Chapter 1
Parallel Programming Models

Vassilios V. Dimakopoulos

1.1 Introduction

Programming models represent an abstraction of the capabilities of the hardware to the programmer. A programming model is a bridge between the actual machine organization and the software that is going to be executed by it. As an example, the abstraction of the memory subsystem leads to two fundamentally different types of parallel programming models, those based on shared memory and those based on message passing. Programming through shared memory can be compared to the use of a bulletin board. Information is exchanged by posting data to shared locations which are agreed upon a priori by the sender and receiver of data. In shared-memory architectures, this is realized directly using ordinary loads and stores. On the other hand, communication via message passing occurs through point-to-point transfers between two computing entities and is thus conceptually similar to the exchange of letters which explicitly identify the sender and receiver of the information. In distributed-memory architectures, these transfers are realized over an actual processor interconnection network.

Because a programming model is an abstraction of the underlying system architecture, it is usually not tied to one specific machine, but rather to all machines with architectures that adhere to the abstraction. In addition one particular system may support more than one programming model, albeit often not with the same efficiency.

A successful parallel programming model must carefully balance opacity and visibility of the underlying architecture; intricate details and idiosyncrasies should be hidden while features which enable the full computational power

V.V. Dimakopoulos (✉)
Department of Computer Science and Engineering, University of Ioannina,
Ioannina, GR-45110, Greece
e-mail: dimako@cs.uoi.gr

M. Torquati et al. (eds.), *Smart Multicore Embedded Systems*,
DOI 10.1007/978-1-4614-8800-2_1,

of the underlying device need to be exposed. The programming model also determines how program parts executed in parallel communicate and which types of synchronization among them are available.

In summary, a programming model enables the developer to express ideas in a certain way. To put the ideas into practice, however, the developer has to choose a specific programming language. The correspondence between programming models and programming languages is close but not one-to-one. Several languages may implement the same programming model. For instance, Java and C++ with Pthreads offer very similar shared-memory programming models although the two languages differ significantly in several other important aspects. Furthermore, it is possible for a programming language and set of APIs to adhere to several programming models.

In what follows we present the most significant parallel programming models available today. In Sect. 1.2 we classify them according to the memory abstraction they offer to the programmer. In Sect. 1.3 we survey models based on shared memory while Sect. 1.4 covers distributed-memory models. Section 1.5 surveys the available models for GPUs and accelerators which represent devices with private memory spaces. Models that try to combine some of the above categories are examined in Sect. 1.6. Finally, Sect. 1.7 visits other promising languages and programming styles that do not fall in the above categories.

1.2 Classification of Parallel Programming Models

Algorithms and whole applications contain parallelism with varying degrees of regularity and granularity. Parallelism can be exploited at the following granularity levels: bits, instructions, data, and tasks. *Bit-level parallelism* is exploited transparently to the programmer by increasing the processor word size thus reducing the number of instructions required to perform elementary arithmetic and bit manipulation operations. Similarly, *instruction-level parallelism* can be exploited by processors and compilers with little or no involvement from the programmer. Processors support overlap of instruction execution through techniques such as instruction pipelining, super-scalar, and out-of-order execution while optimizing compilers can increase the efficiency of these techniques by careful reorderings of program instructions.

At higher granularity levels, such as *data level*, parallelism is not generally exploitable without programmer intervention and the programming model plays a major role. `For` loops in programming languages are a major source of data-level parallelism and therefore this kind of parallelism is also referred to as *loop-level parallelism*. Many processors include single-instruction, multiple-data (SIMD) instructions which operate on short vectors typically containing two to eight data values. These differ from single-instruction, single-data (SISD) instructions as the latter can only compute a single result at a time. Some programming models

allow the programmer to utilize SIMD instructions either explicitly or implicitly, by means of a vectorizing compiler. In any case, it is required that the data-parallel computation can be expressed as operations on vectors, i.e. that a single instruction can be applied to multiple data values.

Another important way of exploiting data-level parallelism is *multiprocessing* or *multi-threading*, where the data-parallel computation is distributed over multiple processors, a single processor capable of executing multiple instruction streams in parallel, or any combination of these. Such architectures are characterized as multiple-instruction, multiple-data (MIMD) machines as they are capable of processing multiple independent instruction streams, each of which operates on separate data items. If data parallelism is found in a loop nest, the workload is distributed among the processing elements by assigning a fraction of the overall iteration count to each of them. Unlike SIMD instructions, the distribution of work can be dynamic if the time to process each of the iterations varies.

The coarsest-grained and least regular kind of parallelism is *task-level parallelism*, which almost uniformly relies on the programmer's ability to identify the parts of an application that may be executed independently. Task-level parallelism is about the distribution of both computation and data among processing elements, whereas data parallelism primarily emphasizes the distribution of data among processing elements. Programming models vary significantly in their support of exploiting task-level parallelism—some are strongly focused on task-level parallelism and provide abstractions which are flexible enough to also exploit data parallelism while others are mainly focused on the exploitation of structured or loop-level parallelism and provide only weak support for irregular and dynamic parallelism at the granularity of tasks.

