Chapter 10
Agent-Supported Programming of Multicore Computing Systems

**Sabri Pllana, Siegfried Benkner,
Eduard Mehofer, Lasse Natvig, and Fatos Xhafa**

Summary In this chapter we argue that an intelligent program development environment that proactively supports the user helps a mainstream programmer to overcome the difficulties of programming multicore computing systems. We propose a programming environment based on intelligent software agents that enables users to work at a high level of abstraction while automating low-level implementation activities. The programming environment supports program composition in a model-driven development fashion using parallel building blocks and proactively assists the user during major phases of program development and performance tuning. We highlight the potential benefits of using such a programming environment with usage scenarios. An experiment with a parallel building block on a Sun UltraSPARC T2 Plus processor shows how the system may assist the programmer in achieving performance improvements.

S. Pllana (✉) · S. Benkner · E. Mehofer
Department of Scientific Computing, University of Vienna, Nordbergstrasse 15, 1090 Vienna, Austria
e-mail: pllana@par.univie.ac.at

S. Benkner
e-mail: sigi@par.univie.ac.at

E. Mehofer
e-mail: mehofer@par.univie.ac.at

L. Natvig
Department of Computer and Information Science, NTNU, Sem Saelands vei 9, 7491 Trondheim, Norway
e-mail: Lasse.Natvig@idi.ntnu.no

F. Xhafa
Department of Computer Science and Information Systems, Birkbeck, University of London, London, UK
e-mail: fatos@dcs.bbk.ac.uk

10.1 Introduction

While multicore processors alleviate several problems that are related to single-core processors, known as *memory wall*, *power wall*, or *instruction-level parallelism wall,* they raise the issue of the *programmability wall.* On the one hand, program development for multicore processors, especially for heterogeneous multicore processors, is significantly more complex than for single-core processors. On the other hand, programmers have been traditionally trained for the development of sequential programs, and only a small percentage of them have experience with parallel programming.

Additionally, there is a portability problem. In the past programmers could trust that compilers succeeded to pass the increased computing power of next processor generations without high porting effort. This was due to relatively homogeneous processor designs even from different hardware vendors with instruction level parallelism (ILP) supported at hardware level. The architectural change to multicore processors, however, affects the programmer in several ways. On the one hand, thread level parallelism (TLP) must be exploited effectively and efficiently. In general, this cannot be done automatically by a compilation system but requires assistance by the programmer. On the other hand, multicore architectures differ significantly requiring that applications must be adapted to the various platforms.

While in the past only a relatively small group of programmers interested in High Performance Computing (HPC) was concerned with the parallel programming issues, the situation has changed dramatically with the appearance of multicore processors on commonly used computing systems. Traditionally parallel programs in HPC community have been developed by *heroic programmers*[1] using a simple text editor as programming environment, programming at a low-level of abstraction, and doing manual performance optimization. It is expected that with the pervasiveness of multicore processors parallel programming will become mainstream, but it cannot be expected that a mainstream programmer will like to become an HPC hero.

In this chapter we argue that the programming productivity of multicore[2] systems is increased if an intelligent programming environment would be available that (1) enables the programmer to work during the process of program development at a higher level of abstraction using domain-specific modeling languages in a model-driven development fashion and (2) provides context-specific knowledge and performs iterative time-consuming tasks involved in program development in a semi automatic/autonomic manner (for instance, performance tuning). We propose a parallel programming methodology that combines model-driven and agent-supported program development with the use of high-level parallel building blocks. The goal is to increase programming productivity without restricting flexibility and creativity, allowing the programmer to fully use his/her intellectual capacity for software

[1]Andrea: “Unhappy is the land that breeds no hero." Galileo: “No, Andrea: Unhappy is the land that needs a hero." – Bertolt Brecht in *Life of Galileo*.

[2]Although some authors have introduced the term *many-core* to denote multicore systems with many cores (i.e., 100 or more), we will stick to the more established term multicore. We do not see a need to make a distinction between multi and many.

design at model-level. Although software development is considered to be an art, we anticipate that there are many implementation activities that can be performed more automatically/autonomically.

The rest of this chapter is organized as follows. An overview of the recent developments in parallel computing systems is given in Sect. 10.2. Section 10.3 describes our vision for programming of multicore computing systems. We illustrate our approach experimentally in Sect. 10.4. Section 10.5 reviews the state-of-the-art in programming multicore computing systems. We conclude the paper with a summary and future work in Sect. 10.6.

10.2 Recent Developments in Parallel Computing Systems

In this section we provide an overview of the recent developments in parallel computing systems focusing on (1) parallel and distributed programming, (2) compilation techniques, and (3) multicore architectures.

10.2.1 Parallel and Distributed Programming

The dominating programming paradigm for parallel systems is based upon standard sequential programming languages, augmented with message passing constructs. In particular the standardized Message-Passing Interface (MPI) is widely used for parallel programming. In this low-level model the user has to deal with all aspects of parallelization, distribution of data and work to processors, and communication and synchronization by means of explicit message passing operations. This leads to high cost for software and error-prone programs that are difficult to write, reuse, and maintain. On smaller-scale Symmetric Multiprocessing (SMP) systems, the use of low-level multithreading libraries, such as POSIX threads, faces similar problems.

Despite significant research efforts, automatic parallelization, i.e., taking a serial program written in a mainstream language and automatically generating an executable program capable of taking advantage of parallel hardware, has not been successful, either for shared-memory or distributed-memory systems, and remains an elusive goal. Data-parallel languages such as High Performance Fortran (High Performance Fortran Forum 1997) provide high-level support for controlling locality by associating distributions with arrays while delegating the generation of explicit message passing code to the compiler. Despite some successes, these languages have not been broadly accepted. OpenMP raises the level of abstraction for multithreaded programming but lacks constructs for controlling data locality and is thus constrained to small-scale, homogeneous shared-memory systems. Partitioned Global Address Space (PGAS) languages like Co-Array Fortran (Numerich and Reid 1998), Unified Parallel C (UPC Consortium 2005), and Titanium (Yelick et al. 1998) have not succeeded in breaking the dominance of low-level message-passing and thread programming either. More recent efforts in the context of the US

High Productivity Computing Systems (HPCS) program, characterized by putting an increased focus on programmability and not just performance, have resulted in the definition of new languages including Fortress (Allen et al. 2008), Chapel (Cray 2008), and X10 (Charles et al. 2006) that address the challenges of programming large-scale (PetaFlop/s) systems. So far, however, none of these languages is seeing user-uptake beyond a rather limited research community.

