CHAPTER

# INTRODUCTION AND OVERVIEW 1

## 1.1 PROCESSES AND THREADS

Multithreading is today a mandatory software technology for taking full advantage of the capabilities of modern computing platforms of any kind, and a good understanding of the overall role and impact of threads, as well as the place they take in the computational landscape, is useful before embarking on a detailed discussion of this subject.

We are all permanently running standalone applications in all kinds of computing devices: servers, laptops, smartphones, or multimedia devices. All the data processing devices available in our environment have a master piece of software—the operating system (OS)—that controls and manages the device, running different applications at the user's demands. A standalone application corresponds to a *process* run by the operating system. A process can be seen as a black box that owns a number of protected hardware and software resources; namely, a block of memory, files to read and write data, network connections to communicate eventually with other processes or devices, and, last but not least, code in the form of a list of instructions to be executed. Process resources are protected in the sense that no other process running on the same platform can access and use them. A process has also access to one or several central processing units—called CPU for short—providing the computing cycles needed to execute the application code.

One of the most basic concepts we will be using is the notion of *concurrency*, which is about two or more activities happening at the same time. Concurrency is a natural part of life: we all can, for example, walk and talk at the same time. In computing platforms, we speak of concurrency when several different activities can advance and make progress at the same time, rather than one after the other. This is possible because CPU cycles are not necessarily exclusively owned by one individual process. The OS can allocate CPU cycles to several different processes at the same time (a very common situation in general). It allocates, for example, successive time slices on a given CPU to different processes, thereby providing the illusion that the processes are running simultaneously. This is called multitasking. In this case, processes are not really running simultaneously but, at a human time scale, they advance together, which is why it is appropriate to speak about concurrent execution.

One special case of concurrency happens when there are sufficient CPU resources so that they do not need to be shared: each activity has exclusive ownership of the CPU resource it needs. In this optimal case, we may speak of *parallel execution*. But remember that parallel execution is just a special case of concurrency.

**Shared Memory Application Programming.** http://dx.doi.org/10.1016/B978-0-12-803761-4.00001-0

As stated, a process incorporates a list of instructions to be executed by the CPU. An ordinary sequential application consists of a single *thread of execution*, i.e., a single list of instructions that are bound to be executed on a single CPU. In a C-C++ programming environment, this single thread of execution is coded in the main() function. In a multithreaded application, a process integrates several independent lists of instructions that execute asynchronously on one or several CPU. Chapter 3 shows in detail how the different multithreading programming environments allow the initial main thread to install other execution streams, equivalent to main() but totally disconnected from it. Obviously, multithreading is focused on enhancing the process performance, either by isolating independent tasks that can be executed concurrently by different threads, or by activating several CPUs to share the work needed to perform a given parallel task.

## 1.2 OVERVIEW OF COMPUTING PLATFORMS

A computing platform consists essentially of one or more CPUs and a memory block where the application data and code are stored. There are also peripheral devices to communicate with the external world, like DVD drives, hard disks, graphic cards, or network interfaces.

Figure 1.1 shows a schematic view of a personal computer. For the time being, think of the processor chip as corresponding to a single CPU, which used to be the case until the advent of the multicore evolution, circa 2004, when processor chips started to integrate several CPUs. The data processing performance of the simple platform shown in Figure 1.1 depends basically on the performance of the CPU, i.e., the rate at which the CPU is able to execute the basic instructions of the instruction set. But it also depends critically on the rate at which data can be moved to and from the memory: a fast processor is useless if it has no data to work upon.

The rate at which the CPU is able to execute basic instructions is determined by the internal processor clock speed. Typically, the execution of a basic low-level instruction may take up to a few cycles, depending on the instruction complexity. The clock frequency is directly related to the *peak*

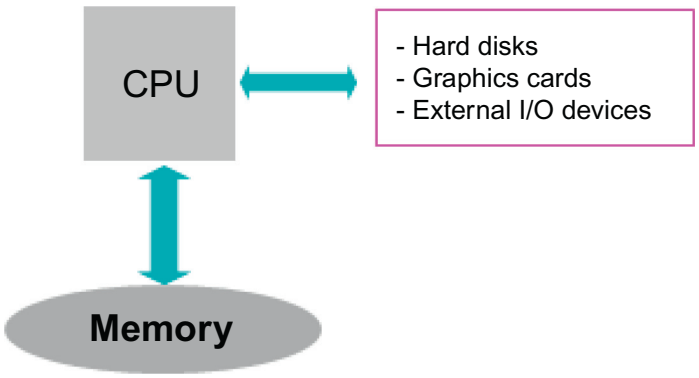


**FIGURE 1.1**

Schematic architecture of a simple computing platform.

