CHAPTER 2

Overview of Platform Architectures

In order to optimize software you need to understand hardware. In this chapter we give you a brief overview of the typical system architectures found in the high-performance computing (HPC) today. We also introduce terminology that will be used throughout the book.

Performance Metrics and Targets

The definition of *optimization* found in Merriam-Webster's *Collegiate Dictionary* reads as follows: "an act, process, or methodology of making something (as a design, system, or decision) as fully perfect, functional, or effective as possible."[1] To become practically applicable, this definition requires establishment of clear success criteria. These objective criteria need to be based on quantifiable metrics and on well-defined standards of measurement. We deal with the metrics in this chapter.

Latency, Throughput, Energy, and Power

Let us start with the most common class of metrics: those that are based on the total time required to complete an action–for example, the time it takes for a car to drive from the start to the finish on a race track, as shown in Figure 2-1. Execution (or *wall-clock*) time is one of the most common ways to measure application performance: to measure its *runtime* on a specific system and report it in seconds (or hours, or sometimes days). In this context, the time required to complete an action is a typical *latency metric*.

Figure 2-1. *Runtime: observed time interval between the start and the finish of a car on a race track*

The runtime, or the period of time from the start to the completion of an application, is important because it tells you how long you need to wait for the results. In networking, *latency* is the amount of time it takes a data packet to travel from the source to the destination; it also can be referred to as the *response time*. For measurements inside the processor, we often use the term *instruction latency* as the time it takes for a machine instruction entering the execution unit until results of that instruction are available—that is, written to the register file and ready to be used by subsequent instructions. In more general terms, *latency* can be defined as the observed time interval between the start of a process and its completion.

We can generalize this class of metrics to represent more of a general class of *consumable resources*. Time is one kind of a consumable resource, such as the time allocated for your job on a supercomputer. Another important example of a consumable resource is the amount of *electrical energy* required to complete your job, called *energy to solution*. The official unit in which energy is measured is the *joule*, while in everyday life we more often use watt-hours. One watt-hour is equal to 3600 joules.

The amount of energy consumption defines your electricity bill and is a very visible item among operating expenses of major, high-performance computing facilities. It drives demand for optimization of the energy to solution, in addition to the traditional efforts to reduce the runtime, improve parallel efficiency, and so on. Energy optimization work has different scales; going from giga-joules (GJ, or 10^9 joules) consumed at the application level, to pico-joules (pJ, or 10^{-12} joules) per instruction.

One of the specific properties of the latency metrics is that they are *additive, so* that they can be viewed as a cumulative sum of several latencies of subtasks. This means that if the application has three subtasks following one after another, and these subtasks take times T_1, T_2 and T_3, respectively, then the total application runtime is $T_{app} = T_1 + T_2 + T_3$.

Other types of metrics describe the amount of work that can be completed by the system per unit of time, or per unit of another consumable resource. One example of car performance would be its speed defined as the distance covered per unit of time; or of its fuel efficiency, defined as the distance covered per unit of fuel—, such as miles per gallon. We call these metrics *throughput metrics*. For example, the number of instructions per second (IPS) executed by the processor, or the number of floating point operations per second (FLOPS) are both throughput metrics. Other widely used metrics of this class are memory bandwidth (reaching tens and hundreds of gigabytes per second these days), and network interconnection throughput (in either bits per second or bytes per second). The unit of power (watt) is also a throughput metric that is defined as energy flow per unit of time, and is equal exactly to 1 joule per second.

You may encounter situations where throughput is described as the inverse of latency. This is correct only when both metrics describe the same process applied to the same amount of work. In particular, for an application or kernel that takes one second to complete 10^9 arithmetic operations on floating point numbers, it is correct to state that its throughput is 1 GFLOPS (gigaFLOPS, or 10^9 FLOPS).

However, very often, especially in computer networks, latency is understood as the time from the beginning of the packet shipment until the first data arrives at the destination. In this context, latency will not be equal to the inverse value of the throughput. To grasp why this happens, compare sending a very large amount of data (say, 1 terabyte (TB), which is 10^{12} bytes) using two different methods[2]:

1. Shipping with overnight express mail
2. Uploading via broadband Internet access

The overnight (24-hour) shipment of the 1TB hard drive has good throughput but lousy latency. The throughput is $(1 \times 10^{12} \times 8)$ bits / $(24 \times 60 \times 60)$ seconds = about 92 million bits per second (bps), which is comparable to modern broadband networks. The difference is that the overnight shipment bits are delayed for a day and then arrive all at once, but the bits we send over the Internet start appearing almost immediately. We would say that the network has much better latency, even though both methods have approximately the same throughput when considered over the interval of one day.

Although high throughput systems may have low latency, there is no causal link. Comparing a GDDR5 (Graphics Double Data Rate, version 5) vs. DDR3 (Double Data Rate, type 3) memory bandwidth and latency, one notices that systems with GDDR5 (such as Intel Xeon Phi coprocessors) deliver three to five times more bandwidth, while the latency to access data (measured in an idle environment) is five to six times lower than in systems with DDR3 memory.

Finally, a graph of latency versus load looks very different from a graph of throughput versus load. As we will see later in this chapter, memory access latency goes up exponentially as the load increases. Throughput will go up almost linearly at first, then levels out to become nearly flat when the physical capacity of the transport medium is saturated. Simply by looking at a graph of test results and keeping those features in mind, you can guess whether it is a latency graph or a throughput graph.

Another important concept and property of a system or process is its degree of concurrency or parallelism. *Concurrency* (or *degree of concurrency*) is defined as the number of work items that can potentially be performed simultaneously. In the example illustrated by Figure 2-2, where three cars can race simultaneously, each on its own track, we would say this system has concurrency of 3. In computation, an example of concurrency would be the simultaneous execution of multiple, structurally different application "threads" by a multicore processor. Presence of concurrency is an intrinsic property of any modern high-performance system. Processes running on different machines of a cluster form a common system that executes application code on multiple machines at the same time. This, too, is an example of concurrency in action.

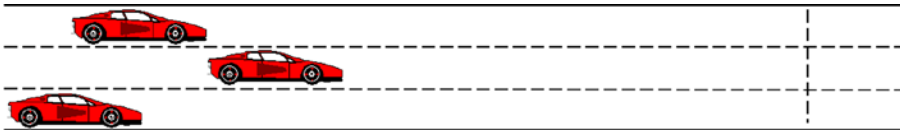

Figure 2-2. *A system with the degree of concurrency equal to 3*

Cantrill and Bonwick describe three fundamental ways of using concurrency to improve application performance.[3] At the same time, these three ways represent the typical optimization targets for either latency or throughout metrics:

- *Increase throughput:* By executing multiple tasks concurrently, the general system throughput can be increased.
- *Reduce latency:* A given amount of work is completed in shorter time by dividing it into parts that can be completed concurrently.
- *Hide latency:* Multiple long-running tasks are executed in parallel by the underlying system. This is particularly effective when some tasks are blocked (for example, if they must wait upon disk or network I/O operations), while others can proceed independently.

Peak Performance as the Ultimate Limit

Every time we talk about performance of an application running on a machine, we try to compare it to the maximum attainable performance on that specific machine, or *peak performance* of that machine. The ratio between the achieved (or measured) performance and the peak performance gives the *efficiency* metric. This metric is often used to drive the performance optimization, for an increase in efficiency will also lead to an increase in performance according to the underlying metric. For example, efficiency for the wall-clock time is the fraction of time that is spent doing useful work, while efficiency for throughout is a measure of useful capacity utilization.

Consider the example of how to quantify efficiency for a network protocol. Network protocols normally require each packet to contain a header and a footer. The actual data transmitted in the packet is then the size of the packet minus the protocol overhead. Therefore, efficiency of using the network, from the application point of view, is reduced from the total utilization according to the size of the header and the footer. For Ethernet, the frame payload size equals 1536 bytes. The TCP/IP header and footer take 40 bytes extra. Hence, efficiency here is equal to 1536 / 1576 × 100, or 97.5 percent.

Understanding the limitations of maximum achievable performance is an important step in guiding the optimization process: the limits are always there! These limits are driven by physical properties of the available materials, maturity of the technology, or (trivially) the cost. Particularly, the propagation of signals along the wires is limited by the speed of light in the respective material. Thus, the latency for completing any work using electronic equipment will always be greater than zero. In the same way, it is not possible to build an infinitely wide highway, for its throughput will always be limited by the number of lanes and their individual throughputs.