Except the above considerations on the level and type of parallelism offered, parallel programming models can also be classified by the memory abstraction they present to the programmer. The memory organization of typical parallel computing systems gives rise to two broad architectural categories. In *centralized* or *shared-memory multiprocessors*, the memory is accessible to all processors with uniform access time (UMA). The processors are usually attached to a bus, sharing its bandwidth, an architecture also known as a *symmetric multiprocessor* (SMP). Unfortunately, buses and centralized memories suffer from increased contention and longer latencies when more processors are added. This limited scalability means that central memories are only used in smaller multiprocessors, usually having up to eight processors. They are also found in the individual nodes of larger systems.

In the second organization, the memory is physically distributed among processors. Two access policies are then possible; each local memory either is accessible only to the respective processor or can be accessed by all processors albeit with nonuniform memory access (NUMA) times through a global address space. The former is called a (pure) *distributed-memory organization*, whereas the latter is a *distributed shared-memory organization* which can be cache coherent (ccNUMA) or not. While scalable ccNUMA systems are a challenge to design, they put smaller burden on the programmer because replication and coherence is managed transparently

by the hardware. Non-cache-coherent distributed shared-memory systems require either the programmer to make sure that each processor view of the memory is consistent with main memory or the intervention of a software layer which manages the consistency transparently in return for some decrease in performance.

The programming models used to target a given platform are closely related to the underlying memory and communication architecture. Shared-memory models offer the programmer the ability to define objects (e.g. variables) that are accessible to all execution entities (e.g. processes or threads) and can be realized most efficiently on machines containing hardware which keeps the memories and caches coherent across the system. Explicit message passing, the dominant representative of distributed-memory models, is commonly used whenever memory is non-centralized and noncoherent. However, both shared- and distributed-memory organizations can support any kind of programming model, albeit with varying degrees of efficiency. For instance, it is possible to support message passing very efficiently by passing a pointer rather than copying data, on shared-memory architectures. A global shared-memory model can also be supported on a distributed-memory machine either in hardware (ccNUMA) or in software with a runtime layer [2], although obtaining acceptable speedups on a wide range of unmodified, parallel applications is challenging and still a subject of active research.

The parallel-computing landscape has been augmented with systems that usually operate as back-end attachments to a general-purpose machine, offering increased execution speeds for special portions of code offloaded to them. *General-purpose graphical processing units* (GPGPUs) and *accelerators* are typical examples. Because of their distinct memory and processing element organization these devices warrant suitable programming models. In addition, their presence gives rise to *heterogeneous* systems and programming models in the sense that the host and the back-end processing elements are no longer of the same type and the programmer must provide different programs for each part of the system.

It is also possible to combine the above paradigms and thus obtain a *hybrid programming model*. This usually requires the programmer to explicitly utilize multiple and different programming styles in the same program and is, for example, popular when targeting clusters of multicores/multiprocessors. Another possibility is represented by the Partitioned Global Address Space (PGAS) languages. Here the notions of shared and distributed memory are unified, treating memory as a globally shared entity which is however logically partitioned, with each portion being local to a processor.

Finally, many other different attributes can be considered in order to classify parallel programming models [3]. For example, they can be categorized based on their application domain or by the way execution entities (workers) are defined, managed, or mapped to actual hardware. Another attribute is the programming paradigm (procedural, object-oriented, functional, streaming, etc). In this chapter we focus on the memory abstraction offered by the programming model.

1.3 Shared-Memory Models

Shared-memory models are based on the assumption that the execution entities (or workers) that carry the actual execution of the program instructions have access to a common memory area, where they can store objects, uniformly accessible to all. They fit naturally UMA architectures (SMPs/multicores with small processor/core counts) and ccNUMA systems. Shared-memory models can be further categorized by the type of execution entities employed and the way they are handled by the programmer (explicit or implicit creation, mapping and workload partitioning).

1.3.1 POSIX Threads

A *thread* is an autonomous execution entity, with its own program counter and execution stack which is usually described as a "lightweight" process. A normal ("heavyweight") process may generate multiple threads; while autonomous, the threads share the process code, its global variables, and any heap-allocated data. The Portable Operating System Interface (POSIX) provides for a standard interface to create threads and control them in a user program. POSIX is a standardization of the interface between the operating system and applications in the Unix family of operating systems. The POSIX.1c thread extensions [4] provide a description of the syntax and semantics of the functions and data types used to create and manipulate threads and is known as *Pthreads* [5].

In parallel applications, multiple threads are created by the `pthread_create` call. Because the execution speed and sequence of different threads is unpredictable and unordered by default, programmers must be aware of race conditions and deadlocks. Synchronization should be used if operations must occur in certain order. POSIX provides *condition variables* as their main synchronization mechanism. Condition variables provide a structured means for a thread to wait (block) until another thread signals that a certain condition becomes true (`pthread_condition_wait`/`pthread_condition_signal` calls). The real-time extensions to POSIX [6] define *barriers* as an additional synchronization mechanism for Pthreads. A thread that calls `pthread_barrier_wait` blocks until all sibling threads perform the same call; they are all then released to continue their execution.

Pthreads provide *mutex objects* as a primary way of achieving mutual exclusion, which is typically used to coordinate access to a resource which is shared among multiple threads. A thread should lock (`pthread_mutex_lock`) the mutex before entering the critical section of the code and unlock it (`pthread_mutex_unlock`) right after leaving it. Pthreads also provide *semaphores* as another mechanism for mutual exclusion.