With the emergence of multicore computing systems consisting of hundreds of processor cores in the near future, the challenge of parallel programming will not be restricted to the HPC community any more but will extend to the broad software industry. Single-chip multicore systems will provide performance levels accommodating applications previously only available on clusters and parallel systems and will give rise to new exciting applications in the embedded and mobile computing domains. However, since emerging heterogeneous multi-core systems provide different types of cores including general-purpose cores, GPU-like cores and specialized accelerators, the challenges of efficient programming and of achieving portability across different architectures and architecture generations will be aggravated compared to traditional parallel systems.

A variety of technologies and tools for programming heterogeneous multicore architectures have been made available recently by hardware vendors including TBB (Threading Building Blocks) (Intel Corp. 2009), CUDA (NVIDIA Corp. CUDA Zone 2009), Cell SDK (IBM 2008), and others. All these technologies are characterized by an extremely low level of abstraction, forcing programmers to take into account a myriad of architecture details, usually beyond the capabilities of average users. Programs relying on these technologies are not portable to other multicore architectures. Although recently with OpenCL (Khronos OpenCL Working Group 2009) and MCAPI two proposals for standardizing parallel programming of heterogeneous multicore systems have been announced, these standards primarily address portability issues but are still at a very low-level of abstraction.

Some of the programming challenges that are posed by emerging heterogeneous manycore architectures are similar to those faced in programming large-scale distributed computing systems (Hall et al. 2008), which are often referred to as Grid (Foster and Kesselman 2003). In the context of Grid computing systems, application developer usually should deal with diversity of programming models and languages, computation partitioning, data flow and dependencies among the application components, legacy code and third-party software components, and diversity of computational resources. Furthermore, application components are usually executed by heterogeneous computational resources that are not known prior to execution. Usually, Grid applications are expressed as workflows, and a workflow planning/optimization system maps workflow components to available resources for execution (Taylor et al. 2006).

10.2.2 Compilation Techniques

Heterogeneous multicore parallel architectures are typically shipped with ANSI C compilers for each type of core, and by default it is the programmer's responsibility to write separate programs for each core, plus glue code so that these programs can interact. Advanced compilation techniques aiming to avoid this manual effort tend to be tied into parallel programming systems, where language extensions are used to indicate where to parallelize and to demarcate code into portions for which distinct compilation strategies are appropriate.

Sequoia (Fatahalian et al. 2006) (Stanford University) provides a high-level abstraction for distributing tasks over multiprocessor systems with hierarchical memory, implemented for the Cell BE processor and for multicore x86 clusters. Efficient scheduling of data-movement is eased by requiring the programmer to specify a priori the working set required by a parallel task. Distribution of data across the memory hierarchy is made efficient via a sequence of compiler optimizations (Knight et al. 2009) and via so-called tunable parameters which may be adjusted by the user. A similar mechanism for specifying data usage upfront is used by CellSs (Barcelona Supercomputer Center) (Perez et al. 2007). PetaBricks provides an autotuning compiler and a programming language that can express multiple algorithms for solving a specific problem and exposes algorithmic choices to compiler (Ansel et al. 2009).

OpenMP (Chandra et al. 2000) has been applied to heterogeneous multicore with an implementation for the Cell processor. The IBM XLC compiler (IBM XL C/C++ 2009) implements a large part of the standard, using a sophisticated suite of optimizations targeting both the PPE and SPE cores (Eichenberger et al. 2006). The Codeplay Sieve C++ language (Lindley 2007) uses the concept of delayed side-effects to ease dependence analysis, making automatic parallelization of C++ more tractable for C++ software. The Sieve system exploits parallelism via speculative execution and provides mechanisms to support common parallel patterns (Donaldson et al. 2007). Sieve C++ has been shown to be portable across homogeneous and heterogeneous architectures, with implementations for multicore x86, the Ageia PhysX (AGEIA Technologies 2009) accelerator card, and the Cell processor.

Specifying data movement explicitly via intent qualifiers, as in Sequoia and CellSs, is suitable for HPC applications over regular data sets but comes at the expense of expressiveness. On the other hand, while programming architectures like Cell via OpenMP or Sieve C++ is more flexible, since the data set required by a task is implicit in the way data is manipulated, programs in these languages are harder to optimize. Recent research into decoupled access/execute specifications (Howes et al. 2009) aims to provide the best of both worlds, by decoupling the execution of a kernel from its access pattern, via programmer-specified access functions. Initial experiments with the access/execute model involve predefined transformations based on common data access patterns; in principle these transformations can be automated via compiler support based on the polyhedral model (Pouchet et al. 2007).

Recent advances in programmable GPU architectures have led to widespread interest in the use of GPUs for scientific programming. Programming GPUs for scientific applications (Owens et al. 2007) has usually been performed using graphics language such as OpenGL (Shreiner et al. 2005) and more recently using stream computing languages such as CUDA (NVIDIA Corp. CUDA Zone 2009) and RapidMind (RapidMind corp. 2009). More recently, approaches based on familiar high-level languages such as C and Fortran have been proposed (Bodin and Bihan 2009). These approaches are directives based, either new ones (Bodin and Bihan 2009) or extension of the OpenMP standard (Chandra et al. 2000). Directives specify computation to be offloaded on the GPU. Parallel loop nests are translated into one of the target GPU-specific programming language. These approaches are very new, and there are still many issues related to performance tuning. These high-level approaches are the most promising for heterogeneous multicore since they help to avoid multiprogramming and to allow the maintenance of a unique source code.

