenvector and the amplitude of motion by the eigenvalue. Linear combinations of a subset of the most important eigenvectors can be used to generate "eigenstructures" for docking. Several studies with both normal mode analysis and elastic network normal mode analysis have demonstrated that a few of the low-frequency modes are able to successfully describe experimentally observed large-scale motions of proteins. The set of low-frequency normal modes can also be used for predicting hinge motions (HingeProt) and for estimating the conformational energy penalty that should be associated with a given conformational change. Tama and Sanejouand have pointed out one potential issue which is that in some cases, the normal modes calculated from open conformations of proteins correlate better with observed conformational changes than those calculated from closed and compact conformations [6]. To avoid this problem, May and Zacharias calculate normal modes from the starting structure, generate new initial conformations along deformations of normal mode eigenvectors, and then recalculate the normal modes assuming the new structure is in equilibrium in an iterative fashion [7]. Another potential issue is that conformational changes involving loop movements are associated with high-frequency rather than low-frequency modes, so new approaches aim to explore and identify conformationally relevant "normal modes" following the observations of Abagyan and coworkers [8].

Scoring Functions for Protein–Protein Docking

As most protein–protein docking approaches use multistage hierarchical methods, scoring and ranking of putative conformations also typically occur at different steps, and have different requirements for accuracy and speed. The first step of scoring is usually simultaneous with a global or reduced rigid body search where a specified number of favorable complexes are selected for additional refinement. Following this refinement, the most accurate scoring functions are applied to select the most favorable complexes for the final prediction. If there is insufficient accuracy in the first step of discrimination, then native-like complexes are never refined and are missed in the final step of more accurate scoring of refined complexes. On the other hand, if the first step of scoring is too computationally expensive, it limits the amount of conformational space that can be searched in the initial phases.

In the global rigid body search phase (6D search of translational/rotational space), experience has shown that allowing some steric overlap between the surfaces of the two proteins is necessary for success; thus some minor or significant rearrangement of the binding partners may be required. This is handled differently in geometric complementarity–based searching strategies like geometric hashing depending on how the surfaces match or geometric complementarity is calculated. Although not all of the most common search strategies employ surface-matching/geometric complementarity as the primary scoring metric during the sampling phase, many successful protocols (BIGGER, ClusPro, 3D-Dock, DOT, Molfit, PatchDock, SKE-DOCK, SmoothDock, and ZDOCK) use various surface-matching/geometric complementarity metrics in combination with other metrics such as electrostatics and desolvation to determine which complexes should move into the refinement phase. For molecular mechanics–based force fields, allowing some steric

overlap is usually accomplished by using a "soft" or smoothed force field with reduced vdW repulsion. In small-molecule protein–ligand docking, it is extremely common to use a soft vdW potential early in the conformational search, and then in the refinement procedure introduce a more "hard" or standard Lennard–Jones potential. This practice has also been incorporated in some variations of protein–protein docking (ROTAFIT).

Although it is true that exposed hydrophobic patches of residues on a protein surface often indicate a protein–protein interaction surface, several studies that have analyzed experimental protein–protein interfaces show that there is no one single determining feature for prediction of the correct native-like interface. Although some propensities for certain residues can be detected at a sequence level, there is no single interface property (i.e., total buried surface area, number of hydrogen bonds, hydrophobicity, electrostatic complementarity, predicted desolvation energy, residue side-chain rotamers) that is a strong predictor of the native interface over a variety of protein complexes. For this reason, most successful scoring functions designed for correct identification of native protein–protein interfaces are increasingly complex and sophisticated, and involve a combination of these types of interface properties (e.g., shape complementarity, hydrophobic interactions and electrostatics). In general, there are three generic categories of scoring functions: (1) molecular mechanics, (2) knowledge-based, and (3) empirical scoring functions. Research groups have demonstrated success and improvements in native-like geometry discrimination using various flavors and combinations of all three of these types of scoring functions.