Pthreads are considered a rather low-level programming model that puts too much burden on the programmer. Explicitly managing and manipulating the execution entities can sometimes give ultimate control to an expert programmer [5], but this comes at the cost of increased complexity and lower productivity since larger Pthreads programs are considered hard to develop, debug, and maintain.

1.3.2 OpenMP

Because threads are a versatile albeit a low-level primitive for parallel programming, it has been argued that they should not be used directly by the application programmer; one should rather use higher-level abstractions which are possibly mapped to threads by an underlying runtime. OpenMP [1] can be seen as a realization of that philosophy. It is a set of compiler directives and library functions which are used to parallelize sequential C/C++ or Fortran code. An important feature of OpenMP is its support for *incremental* parallelism, whereby directives can be added gradually starting from a sequential program.

Like Pthreads, OpenMP is an explicitly parallel programming model meaning that the compiler does not analyze the source code to identify parallelism. The programmer instructs the compiler on how the code should be parallelized, but unlike Pthreads, in OpenMP threads are not an explicit notion; the programmer primarily creates and controls them implicitly through higher-level directives.

OpenMP supports a fork-join model of parallelism. Programs begin executing on a single, master thread which spawns additional threads as parallelized regions are encountered (enclosed in an `omp parallel` directive), i.e. a fork. Parallel regions can be nested although the compiler is not required to exploit more than one level of parallelism. At the end of the outermost parallel region the master thread joins with all worker threads before continuing execution.

A parallel region essentially replicates a job across a set of spawned threads. The threads may cooperate by performing different parts of a job through *worksharing* constructs. The most prominent of such constructs is the `omp for` (C/C++) or `omp loop` (Fortran) directive where the iterations of the adjacent loop are divided and distributed among the participating threads. This allows for easy parallelization of regular loop nests and has been the main strength of OpenMP since these loops are prevalent in scientific codes such as linear algebra or simulation of physical phenomena on rectangular grids.

Since revision 3.0 of the standard [7] the applicability of OpenMP has been significantly broadened in applications with dynamic and irregular parallelism (e.g. when work is created recursively or is contained in loops with unknown iteration counts) with the addition of the `omp task` directive. Tasks are blocks of code that are marked by the programmer and can be executed asynchronously by any thread. Because of their asynchronous nature, tasks must also carry a copy of the data they will operate on when actually executed.

It would not be an exaggeration to say that OpenMP has nowadays become the de facto standard for shared-memory programming. It is used to program

multiprocessors and multicores alike, whether they physically share memory or they have ccNUMA organizations. It has also been implemented successfully for a number of embedded platforms (e.g. [8–10]). There even exist implementations of earlier versions of OpenMP which target computational clusters though shared virtual memory software libraries (albeit without reaching high performance levels). Another important fact is that the directive-based programming style of OpenMP and its latest addition of tasks have a profound influence on many recent programming model proposals for others architectures (cf. Sect. 1.5). However, notice that while an intuitive model to use, OpenMP may not always be able to produce the maximum performance possible since it does not allow fine low-level control.

1.4 Distributed-Memory Models

Distributed-memory systems with no physical shared memory can be programmed in a multitude of ways. To name a few:

- Low-level socket programming
- Remote procedure calls (e.g., SUN RPC, Java RMI)
- Software shared virtual memory [2], to provide the illusion of shared memory
- Message passing

are among the models that have been used. However, *message passing* is by far the dominating programming model for developing high-performance parallel applications in distributed architectures.

Notice also that approaches similar to the above have also been proposed in specific domains. For example, in the context of real-time software for embedded heterogeneous MPSoCs, such as multimedia and signal processing applications, the TTL [11] and Multiflex [12] frameworks provide programming models based on tasks or objects communicating by transferring tokens over channels or through remote procedure calls.

1.4.1 Message-Passing (MPI)

The message-passing model assumes a collection of execution entities (processes, in particular) which do not share anything and are able to communicate with each other by exchanging explicit messages. This is a natural model for distributed-memory systems where communication cannot be achieved through shared variables. It is also an efficient model for NUMA systems where, even if they support a shared address space, the memory is physically distributed among the processors.

Message passing is now almost synonymous to MPI, the Message Passing Interface [13, 14]. MPI is a specification for message-passing operations and is implemented as a library which can be used by C and Fortran programs. An MPI program consists of a number of identical processes with different address spaces,

where data is partitioned and distributed among them (single-program, multiple-data or SPMD style). Interaction among them occurs through messaging operations. MPI provides send/receive operations between a named sender and a named receiver, called point-to-point communications (`MPI_Send` / `MPI_Recv`). These operations are cooperative or *two-sided*, as they involve both sending and receiving processes, and are available in both *synchronous* and *asynchronous* versions.

A synchronous pair of send/receive operations defines a synchronization point between the two entities and requires no buffering since the sender remains blocked until the transfer completes. If the synchronous send/receive pair is not executed simultaneously, either the sender or receiver blocks and is prevented from performing useful work. Asynchronous message passing allows the sender to overlap communication with computation thus increasing performance if the application contains enough exploitable parallelism. In this case buffers are required, and depending on timing, caution is needed to avoid filling them up.

The second version of the MPI [14] added a number of enhancements. One of the most significant is the ability to perform one-sided communications (`MPI_Put`/`MPI_Get`), where a process can perform remote memory accesses (writes or reads) without requiring the involvement of the remote process.