10.2.3 Multi-Core Architectures

For the last 30 years, the makers of General Purpose (GP) CPUs have leveraged continuous improvements in silicon process technology along two axes: the ever decreasing feature size (as described by Moore's Law of 1965) allowed building ever more complex logic into their CPUs, and the transistors could be driven at ever lower voltages and higher frequencies. The former has led to the dominance of just a few advanced highly pipelined, multiscalar processor architectures with out of order execution capability; the latter allowed one to raise clock frequency and thereby CPU performance up to about 3 GHz. Application software developers profited enormously from both trends, since performance improvements were expected to happen automatically at the cost of at most a recompilation, and there was no need to port and optimize applications to many different architectures.

Since about 2001, two significant trends are reshaping the whole computer ecosystem: the traditional CPU evolution did hit the now proverbial "power wall," meaning that further increases in clock speed could only be achieved by dissipating disproportionate amounts of power and were therefore no longer feasible, and special purpose architectures start to rival GP CPUs in the performance/power and performance/cost metrics. Intel's and AMD's answer to the "power wall" are the current line of multicore CPUs that derive their performance from up to eight complex, independent execution units (cores), and no longer from increases in clock frequency.

The need for improvements in power and cost effectiveness is also driving a renaissance of architectures that are tailored to special computational models and use massive parallelism to surpass GP CPUs for these—prime examples are NVIDIA's or AMD's GPUs that combine hundreds of very simple compute units into a very powerful SIMD parallel system, and FPGAs that can be field programmed to perform complex data transformations such as en/decryption and media format conversions at extremely high speeds. These systems need a GP host processor to run the OS and most applications and are coupled to it by a bus interconnect.

Today, explicitly parallel multicore CPUs have taken over the market (90% of all Intel CPUs sold in 2008 had multiple cores), and novel, massively parallel accelerators are making significant inroads. This is not restricted to the high-performance segment—laptop computers offload part of the GUI processing to their GPUs (Apple Macbooks), and low-power parallel accelerators are being used for embedded systems (e.g., for media format conversions or software defined radio). In effect, the era of heterogeneous multicore platforms is already on us.

Today's systems exhibit heterogeneity mainly between the host CPU(s) and the accelerator(s): instruction set and performance characteristics are very much different, and there is usually no shared memory between the components. A typical example is a workstation that combines two Intel Nehalem CPUs with one to four NVIDIA Tesla accelerators. Applications need to be aware of the different capabilities of the CPU vs. the GPU cores, must use different methods to write the respective parallel code components, and have to explicitly manage the data transfer between CPUs and GPUs. Running such an application in an efficient way requires a smart runtime system that optimizes scheduling according to data placement, resource (core) availability, and performance.

On the general purpose CPU side, the future will certainly see further growth in the number of cores: in 2006, Intel has demonstrated the 80 core Polaris chip, Sun has announced a 16-core "Rock" UltraSPARC processor for 2009 availability, and Intel has published their "Larrabee" many-core architecture (Seiler et al. 2008) which is to scale to dozens of cores. A common theme here is to forego some of the advanced architecture features (such as out of order execution) to reduce the complexity and die size of each core and use the "headroom" to both introduce high-performance SIMD compute units per core and increase the core count at the same time. These "small and nimble" cores can deliver great performance for applications that are adapted to them (like graphics processing)—they will perform worse than conventional cores for many nonadapted programs. Thus, designers of future CPUs face a difficult decision: should they go the "all small core" approach which will maximize peak performance, stick with the complex cores that run a wide variety of (nonoptimized) codes well, or create heterogeneous CPUs that combine both kind of cores. Today, it is too early to tell which evolution path the dominating CPU vendors will be taking. A GP CPU that combines different cores remains a distinct possibility. In the embedded systems space, MIPS-based systems have gone up to 64 cores already, and ARM is joining the multicore bandwagon.

The accelerator field is evolving very quickly; GPUs push the number of processing elements and the functionality of them at the same time, thus increasing peak performance and extending their reach from pure SIMD data parallel kernels in the direction of task level or functional parallelism. NVIDIA's recent communications and Intel's entry into the graphics market with the Larrabee architecture provide ample evidence here. A second line of evolution concerns the way of connecting host CPUs with the GPU: higher performing bus connections (like PCI Express 3.0), cache coherent interconnects like Intel's QPI and AMD's Hypertransport, and finally the inclusion of graphics processing elements into the host CPU (as announced from Intel for their Nehalem desktop chips). Combined, these trends will make future GPUs much more similar to the CPUs that they compete with and alleviate the large performance disparity that we see today between local memory and the bus interconnects.

10.3 Intelligent Programming of Multi-Core Systems

In this section we outline our methodology and the corresponding environment for programming multicore systems.

10.3.1 Methodology

Our parallel programming methodology combines model-driven agent-supported program development with the use of high-level *parallel building blocks* (PBB). We propose to address the complexity of programming multicore systems as follows:

- Raise the level of abstraction at which the programmer performs most of the activities during the process of software development, by using a model-driven development approach combined with PBBs;
- Support the programmer during the software development, by using intelligent software agents for providing context-specific knowledge and automation of iterative activities involved in software development and optimization.

10.3.1.1 Model-Driven Development (MDD)

MDD (Model Driven Architecture 2009) is a software development method that advocates to *first model* a program and *then build* the program code. It is inspired by mature engineering disciplines such as *civil engineering*, where before an artifact (for instance, a *bridge*) is built, the corresponding model is first developed. In software engineering the models are usually described graphically using the Unified Modeling Language (UML). The model should preferably describe the program at an abstraction level that is independent from a specific platform. Models may be used to study the functionality, and the performance of the program before the program code for a specific platform is developed. MDD has the potential to reduce software development time and complexity, by using tools for automatic model-to-code transformation and thereby reducing the programmer's effort for manual coding. Since multicore architectures differ significantly from each other, a significant effort is required to adapt (that is, *port*) programs to the various platforms. Since MDD captures the program logic as a platform-independent model, program models remain largely unaffected from the changes in processor architectures. In our previous work we have developed an extension of UML for the domain of performance-oriented parallel/distributed programs (Pllana et al. 2002) and the corresponding tool-support *Teuta* (Pllana et al. 2004). Teuta allows one to build models of parallel programs, enrich them with performance-related information, and generate various textual representations (such as XML or C++).