Most molecular mechanics force fields are quite similar in their overall generic form. The total potential energy of the system is the pair-wise sum of van der Waal interactions modeled from the Lennard–Jones potential, electrostatic interactions, and the sum of all the internal degrees of freedom (bonds, angles, and torsions) that describe the molecular geometry of the bonded atoms:

$$V = \sum_{\text{bonds}} k_b(b-b_0) + \sum_{\text{angles}} k_\theta(\theta-\theta_0) + \sum_{\text{dihedrals}} k_\phi[1+\cos(n\phi-\delta)] + \sum_{\text{nonbonded}} \varepsilon_{\text{vdW}}\left[\left(\frac{R_{\min_{ij}}}{r_{ij}}\right)^{12} - \left(\frac{R_{\min_{ij}}}{r_{ij}}\right)^{6}\right] + \frac{q_i q_j}{\varepsilon_{\text{elec}} r_{ij}} \tag{3}$$

The most common molecular mechanics force field in the protein–protein docking field is CHARMM. Several forms of implicit solvation potential energy can also be included in the CHARMM potential, which are based on continuum electrostatics theories including Poisson–Boltzmann, generalized Born, and more simplified representations that are analytical approximations of these. Including these terms can allow for a more rigorous calculation of the desolvation energy upon formation of a protein–protein interface. The electrostatics of a protein–protein interface is a complicated balance between favorable and unfavorable electrostatic complementarity (e.g., H-bonds) and (un)favorable desolvation energy. Vajda and coworkers have determined that at the interface it is more important to put a larger weight on the vdW terms [9].

The field of knowledge-based scoring functions has its origin in the protein-folding field (for the correct identification of native-like folds), and is based on forming statistical potentials from inverse Boltzmann weighting of what is statistically observed in native-like structures. One of the most basic forms of a knowledge-based potential for protein–protein docking is a potential based on the statistical preference of certain residue–residue pairs across the interface. Additional variations of this include a wide range of sophisticated atom–atom contact potentials and also distance-dependent atom pair potentials. One widely used knowledge-based potential of this type is the "atomic contact potential" (ACP) [10] which is an atom–atom extension of the widely used Miyazawa and Jernigan potential [11]. Delisi and coworkers develop the ACP by considering 18 protein atom types and a contact radii of 6.0 Å where the total contact energy of the interface E_c is defined as

$$E_c = \sum_{i=1}^{18}\sum_{j=1}^{18} E_{ij} n_{ij} \tag{4}$$

where E_{ij} is the absolute contact energy between atoms i and j shown in Eq. 5:

$$E_{ij} = e_{ij} + E_{i0} + E_{0j} - E_{00}. \tag{5}$$

The effective contact energies e_{ij} can be written in terms of the observed contact numbers in a database where $\bar{n}_{ij}$ is the number of i–j contacts in the most probable distribution in an unbiased sample:

$$e_{ij} = -\ln\left(\frac{\bar{n}_{ij}\bar{n}_{00}}{\bar{n}_{i0}\bar{n}_{0j}}\right). \tag{6}$$

It has also been possible to express variations of such atomic contact potentials and variations of distance-dependent pair potentials into functional forms that allow their incorporation into FFT methods such as ZDOCK and PIPER. Incorporation of such potentials has been shown to improve the discrimination of native-like geometries over decoys.

Empirical scoring functions attempt to approximate the relative statistical weights from various parameters to optimize performance from regression-based training and validation studies. One of the best examples of these in protein–protein docking is the complicated scoring function employed in RosettaDock. As outlined in Gray et al. [3], the same functional form has different optimized weights during three phases of the algorithm, i.e., packing, minimization, and final discrimination. In the potential for the discrimination phase, there are 11 terms that all have optimized weights (repulsive vdW, attractive vdW, surface area solvation, Gaussian solvent-exclusion, rotamer probability, hydrogen bonding, residue pair probability, electrostatic short-range repulsion, electrostatic short-range attraction, electrostatic long-range repulsion, and electrostatic long-range attraction) so the final version of the functional form is

$$\begin{aligned} \text{score} = {} & W_{\text{vdw-atr}}S_{\text{vdw-atr}} + W_{\text{vdw-rep}}S_{\text{vdw-rep}} + W_{\text{sol}}S_{\text{sol}} \\ & + W_{\text{sasa}}S_{\text{sasa}} + W_{\text{hb}}S_{\text{hb}} + W_{\text{rotamer}}S_{\text{rotamer}} \\ & + W_{\text{pair}}S_{\text{pair}} + W_{\text{elec sr-rep}}S_{\text{elec sr-rep}} \\ & + W_{\text{elec sr-atr}}S_{\text{elec sr-atr}} + W_{\text{elec lr-rep}}S_{\text{elec lr-rep}} \\ & + W_{\text{elec lr-atr}}S_{\text{elec lr-atr}} \end{aligned} \tag{7}$$

In the RosettaDock algorithm, this empirical functional form is accurate enough to successfully repack the side chains on the protein–protein interface during the refinement and discrimination phases. Many other empirical scoring functions are employed at the final discrimination step in other docking approaches. Another example is FastContact that rapidly estimates the vdW, electrostatic, and desolvation components of the free energy using an empirical contact potential for the desolvation contribution.