Scalability and Maximum Parallel Speedup

The ability to increase performance by using more resources in parallel (for example, more processors) is called *scalability*. The basic approach in high-performance computing is to use many computational resources in parallel to solve one problem, and to add still more resources if higher performance is required. Scalability analysis indicates how efficient an application is using the increasing numbers of parallel computing elements, such as cores, vector units, memory, or network connections.

Increase in performance before and after addition of the resources is called *speedup*. When talking about throughput-related metrics, speedup is expressed as the ratio of the throughput after addition of the resources versus the original throughput. For latency metrics, speedup is the ratio between the original latency and the latency after addition of the resources. This way speedup is always greater than 1.0 if performance improves. If the ratio goes below 1.0, we call this negative speedup, or simply *slowdown*.

Amdahl's Law, also known as Amdahl's argument,[4] is used to find the maximum expected improvement for an entire application when only a part of the application is improved. This law is often used in parallel computing to predict the theoretical maximum speedup that can be achieved by adding multiple processors. In essence, Amdahl's Law says that speedup of a program using p processors in parallel is limited by the time needed for the nonparallel fraction of the program (f), according to the following formula:

$$Speedup \leq \frac{p}{1+f\cdot(p-1)}$$

where f takes values between 0 and 1.

As an example, think about an application that needs 10 hours when running on a single processor core, where a particular portion of the program takes two hours to execute and cannot be made parallel (for instance, since it performs sequential I/O operations). If the remaining 8 hours of the runtime can be efficiently parallelized, then regardless of how many processors are devoted to the parallelized execution of this program, the minimum execution time cannot be less than those critical 2 hours. Hence, speedup is limited by at most five times (usually denoted as 5x). In reality, even this 5x speedup goal is not attainable, since infinite parallelization of code is not possible for the parallel part of the application. Figure 2-3 illustrates Amdahl's law in action. If the parallel component is made 50 times faster, then the maximum speedup with 20 percent of time taken by the serial part will be equal to 4.63x.

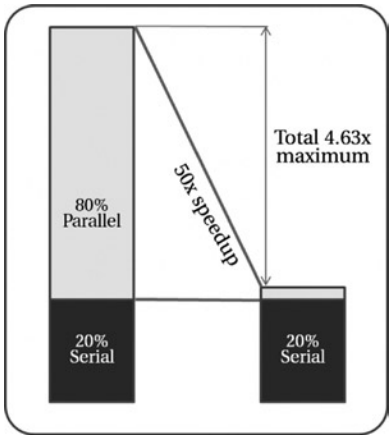

Figure 2-3. *Illustration of Amdahl's Law*

It may be depressing to realize that the maximum possible speedup will be limited by something you can't improve by adding more resources. Even so, consider the same speedup problem from another angle: what happens if the amount of work in the parallelizable part of the execution can be increased?

If the relative share of time taken by the serial portion of the application remains unchanged with the increase of the workload size, there is no inherent speedup factor available, and as illustrated in Figure 2-4 (left), Amdahl's Law still works. However, John Gustafson observed that there was significant speedup opportunity available if the serial component shrank in size relative to the parallel part as the amount of data processed by the application (and consequently the amount of computation) expanded.[5]

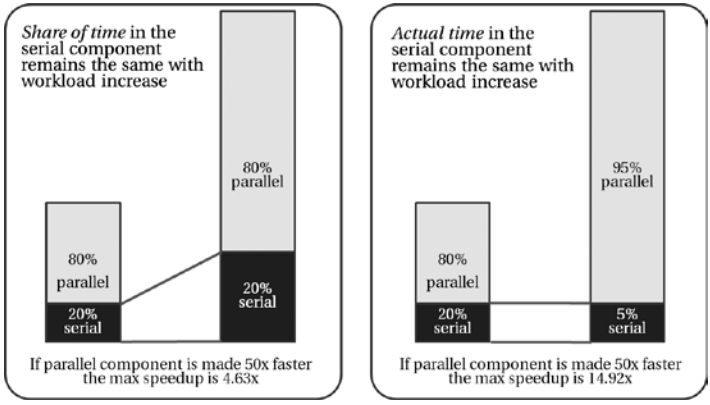

Figure 2-4. *Illustration of Gustafson's observation*

This observation leads to two kinds of scalability metrics:

- *Strong scaling*: How performance varies with the number of computing elements for a fixed *total problem size*. In strong scaling, *perfect scaling* (i.e., when performance improves linearly) is achieved when speedup is equal to the number of computing elements involved.
- *Weak scaling*: How performance varies with the number of computing elements for a fixed problem size *per processor*, and additional computing elements are used to solve a larger total problem. In the case of weak scaling, perfect scaling is achieved if the runtime remains constant while the workload is increased proportionally to the number of computing elements involved.

Bottlenecks and a Bit of Queuing Theory

Performance analysis is a process of identifying bottlenecks and removing them, with the objective of increasing overall application performance. Certain parts of the application that limit performance of the entire application are called *performance bottlenecks*. The significance of the term *bottleneck* can be illustrated with the same car metaphor that we have used before (see Figure 2-5). When there is a toll gate on the road that can process only one car at a time, the rate at which cars will pass along the highway (that is, highway throughput) is limited by the width of the toll gate, irrespective of how many more lanes are on the road before and after it. In other words, the toll gate is a bottleneck. By increasing the width of the toll gate, it is possible to increase the rate of cars on the highway.

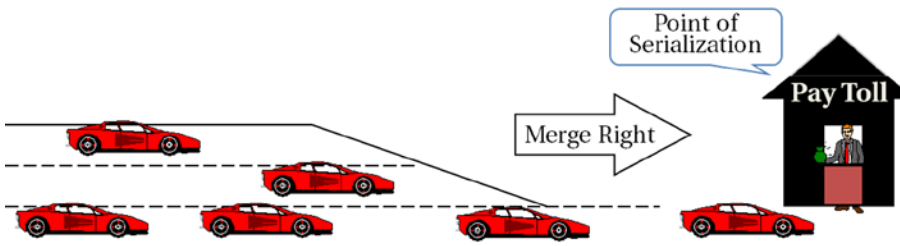

Figure 2-5. *Bottlenecks on the road are commonly known as traffic jams*

As shown in Figure 2-5, bottlenecks can create traffic jams on the highway. Using the terminology of queuing theory,[6] we are talking about the toll gate as a single service center. Customers (here, cars) arrive at this service center at a certain rate, called *arrival rate* or *workload intensity*. There is also certain duration of time required to collect money from each car, which is referred to as *service demand*. For specific parameter values of the workload intensity and the service demand, it is possible to analytically evaluate this model and produce performance metrics, such as *utilization* (proportion of time when the server point is busy), *residence time* (average time spent at the service center by a customer), *length of the queue* (average number of customers waiting at the service center), and *throughput* (rate at which customers depart from the service center).

This approach is widely used by *queuing network modeling*, where a computer system is represented as a network of queues—that is, a collection of service centers that represent system resources and customers who represent users or transactions. This model provides a framework for gathering, organizing, evaluating, and understanding information about the computer system, as well as for identifying possible bottlenecks and testing ideas for system improvement. Such models are widely used for quantitative analysis during computer system design and the application development process.

Roofline Model

Amdahl's law and the queuing network models both offer "bound and bottleneck analysis," and they work quite well in many cases. However, both complexity and the level of concurrency of modern high-performance systems keep increasing. Indeed, even smartphones today have complex multicore chips with pipelines, caches, superscalar instruction issue, and out-of-order execution, while the applications increasingly use tasks and threads with asynchronous communication between them. Quantitative queuing network models that simulate behavior of very complex applications on modern multicore and heterogeneous systems have become very complex. At the same time, the speed of microprocessor development has outpaced the speed of the memory evolution; and in most cases, specifically in high-performance computing, the bandwidth of the memory subsystem is often the main bottleneck.