There also exists a very rich collection of global or *collective* (one-to-many, many-to-many) operations such as *gather*, *scatter*, *reduction*, and *broadcast* which involve more than two processes and are indispensable for both source code structuring and performance enhancement.

MPI dominates programming on computational clusters. Additionally, there exist implementations that allow applications to run on larger computational grids. There also exist lightweight implementations specialized for embedded systems, such as LMPI [15]. MPI is generally considered an efficient but low-level programming model. Like Pthreads, the programmer must partition the work to be done by each execution entity and derive the mapping to the actual processors. Unlike Pthreads, one also needs to partition and distribute the data on which the program operates.

1.5 Heterogeneity: GPGPU and Accelerator Models

General-purpose graphics processing units (GPGPUs) employ the power of a GPU pipeline to perform general-purpose computations instead of solely graphical operations. They have been recognized as indispensable devices for accelerating particular types of high-throughput computational tasks exhibiting data parallelism. They consist typically of hundreds to thousands of elementary processing cores able to perform massive vector operations over their wide vector SIMD architecture.

Such devices are generally nonautonomous. They assume the existence of a *host* (CPU) which will *off-load* portions of code (called *kernels*) for them to execute. As such, a user program is actually divided in two parts—the host and the device

part, to be executed by the CPU and the GPGPU correspondingly, giving rise to heterogeneous programming. Despite the heterogeneity, the coding is generally done in the same programming language which is usually an extension of C.

The two dominant models for programming GPGPUs are Compute Unified Device Architecture (CUDA) and OpenCL. The first is a programming model developed by NVIDIA for programming its own GPUs. The second is an open standard that strives to offer platform-independent general-purpose computation over graphics processing units. As such it has also been implemented on non-GPU devices like general or special-purpose accelerators, the ST P2012/STHORM being a characteristic example [16]. Other models have also been proposed, trying to alleviate the inherent programming heterogeneity to some degree.

1.5.1 CUDA

In CUDA [17, 18] the computation of tasks is done in the GPU by a set of threads that run in parallel. The threads are organized in a two-level hierarchy, namely, the *block* and the *grid*. The first is a set of tightly coupled threads where each thread is identified by a thread ID while the second is a set of loosely coupled blocks with similar size and dimension. The grid is handled by the GPU, which is organized as a collection of "multiprocessors." Each multiprocessor executes one or more of the blocks and there is no synchronization among the blocks.

CUDA is implemented as an extension to the C language. Tasks to be executed on the GPU (kernels) are functions marked with the new `__global__` qualifier. Thread management is implicit; programmers need only specify grid and block dimensions for a particular kernel and do not need to manage thread creation and destruction. On the other hand, workload partitioning and thread mapping is done explicitly when calling the `__global__` kernel using the `«<gs,bs»>` construct, where `gs` (`bs`) specifies the dimensions of the grid (block).

The memory model of CUDA consists of a hierarchy of memories. In particular, there is per-thread memory (registers and *local* memory), per-block memory (*shared* memory, accessed by all threads in a block), and per-device memory (read/write *global* and read-only *constant* memory, accessed by all threads in a grid and by the host). Careful data placement is crucial for application performance.

1.5.2 OpenCL

OpenCL [19] is a standardized, cross-platform, parallel-computing API based on the C99 language and designed to enable the development of portable parallel applications for systems with heterogeneous computing devices. It is quite similar to CUDA although it can be somewhat more complex as it strives for platform independence and portability.

As in CUDA, an OpenCL program consists of two parts: kernels that execute on one or more devices and a host program that manages the execution of kernels. Kernels are marked with the `__kernel` qualifier. Their code is run by *work items* (cf. CUDA threads). Work items form *work groups*, which correspond to CUDA thread blocks. The memory hierarchy is again similar to CUDA with the exception that memory shared among the items of a work group is termed *local memory* (instead of shared), while per-work item memory is called *private* (instead of local).

In OpenCL, devices are managed through *contexts*. The programmer first creates a context that contains the devices to use, through calls like `clCreateContext`. To submit work for execution by a device, the host program must first create a command queue for the device (`clCreateCommandQueue`). After that, the host code can perform a sequence of OpenCL API calls to insert a kernel along with its execution parameters into the command queue. When the device becomes available, it removes and executes the kernel at the head of the queue.

1.5.3 Directive-Based Models

Because programming GPGPUs with the CUDA and OpenCL models is tedious and in a rather low abstraction level, there have been a number of proposals for an alternative way of programming these devices. The common denominator of these proposals is the task-parallel model; kernels are simply denoted as tasks in the user programs and thus blend naturally with the rest of the code, reducing thus the impact of heterogeneity. Moreover, these models are heavily influenced by the OpenMP directive-based style of programming.

GPUSs [20] uses two directives. The first one allows the programmer to annotate a kernel as a task, while also specifying the required parameters and their size and directionality. The second one maps the task to the device that will execute it (in the absence of this directive, the task is executed by the host CPU). This also determines the data movements between the host and the device memories.

Another example is HMPP [21] which offers four types of directives. The first one defines the kernel (termed *codelet*) to be executed on the accelerator. The second one specifies how to use a codelet at a given point in the program, including which device will execute it. The third type of directives determines the actual data transfers before executing the codelet. Finally, a fourth directive is used for synchronization.