*performance*, which is the theoretical performance in an ideal context in which processor cycles are not wasted. The original Intel 8086 processor in the late 70 s ran at 5 MHz (i.e., 5 millions of cycles per second). Today, processors run at about 3 GHz (3000 millions of cycles per second). This represents a 600× increase in frequency, achieved in just over 20 years.

Memory performance is a completely different issue. The time $T$ needed to move to or from memory a data block of $N$ bytes is given by:

$$T = L + N/B$$

where $L$ is the *latency*, namely, the access time needed to trigger the transfer, and $B$ is the *bandwidth*, namely, the speed at which data is moved once the transfer has started.

It is important to keep in mind that, in the case we are considering, moving data from memory to the processor or vice versa:

- The *bandwidth* is a property of the network connecting the two devices. Bandwidths are controlled by network technologies.
- The *latency* is an intrinsic property of the memory device.

It turns out that memory latencies are very difficult to tame. At the time of the 8086, memory latencies were close to the processor clock, so it was possible to feed the processor at a rate guaranteeing roughly the execution of one instruction per clock cycle. But memory latencies have not decreased as dramatically as processor frequencies have increased. Today, it takes a few hundreds of clock cycles to trigger a memory access. Therefore, in ordinary memory bound applications the processors tend to starve, and typically their *sustained performances* are a small fraction (about 10%) of the theoretical peak performance promised by the processor frequency. As we will see again and again, this is the crucial issue for application performance, and lots of ingenuity in the design of hardware and system software has gone into taming this bottleneck, which is called the *memory wall*.

### 1.2.1 SHARED MEMORY MULTIPROCESSOR SYSTEMS

The next step in computing platform complexity are the so-called *symmetric multiprocessor* (SMP) systems, where a network interconnect links a number of processor chips to a common, shared memory block. Symmetric means that all CPUs have an equivalent status with respect to memory accesses. Memory is shared, and all CPUs can access the whole common memory address space. They are shown in Figure 1.2.

For reasons that will be discussed in Chapter 7, coherency in shared memory accesses prevents SMP platforms from scaling to very large numbers of CPUs. Today, shared memory systems do not exceed 60 CPUs. This limit will perhaps increase in the future, but it is bound to remain limited. These systems are very useful as medium-size servers. They can perform efficient multitasking by allocating different applications on different CPUs. On the other hand, a multithreaded application can benefit from the availability of a reasonable number of CPUs to enhance the process performance.

However, note that the SMP computing platforms are not perfectly symmetric. The reason is that a huge, logically shared memory block is normally constructed by using, say, $M$ identical memory devices. Therefore, the SMP network interconnect is really connecting all $N$ processor chips to all $M$ memory blocks. According to the topology and the quality of the network, a given CPU may

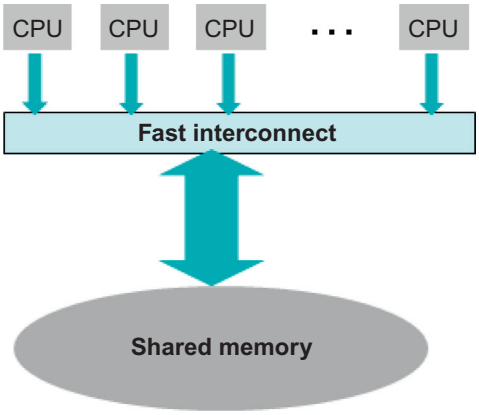


**FIGURE 1.2**

Symmetric multiprocessor shared memory platform.

not be at the same "effective distance" of all $M$ memory blocks, and the access times may be non-uniform with mild differences in memory access performance. This kind of computing platform is called a NUMA (Non-Uniform Memory Access) architecture. This observation is sometimes relevant in multithreaded programming, because performance may be optimized by placing data items as close as possible to the CPU running the thread that is accessing them. This issue—placing threads as near as possible to the data they access—is called *memory affinity*. It plays a significant role in multithreaded programming.