In search of a simplified model that would relate processor performance to the off-chip memory traffic, Williams, Waterman, and Patterson observed that that "the Roofline [model] sets an upper bound on performance of a kernel depending on the kernel's operational intensity."[7] The *Roofline model* subsumes two platform specific ceilings in one single graph: floating-point performance and memory bandwidth. The model, despite its apparent simplicity, provides an insightful visualization of the system bottlenecks. Peak floating point and memory throughput performances can usually be found from the architecture specifications. Alternatively, it is possible to find sustained memory performance by running the STREAM benchmark.[8]

Figure 2-6 shows a roofline plot for a platform with peak performance $P = 518.4$ GFLOPS (such as a dual-socket server with Intel Xeon E5-2697 v2 processors) and bandwidth $B = 101$ GB/s (gigabytes per second) attainable with the STREAM TRIAD benchmark on this system.

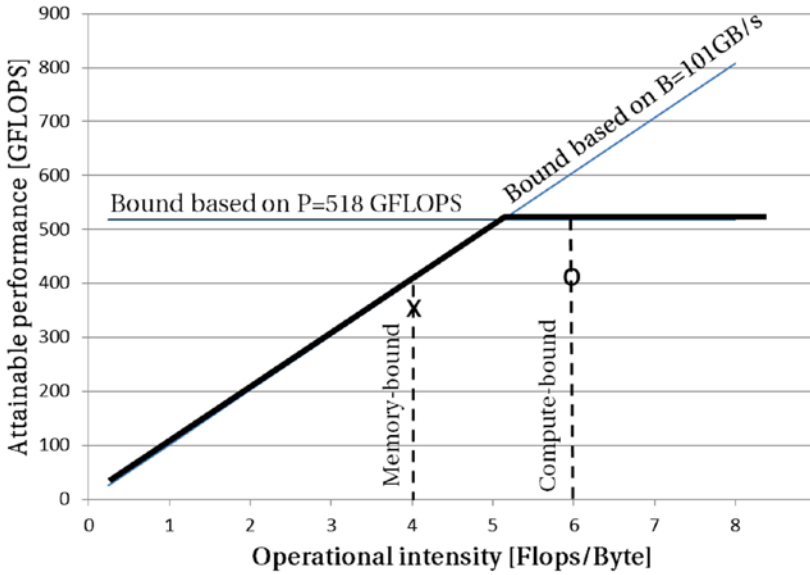

Figure 2-6. *Roofline model for dual Intel Xeon E5-2697 v2 server with DDR3-1866 memory*

The horizontal line shows peak performance of the computer. This is a hardware limit for this server. The *X*-axis represents amount of work (in number of floating point operations, or Flops) done for every byte of data coming from memory: Flops/byte (here, "Flops" stands for the plural of "Flop"–the number of floating point operations, rather than FLOPS, which is Flops per second). And the *Y*-axis represents gigaFLOPS (10^9 FLOPS), which is a throughput metric showing the number of floating point operations executed every second (Flops/second, or FLOPS). With that, taking into account that $\text{bytes/second} = \frac{\text{Flops/second}}{\text{Flops/byte}}$, the memory throughput metric gigabytes/second is represented by a line of unit slope in Figure 2-6. Thus, the slanted line shows the maximum floating point performance that the memory subsystem can support for the given operational intensity. The following formula drives the two performance limits in the graph shown in Figure 2-6:

$$Attainable\ performance[GLOPS] = \min\left\{\begin{array}{c} Peak\ floating\ point\ performance, \\ Peak\ memory\ bandwidth \times Operational\ intensity \end{array}\right\}$$

The horizontal and diagonal lines form a kind of roofline, and this gives the model its name. The roofline sets an upper bound on performance of a computational kernel depending on its operational intensity. Improving performance of a kernel with operational intensity of 6 Flops/byte (shown as the dotted line marked by "O" in the plot) will hit the flat part of the roof, meaning that the kernel performance is ultimately

compute-bound. For another kernel (the one marked by "X"), any improvement will eventually hit the slanted part of the roof, meaning its performance is ultimately memory bound. The roofline found for a specific system can be reused repeatedly for classifying different kernels.

Performance Features of Computer Architectures

We have discussed the major types of performance characteristics and approaches to estimate maximum attainable performance. Let's turn to a discussion of where the potential performance increases can come from.

Increasing Single-Threaded Performance: Where You Can and Cannot Help

We will refer to the basic execution context as a *thread*—a sequence of machine instructions executed by a processor core. Typically, a thread is the smallest context of execution that is independently managed by the operating system (OS). A thread can be granted a processor core to execute instructions on, or it can be put to sleep to free execution resources for other threads in a queue. Under Linux OS, the most widely used operating system in HPC these days,[9] kernel threads and processes are the same entity: simply a runable task. Later, when we talk about hybrid programing, we will want to distinguish processes and threads. But for now let us leave them as a software thread or task, understanding that at any given moment each processor core executes instructions from a single task. Making these instructions run faster is the essence of application optimization.

Performance of a single thread can be defined by number of instructions executed per second (IPS) and calculated as a product of two values (IPS = CPS × IPC):

1. Number of processor clock *cycles per second* (CPS). It is more often called *processor clock frequency*, or simply frequency, and is measured in Hertz (Hz), or for most processors in gigaHertz (GHz), which is 10^9 Hertz.

2. Number of instructions executed per processor clock tick, *instructions per cycle* (IPC).

An application usually cannot do anything about the processor frequency: it is something defined at the manufacturing time and considered fixed or at least not directly changeable when an application is running. In contrast, the IPC is a function of both the processor microarchitecture and your application. The microarchitecture is an internal implementation of the processor. Very simple microarchitectures can execute a maximum of only one instruction per cycle; they are called *scalar*. More sophisticated ones can execute concurrently several instructions at every clock cycle and are known as *superscalar*.

The ability of a processor to produce results for several instructions in parallel is a very important first step toward achieving greater application performance. Since processors have reached the limit of the affordable heat dissipation (that happens around 2.5 to 3.5 GHz, depending on complexity of the chip), the frequency of modern processors does not grow as fast as required to deliver new levels of performance to

demanding customers. Superscalar microarchitectures that are predominant among high-performance focused processors these days provide a much needed solution to the frequency problem. Modern x86 superscalar processors (such as the Intel Core family) can complete up to four instructions per cycle, so it would be as if the frequency was effectively increased four times in a scalar processor. This book was written using a computer with 2.5 GHz Intel Core i5 processor. If it were written on a scalar processor, such as the older Intel 486, the processor would need to run at approximately 10 GHz to be equal in peak performance.

Superscalar execution provides a great way to improve application performance. However, it is to a large extent simply a capability of the processor that needs to be exploited to yield real benefit in application performance. When we talk about microarchitecture optimization in Chapter 7, we review cases when a superscalar processor does not execute as many instructions as it could, and how to fix that. But before we go any further, it is important to note that very often during the optimization process we use a multiplicative inverse of IPC called *CPI*, or *clocks per instruction* . With some simplification we can use the relationship CPI = 1 / IPC without losing many details. Many performance profiling tools (such as Intel VTune Amplifier XE,[10] discussed in greater details in Chapters 6 and 7) use CPI instead of IPC to make it easier to correlate observed CPI metrics with table data on latencies that are traditionally provided for each instruction in processor cycles.

It is important to familiarize yourself with both metrics and be able to assess their values. For example, when a profiler tells you that the average CPI is equal to 2 (meaning it takes two cycles on average to complete every instruction), that means IPC is equal to 0.5 (or one instruction completed every two processor cycles). This level of performance would be rather bad for a modern processor that can (theoretically) reach 4 IPC (delivering results for four instructions every cycle) and the best achievable average CPI of 0.25. Luckily, such an application or piece of code under consideration provides great opportunity for optimization.

Process More Data with SIMD Parallelism

Another way to increase performance of each thread is to look at the data being processed. So far we have only discussed the limit of each processor core with respect to the instructions, but not with respect to the data each instruction works with. The next natural way to optimize an application execution is to let each instruction deal with more than one element of data at a time. Michael J. Flynn gave this approach the name SIMD, standing for Single Instruction Multiple Data single instruction, multiple data.[11] As it obviously follows from its name, in this approach a compute instruction produces results for multiple elements of data using the same instruction on those multiple elements. As illustrated in Figure 2-7, the addition symbol + simultaneously produces results for four elements of the arrays `a` and `b`. To execute this way, elements of the arrays are packed into vectors of length 4 with the operation applied to each separate element pair concurrently.