10.3.1.2 Parallel Building Blocks

The PBBs are inspired from research in programming concepts such as *skeletons* (Alba et al. 2002; Alind et al. 2008; Cole 2004) or *dwarfs* (Asanovic et al. 2006). Basically, PBBs may be thought of as program-independent generic programming units that support software reusability. A set of parameters is used to specify the functionality of a PBB in the context of a certain program. For instance, as parameter may serve the program-specific code (that is, the code that PBB requires to perform the expected functionality in the context of a certain program). PBBs may be implemented, for instance, using *C++ Templates* or *Java Generics*. Parallelism is described within the PBB, and therefore the programmer is not exposed directly to the parallel programming complexity (such as dealing explicitly with the *communication and synchronization* among processing units or *deadlock avoidance*). Commonly various combinations of PBBs may be used for solving a certain problem. In the context of programming environments, PBBs lend themselves to an increased level of automation of various activities such as program transformation, code generation, performance optimization, and resource usage optimization. In our previous work, in the context of MALLBA project (Alba et al. 2002), we have developed a library of parallel skeletons (such as *branch and bound*, *metropolis*, *simulated annealing*, *genetic algorithms*, or *tabu search*) for solving various optimization problems.

10.3.1.3 Intelligent Software Agents

Software agents are programs that are *reactive*, *proactive*, *autonomic*, and *social* (Wooldridge 2002). Software agents that have *learning* and *adapting* abilities are known as *intelligent software agents*. *Reactiveness* indicates the ability to respond adequately to changes in the context in which it operates. A *proactive* program performs activities to achieve a specific goal based on its initiative (it does not wait passively for a request of another entity to perform a certain activity). *Autonomy* indicates the ability to perform activities independently of user intervention in order to achieve a specific goal. *Social* programs are able to communicate and coordinate activities with other programs (that is, agents). A program is considered intelligent if it is able to learn from the previous experience (for instance, via trial-and-error or generalization) and is able to adapt accordingly to the perceived changes in the environment. We have a vision about several intelligent software agents cooperating with each other and the programmer during the process of program development. Our vision is based on the idea that the programming environment should be better at helping the programmer as a more active partner. In our previous work, in the context of the AURORA project (AURORA 2009), we have used intelligent software agents to automate systematic performance analysis for parallel and distributed programs. Although software development is considered to be an art, we anticipate that there are many implementation activities that can be performed more automatically/autonomically using intelligent software agents.

In the following subsection we propose a programming environment for multicore computing systems that uses MDD, PBBs, and intelligent software agents.

10.3.2 Programming Environment

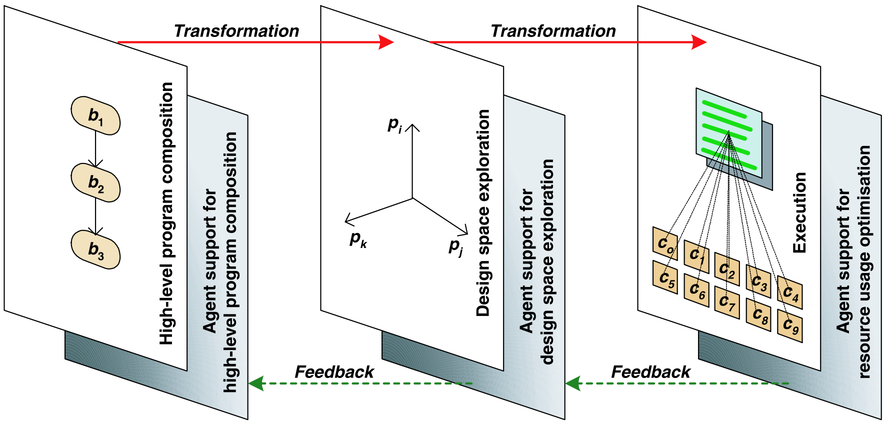
The proposed programming environment comprises a set of intelligent software agents that may help to automate the programming process at several levels. Some agents will advice the composition of programs using PBBs, while others will guide the exploration of different possible parallel strategies, load balancing, and performance optimization (see Fig. 10.1).

The programming environment provides the programmer with information feedback useful in the process of developing a program for a multicore system. This information is collected at several levels, from program composition to information about resource usage (such as the cache behavior) obtained by execution or simulated execution. Also, information is exchanged between the agents at the system level in an automated manner continuously looking for ways of obtaining and improving knowledge about the performance of the program being developed. In this way, a parallel program with good performance can be developed with high programmer productivity.

In what follows in this section we highlight the major program development and tuning phases: (1) high-level program composition, (2) design space exploration, and (3) resource usage optimization.

10.3.2.1 High-level Program Composition

This phase deals with the composition and coordination of PBBs. The granularity of PBBs may range from frequently used programming idioms to larger patterns or dwarfs (Asanovic et al. 2006). High-level descriptors are used to capture the main parallelization aspects of PBBs and serve as interface to agents in the design



Transformation
Transformation
b_1
b_2
b_3
High-level program composition
Agent support for high-level program composition
p_i
p_k
p_j
Design space exploration
Agent support for design space exploration
Execution
c_0
c_1
c_2
c_3
c_4
c_5
c_6
c_7
c_8
c_9
Agent support for resource usage optimisation
Feedback
Feedback

Fig. 10.1 Agent-supported program development. The programming environment comprises multiple intelligent software agents that support program composition, design space exploration, and resource usage optimization

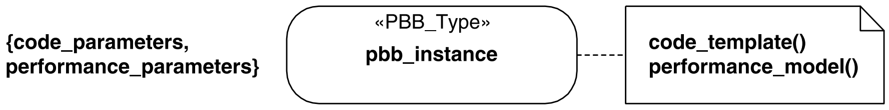


Fig. 10.2 UML representation of a PBB

space exploration phase. The user composes the program graphically using a UML extension for multicore systems.