### 1.2.2 DISTRIBUTED MEMORY MULTIPROCESSOR SYSTEMS

The only way to move beyond SMPs to large massive parallel computers is to abandon the shared memory architecture and move to a *distributed memory* systems, as shown in Figure 1.3. These platforms are essentially clusters of SMPs; namely, a collection of SMP computing platforms linked by

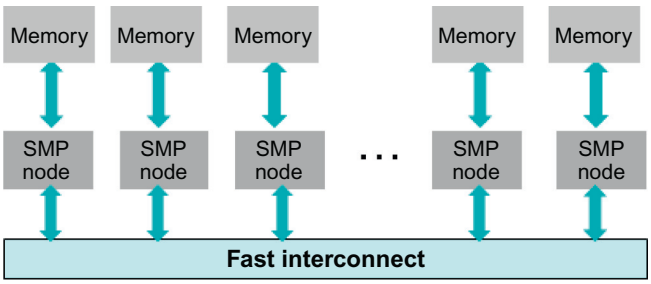


**FIGURE 1.3**

Distributed memory computing platform.

yet another, higher level network interconnect. Each SMP has its own private memory address space. When running a huge parallel application that engages several SMP nodes, CPUs in different SMP nodes communicate by explicit message-passing protocols.

The quality of these platforms as parallel platforms depends essentially on the quality of the network interconnect. Standard clusters integrate commodity networks, while more ambitious supercomputers tuned for extreme massive parallelism—like the IBM BlueGene systems—incorporate proprietary networks that support highly optimized point-to-point and collective communication operations. There is no limit to the scalability of these systems other than those arising from network performance. Today, systems incorporating several hundred thousands of CPUs are in operation.

Large parallel applications consist of independent processes running on different SMPs and communicating with one another via an explicit communication protocol integrated in a programming model, called the *Message Passing Interface* (MPI). Each process acts on its own private data set, and a number of synchronization and data transfer primitives allow them to communicate to other processes the updated data values they need to proceed. Programming these systems is more difficult than programming ordinary SMP platforms because the programmer is in general lacking a global view of the application's data set: data is spread across the different SMP nodes, and each autonomous process accesses only a section of the complete data set. However, on the other hand, MPI is the only programming model that can cope with scalable, massive parallelism.

### 1.2.3 MULTICORE EVOLUTION

Since the early 1970s the sustained evolution of silicon technologies has followed a trend initially underlined by Gordon Moore, one of the Intel founders: *the number of transistors that can be placed inexpensively on an integrated circuit doubles approximately every two years*, and observation that is known under the name of Moore's law. This exponential growth of the transistor count in a chip—arising from the smaller transistor sizes—generated a golden age in computing systems, not only because a given surface of silicon real state could accommodate more transistors, but also because the transistor performance increased with decreasing size: smaller transistors commuted faster—which meant faster chips—and used less working voltages and currents, i.e., less power.

A seminal paper written in 1974 by Robert Dennard and colleagues [4] at the IBM T.J. Watson Research Center, described the scaling rules for obtaining simultaneous improvements in transistor density, switching speeds and power dissipation. These scaling principles—known as Dennard scaling—determined the roadmap followed by the semiconductor industry for producing sustained transistor improvements. Table 1.1, reproduced from Dennard's paper, summarizes the transistor or circuit parameters under ideal scaling. In this table $k$ is the scaling factor for transistor size.

The implications of Dennard scaling are obvious: reducing by a half the transistor size multiplies by 4 the number of transistors in a given area *at constant power dissipation*, and in addition each transistor is operating twice as fast. This is the reason why for more than three decades we enjoyed the benefits of automatic performance enhancements: processor working frequencies (and, therefore, their peak performance) doubled every 18 months. This was a very comfortable situation; it was sometimes sufficient to wait for the next generation of CPU technology to benefit from an increased applications performance. Parallel processing was mainly an issue of choice and opportunity, mostly limited to supercomputing or mission critical applications.

**Table 1.1 Dennard Scaling Results for Circuit Performance**

| Dennard Scaling | |
|---|---|
| **Device or Circuit Parameter** | **Scaling Factor** |
| Device dimension | 1/k |
| Voltage V | 1/k |
| Current I | 1/k |
| Capacitance | 1/k |
| Delay time per circuit VC/I | 1/k |
| Power dissipation per circuit VI | 1/k |
| Power density VI/area | 1 |

This golden age came to an end about 2004, when a power dissipation wall was hit. As transistors reach smaller sizes, new previously negligible physical effects emerge—mainly quantum effects—that invalidate Dennard's scaling. Working voltages essentially stopped shrinking at this time, and today clock speeds are stuck because it is no longer possible to make them work faster. Moore's law still holds: it is always possible to double the transistor count about every 2 years, but shrinking the transistor size does not make them better, as was the case when Dennard scaling applied.