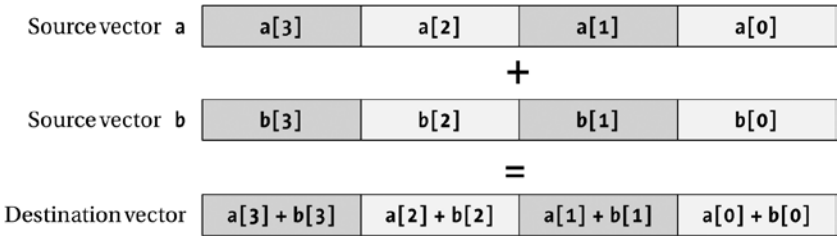

Figure 2-7. *SIMD approach: single instruction produces results for several data elements simultaneously*

Following this principle, the SIMD vector instruction sets implement not only basic arithmetic operations (such as additions, multiplications, absolute values, shifts, and divisions) but also many other useful instructions present in nonvectorized instruction sets. They also implement special operations to deal with the contents of the vector registers—for example, any-to-any permutations—and gather instructions that are useful for vectorized code that accesses nonadjacent data elements.

SIMD extensions for the x86 instruction set were first brought into the Intel architecture under the Intel MMX brand in 1996 and were used in Pentium processors. MMX had a SIMD width of 64 bits and focused on integer arithmetic. Thus, two 32-bit integers, or four 16-bit integers (as type `short` in C), or eight 8-bit integer numbers (C type `char`), could be processed simultaneously. Note also that the MMX instruction set extensions for x86 supported both signed and unsigned integers.

New SIMD instruction sets for x86 processors added support for new operations on the vectors, increased the SIMD data width, and added vector instructions to process floating point numbers much demanded in HPC. In 1999, SIMD data width was increased to 128 bits with SSE (Streaming SIMD Extensions), and each SSE register (called `xmm`) was able to hold two double precision floating point numbers or two 64-bit integers, four single precision floats or four 32-bit integers, eight 16-bit integers or 16 single-byte elements.

In 2008, Intel announced doubling of the vector width to 256 bits in Intel AVX (Advanced Vector eXtensions) instruction set. The extended register was called `ymm`. The `ymm` registers can hold twice as much data as the SSE's `xmm` registers. They support packed data types for modern x86 processor cores (for instance, in the fourth-generation Intel Core processors with microarchitecture, codenamed Haswell), as shown in Figure 2-8.

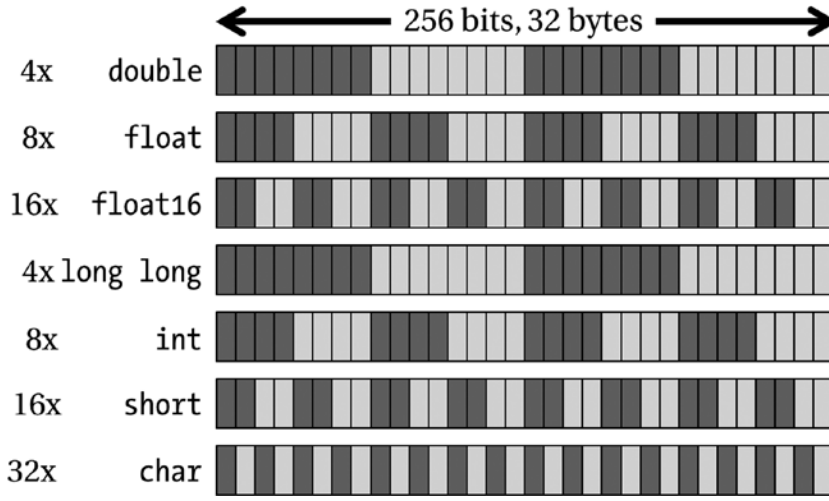

Figure 2-8. *AVX registers and supported packed data types*

The latest addition to Intel AVX, announced in 2013, includes definition of Intel Advanced Vector Extensions 512 (or AVX-512) instructions. These instructions represent a leap ahead to the 512-bit SIMD support (And guess what? The registers are now called `zmm`). Consequently, up to eight double precision or 16 single precision floating point numbers, or eight 64-bit integers, or 16 32-bit integers can be packed within the 512-bit vectors. Figure 2-9 shows the relative sizes of SSE, AVX, and AXV-512 SIMD registers with highlighted packed 64-bit data types (for instance, double precision floats).

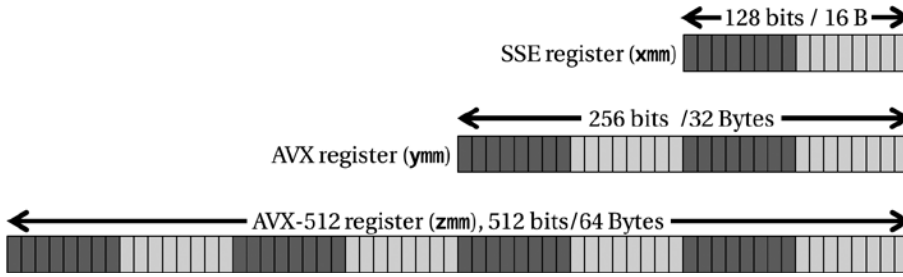

Figure 2-9. *SSE, AVX, and AVX-512 vector registers with packed 64-bit numbers*

Now that you're familiar with the important concepts of SIMD processing and superscalar microarchitectures, the time has come to discuss in greater detail the FLOPS (floating point operations per second) metric, one of the most cited HPC performance metrics. This measure of performance is widely used as a performance metric in the field of scientific computing where heavy use of calculations with the floating point numbers is very common. The last "S" designates not the plural form for FLOP but a ratio "per second" and is historically written without a slash (/) and avoiding double "S" (i.e., FLOPS instead of Flops/S). In our book we will stick to the common practice. In some situations, we will need to refer to floating point operations, so abbreviate it as Flops and produce required ratios as needed. For example, we will write Flops/cycle when there is a need to count number of floating point operations per processor cycle of the processor core.

One of the most often quoted metrics for individual processors or complete HPC systems is their peak performance. This is the theoretically maximum possible performance that could be delivered by the system. It is defined as follows:

- Peak performance of a system is a sum of peak performances of all computing elements (namely, cores) in the system.
- Peak performance for a vectorized superscalar core is calculated as the number of independent floating point arithmetic operations that the core can execute in parallel, multiplied by the number of vector elements that are processed in parallel by these operations.

As an example, if you have a cluster of 16 nodes, each with a single Intel Xeon E3-1285 v3 processor that has four cores with Haswell microarchitecture running at 3.6 GHz, it will have peak performance of 3686.4 gigaFLOPS (or 10^9 FLOPS). Using the FMA (fused multiply add, which is $b = a \times b + c$) instruction, a Haswell core can generate four Flops/cycle (via execution of two FMAs per cycle) with a SIMD vector putting out four results per cycle, thus delivering peak performance of 57.6 gigaFLOPS at the frequency of 3.6 GHz: $4\frac{Flops}{cycle} \times 4SIMD \times 3.6GHz = 57.6GFLOPS$. Multiplying this by total number of cores in the cluster $(64 = 16\,nodes \times 1\frac{processor}{node} \times 4\frac{cores}{processor})$ gives 3686.4 gigaFLOPS, or 3.68 teraFLOPS.

Peak performance usually cannot be reached, but it serves as a guideline for the optimization work. Actual application performance (often referred as *sustained performance*) can be obtained by counting the total number of floating point operations executed by the application (either by analyzing the algorithm or using special processor counters), and then dividing this number by the application runtime in seconds. The ratio between measured application performance (in FLOPS) and the peak performance of the system it was run on, multiplied by 100 percent, is often referred to as *computational efficiency*, which demonstrates what share of theoretically possible performance of the system was actually used by the application. The best efficiencies close to 95 percent are usually obtained by highly tuned computational kernels, such as BLAS (Basic Linear Algebra Subprograms), while mainstream HPC applications often achieve efficiencies of 10 percent and lower.

Distributed and Shared Memory Systems

So far we have discussed how application performance can be improved by increasing the amount of work done in parallel inside a processor core: by allowing more instructions to execute in parallel in superscalar microarchitectures, and by making each instruction process more data using the SIMD paradigm. As the next step we discuss two types of parallelism that can be employed to further enhance application performance. The main difference visible to you as a software developer is how the memory is shared and accessed by the processors. In the *shared memory* approach, multiple application threads can access all the memory simultaneously in a transparent. In the *distributed memory* approach, there is local and remote memory, and in order to work on any piece of data, that data has to be first copied into the local memory of the thread or process.