The UML may be extended by defining new modeling elements, *stereotypes*, based on existing elements (also known as *base classes* or *metaclasses*). Stereotypes are notated by the stereotype name enclosed in guillemets «`Stereotype Name`». Figure 10.2 depicts the graphical representation of a PBB. «`PBB_Type`» indicates the kind of PBB. With a PBB is associated the corresponding *parameterized code and performance model*. Parameters determine the behavior of the PBB instance in the context of a specific program.

The programming environment assists the user proactively during the program composition. For instance, while the user is loading some old BLAS code for some dense linear algebra operations, the *composer agent* interrupts and suggests using the PBB for dense linear algebra tailored for efficient execution on multicore systems. Additionally, it may offer a list of other PBBs that often are used together with this one, also presenting typical compositional patterns in a graphical way.

10.3.2.2 Design Space Exploration

High-level discrete-event simulation is used for rapid model-based performance evaluation of programs, using a hybrid method that combines mathematical modeling with high-level discrete-event simulation (Pllana et al. 2008).

For instance, after the completion of the program composition phase, the programming environment may suggest to the user doing some high-level rapid design space exploration. The estimated performance of various possible program implementations is presented by a *visualization agent*. While the user is studying the graphs and gets some ideas for improvement, the programming environment is also analyzing the results and suggests changing some of the parameters in one of the PBBs (such as the parallelization granularity) and performing some more detailed simulations for getting better knowledge of the performance that can be obtained with different task allocation and scheduling policies.

10.3.2.3 Resource Usage Optimization

Instruction-level simulation is used for more detailed studies of the utilization of shared resources such as shared on-chip memory and off-chip bandwidth. For instance, in Dybdahl et al. (2006) an efficient utilization of the shared cache resources has been found to have great affect on multicore performance. This is integrated with the use of performance counters. A performance monitoring agent provides information about the state of the system (resource characteristics and usage). Instruction-level simulation is time consuming (may take several hours or days) and therefore should run in background. When finished, the findings will be propagated upwards back to the higher-level performance models, as a model calibration process. It is a systematic way of bringing performance information from the execution (or simulated execution) environment back to the development environment. Please note that this kind of optimization is architecture-dependent.

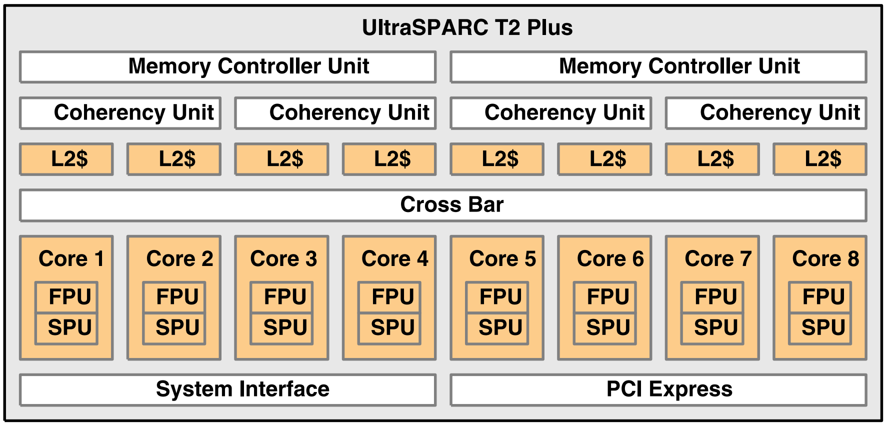
For instance, the user may get hints from the programming environment for changes that will improve performance of the program. The programming environment may offer some detailed simulations at the instruction level and helps the user to select those simulation experiments that are likely to be the most relevant. For instance, if higher-level simulations show that some of the processor cores were waiting for data for long periods, a more detailed study of the on-chip shared memory resources should be done.

10.4 Example

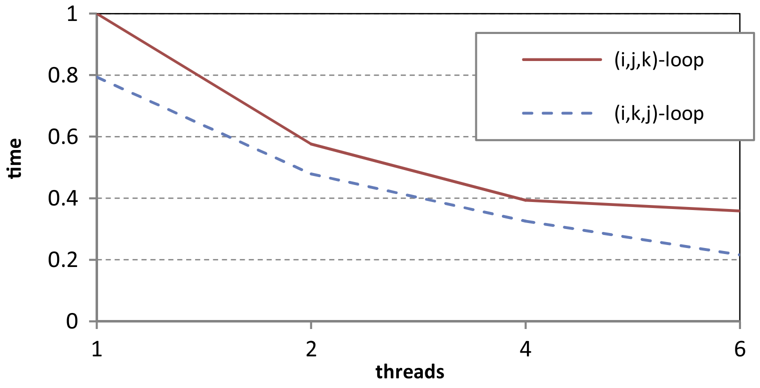
In this section we illustrate how best practices from HPC, combined with agent-based program development, offer new opportunities to obtain efficient solutions.

PBBs allow a programmer to specify various parallelization strategies together with the code and a first guess for individual parameters which are subject to the tuning process. This follows our assumption that only semi-automatic parallelization is reasonable. The programmer specifies the main strategies for parallelizing the code, and the system explores this restricted optimization space to generate efficient code. Two factors back up this approach. First, rich analysis work has been done in the past by the HPC community, including the authors institutions (Vienna Fortran Compilation System, Benkner et al. 1996), which can be reused. Second, in the past the strong emphasis on the target-code performance and manual performance tuning resulted in low programming productivity. The increasing importance of development of economically viable software nowadays reveals opportunities for semiautomatic parallelization, even at the price of achieving lower performance compared to a hand-tuned version.