Critical Assessments of Protein–Protein Docking Methods

The protein–protein docking field has significantly benefited from the Critical Assessment of Predicted Interactions (CAPRI) experiment. CAPRI is a comm unity-wide blind prediction experiment, where experimentalists provide newly solved protein–protein complexes and withhold the coordinates. CAPRI participants are given the 3D coordinates of each unbound subunit and predictions are due 6–8 weeks later. Using available experimental data in the literature (e.g., mutations that may indicate the location of the binding site) is also allowed, so some targets offer assessments of protein–protein docking where some information is known which can limit the search space, while for most of the targets no information is available. Each participating group is allowed to submit 10 top-ranked models of the protein–protein complex. The accuracies of the predictions are categorized as: (1) high, (2) medium, (3) acceptable, and (4) incorrect according to several evaluation metrics that are summarized in Table 1. The three primary metrics are: (1) C_a RMSD of the ligand (smaller protein) from the native complex structure after the superposition of the receptor (larger protein), (2) C_a RMSD of the backbone of the interface residues, and (3) the percentage of native contacts on the interaction interface.

The initial rounds 1–5 of CAPRI occurred from 2001 to 2004 and the results have been published in the literature [12, 13]. The next rounds 6–12 occurred from 2005 to 2007 and these results have also been published [14]. The most recent rounds 13–19 were recently evaluated in a CAPRI meeting held in Barcelona, Spain, in December 2009. Over the 19 rounds of CAPRI so far, 42 targets

Protein Docking. Table 1 CAPRI evaluation metrics for predicted protein–protein complexes

Rank	Ligand (C_α RMSD)	Interface (C_α RMSD)	Interface $f_{\text{Nat. Cont.}}$
High***	≤ 1.0	≤ 1.0	≥ 0.5
Medium**	1.0–5.0	1.0–2.0	≥ 0.3
Acceptable*	5.0–10.0	2.0–4.0	≥ 0.1
Incorrect			< 0.1

have been released and predicted by various groups, although some of the targets were canceled during the evaluation period. Over rounds 1–2, the best performing methods were: ICM-DISCO, SmoothDock, Molfit, 3D-Dock, and DOT [12]. Over rounds 3–5 the best overall performance was from ICM-DICSO, PatchDock, ZDOCK, FT-Dock, RosettaDock, and SmoothDock [13]. Over rounds 6–12, the best overall performance was from ZDOCK, HADDOCK, Molfit, 3D-Dock, ClusProt, RosettaDock, and ICM-DISCO [14]. From the most recent rounds 13–19, which were evaluated at the Barcelona meeting, it was shown that some of the Web servers are also now performing as well as some of the other participating groups. The best performing Web servers were ClusPro and PatchDock over rounds 6–12. The performance of the best performing Web servers that participated in rounds 13–19 is shown in Table 2, along with some of the other most popular Web servers. The improved performance of some of the Web servers is very important to the experimental community, as many groups use these servers to predict likely binding modes of protein complexes of interest. These top-ranked binding modes are then often used by experimental groups to design experiments aimed at validating the complex interface using experimental methods such as site-directed mutagenesis, enzymatic proteolysis, mass spectrometry, crosslinking, and FRET.

The most recent CAPRI rounds have shown that some of the more easy targets that did not involve significant conformational changes could be predicted by many groups and Web servers using a variety of methods. This suggests that there is an emerging consensus among the various docking methods for adequate predictions for the easiest of the test cases. The most challenging targets were clearly those where backbone flexibility was very important, as well as targets where one of the protein complex components needed to be predicted by homology modeling. Another clear lesson is that some limited knowledge from biochemical information, when used to limit the search to a certain space area, dramatically improves predictions for several of the methods. The greatest challenge in the field is clearly the issue of incorporating backbone flexibility into the predictions. One reason why this is challenging is that incorporation of backbone flexibility can further increase the number of false positives by nonnative complexes that score better than the native interface.