Use More Independent Threads on the Same Node

The first approach we will discuss harnesses several threads belonging to one program that can simultaneously access the same memory locations. Application threads can communicate through this *shared memory* with each other and avoid redundant copies of data. Shared memory is an efficient means of passing data between program threads. To connect multiple processors (each with multiple cores), the underlying system needs to have robust hardware to support arbitration and ordering of the memory requests.

In a shared memory system, the memory is presented to the application as a uniform, contiguous address range, while in fact the cost of accessing different parts of the memory by different processors may not be the same. Since most modern high-performance processors contain integrated memory controllers, there is some memory attached to each processor that is called *local memory* of that processor. Memory attached to other processors in the same system then needs to be accessed through an internal interconnect, such as Intel QuickPath Interconnect (QPI), that provides hardware mechanisms for all memory in the system to appear as one contiguous address space. There may be additional latency associated with accessing this *remote memory* over the latency for accessing the local memory. Shared memory systems that have this extra latency are called Non-Uniform Memory Access (NUMA) systems.

Impact from NUMA can be characterized by the ratio between the latencies for remote and local memory access. This ratio is called the *NUMA factor*. For example, in a dual-processor server with Intel Xeon E5-2697 v2 processors, local memory access latency (measured in the idle case) is around 50-70 ns (nanoseconds, or 10^{-9} second), while for remote memory access latency is equal to 90-110 ns, which leads to the NUMA factor for this system of approximately 1.5. The larger the shared memory system is, the larger the NUMA factor normally becomes. In fact, you may even find several different NUMA factors within larger systems. As a result, it is more difficult to optimize applications for these systems.

A generic diagram of a shared memory system in Figure 2-10 shows four processors, `P0...P3`, accessing shared memory divided into two NUMA regions, where memory local to `P0` and `P1` will be remote for `P2` and `P3`, and vice versa.

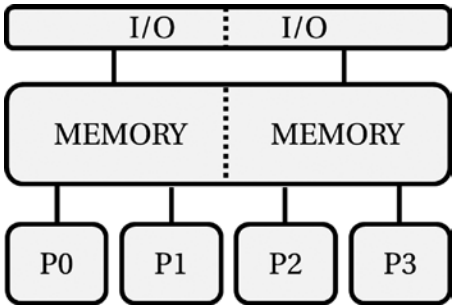

Figure 2-10. *Shared memory system diagram*

To get details on the NUMA topology of your system, use the `numactl` tool that is available for all major Linux distributions. On our workstation, the execution of `numactl` tool with the `--hardware` argument displays the following information (see Listing 2-1):

Listing 2-1. Output of the numactl --hardware Command

```
available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 24 25 26 27 28 29 30 31 32 33 34 35
node 0 size: 65457 MB
node 0 free: 57337 MB
node 1 cpus: 12 13 14 15 16 17 18 19 20 21 22 23 36 37 38 39 40 41 42 43 44
             45 46 47
node 1 size: 65536 MB
node 1 free: 59594 MB
node distances:
node   0   1
  0:  10  21
  1:  21  10
```

The output of the `numactl` tool shows two NUMA nodes, each with 24 processors (and just a hint-these are twelve physically independent cores with two threads each), and 64 GB of RAM per NUMA node, or 128 GB in the server in total.

In a similar manner to physical memory, the Input/Output subsystem and the I/O controllers are shared inside the multiprocessor systems, so that any processor can access any I/O device. Similarly to memory controllers, the I/O controllers are often integrated into the processors, and latency to access local and remote devices may differ. However, since latency associated with getting data from or to external I/O devices is significantly higher than the latency added by crossing the inter-processor network (such as QPI), this additional inter-processor network latency can be ignored in most cases. We will discuss specific I/O related issues in greater detail in Chapter 4.

Don't Limit Yourself to a Single Server

Unfortunately, there are practical limits to the size of a single system with shared memory, mostly driven by cost of building the hardware, as well as by overheads associated with the memory arbitration logic.

To achieve higher performance than a single shared memory system could offer, it is more beneficial to put together several smaller shared memory systems, and interconnect those with a fast network. Such interconnection does not make the memory from different boxes look like a single address space. This leads to the need for software to take care of copying data from one server to another implicitly or explicitly. Figure 2-11 shows an example system.

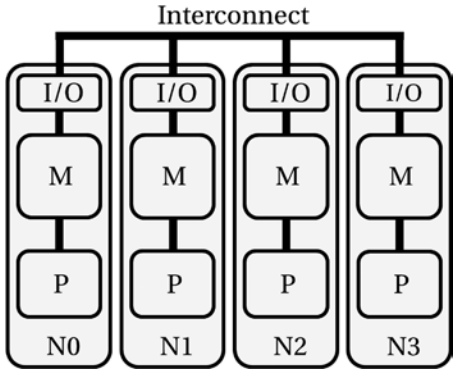

Figure 2-11. *Diagram of a distributed memory system*

Figure 2-11 shows a computer with four nodes, N0–N3, interconnected by a network, also called *interconnect* or *fabric*. Processors in each node have their own dedicated private memory and their own private I/O. In fact, these nodes are likely to be shared memory systems like those we have reviewed earlier. Before any processor can access data residing in another node's private memory, that data should be copied to the private memory of the node that is requesting the data. This hardware approach to building a parallel machine is called *distributed memory*. The additional data copy step, of course, has additional penalty associated with it, and the performance impact greatly depends on characteristics of the interconnect between the nodes and on the way it is programmed.

HPC Hardware Architecture Overview

Modern HPC hardware is quite complex, following several levels of integration, as presented in Figure 2-12. Each processor core contains several execution units, driven by out-of-order execution pipelines. Several cores in each processor may run at different frequencies to optimize the total system power consumption and keep it in balance with the application throughput. Complexity is further increased by the hierarchical cache subsystems and nonuniform memory access at the system level. One level up, several shared memory servers are assembled into a distributed memory cluster, using one or more dedicated interconnection networks.

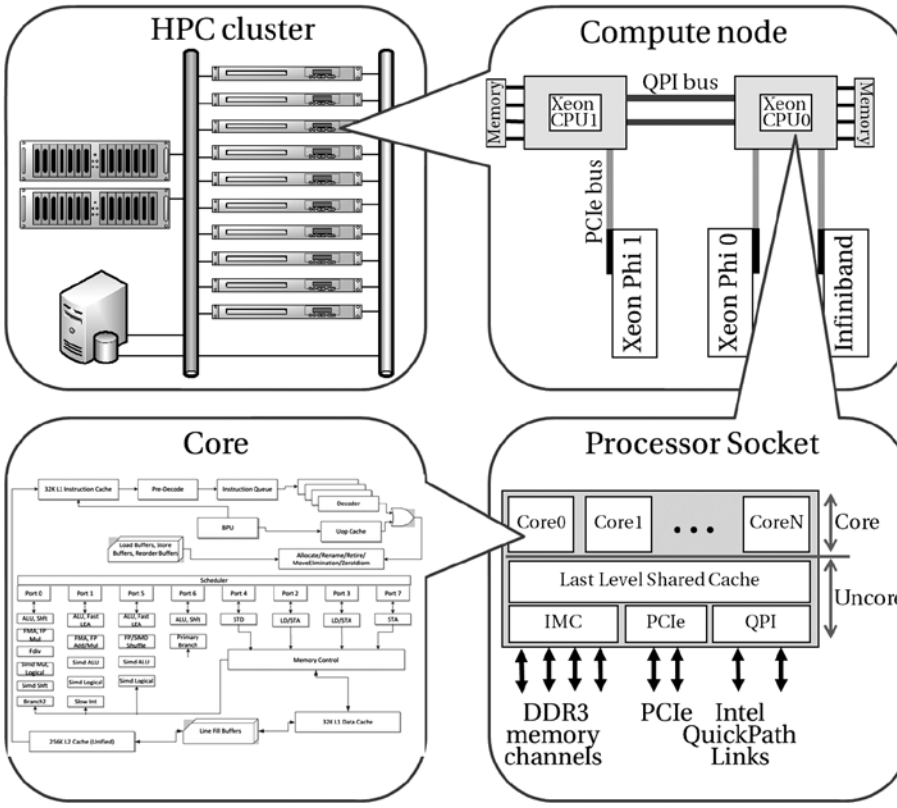

Figure 2-12. *The complexity of a modern cluster with multi-processor, multicore systems*

A Multicore Workstation or a Server Compute Node

Let us start with an overview of a simple workstation or a desktop computer. It has at least one processor and that processor very likely has multiple cores.