Today, the only benefits of shrinking transistor sizes are disposing of more functions per chip and/or lower costs per function.

For this reason, the last decade has witnessed the multicore evolution. Processor chips are no longer identified with a single CPU, as they contain now multiple processing units called cores. Since clock speeds can no longer be increased, the only road for enhanced performance is the cooperative operation of multiple processing units, at the price of making parallel processing mandatory. Virtually all processor vendors have adopted this technology, and even the simplest laptop is today a SMP platform. Multicore processor chips are now called *sockets*. The present situation can be summarized as follows:

- The number of cores per socket doubles every 2 years.
- Clock speeds tend to mildly decrease. In any case, they are stabilized around a couple of GHz.

It follows that the most general computing platform has the following hierarchical structure, as shown in Figure 1.4:

- Several cores inside a socket
- A few sockets interconnected around a shared memory block to implement a SMP node
- A substantial number of SMP nodes interconnected in a distributed memory cluster

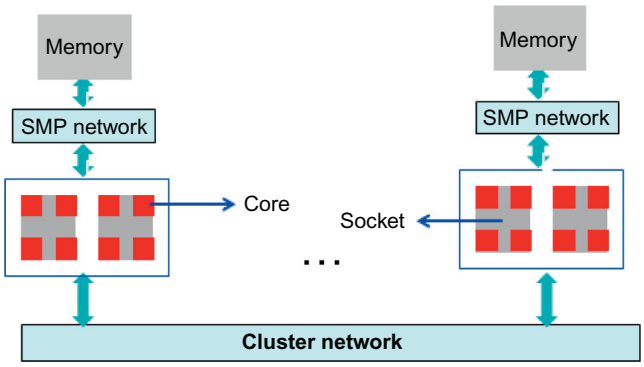


**FIGURE 1.4**

Generic cluster computing platform.

In such a system, three programming models can be implemented:

- Shared memory multithreaded programming inside a SMP node. From a programmer's point of view, the logical view of a SMP node is just a number of virtual CPUs—the cores—sharing a common memory address space. It does not matter whether the different cores are in the same or in different sockets.
- Flat MPI distributed memory programming across cores. In this approach, each core in the system runs a full MPI process, and they all communicate via MPI message passing primitives. It does not matter whether the MPI processes are all in the same or in different SMP nodes.
- A hybrid MPI-Threads model in which each MPI process is internally multithreaded, running on several cores. These MPI processes communicate with one another via the MPI message passing protocol.

This book covers the first programming model discussed above; namely, shared memory application programming on multicore platforms. Chapter 2 presents the different ways in which a multithreaded process can be scheduled and run on these platforms.

## 1.3 MEMORY SYSTEM OF COMPUTING PLATFORMS

A number of programming issues that will be encountered later on require a clear understanding of the operation of a computing platform memory system. As stated, one of the most important performance issues is the *memory wall*; namely, the mismatch between the processor clock and the high latencies involved in each main memory access. The memory system of modern computing platforms is designed to reduce as much as possible the memory wall impact on the application's performance.

### 1.3.1 READING DATA FROM MEMORY

The first observation is that the huge latencies of a few hundreds of processor cycles are a characteristic of the *sdram* technologies used in large *main memory* blocks, typically of the order of 2-4 gigabytes per core in current platforms. Other, faster technologies are available, but it would be prohibitively expensive to use them for main memories of these sizes.

The next observation is about the *locality* of memory accesses. In most applications, access to a memory location is often followed by repeated accesses to the same or to nearby locations. Then, the basic idea is to store recently accessed memory blocks in smaller but faster memory devices. When a memory location is accessed for a read, a whole block of memory surrounding the required data is copied to a *cache memory*, a smaller capacity memory device with a significantly lower latency, based on faster memory technologies. Future read accesses to the same or to nearby memory locations contained in the block will retrieve the data directly from the cache memory in a much more efficient way, avoiding costly main memory access. The memory blocks copied in this way are called *cache lines*.

Cache memories are significantly smaller than the main memory. It is then obvious that, when reads are very frequent, the cache lines will often be overwritten with new data, thereby invalidating previously cached data values. Various algorithms are used to choose the cache lines that are to be reused, with different impacts on performance.