In our example we use as hardware platform the Sun UltraSPARC T2 Plus, codenamed Niagara-2, multicore processor (shown in Fig. 10.3(a)) which is an SMP extended version of the T2 allowing multiple Chip-level Multithreading (CMT) processors to be used within a single system. The T2 Plus was presented in April 2008 and has up to 8 cores per processor with 8 hardware threads per core resulting in a maximum number of 64 threads per processor or logical CPUs as reported by the operating system. T2 Plus offers only poor support for instruction-level parallelism emphasizing thread-level parallelism. Two integer units are provided per core with four threads sharing one unit, and one FPU is provided per core with all eight threads sharing it. The L1 data cache has 8 KB per core, and the on-chip L2 cache offers 4 MB which are shared between the cores.



(a) Sun UltraSPARC T2 Plus.



(b) Performance improvements.

Fig. 10.3 Processor block diagram and optimization results

In what follows in this section we present an example scenario to illustrate the agent-supported software development cycle. Different forms of PBBs are possible, but in the simplest case a PBB can be some loop nest together with data layout and work distribution annotations. Consider, e.g., an application written in C consisting of a series of PBBs with one of them denoting a floating point matrix–matrix multiplication, i.e., `C[i,j] = C[i,j] + A[i,k] * B[k,j]` with loop nest (i, j, k). As parallelization strategy, the programmer specifies that the elements of result matrix C should be assigned to processor cores in a row-wise manner and calculated by them. Since the target architecture is a Sun T2 Plus with 8 cores and 8 FPUs, the programmer specifies that the rows shall be assigned to 8 threads.

When submitted to the *design space exploration agent* and its analysis framework (cf. Benkner et al. 1996; Benkner 1999), the framework detects poor spatial cache locality and performs loop interchange resulting in loop nest (i, k, j). Then the code is split up in 8 threads, as suggested by the programmer, assigned to the 8 cores of T2 Plus, and executed. The monitoring component of the *resource usage agent* reveals low memory bandwidth utilization and low FPU utilization for this PBB and reports this feedback information to the agent. The *resource usage agent* is aware of the hardware characteristics of T2 Plus and knows about the hyper-threading (HT) technology provided by this kind of architecture with up to 8 hardware threads. Therefore the agent suggests to use HT technology to increase FPU utilization and reports to the *design space agent* to explore possibilities to increase the number of threads. Consequently, the *design space agent* proposes to assign the rows of result matrix C to 2, 4, 6 hardware threads per core resulting in a total number of 16, 32, 48 threads, respectively. Three versions are generated and submitted for execution. Moreover, feedback information is used by the compilation system to perform further optimizations (cf. Gupta and Mehofer 2002).

The key point is that this time-consuming tuning task is done automatically by the system and not by the programmer. The different versions are automatically generated and run on T2 Plus, and the monitoring results are reported back to the agents and the programmer. Figure 10.3(b) shows the normalized execution times (longest execution time denoted by time unit 1.0) for the different versions with 1, 2, 4, 6 threads per core and the improvements achieved by the optimizations taking programmer annotations and hardware characteristics into account. The performance improvement of loop interchange is considerable and amounts to 26% for 1 thread per core, approximately 20% for 2 and 4 threads per core, and 66% for 6 threads per core. The performance improvement for increasing the number of threads per core to deal with memory latency is even more significant. The performance improvement assigning 2 and 4 threads to one core was for both loop nest versions approximately a factor of 1.7 and 2.5, respectively. For 6 threads per core, we got for (i, j, k) loop nest a factor of 2.8 and for (i, k, j) loop nest up to 3.7. Based on this experience, the *resource usage agent* classifies increasing the number of threads to deal with memory latency as valuable optimization which has proven beneficial for this processor. The programming environment may suggest this kind of optimization for similar processor architectures as well.

10.5 Related Work

An increasing number of research projects is addressing the challenge of programming multicore computing systems. The *Habanero project* (Habanero Multicore Software Project 2009), which started in Fall 2007 at Rice University, aims to develop languages and compilers for the development of portable software for multicore systems. The *SALSA project* (Service Aggregated Linked Sequential Activities 2009) at Indiana University is investigating the use of services as building blocks for composing parallel data-mining applications based on the workflow paradigm. *Linked Sequential Activities* in SALSA, which are conceptually based on Communicating Sequential Processes of Hoare, are used to build services. The *Berkeley View* (Asanovic et al. 2006) project investigates the influence of multicore processors in applications, hardware, programming models, and systems software for parallel computing. The Berkeley View proposes to use a set of *dwarfs* (a dwarf defines a specific computation and communication pattern) for evaluation of parallel programming models. The recently established *Pervasive Parallelism Laboratory (PPL)* (Pervasive Parallelism Laboratory 2009) at Stanford University is investigating future parallel computing platforms. PPL is supported by six computer and chip makers that are convinced that their product sales may decline if software is not able to use effectively the new multicore-based hardware. *SWARM* (Bader et al. 2007), developed at Georgia Institute of Technology, is a parallel programming framework that provides a collection of primitives for programming multicore processors. The *Programming Environments Laboratory (PELAB)* (Programming Environments Laboratory 2009) at Linköping University is investigating the applicability of round-trip engineering techniques to parallelization of sequential programs. The *Cell Superscalar (CellSs)* (Perez et al. 2007) project at Barcelona Supercomputing Center focuses on parallelization of sequential programs for Cell BE processor. The CellSs parallelization involves the functional decomposition, code annotation, and the use of a source-to-source compiler. The *IT Research Division of the NEC Laboratories Europe* (Wagner et al. 2008) is investigating the use of work stealing concept to achieve load balancing. *Performance Portability and Programmability for Heterogeneous Many-core Architectures* (PEPPHER) (PEPPHER 2009) is a related project that is funded under the Seventh Framework Programme of the European Commission. PEPPHER aims at providing a unified framework for programming architecturally diverse, heterogeneous manycore processors to ensure performance portability.