Finally, OpenACC [22] is another attempt to provide a simplified programming model for heterogeneous CPU/GPU systems. It is developed by Cray, CAPS, NVIDIA, and PGI and consists of a number of compiler directives and runtime functions. OpenACC is expected to be endorsed by the upcoming version of OpenMP.

1.6 Hybrid Models

Hybrid programming models try to combine two or more of the aforementioned models in the same user program. This can be advantageous for performance reasons when targeting systems which do not fall clearly in one architectural category. A characteristic example is a cluster of multicore nodes. Clusters are distributed-memory machines. Their nodes are autonomous computers, each with its own processors and memory; nodes are connected through an interconnection network which is used for communicating with each other. Within a node, however, the processors (or cores) have access to the same local memory, forming a small shared-memory subsystem.

Using one programming model (e.g. OpenMP combined with software shared virtual memory layers, or MPI only) to leverage such platforms is a valid option but may not be the most efficient one. Hybrid programming utilizes multiple programming models in an effort to produce the best possible code for each part of the system.

A similar situation occurs in systems consisting of multiple nodes of CPUs and/or GPU cards. There have been works that combine, for example, OpenMP and CUDA [23] or even CUDA and MPI [24]. However, this case is not considered in detail here as the various GPU models are more or less hybrid by nature in the sense of already supporting heterogeneous CPU/GPU programming.

1.6.1 Pthreads + MPI

Under this model, Pthreads are used for shared-memory programming within a node while MPI is used for message passing across the entire system. While the first version of the MPI standard was not designed to be safely mixed with user-created threads, MPI-2 [14] allows users to write multithreaded programs easily. There are four levels of thread safety supported by systems and selectable by the programmer (through the `MPI_Init_thread` call): `MPI_THREAD_SINGLE`, `MPI_THREAD_FUNNELED`, `MPI_THREAD_SERIALIZED`, and `MPI_THREAD_MULTIPLE`. Except for `SINGLE` all other levels allow multiple threads. In the `FUNNELED` level, however, only one of the threads is allowed to place MPI calls. The other two levels allow MPI calls by all threads (albeit nonsimultaneously in the `SERIALIZED` level). Notice that not all implementations support all thread safety levels.

While there have been some application studies reported with the hybrid Pthreads + MPI model (e.g. [25, 26]), this is not an extensively used approach; OpenMP is the most popular choice for spawning shared-memory parallelism within a node.

1.6.2 OpenMP + MPI

This is the most widely used combination, where OpenMP is employed to leverage the shared-memory features of a node [27–29]. This, of course, guarantees portability since both OpenMP and MPI are industry standards. Many application classes can benefit from the usage of this model. For example, some applications expose two levels of parallelism, coarse-grained (suitable for MPI) and fine-grained (suitable for OpenMP). The combination may also help in situations where the load of the MPI processes is relatively unbalanced. Finally, OpenMP threads may provide a faster communication mechanism than multiple MPI processes within a node.

On the other hand, some applications possess only one level of parallelism. In addition, the utilization of OpenMP is not without its costs; it introduces the overheads of thread creation, synchronization, and worksharing. Consequently, the hybrid OpenMP + MPI model may not always be the better choice (see, e.g. [30, 31]).

The user program is structured as a collection of MPI processes. The code of each process is enhanced with OpenMP directives to take advantage of the presence of shared memory. Depending on the programming and the capabilities of the MPI implementation MPI calls may be made by the master thread only, outside `parallel` regions. The other option is to allow MPI calls within `parallel` regions and thus have some thread(s) communicate while others compute (the `SERIALIZED` or `MULTIPLE` safety levels are required). As a result, one can overlap communication with computation. This requires the most complicated control, but can result in performance improvements.

The hybrid OpenMP + MPI model has been used successfully in many cases (e.g. [32–34]). We should also note the existence of frameworks that facilitate programming with this model while also combining it with others, such as task-centric ones (e.g. HOMPI by [35]).

1.6.3 PGAS

PGAS stands for Partitioned Global Address Space and represents a class of programming languages and runtime libraries that try to marry the shared- and distributed-memory models when targeting clusters of SMPs or multicores. However, while the other hybrid approaches force the programmer to mix two different models in the same code, PGAS presents a single, unified model which inherits characteristics of both.

In the PGAS model multiple SPMD threads (or processes) share a part of their address space. However, this globally shared area is logically partitioned, with each portion being local to a processor. Thus, programs are able to use a shared address space, which generally simplifies programming, while also exposing data/computation locality in order to enhance performance. Because of this, PGAS is sometimes termed *locality-aware* shared-memory programming.

Two characteristic examples of the PGAS model are Co-array Fortran (CAF, [36]) and Unified Parallel C (UPC, [37]). The first one is an extension to Fortran 95 and is now incorporated in the recent Fortran 2008 standard. A program in CAF consists of SPMD processes (*images*). The language provides for defining private and shared data, accessing shared data with one-sided communications and synchronization among images. Similarly, UPC extends the C programming language to allow declaring globally shared variables, partitioning and distributing their storage across the available nodes. It offers an affinity-aware loop construct (`upc_forall`) as well as built-in calls for collective communications among threads.