These observations lead to a hierarchical multilevel memory system in which cache lines at one level are copied to a faster, lower level memory device. As long as data is read, it is retrieved from the lowest level cache memory holding a valid data value (we will soon discuss what a valid data value is). Figure 1.5 shows a typical organization of the memory hierarchy, exhibiting two intermediate cache levels—called L1 and L2—between the main memory and the *core registers*, which are the ultrafast memory buffers internal to the executing core, where data is deposited before being processed by the core functional units. Typically, core registers are 4(8) bytes wide for 32(64) bit architectures. It is important to keep in mind that current CPUs are *load-store architectures* in which data is *always* moved to core registers before being processed. Direct manipulation of memory locations does not occur in these architectures.

The L1 caches are rather small but fast memory blocks used both as data caches—to store the most recently accessed data—as well as instruction caches—to store the instructions to be executed by the core. Because they are partly used to store the instructions to be executed, the L1 caches *are never*

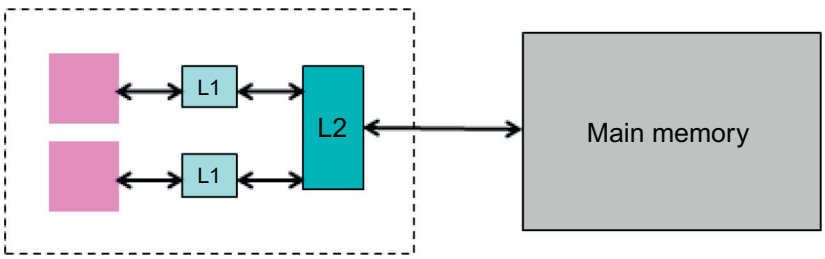


**FIGURE 1.5**

Hierarchical memory system.

*shared*: threads running on different cores have their own, proprietary instruction stack. The L2 caches, instead, only store data, and they may be shared by several cores. In most systems, there is yet another cache level—called L3—between the L2 caches and the main memory, but this is not critical to our discussion.

Typical values of cache lines are 128 (or more) bytes for L2 and 32 bytes for L1. Typical latencies are 4 cycles for the L1 cache, 12 cycles for the L2 cache, and roughly 150-200 cycles for main memory. This memory hierarchy enhances memory performance in two ways. On one hand, it reduces the memory latency for recently used data. On the other hand, it reduces the number of accesses to the main memory, thereby limiting the usage of the network interconnect and the bandwidth demand. Indeed, accesses to L1 and L2 caches do not require network activation because they are part of the processor socket.

### 1.3.2 WRITING DATA TO MEMORY

When a data value is changed in a processor register, the CPU must proceed to update the original main memory location. Typically, the new data value is updated first in the L2 cache, and from there on the network interconnect is activated to update the main memory. The following issues arise every time a new data value is updated in the L2 cache:

- First, the cache memory is no longer coherent with the main memory. How and when the main memory is updated is system dependent, but there is in most cases a time delay: the memory update is not *atomic* (instantaneous). Some systems choose to delay the writes in order to collect several data items in a *write buffer*, and then move them in a unique memory access in order to pay the big latency cost only once.
- Secondly, the updated cache memory is no longer coherent with other L2 caches in other sockets that may contain the old invalid value of the updated variable. The CPU that has performed the update must therefore inform all other CPUs engaged in the application that the relevant cache line is invalid, and that further reads of data in this cache line must get the data values from main memory. This is the *cache coherency issue*.
- Finally, because of the time delays in memory updates mentioned before, those threads that must get updated data values from main memory must know, in one way or another, *when the new updated values are available*. They must make sure that, in performing a read, they recover the last updated value of the target data item. This is the *memory consistency issue*.

These two issues—memory coherency and consistency—are analyzed in more detail in Chapter 7. It is obvious that the cache coherency mechanism requires a persistent communication context among sockets in a SMP platform, and this is the main reason why it is not possible to extend a shared memory SMP architecture into the massive parallel domain. It is just too expensive and unrealistic to try to enforce cache coherency among hundreds or thousands of sockets in a SMP node.

In discussing the performance and the behavior or several real applications in Chapters 13–15, we will have the opportunity to discuss how the memory access patterns impacts the application performance.

## 1.4 PARALLEL PROCESSING INSIDE CORES

The essence of multithreaded programming is the capability of coordinating the activity of several CPUs in the execution of a given application. In principle, whatever happens inside a single core is not directly relevant to multithreaded programming. However, it is extremely relevant to the overall performance of the application, and there are some capabilities of current CPU architectures that must be integrated into the developer's options.