A *core* is an independent piece of hardware that does not share any hardware resources with other cores inside the processor. The core executes instructions of a computer program by performing requested arithmetical, logical, input/output, and other operations. Supported instructions are usually hardwired into the cores. They are called the *instruction set*. This is the language that the processor speaks, and it won't understand a different one. All instructions for mainstream Intel processors are based on the x86 instruction set, with multiple extensions, known as MMX, SSE, AES-NI, AVX, etc. The supported instruction set and the architecture state (including all the registers visible to the instructions, flags, etc.) define a *core architecture*.

The internal implementation that defines how exactly the instructions are handled to produce expected results may, and in fact does, vary from one processor to another. As an example, an Intel Atom processor and an Intel Xeon processor share the same instruction set architecture, meaning that you can run exactly the same operating system

and application software on these two. However, internal implementations of these two processor cores are very different.

We refer to the internal implementations as *microarchitecture*. Thus, the Haswell microarchitecture that is the basis for Intel Xeon E3-1200 v3 processors is very different from the Silvermont microarchitecture used to build cores for Intel Atom C2000 processors. Detailed microarchitecture differences and specific optimization techniques are described in the *Intel 64 and IA-32 Architectures Optimization Reference Manual.*[12] This 600-page document describes a large number of Intel x86 cores and explains how to optimize software for IA-32 and Intel 64 architecture processors.

The addendum to the aforementioned *Intel 64 and IA-32 Architectures Optimization Reference Manual* contains data useful for quantitative analysis of the typical latencies and throughputs of the individual processor instructions. The primary objective of this information is to help the programmer with the selection of the instruction sequences (to minimize chain latency) and in the arrangement of the instructions (to assist in hardware processing).

However, this information also provides an understanding of the scale of performance impact from various instruction choices. For instance, typical arithmetic instruction latencies (reported in the number of clock cycles that are required for the execution core to complete the execution of the instruction) are one to five cycles (or 0.4-2 ns when running at 2.5 GHz) for simple instructions such as addition, multiplication, taking maximum or minimum value. Latency can reach up to 45 cycles (or 18 ns at 2.5 GHz) for division of double precision floating point numbers.

Instruction throughput is reported as the number of clock cycles that need to pass before the issue ports can accept the same instruction again. This helps to estimate the time it would take, for example, for a loop iteration to complete in presence of a cross-loop dependency. For many instructions, throughput of an instruction can be significantly smaller than its latency. Sometimes latency is given as just one half of the clock cycle. This occurs only for the double-speed execution units found in some microprocessors.

The same manual provides estimates for the best-case latencies and throughput of the dedicated caches: the first (L1) and the second (L2) level caches, as well as the translation lookaside buffers (TLBs). Particularly, on the latest Haswell cores, the load latency from L1 data cache may vary from four to seven cycles (or 1.6-2.8 ns at 2.5 GHz), and the peak bandwidth for data is equal to 64 (Load) + 32 (Store) bytes per cycle, or up to 240 GB/s aggregate bandwidth (160 GB/s to load data and 80 GB/s to store the data).

The architecture of modern Intel processors supports flexible integration of multiple processor cores with a shared *uncore* subsystem. Uncores usually contain integrated DRAM (Dynamic Random Access Memory) controllers, PCI Express I/O, Quick Path Interconnect (QPI) links, and the integrated graphics processing units (GPUs) in some models, as well as a shared cache (L2 or L3, depending on the processor, which is often called the Last Level Cache, or LLC). An example of the system integration view of four cores with uncore components is shown in Figure 2-13.

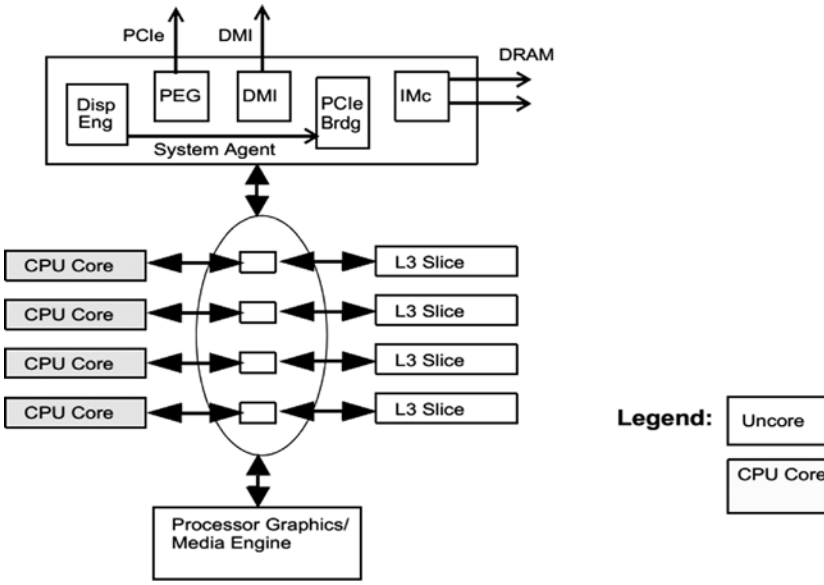

Figure 2-13. *Four-core processor integration of Intel microarchitecture, codenamed Haswell*

Uncore resources typically reside farther away from the cores on the processor die, so that typical latencies to access uncore resources (such as LLC) are normally higher than that for a core's own resources (such as L1 and L2 caches). Also, since the uncore resources are shared, the cores compete for uncore bandwidth. The latency of accessing uncore resources is not as deterministic as the latency inside the core. For example, the latency of loading data from LLC may vary from 26 to 60 cycles (or from 10.4 to 24 ns for a 2.5 GHz processor), comparing to the typical best case of 12 cycles (or 4.8 ns) load latency for the L2 cache.

Cache bandwidth improvements in the Haswell microarchitecture over the older Sandy Bridge/Ivy Bridge microarchitectures doubled the number of bytes loaded and stored per clock cycle from 32 and 16 to 64 and 32, respectively. Last Level Cache bandwidth also jumped from 32 bytes per cycle to 64 bytes. At the same time, typical access latencies stayed unchanged between the microarchitecture generations. This confirms the earlier observation related to the bandwidth vs. latency development.

As for the next level in the memory hierarchy, the computer main memory, its latency further increases and its bandwidth drops. Figure 2-14 shows schematically the relative latency and bandwidth capabilities in the memory hierarchy of a quad-core Haswell-based Intel Xeon Processor E3-1265L v3 processor.

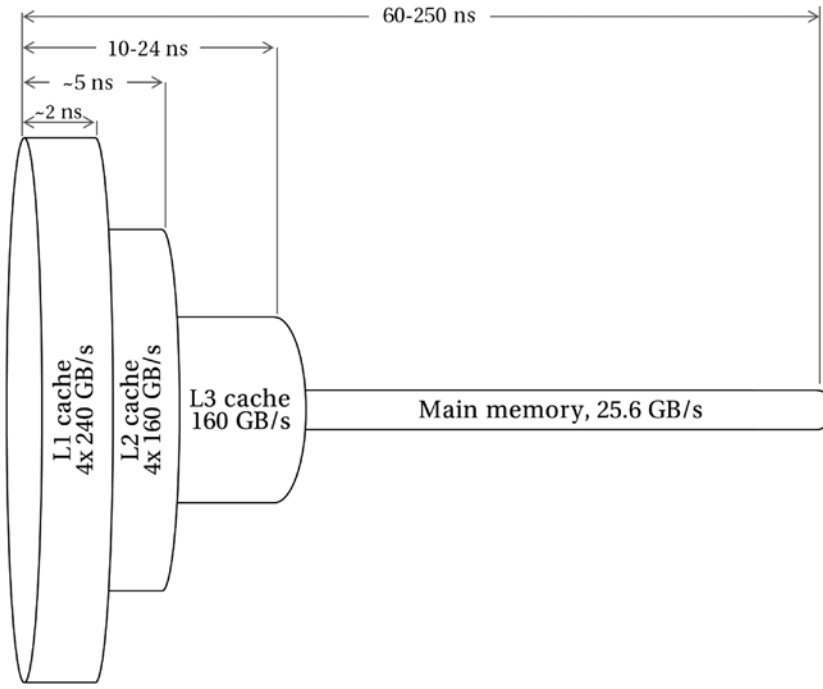

Figure 2-14. *Bandwidth and latency characteristics of a quad-core Haswell-based processor*

Another important aspect of the memory latency is that the effective time to load or store data goes up with higher utilization of the memory busses. Figure 2-15 shows the results of the latency measurement performed as a function of intensity of the memory traffic for a dual-socket server. Here, two generations of server processors are compared, with cores based on the Sandy Bridge and Ivy Bridge. The newer Ivy Bridge-based processors (specifically Intel Xeon E5-2697 v2) support faster memory running at 1866 MHz and contain improvements in the efficiency of the memory controller implementation over the previous generation, Intel Xeon E5-2690 processor built with eight Sandy Bridge cores and memory running at 1600 MHz. Despite the increase of the core count and faster DRAM speed, latency is about the same in both cases when the concurrency of memory requests is low (and thus the consumed memory bandwidth is far below the physical limits).

