Chapter 2

Parallel Computer Architectures

William Gropp, Rick Stevens, and Charlie Catlett (Ian Foster, editor)

Target Length: 20 to 30 pages

Parallel computers provide great amounts of computing power, but they do so at the cost of increased difficulty in programming and using them. Certainly, a uniprocessor that was fast enough would be simpler to use. To explain why parallel computers are inevitable and to identify the challenges facing developers of parallel algorithms, programming models, and systems, in this chapter we describe briefly (but in more detail than in Chapter 2) the architecture of both uniprocessor and parallel computers. We will see that while computing power can be increased by adding processing units, memory latency (the irreducible time to access data) is the source of many challenges in both uniprocessor and parallel processor design.

Parallel architectures and programming models are not independent. While most architectures can support all programming models, they may not be able to do so efficiently. An important part of any parallel architecture is any feature that simplifies the process of building, testing, and tuning an application. Some parallel architectures put a great deal of effort into supporting a parallel programming model; others provide little or no extra support. All architectures represent a compromise between cost, complexity, timeliness, and performance.

This chapter is organized as follows. In Section 2.1 we briefly describe the important features of single processor (or uniprocessor) architecture. From this background, the basics of parallel architecture are presented in Section 2.2; in particular, we describe the opportunities for performance improvement through parallelism at each level in a parallel computer, with references to machines of each type. Section 2.3 reviews current parallel systems. In Section 2.4, we examine potential future parallel computer architectures. We conclude the chapter with a brief summary of the key issues motivating the development of parallel algorithms and programming models.

2.1 Uniprocessor Architecture

In this section we briefly describe the major components of a conventional, single-processor computer, emphasizing the design tradeoffs faced by the hardware architect. This description lays the groundwork for a discussion of parallel architectures, since parallelism is entirely a response to the difficulty of providing ever greater performance (or reliability) in a system that inherently performs only one task at a time. Those interested in a more detailed discussion of these issues should consult [8].

The major components of a computer are the central processing unit that executes programs, the memory system that stores executing programs and the data that the programs are operating on, and input/output systems that allow the computer to communicate with the outside world (e.g., through keyboards, networks, and displays) and with permanent storage devices such as disks. The design of a computer reflects the available technology; constraints such as power consumption, physical size, cost, and maintainability; the imagination of the architect; and the software (programs) that will run on the computer (including compatibility issues). All of these have changed tremendously over the past fifty years.

Perhaps the best known change is captured by Moore's law [18], which says that microprocessor CPU performance doubles roughly every eighteen months. This is equivalent to a thousandfold increase in performance over fifteen years. Moore's law has been remarkably accurate over the past thirty-six years (see Figure 2.1), even though it represents an observation about the rate of engineering progress and is not a law of nature (such as the speed of light). In fact, it is interesting to look at the clock speed of the *fastest* machines in addition to (and compared with) that of microprocessors. In 1981, the Cray 1 was one of the fastest computers, with a 12.5 ns clock. In 2001, microprocessors with 0.8 ns clocks are becoming available. This is a factor of 16 in twenty years, or equivalently a doubling every five years.

Remarkable advances have occurred in other areas of computer technology as well. The cost per byte of storage, both in computer memory and in disk storage, has fallen along a similar exponential curve, as has the physical size per byte of storage (in fact, the cost and size are closely related). Dramatic advancements in algorithms have reduced the amount of work needed to solve many classes of important problem; for example, the work needed to solve n simultaneous linear equations has fallen, in many cases, from n^3 to n. For 1 million equations, this is an improvement of 12 orders of magnitude!

Unfortunately, these changes have not been uniform. For example, while the density of storage (memory and disk) and the bandwidths have increased dramatically, the time to access storage (latency) has not kept up. As a result, over the years, the balance in performance between the different parts of a