1.7 Other Parallel Programming Models

In this section we briefly discuss other approaches to parallel programming. We take a different view from the previous sections in that we do not categorize them according to their memory abstraction. We first take a look at new languages or notable parallel extensions to well-known sequential languages, irrespectively of the memory model they follow. Then we visit a different way of parallel programming—skeleton-based models.

1.7.1 Languages and Language Extensions

Cilk [38] is a language extension to C which adds support for parallel programming based on tasks or *Cilk procedures*. Syntactically, a Cilk procedure is a regular C function where the `cilk` keyword has been added to its definition to allow asynchronous invocation. Cilk procedures can then be invoked or spawned by prefixing the function invocation with the `spawn` keyword. A key strength of Cilk is its support for irregular and dynamic parallelism. Work stealing, a provably optimal technique for balancing the workload across processing elements, was developed as part of the Cilk project. It has subsequently been adopted by numerous other parallel programming frameworks. Cilk++ is a commercial implementation of the language.

Sequoia [39] is a language extension to a subset of C. In Sequoia tasks are represented via a special language construct and are isolated from each other in that each task has its own address space and calling a subtask or returning from a task is the only means of communication among processing elements. To achieve isolation tasks have call-by-value-result semantics and the use of pointer and reference types inside tasks is disallowed. First-class language constructs are used to express data decomposition and distribution among processing elements and locality-enhancing transformations such as loop blocking; for instance, the `blkset` data type is used to represent the tiles or blocks of a conventional array. Another characteristic of Sequoia is that the memory hierarchy is represented explicitly via trees of memory

modules. Generic algorithmic expression and machine-specific optimizations are kept (mostly) separate. The source code contains tunable parameters and variants of the same task which are optimized for different hardware architectures. A separate set of mapping files stores values of tunable parameters and choices of task variants for the individual execution platform.

Hierarchically tiled arrays (HTA, [40]) are an attempt to realize efficient parallel computation at a higher level of abstraction solely by adding specialized data types in traditional, imperative languages such as C++. As the name implies, HTAs are hierarchies of tiles where each tile is either a conventional array or an HTA itself. Tiles serve the dual purpose of controlling data distribution and communication at the highest level and of attaining locality for the sequential computation inside each task. Like Sequoia, HTA preserves the abstraction of a global shared memory while automatically generating calls to a message-passing library on distributed-memory architectures.

Java is one of the most popular application-oriented languages. However, it also provides support for parallel programming. For one, it was designed to be multi-threaded. Additionally, Java provides Remote Method Invocation (RMI) for transparent communication among virtual machines. A number of other programming models can also be exercised using this language. For example, *MPJ Express* [41] is a library that provides message-passing facilities to Java. Finally, Java forms the basis for notable PGAS languages such as *Titanium* [42].

1.7.2 Skeletal Programming

Algorithmic skeletons [43] represent a different, higher-level pragmatic approach to parallel programming. They promote *structured parallel programming* where a parallel program is conceived as two separate and complementary concerns: *computation*, which expresses the actual calculations, and *coordination*, which abstracts the interaction and communication. In principle, the two concepts should be orthogonal and generic, so that a coordination style can be applied to any parallel program, coarse- or fine-grained. Nonetheless, in conventional parallel applications, computation and coordination are not necessarily separated, and communications and synchronization primitives are typically interwoven with calculations.

Algorithmic skeletons essentially abstract commonly used patterns of parallel computation, communication, and interaction (e.g. map-reduce, for-all, divide and conquer) and make them available to the programmer as high-level programming constructs. Skeletal parallel programs can then be expressed by interweaving *parametrized* skeletons using composition and control inheritance throughout the program structure. Based on their functionality, skeletons can be categorized as data parallel (which work on bulk data structures and typically require a function or sub-skeleton to be applied to all elements of the structure, e.g. *map* or *reduce*), task-parallel (which operate on tasks, e.g. *farm*, *pipe*, *for*), or resolution (which outline algorithmic methods for a family of problems, e.g. *divide and conquer* or

branch and bound). Notice that the interface of the skeleton is decoupled from its implementation and as a consequence the programmer need only specify *what* is to computed and not *how* it should be deployed on a given architecture.

Most skeleton frameworks target distributed-memory platforms, e.g. eSkel [44], SKELib [45], and ASSIST [46], which deliver task- and data-parallel skeletal APIs. Java-based Skandium [47] and C++-based FastFlow [48], on the other hand, target shared-memory systems. The FastFlow framework is differentiated as it focuses on stream parallelism, providing farm, divide and conquer, and pipeline skeletons. A detailed survey of algorithmic skeleton frameworks is given in [49].

1.8 Conclusion

In this chapter we attempted to outline the parallel programming models landscape by discussing the most important and popular ones. The reader should be aware that what we presented is just a portion of what is available for the programmer or what has been proposed by researchers. We classified the models mainly according to the memory abstraction they present to the programmer and then presented the representative ones in each category. We covered shared-memory models, distributed-memory models, and models for GPUs and accelerators. Hybrid models combine the aforementioned ones in some way. We finally concluded with a summary of other models that do not fit directly in the above categories.

Parallel programming models is a vast area of active research. The multitude of platforms and architectures and the variety of their combinations are overwhelming but at the same time a source of new problems and ideas. The domination of multi- and many-core systems even on the desktop has pushed research on parallel programming even further. This is not without reason since the "concurrency revolution is primarily a software revolution. The difficult problem is not building multicore hardware, but programming it in a way that lets mainstream applications benefit from the continued exponential growth in CPU performance" [50].