Applications of Protein–Protein Docking and Large-Scale Predictions

Due to the importance of protein–protein interactions in understanding the connections and regulations between biochemical pathways, many high-throughput

Protein Docking. Table 2 Protein–protein docking Web servers and performance in CAPRI rounds 13–19 (CAPRI meeting Barcelona Dec 2009)

	Web Servers (CAPRI rounds 13–19)	Total	High	Medium
1	*CLUSPRO* (Vajda group, Boston University, Boston MA) http://cluspro.bu.edu	5	1***	3**
2	*HADDOCK* (Bonvin group, Utrecht University, the Netherlands) http://haddock.chem.uu.nl	4	1***	1**
3	*GRAMM-X* (Vakser group, U. of Kansas, Lawrence, KS) http://vakser.bioinformatics.ku.edu/resources/gramm/grammx	2	2***	-
4	*SKE-DOCK* (Takeda-Shitaka group, Kitasato U., Japan) http://www.pharm.kitasato-u.ac.jp/bmd/files/SKE_DOCK.html	2	1***	-
5	*PatchDock, FiberDock, FireDock* (Wofson group, Tel Aviv U.) http://bioinfo3d.cs.tau.ac.il/PatchDock/	1	1***	-
6	*TOP DOWN* (Nakamura group, Inst. for Prot. Res. Osaka Japan) http://pdbjs6.pdbj.org/TopDown/	1	1**	-
	Other Web Servers:			
	ZDOCK (Z. Weng group U. of Massachusetts, Worcester, MA) http://zdock.bu.edu/			
	RossettaDock (J. Gray group John Hopkins U., Baltimore, MD) http://rosettadock.graylab.jhu.edu/			
	HEX (Ritchie group, Nancy, France) http://www.loria.fr/~ritchied/hex/			
	FiberDock (Wofson group, Tel Aviv U.) http://bioinfo3d.cs.tau.ac.il/FiberDock			
	3D-Garden (Sternberg group, Imperial College, London, UK) http://www.sbg.bio.ic.ac.uk/~3dgarden/			

proteomics efforts have also been aimed at creating large-scale comprehensive catalogues (an *Interactome*) of experimentally verified protein–protein interactions. Significant progress has been made in this area in the model organism yeast, and similar efforts are underway in many other areas, including important pathogenic organisms and human cancer cell types. Experimental studies of these protein–protein interaction networks in cells have shown that some proteins may interact with up to 100 other proteins. Despite these ongoing experimental efforts, it is clear that structural genomics efforts alone (which systematically determine the experimental 3D structures of high interest proteins) will not be able to determine the structures of all of the detected and important protein–protein interactions and reliable computational prediction of these complexes will be more and more important in the future. To successfully perform large-scale protein–protein docking predictions, it will be necessary to employ homology-based modeling methods. The necessity to rely on homology models as the initial input to protein–protein docking, increases the need for improved accuracy both for the initial homology modeling steps and for the reliable incorporation of flexibility in the protein–protein docking steps.

The European 3D-Repertoire project (http://www.3drepertoire.org) aims to experimentally determine the 3D structures of some 100 high–interest protein–protein complexes from yeast that are amenable to X-ray crystallography, NMR spectroscopy, and cryo-electron microscopy methods. Still a large number of the total number of yeast protein–protein interactions (10,000 high confidence complexes from a total of 6,000 genes) will be predicted by computational methods. It is also true that for some fraction of the protein–protein targets, the entire complex itself including the interface may be modeled reliably based on homology to other known complexes that are sufficiently similar in sequence. However, for the vast majority of the targets, this approach is not possible and it becomes necessary to perform protein–protein docking of homology models for each individual component of the complex (given that sufficient templates are available for each component of the complex). The first publication describing this effort came out in 2009, describing the prediction of 3,000 protein–protein complexes starting from 217 experimental structures and 1,023 homology models [15]. The authors used both ZDOCK 3.0 along with the pyDOCK scoring scheme and provided results for both a widely used protein–protein docking benchmark dataset [15]. Despite the fact that the accuracy of the benchmark is only in the range of 11–42% (depending on if the top 1, 3, or 10 complexes are considered), these predictions of 3,000 complexes from the yeast *Interactome* are still a very exciting preliminary step toward more large-scale prediction efforts like this in the future. These preliminary efforts highlight the ongoing need for improved accuracy in protein–protein docking methods.

Bibliographic Notes and Further Reading

More on genetic algorithms and protein–protein docking: (1) Morris GM, Goodsell DS, Halliday RS, Huey R, Hart WE, Belew RK, Olson AJ (1998) Automated docking using a Lamarckian genetic algorithm and an empirical binding free energy function. J Comput Chem 19(14):1639–1662. (2) Taylor JS, Burnett RM (2000) DARWIN: A program for docking flexible molecules. Proteins 41:173–191. (3) Gardiner EJ, Willett P, Artymiuk PJ (2001) Protein docking using a genetic algorithm. Proteins 44:44–56.