In contrast to the related work, we propose an intelligent programming environment that proactively supports the user during major phases of program development and performance tuning by providing context-specific knowledge and performing iterative time-consuming tasks involved in program development in a semi automatic/autonomic manner.

10.6 Conclusions

We have outlined an intelligent programming environment, which proactively supports the user during high-level program composition, design space exploration, and resource usage optimization. We have highlighted the potential benefits of using such a programming environment with usage-scenarios.

We have observed that even for a rather simple parallel building block such as matrix multiplication, the exploration of the parameter space may be, on one hand, time prohibitive, but, on the other hand, there is a big potential for performance improvement. The example scenario described a first and manageable step toward an intelligent program environment for multicore architectures. Several projects at the authors' home institutions are currently pursued toward the realization of such an intelligent programming environment for multicore computing systems.

Acknowledgements Fatos Xhafa's research work is supported by a grant from the General Secretariat of Universities of the Ministry of Education, Spain. The authors are grateful for numerous discussions and contributions related to Sect. 10.2 to Alastair Donaldson, Hans-Christian Hoppe, Christoph Kessler, David Moloney, Raymond Namyst, Peter Sanders, Jesper Larsson Traff, and Philippas Tsigas.

References

AGEIA Technologies (now a subsidiary of NVIDIA): The PhysX processor, http://www.ageia.com, accessed November 2009.

E. Alba, F. Almeida, M. Blesa, J. Cabeza, C. Cotta, M. Diaz, I. Dorta, J. Gabarro, C. Leon, J. Luna, L. Moreno, C. Pablos, J. Petit, A. Rojas, and F. Xhafa. MALLBA: A library of skeletons for combinatorial optimisation (research note). In Euro-Par 2002. Springer, Berlin, 2002.

M. Alind, M. Eriksson, and C. Kessler. BlockLib: a skeleton library for cell broadband engine. In International Workshop on Multicore Software Engineering (IWMSE-2008) at ICSE-2008. Leipzig, Germany, May 2008. ACM, New York, 2008.

E. Allen, D. Chase, J. Hallett, V. Luchangco, J. Maessen, S. Ryu, G. Steele Jr., and S. Tobin-Hochstadt. The Fortress language specification, version 1.0 (available at http://research.sun.com/projects/plrg/Publications/fortress.1.0.pdf), March 2008.

J. Ansel, C. Chan, Y. Wong, M. Olszewski, A. Edelman, and S. Amarasinghe. PetaBricks: a language and compiler for algorithmic choice. In ACM SIGPLAN Conference on Programming Language Design and Implementation, June 2009.

K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams, and K. Yelick. The landscape of parallel computing research: a view from Berkeley. EECS Department, University of California, Berkeley, Technical Report No. UCB/EECS-2006-183, December 18, 2006.

AURORA: a priority research program on advanced models, applications and software systems for high performance computing (1997–2007). http://www.vcpc.univie.ac.at/aurora/, accessed November 2009.

D. Bader, V. Kanade, and K. Madduri. SWARM: A parallel programming framework for multicore processors. First Workshop on Multithreaded Architectures and Applications (MTAAP) at IPDPS 2007. Long Beach, CA, USA, March 2007. IEEE, New York, 2007.

S. Benkner. VFC: The Vienna Fortran compiler. Scientific Programming, 7(1):67–81, 1999.

S. Benkner, S. Andel, R. Blasko, P. Brezany, A. Celic, B. Chapman, M. Egg, T. Fahringer, J. Hulman, E. Kelc, E. Mehofer, H. Moritsch, M. Paul, K. Sanjari, V. Sipkova, B. Velkov, B. Wender, and H. Zima. Vienna Fortran compilation system, Version 1.2, user's guide. Technical report, Institute for Software Technology and Parallel Systems, University of Vienna, February 1996.

F. Bodin, S. Bihan. Heterogeneous multicore parallel programming for graphics processing units. Scientific Programming, 17(4):283–348, 2009.

R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon. Parallel Programming in OpenMP. Morgan Kaufmann, San Francisco, 2000.

P. Charles et al.: X10: An object-oriented approach to non-uniform cluster computing. In Proc. ACM OOPSLA'05, Oct. 2005. See also: Report on the Experimental Language X10, Draft 0.41 (available at http://www.research.ibm.com/x10), Feb. 2006.

M. Cole. Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. Parallel Computing 30(3):389–406, 2004.

Cray Inc., Seattle, WA. Chapel specification, version 0.780, February 2008. (http://chapel.cs.washington.edu).

A. Donaldson, C. Riley, A. Lokhmotov, and A. Cook. Autoparallelisation of sieve C++ programs. In Proceedings of the 1st Euro-Par Workshop on Highly Parallel Processing on a Chip (HPPC), volume 4854 of *Lecture Notes in Computer Science*, pages 18–27. Springer, Berlin, 2007.

H. Dybdahl, P. Stenström, and L. Natvig. A cache-partitioning aware replacement policy for chip multiprocessors. In 13th Intern. Conf. of High Perform. Comput., HiPC 2006. Springer, Berlin, 2006.

A. Eichenberger et al. Using advanced compiler technology to exploit the performance of the cell broadband engineTM architecture. IBM Systems Journal, 45(1):59–84, 2006.

K. Fatahalian, T. Knight, M. Houston, M. Erez, D. Horn, L. Leem, J. Park, M. Ren, A. Aiken, W. Dally, and P. Hanrahan. Sequoia: programming the memory hierarchy. In Proceedings of the ACM/IEEE SC2006 Conference on High Performance Networking and Computing, November 11–17, 2006, Tampa, FL, USA. ACM, New York, 2006.

I. Foster and C. Kesselman (Editors). The Grid 2: Blueprint for a New Computing Infrastructure. The Elsevier Series in Grid Computing, 2 edition. Morgan Kaufmann, San Mateo, 2003.