There is an amazing amount of parallel processing inside a core. After all, millions of transistors operate concurrently most of the time. After translating the application code into basic assembler language, hardware, system, and compiler software cooperate to take as much as possible advantage of *instruction level parallelism*. This means running the basic instructions in parallel whenever it is possible to do so while respecting the program integrity.

Instruction-level parallelism is too low level to constitute a direct concern to programmers. But there are other parallel structures that are definitely worth their attention: hyperthreading and vectorization.

### 1.4.1 HYPERTHREADING

Hyperthreading is the CPU capability of *simultaneously* running several threads. This means, very precisely, that the core has the capability of interleaving the execution of instructions arising from different execution streams, while maintaining the program integrity. These different execution streams interleaved in hyperthreading are in general targeting different data sets, and Chapter 2 will explain how the multithreaded execution environment is organized in such a way as to make this possible.

Hyperthreading should not be confused with another feature also to be discussed in the next chapter: a core can service several threads by running them one at a time, in a round robin fashion, allocating CPU time slices to all of them. This is a very general operating system feature that has always existed in multithreading, enabling the possibility of over-committing threads on a given core. In this case, different threads access *successively* the CPU cycles. In a hyperthreading operation, they are *simultaneously* sharing CPU cycles.

The main reason for introducing hyperthreading capabilities in modern CPU architectures is to make better use of the CPU cycles, e.g., in accumulating efforts to beat the memory wall. If a thread needs to wait for memory data, there may be tens or hundreds of cycles wasted doing nothing. These cycles can then be used to run other thread instructions. Hyperthreading has different impacts on different architectures. In general-purpose CPUs, hyperthreading hides occasional latencies, and its impact on performance strongly depends on the code profile. However, there are more specialized architectures—like the IBM BlueGene platforms, or Intel Xeon Phi coprocessor discussed in the following—where for different reasons hyperthreading is required to benefit from the full single-core performance.

The name *hardware threads* is used to refer to the number of threads that a core can simultaneously execute.

### 1.4.2 VECTORIZATION

Vectorization is another parallel processing technique that enhances the single core performance. In a multithreaded environment, different threads execute different instruction streams acting on different data sets. This parallel pattern is called MIMD, meaning Multiple Instruction, Multiple Data.

Vectorization, instead, implements a totally different parallel pattern called SIMD—Single Instruction, Multiple Data—in which a single instruction operates on several data items in one shot, as is explained in detail below.

Let us consider a simple example, the execution of the operation a += b; where a and b are vectors of doubles of size N. This corresponds to the execution of the loop:

```
double a[N], b[N];
...
for(int n=0; n<N; n++)
   {
   a[n] += b[n];
   }
```

**LISTING 1.1**

Executing a vector operation

Figure 1.6 shows the internal CPU registers, where the target data for operations is stored. The scalar registers are 64 bits wide, and can hold a double. In the default scalar mode of operation, the vector addition is computed by adding one component at a time. The values of a[n] and b[n] are loaded into Ra and Rb registers, respectively. Then, the operation Ra = Ra+Rb is performed, and the value of Ra is copied to a[n].

Cores with SIMD capabilities have wide vector registers that can hold several vector components. In the Intel Sandy Bridge processor, vector registers are 256 bits wide, holding either four doubles or eight floats, and we speak in this case of four or eight SIMD lanes, respectively. Vector instructions can act simultaneously in one shot on all the SIMD lanes, boosting the floating-point performance. In vector mode, the loop above is computed by loading in RVa and RBv a block of four a[] and b[] components, and acting simultaneously on all of them.

Implementing wide vector registers in the core architecture is not sufficient. The capability of loading the wide vector registers as fast as the scalar registers is also required, which in turn demands an

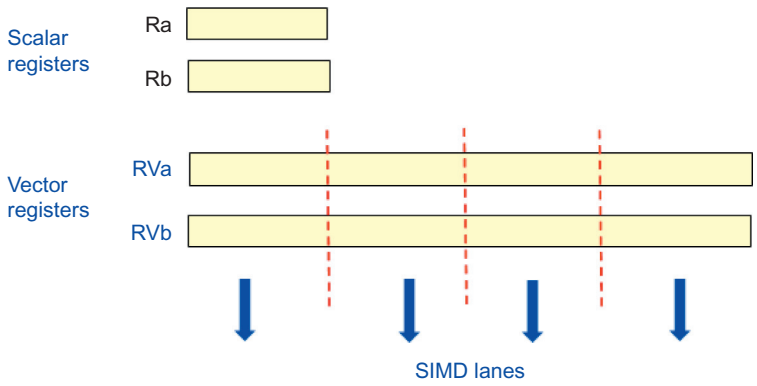


**FIGURE 1.6**

Scalar versus vector processing.

enhanced communication bandwidth between the core and the L2 cache. When the code profile is well adapted, vectorization can provide significant performance enhancements in computationally bound applications. Vectorization is re-examined in more detail in Chapter 10, when discussing the OpenMP 4.0 new vectorization directives. In Chapters 13–14, the impact of vectorization on real application examples is assessed.