Introduction in molecular dynamics: (1) Leach A (2001) Molecular modelling: principles and applications, 2nd edn. Prentice Hall, Englewood Cliffs. (2) Rapaport DC (2004) The art of dynamics simulation, 2nd edn. Cambridge University Press, Cambridge.

Key recent reviews on protein docking: (1) Zacharias M (2010) Accounting for conformational changes during protein-protein docking. Curr Opin Struct Biol 20:180–186. (2) Vajda S, Kozakov D (2009) Convergence and combination of methods in protein-protein docking. Curr Opin Struct Biol 19:164–170. (3) Andrusier N, Mashinach E, Nussinov R, Wolfson HJ (2008) Principles of flexible protein-protein docking. Proteins 73:271–289.

Bibliography

1. Katchalski-Katzir E, Shariv I, Eisenstein M, Friesem AA, Aflalo C, Vakser IA (1992) Molecular surface recognition: determination of geometric fit between proteins and their ligands by correlation techniques. Proc Natl Acad Sci 89:2195–2199
2. Ritchie DW, Kemp GJL (2000) Protein docking using spherical polar Fourier correlations. Proteins 39:178–194

3. Gray JJ, Moughon S, Wang C, Schueler-Furman O, Kuhlman B, Rohl CA, Baker D (2003) Protein-protein docking with simultaneous optimization of rigid-body displacement and side-chain conformations. J Mol Biol 331:281–299
4. Fernandez-Recio J, Totrov M, Abagyan R (2003) ICM-DISCO Docking by global energy optimization with fully flexible side-chains. Proteins 52:113–117
5. Dominguez C, Boelens R, Bonvin MJJ (2003) HADDOCK: a protein-protein docking approach based on biochemical or biophysical information. J Am Chem Soc 125:1731–1737
6. Tama F, Sanejouand YH (2001) Conformational change of proteins arising from normal mode calculations. Protein Eng 14:1–6
7. May A, Zacharias M (2008) Energy minimization in low-frequency normal modes to efficiently allow for global flexibility during systematic protein-protein docking. Proteins 70:794–809
8. Cavasotto CN, Kovacs JA, Abagyan RA (2005) Representing receptor flexibility in ligand docking through relevant normal modes. J Am Chem Soc 127:9632–9640
9. Camacho CJ, Vajda S (2001) Protein docking along smooth association pathways. Proc Natl Acad Sci 98:10636–10641
10. Zhang C, Vasmatzis G, Cornette JL, DeLisi C (1999) Free energy landscapes of encounter complexes in protein-protein association. Biophys J 76(3):1166–1178
11. Miyazawa S, Jernigan RL (1985) Estimation of effective interresidue contact energies from protein crystal structures: quasi-chemical approximation. Macromolecules 18:534–552
12. Mendez R, Leplae R, De Maria L, Wodak SJ (2003) Assessment of blind predictions of protein–protein interactions: current status of docking methods. Proteins 52:51–67
13. Mendez R, Leplae R, Lensink MF, Wodak SJ (2005) Assessment of CAPRI predictions in rounds 3–5 shows progress in docking procedures. Proteins 60:150–169
14. Lensink MF, Wodak SJ, Mendez R (2007) Docking and scoring protein complexes: CAPRI 3rd edition. Proteins 69:704–718
15. Mosca R, Pons C, Fernandez-Recio J, Aloy P (2009) Pushing structural information into the yeast interactome by high-throughput protein docking experiments. PLOS Comp Biol 5(8):1–13

Pthreads (POSIX Threads)

►POSIX Threads (Pthreads)

PVM (Parallel Virtual Machine)

Al Geist
Oak Ridge National Laboratory, Oak Ridge, TN, USA

Synonyms

Message passing

Definition

Parallel Virtual Machine (PVM) is a software package that permits a heterogeneous collection of Unix and/or Windows computers hooked together by a network to be used as a single large parallel computer. Thus large computational problems can be solved more cost-effectively by using the aggregate power and memory of many computers. The software is very robust and portable. The source, which is available free from the PVM website, has been compiled on everything from laptops to CRAYs. Because PVM supports fault tolerance and dynamic adaptability, hundreds of sites around the world continue using PVM to solve important scientific, industrial, and medical problems. PVM has also been popular as an educational tool to teach parallel programming because of its simple intuitive user interface.