R. Gupta, E. Mehofer, and Y. Zhang. Profile guided code optimizations. In Y.N. Srikant, and P. Shankar, editors, The Compiler Design Handbook: Optimizations & Machine Code Generation. CRC Press, Boca Raton, 2002.

Habanero Multicore Software Project. http://www.cs.rice.edu/~vs3/habanero/, accessed November 2009.

M. Hall, Y. Gil, and R. Lucas. Self-configuring applications for heterogeneous systems: program composition and optimization using cognitive techniques. In Proceedings of the IEEE, Special Issue on Cutting-Edge Computing: Using New Commodity Architectures, Volume 96, Issue 5, 2008.

High Performance Fortran Forum. High performance Fortran language specification, version 2.0. Technical report, January 1997.

L. Howes, A. Lokhmotov, A. Donaldson, and P. Kelly. Deriving efficient data movement from decoupled access/execute specifications. In Proceedings of the 4th International Conference on High Performance and Embedded Architectures and Compilers (HiPEAC), volume 5409 of *Lecture Notes in Computer Science*, pages 168–182. Springer, Berlin, 2009.

IBM Cell/B.E. SDK for multicore acceleration version 3.1, available at http://www.ibm.com/developerworks/power/cell/, 2008.

IBM XL C/C++ for Multicore Acceleration for Linux. http://www.alphaworks.ibm.com/tech/cellcompiler, accessed November 2009.

Intel Corp.: Intel Threading Building Blocks 2.1, Reference Manual, 2009 (available at http://www.threadingbuildingblocks.org).

Khronos OpenCL Working Group. The OpenCL specification, version 1.0. Updated, May 16, 2009. http://www.khronos.org/registry/cl/specs/opencl-1.0.43.pdf

T. Knight, J. Park, M. Ren, M. Houston, M. Erez, K. Fatahalian, A. Aiken, W. Dally, and P. Hanrahan. Compilation for explicitly managed memory hierarchies. In Proceedings of the 12th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming, San Jose, California, USA, pages 226–236, ACM Press, New York, 2009.

S. Lindley. Implementing deterministic declarative concurrency using sieves. In Proceedings of the ACM SIGPLAN Workshop on Declarative Aspects of Multicore Programming (DAMP), ACM, New York, 2007.

Model Driven Architecture. http://www.omg.org/mda/, accessed November 2009.

R.W. Numerich and J. Reid. Co-array FORTRAN for parallel programming. SIGPLAN Fortran Forum, 17(2):1–31, 1998.

NVIDIA Corp. CUDA Zone: http://developer.nvidia.com/object/cuda.html, accessed November 2009.

J. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. Lefohn, and T. Purcell. A survey of general-purpose computation on graphics hardware. Computer Graphics Forum, 26(1):80–113, 2007.

J. Perez, P. Bellens, R. Badia, and J. Labarta. CellSs: Making it easier to program the cell broadband engine processor. IBM Journal of Research and Development 51(5): 593–604, 2007.

Performance Portability and Programmability for Heterogeneous Many-core Architectures (PEPPHER). The Seventh Framework Programme of the European Commission. Project Reference: 248481. http://www.peppher.eu/, accessed November 2009.

Pervasive Parallelism Laboratory. http://ppl.stanford.edu/wiki/index.php/Pervasive_Parallelism_Laboratory, accessed November 2009.

S. Pllana et al. On customizing the UML for modeling performance-oriented applications. In Proceedings of «UML» 2002, Model Engineering, Concepts and Tools, *LNCS* 2460, Springer, Dresden, 2002.

S. Pllana et al. Teuta: tool support for performance modeling of distributed and parallel applications. In International Conference on Computational Science, Tools for Program Development and Analysis in Computational Science. Krakow, Poland, June 2004. Springer, Dresden, 2004.

S. Pllana, S. Benkner, F. Xhafa, and L. Barolli. Hybrid performance modeling and prediction of large-scale computing systems. In 2008 International Conference on Complex, Intelligent and Software Intensive Systems (CISIS 2008). Barcelona, Spain, March 2008. IEEE CS, Los Alamitos, 2008.

L. Pouchet, C. Bastoul, A. Cohen, and N. Vasilache. Iterative optimization in the polyhedral model: Part I, one-dimensional time. IEEE/ACM Fifth International Symposium on Code Generation and Optimization (CGO'07), San Jose, California, pages 144–156. IEEE Computer Society Press, Los Alamitos, 2007.

Programming Environments Laboratory (PELAB). http://www.ida.liu.se/labs/pelab/, accessed November 2009.

RapidMind corp. home page: http://www.rapidmind.net/, accessed November 2009.

L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: a many-core x86 architecture for visual computing. ACM Transactions on Graphics 27, 3 (2008), 1–15.

Service Aggregated Linked Sequential Activities (SALSA). http://www.infomall.org/multicore/, accessed November 2009.

D. Shreiner, M. Woo, J. Neider, and T. Davis. OpenGL Programming Guide: The Official Guide to Learning OpenGL. Addison-Wesley Professional, Reading, 2005.

I.J. Taylor, E. Deelman, D.B. Gannon, M. Shields (Editors). Workflows for E-science: Scientific Workflows for Grids. Springer, Berlin, 2006.

The UPC Consortium. UPC Language Specification (v 1.2), June 2005. available at http://upc.gwu.edu.

J. Wagner, A. Jahanpanah, and J. Träff. User-land work stealing schedulers: Towards a standard. 2008 International Workshop on Multi-Core Computing Systems (MuCoCoS'08) at CISIS 2008. Barcelona, Spain, March 2008. IEEE CS, Los Alamitos, 2008.

M. Wooldridge. An Introduction to MultiAgent Systems. Wiley, New York, 2002.

K. Yelick, L. Semenzato, G. Pike, C. Miyamato, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: a high-performance Java dialect. Concurrency: Practice and Experience, 10(11–13):825–836, 1998.