## 1.5 EXTERNAL COMPUTATIONAL DEVICES

The last few years have witnessed impressive development of external computational devices that connect to a socket via the standard network interfaces for external devices. They act as a co-processor executing code blocks offloaded from the CPU cores, boosting the execution performance for suitable computationally intensive code blocks—called *kernels*—in the application. They are seen from the host CPU as another computational engine available in the network.

External computational devices are shown in Figure 1.7. They all have an internal device memory hierarchy as well as a large number of cores for computation. There are today two very different kinds of external devices: GPUs (Graphical Processing Units)—accelerator devices capable of executing basic computational kernels with very high performance and very low power consumption—and the Intel's Xeon Phi coprocessor.

### 1.5.1 GPUs

Initially, graphics accelerator cards—called GPUs, for Graphical Processing Units, used for rendering visualization images—were occasionally used to perform other computations that could be rephrased in terms of the graphical programming API. Working on a two-dimensional image—a two-dimensional array of pixels—is a highly parallel affair, because most of the time sets of pixels can be simultaneously updated. Graphics hardware and software are strongly data parallel. GPUs map the graphics algorithms to a large set of independent execution streams (threads) all executing the same operations on different sectors of the target data set.

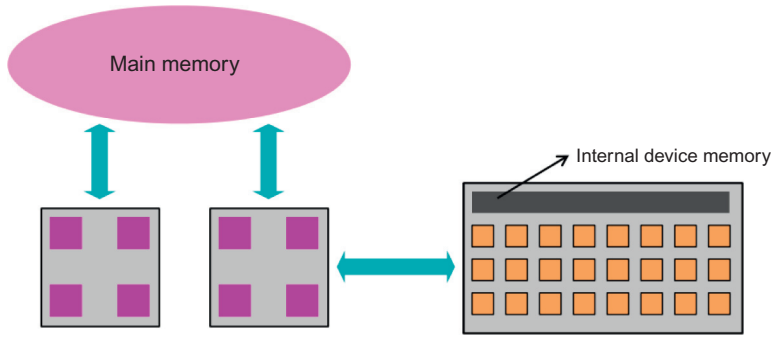


**FIGURE 1.7**

Generic external computational device.

A major step was taken in 2007 when NVIDIA Corporation realized the interest in allowing programmers to think of a GPU as a processor. Besides a substantial number of hardware improvements to reduce the gap between graphics accelerators and more standard computational engines, the CUDA C/C++ compiler, libraries and runtime software were introduced, enabling programmers to access the underlying data parallel computation model and develop applications adapted to this model. Graphics APIs were no longer needed to benefit from the GPU computing environment. Since then, the progress accomplished at the hardware and software levels is quite remarkable. Today, GPUs are very powerful computational engines, and part of their success is due to the fact that the computing cores, being very lightweight, beat the power dissipation wall providing an excellent ratio of computing power (flops) per dissipated watt.

When discussing GPU accelerators, the concept of *heterogeneous computing* is important, because the GPU computing engine's design requirements are different from those guiding CPU design. The CPU design is *latency driven*: making a single thread of execution as efficient as possible. A substantial amount of the silicon real state in a CPU is devoted to control logic—needed to implement instruction-level parallelism or to provide the low-level interfaces required to run an operating system—or to cache memories needed to optimize memory accesses. Further, CPUs have a limited number of cores. GPUs, instead, are *throughput driven*. They have today tens of thousands of *very lightweight* cores, each one involving very limited silicon real state because they are not intended to run an operating system. Indeed, GPU cores execute a restricted number of instructions for computation and synchronization with other cores. The code blocks they execute are offloaded from the application running in the main computing platform they are connected to. GPU cores are in absolute terms substantially less efficient than CPU cores, but the performance impact comes from their important number mentioned above, which provides a huge potential throughput. An application's performance strongly depends on its capability to take advantage of the important amount of potential parallelism offered by the GPU. The interest of heterogeneous computing lies in the possibility of executing parts of the application in a standard platform and offloading to the GPU the execution of code blocks adapted to its architecture.