Discussion

Introduction

PVM (Parallel Virtual Machine) is often lumped together with the Message Passing Interface (MPI) standard, because PVM was the precursor to MPI and the PVM developers, most notably, Jack Dongarra started and lead the initial MPI forum that defined the MPI 1.0 standard. But message passing is only a small part of the PVM package. PVM is an integrated set of software tools and libraries that emulates a general-purpose, fault-tolerant, heterogeneous parallel computing framework on interconnected computers of varied architectures and operating systems. As a demonstration at the 1992 Supercomputing conference, PVM was used to hook together all the supercomputers on the exhibit floor into one giant parallel virtual machine. PVM is a byproduct of the heterogeneous-distributed computing research project at Oak Ridge National Laboratory, Emory University, and the University of Tennessee. The programming model upon which PVM is based differs substantially from MPI. The PVM programming model has:

- *Dynamic, user configured, host pool*. The set of computers that make up the virtual machine may themselves be parallel or serial computers and PVM can add or delete machines while programs are running. For example, to add more computing power, or delete hosts that have failed.

P

- *PVM tasks are dynamic and asynchronous*. The tasks that make up a parallel program do not have to start all together. New tasks can be spawned off in the middle of a computation, tasks can be killed and programs started outside the parallel virtual machine can join PVM and become another task in the dynamic set of tasks that make up the parallel program.
- *Dynamic groups*. Users can define any number of subgroups of PVM tasks that can then perform collective operations such as broadcast data to all members of the group or set a barrier. The members of the groups can change dynamically during a computation (an important feature for fault tolerance). PVM provides calls to manage dynamic groups.
- *Translucent access to hardware*. Application programs may view the hardware environment as an attributeless collection of hosts or may choose to exploit the capabilities of specific machines in the host pool by positioning certain computational tasks on the most appropriate computers. PVM provides calls to convey host attributes to a running application.
- *Transparent heterogeneity support*. A single PVM job can run across a collection of Linux, Windows, and Unix computers, that themselves can be multicore, serial or parallel computers. PVM transparently takes care when transferring data between computers that have different data representations, for example, big Endian and little Endian and different word lengths.
- *Explicit message-passing*. Data is transferred between PVM tasks using a small number of point-to-point and collective message-passing calls. When a sender or receiver is detected to be dead, PVM automatically deletes it from the virtual machine and notifies the application.

The PVM system is composed of two parts. The first part is a daemon, called *pvmd*, that resides on all computers making up the virtual machine. PVM is designed so that anyone with a valid login can install this daemon on a machine. When a user wishes to run a PVM application, he first creates a parallel virtual machine by starting PVM with a list of hosts upon which he has already installed pvmd. Multiple users can configure overlapping virtual machines and each user can execute several PVM applications simultaneously on the same parallel virtual machine. The second part of the system is a library of PVM interface routines that is linked with the users PVM application. The library contains routines for resource management (modifying the pool of hosts), process control (spawning and killing tasks), managing dynamic task groups, passing messages, and fault tolerance. The basic PVM system supports C, C++, and Fortran languages, but the PVM user community has extended PVM so it can be used with Java, Python, Perl, Lisp, S-lang, R, tk/tcl, and IDL languages.

Resource Management

The underlying set of hosts that make up a particular PVM instantiation is a dynamic resource. Typically, a host list is just fed into the PVM startup routine to create a custom parallel virtual machine tailored to the problem being solved, but PVM provides much more resource management flexibility. PVM provides pvm_addhosts and pvm_delhosts routines that allow any PVM task to add or delete a set of hosts in the parallel virtual machine at any time. These routines are sometimes used to set up a virtual machine, but more often they are used to increase the flexibility and fault tolerance of a large scientific simulation. These routines allow a simulation to increase the available computing power (adding hosts) if it determines that the problem is getting harder to solve, then to shrink back in size when the problem does not need all the extra resources. Another use of these routines is to increase the fault tolerance of a simulation by having it detect the failure of a host and replacing the failed host on-the-fly with pvm_addhosts.

PVM provides routines to return information about the present set of hosts in the parallel virtual machine. The information includes the host names, their architecture types (from which the individual data representations can be determined), and their relative CPU speed (which can be useful to an intelligent load balancing program). A PVM task can also find out what host it is running on and use this architectural awareness to run algorithms tuned for that architecture.

The PVM resource management system also includes a way to set/get various options that determines how the underlying PVM system works. For example, if the user prefers speed to flexibility, the communication can be set so that it is as fast as MPI. Similarly, if the user

knows that the task groups are going to be static then the underlying system can be told and it will utilize local cache information to speed up all the group routines. If tracing or debugging are desired there are options in PVM to set which tasks to trace, and where to send the trace output. PVM has over a dozen options that can be set and the list is extensible to add additional system hints or options in the future.