References

1. OpenMP ARB, "OpenMP Application Program Interface V3.1," July 2011.
2. J. Protic, M. Tomasevic, and V. Milutinovic, "Distributed shared memory: Concepts and systems," *IEEE Concurrency*, vol. 4, pp. 63–79, 1996.
3. K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The landscape of parallel computing research: A view from Berkeley," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2006-183, Dec 2006.
4. IEEE, "IEEE P1003.1c/D10: Draft standard for information technology – Portable Operating System Interface (POSIX)," Sept 1994.

5. D. Butenhof, *Programming With Posix Threads*, ser. Addison-Wesley Professional Computing Series. Addison-Wesley, 1997.
6. IEEE, "IEEE Std 1003.1j-2000: Standard for information technology – Portable Operating System Interface (POSIX)- part 1: System Application Program Interface (API)-Amendment J: Advanced real-time extensions," pp. 1–88, 2000.
7. OpenMP ARB, "OpenMP Application Program Interface V3.0," May 2008.
8. F. Liu and V. Chaudhary, "A practical OpenMP compiler for system on chips," in *WOMPAT '03, Int'l Workshop on OpenMP Applications and Tools*, Toronto, Canada, 2003, pp. 54–68.
9. T. Hanawa, M. Sato, J. Lee, T. Imada, H. Kimura, and T. Boku, "Evaluation of multicore processors for embedded systems by parallel benchmark program using OpenMP," in *IWOMP 2009, 5th International Workshop on OpenMP*, Dresden, Germany, June 2009, pp. 15–27.
10. S. N. Agathos, V. V. Dimakopoulos, A. Mourelis, and A. Papadogiannakis, "Deploying OpenMP on an embedded multicore accelerator," in *SAMOS XIII, 13th Int'l Conference on Embedded Computer Systems: Architectures, MOdeling, and Simulation*, Samos, Greece, July 2013.
11. P. Vanăder Wolf, E. deăKock, T. Henriksson, W. Kruijtzer, and G. Essink, "Design and programming of embedded multiprocessors: an interface-centric approach," in *Proc. CODES+ISSS '04, 2nd IEEE/ACM/IFIP Int'l Conference on Hardware/software Codesign and System Synthesis*, New York, USA, 2004, pp. 206–217.
12. P. Paulin, C. Pilkington, M. Langevin, E. Bensoudane, and G. Nicolescu, "Parallel programming models for a multi-processor SoC platform applied to high-speed traffic management," in *Proc. CODES+ISSS '04, 2nd IEEE/ACM/IFIP Int'l Conference on Hardware/software Codesign and System Synthesis*, New York, USA, 2004, pp. 48–53.
13. W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message Passing Interface,* 2nd edition. Cambridge, MA: MIT Press, 1999.
14. W. Gropp, E. Lusk, and R. Thakur, *Using MPI-2: Advanced Features of the Message-Passing Interface*. Cambridge, MA: MIT Press, 1999.
15. A. Agbaria, D.-I. Kang, and K. Singh, "LMPI: MPI for heterogeneous embedded distributed systems," in *ICPADS '06, 12th International Conference on Parallel and Distributed Systems - Volume 1*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 79–86.
16. L. Benini, E. Flamand, D. Fuin, and D. Melpignano, "P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, 2012, pp. 983–987.
17. NVIDIA, *NVIDIA CUDA Programming Guide 2.0*, 2008.
18. D. B. Kirk and W.-m. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2010.
19. Khronos Group, *The OpenCL Specification Version 1.0*, Beaverton, OR, 2009.
20. E. Ayguadé, R. M. Badia, F. D. Igual, J. Labarta, R. Mayo, and E. S. Quintana-Ortí, "An extension of the StarSs programming model for platforms with multiple GPUs," in *Euro-Par '09, 15th International Euro-Par Conference on Parallel Processing*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 851–862.
21. R. Dolbeau, S. Bihan, and F. Bodin, "HMPP: A hybrid multi-core parallel programming environment," in *GPGPU 2007, Workshop on General Purpose Processing on Graphics Processing Units*, 2007.
22. OpenACC, *The OpenACC™ Application Programming Interface Version 1.0*, Nov 2011.
23. Y. Wang, Z. Feng, H. Guo, C. He, and Y. Yang, "Scene recognition acceleration using CUDA and OpenMP," in *ICISE '09, 1st International Conference on Information Science and Engineering*, 2009, pp. 1422–1425.
24. Q.-k. Chen and J.-k. Zhang, "A stream processor cluster architecture model with the hybrid technology of MPI and CUDA," in *ICISE, 2009, 1st International Conference on Information Science and Engineering*, 2009, pp. 86–89.
25. C. Wright, "Hybrid programming fun: Making bzip2 parallel with MPICH2 & Pthreads on the Cray XD1," in *CUG '06, 48th Cray User Group Conference*, 2006.