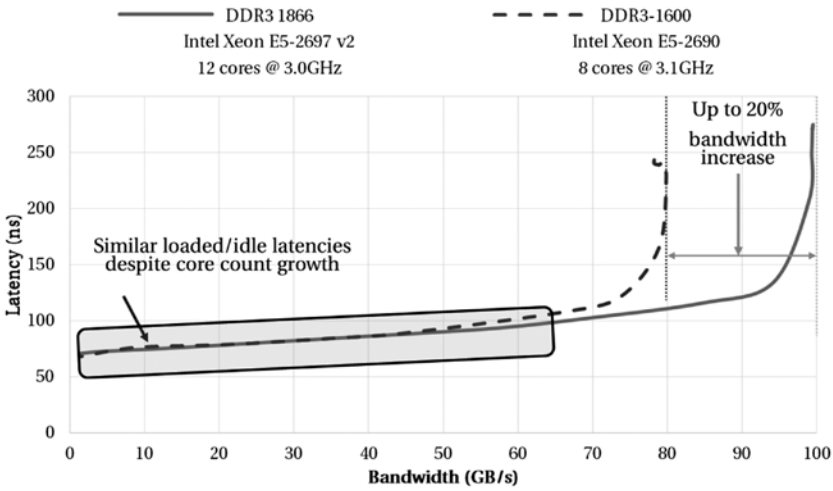

Figure 2-15. *Memory latency dependency from memory bus load**

**Based on measurements done using a latency/bandwidth tool internal to Intel Corporation. Memory traffic mix: 66% Reads and 33% writes (Request for ownership). OS: Windows 2008 R2 SP1, System configurations: Intel Xeon E5 2697 v2 (Ivy Bridge-EP): 12C, nominal 2.7 GHz [July 2013], Xeon E5 2690 (Sandy Bridge-EP): 8C, nominal 2.9 GHz [Sept 2011], 1 dual-ranked RDIMM per channel, 4 channels, varied DDR frequencies, pre-production BIOS*

In this case, latency is determined by the internal organization of the memory hierarchy rather than by the DRAM speed and technology. Only when concurrency increases, generating more load/store requests, and the consumed memory bandwidth reaches the wire speed limit, the latency difference becomes noticeable. Another outcome from this measurement is that memory latency does vary significantly, depending on the load of the memory bus: from 60 to 70 ns in idle case (that will be 160-170 processor core cycles at 2.5 GHz), up to around 250 ns for the loaded case (over 600 cycles).

Coprocessor for Highly Parallel Applications

Recent years have seen the rise of accelerators and coprocessor targeting highly parallel applications. One example would be Intel's Xeon Phi family of coprocessors that feature a highly parallel microprocessor with up to 61 cores running at up to 1.2 GHz, with 16 GB of GDDR5 memory clocked at up to 5.5 GHz, and an integrated system management controller. The coprocessor runs Linux OS. It can even be seen as a standalone computational node, although the presence of a host processor is still required to boot and initialize the coprocessor.

The coprocessors found in HPC these days focus on delivering higher throughput. They can achieve over 1 TFLOPS of peak floating point performance with peak memory

bandwidth reaching 350 GB/s. However, great throughput comes at the expense of latency: the coprocessors usually run at frequencies around 1 GHz (2.5-3x slower than standalone processors), and GDDR5 access latency is at least a factor of two times higher versus DDR3 in a standard server. However, for a subset of applications, where higher latency can be hidden by much higher concurrency, noticeable performance benefit comes from the significantly higher throughput in hardware.

One important performance and programmability aspect of coprocessor is that they are attached to the main processor(s) over the PCI Express (PCIe) bus. Often they have to involve the host processor to execute I/O operations or perform other tasks. The second-generation PCIe bus that is used in Intel Xeon Phi coprocessors can deliver up to 80 Gbps (gigabits per second) of peak wire bandwidth in every direction via a x16 connector. This translates into approximately 7 GB/s of sustained bandwidth, for the overhead includes 8/10 encoding scheme used to increase reliability of data transfers over the bus. This also adds latency between the host processor and the coprocessor, on the order of 200-300 ns, or more if the bus is heavily loaded. In heterogeneous applications that use both the central processors and the coprocessors in a system, it is important to hide this added latency of communications over the PCIe bus.

Group of Similar Nodes Form an HPC Cluster

When a single server is not enough to solve a scientific or technology problem in a sufficiently short time, people put together several nodes and wire them with a dedicated communication network to form a distributed memory system, called a *cluster*. For this approach to work, every node adds a special adapter for a fast network, such as 10 gigabit Ethernet or InfiniBand. The software stack needs to support message passing between the nodes, so that it becomes more sophisticated. Two dual-processor servers with two Intel Xeon Phi coprocessors each clustered together with InfiniBand interconnect are shown in Figure 2-16.

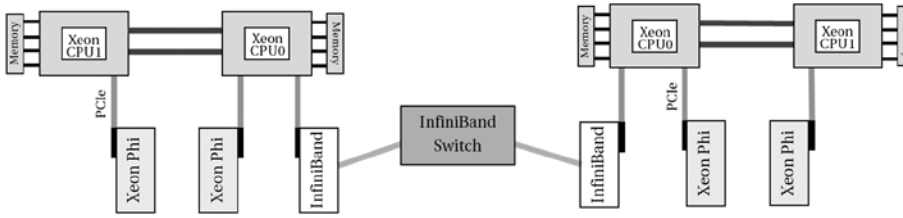

Figure 2-16. *Cluster of two nodes*

All of the most popular interconnects use the PCIe bus to connect to the processors in the system, and thus they inherit all the latency and bandwidth limitations specific to PCIe. On top of this, since both Ethernet and InfiniBand are designed to scale to a much larger number of communicating agents in the network (at least tens of thousands) than PCIe, their protocol overheads and cost of packet routing are significantly higher compared to the PCIe bus used inside the server.

Typical latencies of modern, widely used interconnects are around 1.5 to 15 microseconds (which is 1,500-15,000 ns, or thousands and tens of thousands of processor cycles) for point-to-point communication between two application processes, including the overheads associated with the utilized message passing protocol and its software implementation. However, bandwidth of these fast interconnects is closer to what can be found inside the server. For instance, the fastest InfiniBand peak data rate is 56 Gbps. This results in approximately 6.5 GB/s attainable bandwidth between two nodes in one direction. Latency is often a higher limiting factor, unless it is hidden by the applications via overlapping communications and computations, and using optimized algorithms for collective communication between large numbers of nodes.

Another important factor that influences performance of a parallel application in a cluster environment, especially with a very large number of nodes, is the interconnect topology. The PCIe bus used inside each node usually provides a very simple point-to-point or star topology. More complex, though scalable, topologies are used in the HPC cluster interconnects. The Fat Tree topology is probably the most popular one, despite a cost that grows with the size of the cluster. The InfiniBand network supports several multiple topology choices, including All-to-All, Fat Tree, Torus, and Hypercube topologies, as shown in Figure 2-17. There is no single, best topology; its choice and suitability are determined by the needs of the application and by the target metrics and cost implications. Here is a brief outline of the advantages and drawbacks of several interconnect topologies:

- *All-to-All topologies* are ideal for applications that are highly sensitive to communication latency, since the All-to-All topology requires the minimum number of hops between the communicating agents. Even though an asynchronous fabric with high bisection bandwidth can be built using the All-to-All topology, it is restricted to relatively small clusters due to the limited switch port counts.