Process Control

PVM provides routines to dynamically spawn and kill tasks and allows tasks to join a running PVM system and to exit a PVM system. The pvm_spawn routine is the central function in the PVM process control. The pvm_spawn routine starts up one or more copies of an executable file. The executable is usually a PVM program, but it does not have to be. Spawn can launch any command or executable program that the user has permission to execute on the hosts. The spawn routine can pass an argument list to the launched executable(s) and can specify where the tasks should be started. If unspecified, then PVM will spread the tasks evenly across the entire virtual machine. Pvm_spawn can be called multiple times to start many different types of tasks, which is needed to support functional parallelism. It can be called anytime during program execution to add additional computation tasks, or to replace a failed task, or to add a different type of task in order to create a program that adapts on-the-fly to the problem being solved. Besides being able to place tasks on specific hosts, PVM has a plug-in interface that allows users to plug-in their own task placement/load balancing routines into the parallel virtual machine. The flexibility of the pvm_spawn routine allows PVM to support all types of programming methodologies, not just SPMD.

To complement the ability to create new tasks dynamically, PVM provides a routine that allows any task to kill any other task, and a routine to allow a task to exit the PVM system, i.e., to cleanly kill itself. The key to these routines is to be sure that all pending parallel virtual machine operations that involve the particular tasks are taken care of before they are killed.

The PVM process control system has a routine that returns information about the tasks running on PVM at any given moment. The routine, called pvm_tasks, has three options: it allows a user's PVM program to determine information about all tasks in the parallel virtual machine, information about all tasks running on a specific host (useful if there is a fault or a load balance issue), or information about a particular task.

Message Passing

The PVM communication model provides point-to-point, and collective message-passing routines. It also provides the concept of persistent messages and message handlers. The number of communication routines is quite small compared to the MPI standard, which has over 300 functions. For point-to-point communication PVM only provides asynchronous blocking send, asynchronous blocking receive, and nonblocking receive functions. There is also a nonblocking probe routine to check if a particular message has arrived. For collective communication PVM provides multicast to a set of tasks, broadcast to a user-defined group of tasks, reduce operation across a group, i.e., global max, min, sum, etc., and barrier across a group.

There is a four-step process in sending a PVM message. First the message is "packed." Each message can contain an arbitrary structure of data types and arrays. For example, a message may have an integer defining how long a vector is, a vector of integers defining the indices of a sparse array, and a vector of double-precision floating point numbers. Each of these data types could be packed into a single message. In the second step the message is sent to its destination. In the third step the message is received by one or more PVM tasks. In the fourth step the message is unpacked. PVM supplies a pack and unpack routine for every data type supplied by computer manufacturers.

Persistent messages were added to the PVM interface in version 3.4. Persistent messages are packed as usual with an arbitrarily complex or simple data structure, but instead of being sent to a destination, they are stored and retrieved by "name." The sender of the message can specify if other tasks are allowed to update the message, or delete it. The sender also specifies if the message should persist beyond the sender exiting PVM or if it should be cleaned up on exit. Persistent messages have many powerful uses in the dynamic, fault-tolerant environment of PVM's programming model. It allows a message to be sent to a task that does not exist yet. An example of this use is rendezvous – the first task leaves a persistent message defining where it

can be found and how to attach, another example of this use case is in-memory checkpointing. The feature also allows a message to be stored persistently and retrieved by a set of tasks that will not be defined until sometime in the future. An example of this is a fault in the system affects a set of tasks and they need the stored information to recover. Persistent messages provide a distributed information database for dynamic programs inside PVM. For example, a monitoring or computational steering application can attach to a PVM program and leave information in a named space without having to know anything about the tasks in the PVM program, which will retrieve and act on the information. The concept is similar to tuple space used by the Linda system [1]. Persistent messages are implemented in PVM 3.4 using only four additional routines: pvm_putinfo(), pvm_recvinfo(), pvm_delinfo() to delete a persistent message, and lastly, pvm_getmboxinfo() that allows a user to search the persistent message namespace with a regular expression.

User-defined message handlers were also added to PVM in version 3.4. There are no restrictions on the handler function that can be set up. The handler is triggered whenever a message arrives matching the specified context, message tag, and source. This feature has many uses. For examples, it allows users to define new control features inside a parallel virtual machine and it provides the ability to implement active messages to increase communication performance.