The fundamental performance-limiting factor in GPUs is, again, the memory wall. Current GPUs have no direct access to the main memory, and the target data set must be offloaded to the GPU together with the code to be run. Then, output results must be returned to the master code running in the main platform. In this context, the real achieved sustained performance strongly depends on the application profile. Excellent efficiency is obtained only if a substantial amount of computation can be performed with a limited amount of data transfers to and from the host CPU. GPU architectures are nevertheless evolving very fast, and the possibility of directly exchanging data between different GPUs is available today. Substantial improvements are expected in the future concerning their access performance to the platform main memory.

GPU programming requires specific programming environments (CUDA, OpenCL, OpenACL) capable of producing and offloading code and data to the GPU. These programming environments have matured greatly in the last few years, facilitating the dissemination and adoption of GPU programming. An important recent evolution is the recent OpenMP 4.0 extensions incorporating directives for offloading data and code blocks to accelerators, which will be reviewed in Chapter 10.

Code executed in GPUs must conform to the target architecture capabilities. Threads running on the GPU cores *are not* the general-purpose threads, running on a general-purpose CPU, which are the main subject of this book.

Today there is vast literature on GPUs. A good reference providing a broad coverage of this subject is D. Kirk and W. Hwu book, "Programming Massively Parallel Processors" [5]. Other useful references are "The CUDA Handbook" by N. Wilt [6] and "CUDA by Example" by J. Sanders and E. Kandrot [7].

### 1.5.2 INTEL XEON PHI

Intel's Xeon Phi chip is very different from GPUs. Like GPUs, it must be connected to a CPU host and can communicate with it. However, *Intel Xeon Phi is really a complete SMP node*. Rather than integrating hundreds or thousands of very lightweight cores that execute a limited instruction set, Intel Xeon Phi incorporates today 60 normal CPU cores that all execute a full and improved x86 instruction set and run a complete Linux operating system. The Xeon Phi chip integrates today 8 GB of shared memory (extension to 16 GB will soon be available). In fact, the chip is just a huge SMP node that can even work as a standalone computing platform. The only point to be underlined is that the Xeon Phi cores only access the internal chip shared memory. The access to the main memory of the computing platform goes, as in the GPU case, through the external device network interfaces that connect to the master CPU in the computing platform.

Since the Intel Xeon Phi is an ordinary SMP node, standard programming models are used to compile and run applications. All the distributed memory (MPI) and shared memory programming environments (Pthreads, OpenMP, TBB) we will be discussing in this book operate "as such" inside the Intel Xeon Phi. If needed, these programming environments are completed with a few directives that offload code blocks for execution on the coprocessor, or vice-versa. In fact, the coprocessor has three modes of execution:

- The main application runs on the master CPU, and specific code blocks are offloaded for execution in the coprocessor
- The main application runs on the coprocessor, and specific code blocks are offloaded for execution in the master CPU. This is very useful, for example, to perform computationally intensive operations on the Intel Xeon Phi, while offloading I/O operations for execution on the master CPU.
- Standalone SMP node: the whole application is run on the Intel Xeon Phi.

A very useful and complete reference to the Intel Xeon Phi coprocessor is the "Intel Xeon Phy Coprocessor" book by J. Jeffers and J. Reinders [8]. The online Intel Xeon Phy documentation can also be consulted [9].

## 1.6 FINAL COMMENTS

This chapter has reviewed the basic technological concepts related to shared memory programming. A somewhat more detailed discussion is available in the early chapters of D. Gove book "Multicore Application Programming" [10].

Threads have been with us for a very long time. All current operating systems support multi-threading, and threads are needed—even for programmers not concerned by parallel programming—to enhance the performance of interactive applications. We all see every day Web pages where several animations are running without interruption. Each one of these animations is a Java applet running as a thread. These applets are just sequential flows of control executing inside the application, which is the

browser himself. When the browser runs on a single processor system, the operating system creates the illusion that the different animations are simultaneous in time. In fact, the different applets get, one at a time in a round-robin fashion, access to the processor for a short interval of time called a *time slice*. This time slice is very short—typically a few milliseconds—as compared to the time scale for human perception.

The multicore evolution, and the fact that the number of cores per processor chip will keep increasing in the foreseeable future, has neatly enhanced the impact of multithreading. This software technology is today the only way to take full advantage of the enhanced performance of current computing systems.