- *Fat Tree* topologies are well suited for the majority of clusters and applications. Fat Tree topologies can provide asynchronous fabrics and predictable latency between the nodes. However, cabling and switching become increasingly difficult and expensive as the cluster size grows, with very large switches required for larger clusters. Anyway, there are clusters comprising several thousands of nodes and reaching PetaFLOPS of performance that are organized in the nonblocking Fat Tree topology.

- *Hypercube*, *Torus,* and other topologies are best suited for very large node counts. They provide rich bandwidth capabilities, and they scale easily from small to extremely large clusters. These topologies are usually much harder to design and implement. They can present additional scalability challenges and introduce variable hop count and latency with the increasing cluster size. Inconsistent hop counts can result in unpredictable application behavior owing to unequal latency between the nodes.

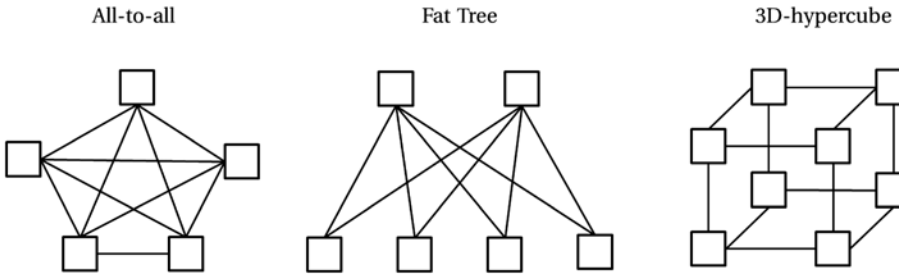

Figure 2-17. *Typical interconnection topologies found in HPC*

Other Important Components of HPC Systems

In reality, to get the most out of an HPC system, it is not enough to just rely on the best components in servers, packed together in a balanced way for your applications. Several other elements are important from the point of view of overall cluster performance and efficiency.

Specifically, HPC systems are used for applications that require a lot of memory, so that multiple nodes are used simultaneously in a scale-out fashion to provide the required amount of memory. Quite often, such applications either get a lot of data as their input (so-called Big Data applications) or produce huge amounts of output (typical scientific simulations). Handling huge amounts of data requires good storage arrangements. Development of scalable parallel storage and file systems to meet specific demands of HPC or Big Data applications can be viewed as a special art.

A single storage server along with a large compute cluster is likely not the best possible setup. Although it will work, a single network link, a single RAID adapter, or, in the worst case, a single disk will likely become a bottleneck, serializing all your cluster nodes around the single storage node. A better approach widely used in HPC today is to parallelize the storage or cluster it. Following the approach similar to the computing capacity, you can use several storage nodes to provide throughput and increase the level of concurrency of disk operations to sustain the high number of I/O requests issued by the computing cluster nodes. On top of clustered hardware, use of parallel I/O operations is usually implemented in the application or system software.

However, latency of single I/O operations is usually quite high. Here we see the milliseconds of time (as visible by the user application) that small I/O transactions can take. If your application does a lot of small-size reads/writes at random locations or from/to many different files, solid state drives attached to PCIe bus on the storage nodes promise significant increase in performance for small I/O operations. With an optimized file system and network stack, latencies of single I/O operations with small payloads drop down to a few hundred or even tens of microseconds.

Another really important component of an HPC system is a job scheduler. This is a software component, but it influences hardware utilization and overall cluster utilization efficiency. Its main purposes are planning the execution of user batch jobs, scheduling them for execution, deploying the user run script and executable file(s) to the allocated cluster nodes, organizing input and gathering output, terminating the application, and collecting accounting information. There are multiple open-source and commercial schedulers available to choose from.

It is worth noting that job schedulers take their portion of time for every job execution, and this time can reach seconds per job submission. The good news is that scheduling takes place only before the application starts and may add some time after the job ends (for the clean-up). So, if your job takes several days to run on a cluster, these few seconds have a small relative impact.

However, sometimes people need to run a large number of smaller jobs. For example, if each job takes a couple of minutes to run, but there are many jobs (up to tens of thousands have been observed in real life), the relative time taken by the job scheduler becomes very visible. Most job schedules offer special support for large number of smaller jobs with identical parameters via so-called job arrays. If your application is of that kind, please take some time to study how to make effective use of your cluster's scheduling software.

Summary

This chapter briefly overviewed the main terms and concepts pertaining to the performance analysis and gave an overview of the modern high-performance computing platforms. Certainly, this is the minimum information needed to help you get started on the subject or to refresh your existing knowledge.

If you are interested in computer architecture, you may enjoy the book *Computer Architecture: A Quantitative Approach.*[13] In the fourth edition of this book, the authors increase their coverage of multiprocessors and explore the most effective ways of achieving parallelism as the key to unlocking the power of modern architectures.

We also found an easy-to-read guide in the book *Introduction to High Performance Computing for Scientists and Engineers,* written by Georg Hager and Gerhard Wellein.[14] It contains a great overview of platforms architectures, as well as recommendations for application optimization specific to the serial, multi-threaded, and clustered execution.

In his article *Latency Lags Bandwidth,* David Patterson presents an interesting study that illustrates a chronic imbalance between bandwidth and latency.[15] He lists half a dozen performance milestones to document this observation, highlights many reasons why this happens, and proposes a few ways to cope with the problem, as well as gives a rule of thumb to quantify it, plus an example of how to design systems based on this observation.

For readers interested in the queuing network modeling, we recommend the book *Quantitative System Performance: Computer System Analysis Using Queuing Network Models.*[16] It contains an in-depth description of the methodology and a practical guide to and case studies of system performance analysis. It also provides great insight into the major factors affecting the performance of computer systems and quantifies the influence of the system bottlenecks.

The fundamentals and practical methods of the queuing theory are described in the book *Queueing Systems: Theory.*[17] Step-by-step derivations with detailed explanation and lists of the most important results make this treatise useful as a handbook.

References

1. *Merriam-Webster Collegiate Dictionary*, 11th ed. (Springfield, MA: Merriam-Webster, 2003).
2. A. S. Tanenbaum, *Computer Networks* (Englewood Cliffs, NJ: Prentice-Hall, 2003).
3. B. Cantrill and J. Bonwick, "Real-World Concurrency," *ACM Queue* 6, no. 5 (September 2008): 16–25.
4. G. M. Amdahl, "Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities," AFIPS '67 (Spring) Proceedings of the 18–20 April 1967, spring joint computer conference, 483–85.
5. J. L. Gustafson, "Reevaluating Amdahl's law," *Communications of the ACM* 31, no. 5 (May 1988): 532–33.
6. "Queueing theory," Wikipedia, http://en.wikipedia.org/wiki/Queueing_theory.
7. S. Williams, A. Waterman, and D. Patterson, "Roofline: An Insightful Visual Performance Model for Multicore Architectures," *Communications of the ACM - A Direct Path to Dependable Software* 52, no. 4 (April 2009): 65–76.
8. J. McCalpin, "Memory Bandwidth and Machine Balance in Current High Performance Computers," *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, December 1995.
9. IDC (International Data Corporation), "HPC Market Update: 2012," September 2012, www.hpcuserforum.com/presentations/dearborn2012/IDCmarketslidesChirag-Steve.pdf.
10. Intel Corporation, "Intel VTune Amplifier XE 2013," http://software.intel.com/en-us/intel-vtune-amplifier-xe.
11. M. J. Flynn, "Very High-speed Computing Systems," *Proceedings of IEEE* 54 (1966): 1901–909.
12. Intel Corporation, *Intel 64 and IA-32 Architectures Optimization Reference Manual,* www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html.
13. J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 4th ed. (Burlington, MA: Morgan Kaufmann, 2006).
14. G. Hager and G. Wellein, *Introduction to High Performance Computing for Scientists and Engineers* (Boca Raton, FL: CRC Press, 2010).
15. D. A. Patterson, "Latency Lags Bandwith," *Communications of the ACM - Voting Systems,* January 2004, pp. 71–75.
16. E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik, *Quantitative System Performance: Computer System Analysis Using Queueing Network Models* (Upper Saddle River, NJ: Prentice-Hall, 1984).
17. L. Kleinrock, *Queueing Systems: Theory,* vol. 1 (Hoboken, NJ: John Wiley, 1976).