Dynamic Task Groups

The dynamic process group functions are built on top of the core PVM routines. There was some debate about how groups should be handled in PVM. The issues include efficiency, fault tolerance, and robustness. There are tradeoffs between static versus dynamic groups and tradeoffs if use of the function calls is restricted to only tasks in a group. In keeping with the PVM philosophy, the group functions are designed to be very general and transparent to the user, at some cost in efficiency. Any PVM task can join or leave any group at any time without having to inform any other task in the affected groups. Tasks can broadcast messages to groups of which they are not members. Tasks can be members of multiple groups. In general, any PVM task can call any of the group functions at any time with only two exceptions: pvm_lvgroup() and pvm_barrier() which by their nature require the calling task to be a member of the specified group. To boost performance, PVM provides a user function to freeze a group, which allows the underlying PVM system to treat that group as static from that point forward and exploiting all the efficiency gains afforded by having a static group structure.

Dynamic group functions in PVM include joining a group, leaving a group, getting information about a group such as size and members, broadcasting to all members of a group, doing a reduce across all members of a group with predefined or arbitrary functions, and lastly setting a barrier across a group where tasks in the group wait until all tasks in the group have called the barrier routine.

Fault Tolerance

The parallel virtual machine constantly monitors itself even when no PVM applications are running. If a fault is detected the set of pvmd coordinate with each other and automatically reconfigure the parallel virtual machine to remove the fault and keep running. The PVM interface also has a notify function that allows a running PVM task to request to be notified of changes in the system including where and what type of fault was detected. Typical notifications are for task exit, host deletion, and a host being added. Note that the latter notification is useful during a recovery phase when replacement resources are being brought into the virtual machine. A fault-tolerant PVM application has many choices about how to utilize the notification function in PVM. The application may have all the tasks notified if a fault occurs (SPMD style), it could have one specially written task dedicated to monitoring and recovery of the application if a fault occurs anywhere (MIMD). It could have a set of tasks that get spawned dynamically with a recovery expertise specific to the notification. It could have a set of monitoring and recovery tasks that back each other up. The flexibility of PVM and its dynamic programming model make it very attractive to groups who need fault-tolerant applications. One real life example – PVM was used to create a monitoring system across a hospital's operating rooms with information from individual patient monitors being fed to a central observation/control room, the system needed to be tolerant as patients were constantly being plugged and unplugged from various resources as they went from prep to operating room to recovery room, etc.

The PVM notification feature is not limited to fault recovery. It can also be used by load-balancing programs that increase and decrease the host pool to meet the changing load of the application over time. It can be used by logging and accounting tasks to keep track of system usage by PVM users. As with many of the PVM functions they are design to provide the maximum use and flexibility to the user.

Related Entries

►Broadcast
►Clusters
►Collective Communication
►Distributed-Memory Multiprocessor
►Fault Tolerance
►MPI (Message Passing Interface)
►Parallel Computing

Bibliographic Notes and Further Reading

The PVM project began in the summer of 1989 at Oak Ridge National Laboratory when Vaidy Sunderam, a visiting faculty from Emory University, and Al Geist designed and built the prototype PVM 1.0 to explore the concept of heterogeneous parallel computing. This prototype was only used internally at the lab and was not released. Jack Dongarra joined the project in 1990 and saw the value of PVM for the larger community. A robust and hardened version of the prototype was written at the University of Tennessee and publicly released as PVM 2.0 in March 1991. During the next 2 years PVM rapidly grew in popularity. After user feedback and a number of evolutionary changes (PVM 2.1–2.4), a complete rewrite was undertaken, and PVM 3.0 was released in February 1993. PVM 3.0 continued to evolve and incorporate new features and capabilities to meet the needs of the exponentially growing number of users and applications. In 1999, 10 years after its inception, the PVM feature set was frozen at PVM 3.4. Since then PVM has continued to be supported and new versions released to port to new architectures and operating systems, for example, in 2009, 20 years after PVM started, PVM 3.4.6 was released and distributed freely as has every other version of PVM. For further reading the PVM website (http://www.csm.ornl.gov) contains tutorials, tools, and information about the PVM development team. For the past 15 years there has been an annual European conference dedicated to advances to PVM and MPI. The topics and papers from all these conferences can be found at http://pvmmpi08.ucd.ie/previous.

Bibliography

1. Bjornson R, Carriero N, Gelernter D, Leichter J (January 1988) Linda, the portable parallel. Research Report Yale/DCS/RR-520
2. Geist A, Beguelin A, Dongarra J, Jiang W, Manchek R, Sunderam V (1994) PVM: parallel virtual machine. MIT Press, Cambridge, Massachusetts
3. Snir M, Otto S, Huss-Lederman S, Walker D, Dongarra J (1996) MPI: the complete reference. MIT Press, Cambridge, Massachusetts