26. W. Pfeiffer and A. Stamatakis, "Hybrid MPI / Pthreads parallelization of the RAxML phylogenetics code," in *9th IEEE International Workshop on High Performance Computational Biology*, Atlanta, GA, Apr 2010.
27. L. Smith and M. Bull, "Development of mixed mode MPI / OpenMP applications," *Scientific Programming*, vol. 9, no. 2,3, pp. 83–98, Aug. 2001.
28. R. Rabenseifner, "Hybrid parallel programming on HPC platforms," in *EWOMP '03, 5th European Workshop on OpenMP*, Aachen, Germany, Sept 2003, pp. 185–194.
29. B. Estrade, "Hybrid programming with MPI and OpenMP," in *High Performance Computing Workshop*, 2009.
30. F. Cappello and D. Etiemble, "MPI versus MPI+OpenMP on IBM SP for the NAS benchmarks," in *SC '00, ACM/IEEE Conference on Supercomputing*, Dallas, Texas, USA, 2000.
31. D. S. Henty, "Performance of hybrid message-passing and shared-memory parallelism for discrete element modeling," in *SC '00, ACM/IEEE Conference on Supercomputing*, Dallas, Texas, USA, 2000.
32. K. Nakajima, "Parallel iterative solvers for finite-element methods using an OpenMP/MPI hybrid programming model on the earth simulator," *Parallel Computing*, vol. 31, no. 10–12, pp. 1048–1065, Oct. 2005.
33. R. Aversa, B. Di Martino, M. Rak, S. Venticinque, and U. Villano, "Performance prediction through simulation of a hybrid MPI/OpenMP application," *Parallel Comput.*, vol. 31, no. 10–12, pp. 1013–1033, Oct. 2005.
34. P. D. Mininni, D. Rosenberg, R. Reddy, and A. Pouquet, "A hybrid MPI-OpenMP scheme for scalable parallel pseudospectral computations for fluid turbulence," *Parallel Computing*, vol. 37, no. 6–7, pp. 316–326, 2011.
35. V. V. Dimakopoulos and P. E. Hadjidoukas, "HOMPI: A hybrid programming framework for expressing and deploying task-based parallelism," in *Euro-Par'11, 17th International Conference on Parallel processing*, Bordeaux, France, Aug 2011, pp. 14–26.
36. R. W. Numrich and J. Reid, "Co-arrays in the next Fortran standard," *SIGPLAN Fortran Forum*, vol. 24, no. 2, pp. 4–17, Aug. 2005.
37. UPC Consortium, "UPC language specifications, v1.2," Lawrence Berkeley National Lab, Tech. Rep. LBNL-59208, 2005.
38. M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the Cilk-5 multithreaded language," in *PLDI '98, ACM SIGPLAN 1998 conference on Programming language design and implementation*, Montreal, Quebec, Canada, 1998, pp. 212–223.
39. K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan, "Sequoia: programming the memory hierarchy," in *SC '06, 2006 ACM/IEEE Conference on Supercomputing*, Tampa, Florida, 2006.
40. G. Bikshandi, J. Guo, D. Hoeflinger, G. Almasi, B. B. Fraguela, M. J. Garzarán, D. Padua, and C. von Praun, "Programming for parallelism and locality with hierarchically tiled arrays," in *PPoPP '06, 11th ACM SIGPLAN symposium on Principles and practice of parallel programming*, New York, New York, USA, 2006, pp. 48–57.
41. A. Shafi, B. Carpenter, and M. Baker, "Nested parallelism for multi-core HPC systems using Java," *J. Parallel Distrib. Comput.*, vol. 69, no. 6, pp. 532–545, Jun. 2009.
42. K. A. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. N. Hilfinger, S. L. Graham, D. Gay, P. Colella, and A. Aiken, "Titanium: A high-performance Java dialect," *Concurrency: Practice and Experience*, vol. 10, no. 11–13, pp. 825–836, 1998.
43. M. Cole, *Algorithmic skeletons: structured management of parallel computation*. London: Pitman / MIT Press, 1989.
44. M. Cole, "Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming," *Parallel Computing*, vol. 30, no. 3, pp. 389–406, 2004.
45. M. Danelutto and M. Stigliani, "SKElib: parallel programming with skeletons in C," in *Proc. of 6th Intl. Euro-Par 2000 Parallel Processing*, ser. LNCS, A. Bode, T. Ludwing, W. Karl, and R. Wismüller, Eds., vol. 1900. Munich, Germany: Springer, Aug. 2000, pp. 1175–1184.
46. M. Vanneschi, "The programming model of ASSIST, an environment for parallel and distributed portable applications," *Parallel Computing*, vol. 28, no. 12, pp. 1709–1732, 2002.

47. M. Leyton and J. M. Piquer, “Skandium: Multi-core programming with algorithmic skeletons,” in *PDP ’10, 18th Euromicro Int’l Conference on Parallel, Distributed and Network-Based Processing*, 2010, pp. 289–296.
48. M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati, “FastFlow: high-level and efficient streaming on multi-core,” in *Programming Multi-core and Many-core Computing Systems*, ser. Parallel and Distributed Computing, S. Pllana and F. Xhafa, Eds. Wiley, Jan. 2013, ch. 13.
49. H. González-Vélez and M. Leyton, “A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers,” *Software: Practice and Experience*, vol. 40, no. 12, pp. 1135–1160, 2010.
50. H. Sutter and J. Larus, “Software and the concurrency revolution,” *ACM Queue*, vol. 3, no. 7, pp. 54–62, Sep. 2005.