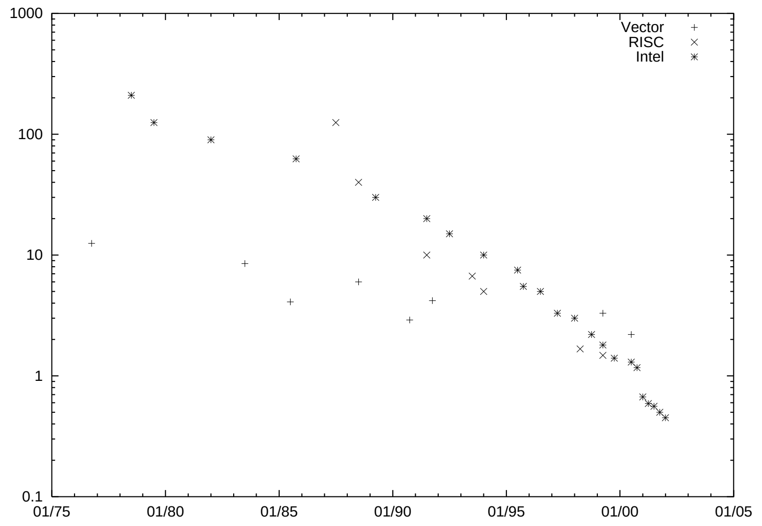

Figure 2.1: Improvement in CPU performance measured by clock rate

computer has changed. In the case of storage, increases in clock rates relative to storage latency have translated Moore's law into a description of *inflation* in terms of the relative cost of memory access from the point of view of potentially wasted CPU cycles. This has forced computer architectures to evolve over the years, for example moving to deeper and more complex memory hierarchies.

2.1.1 The CPU

The CPU is the heart of the computer; it is responsible for all calculations and for controlling or supervising the other parts of the computer. A typical CPU contains the following (see Figure 2.2):

Arithmetic Logic Unit (ALU): Performs computations such as addition and comparison.

Floating Point Unit (FPU): Performs operations on floating-point numbers.

Load/Store Unit: Performs loads and stores for data.

Registers: Fast memory locations that can be used to store intermediate results. These are often subdivided into floating-point registers (FPR) and general purpose registers (GPR).

Program Counter (PC): Contains the address of the instruction that is executing.

Memory Interface: Provides access to the memory system. In addition, the CPU chip often contains the fastest part of the memory hierarchy (the top level cache); this part is described in Section 2.1.2.

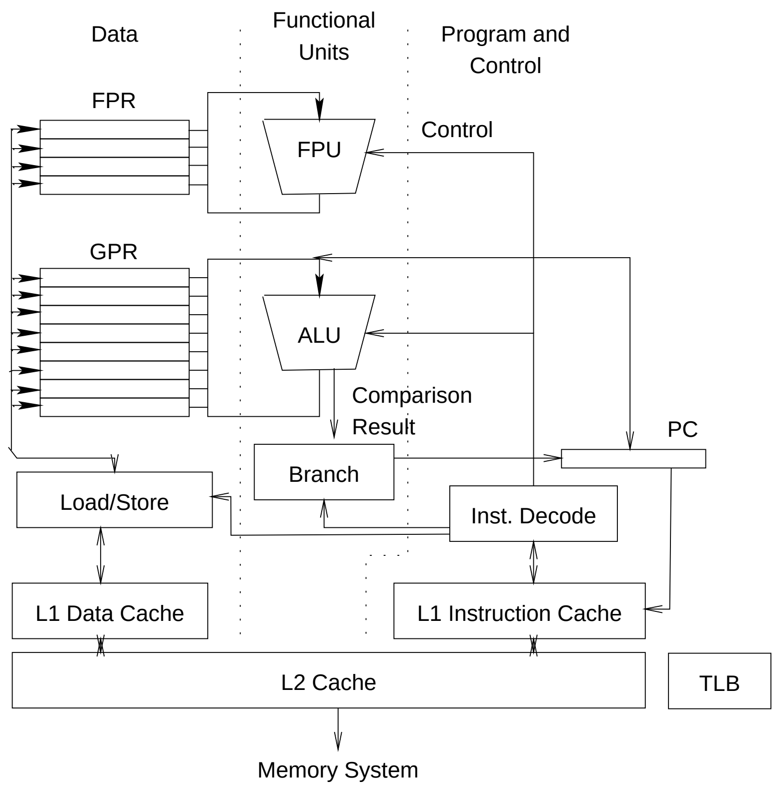

Figure 2.2: Generic CPU diagram. This example has a separate L1 cache for data and for program instructions and a unified (both data and instructions) L2 cache. Not all data paths are shown.

Other components of a CPU are needed for a complete system, but the ones listed are most important for our purpose.

The CPU operates in steps controlled by a clock: in each step, or *clock cycle*, the CPU performs an operation.[1] The speed of the CPU clock has increased dramatically; desktop computers now come with clocks that run at over 1 GHz (10^9 Hz).

One of the first decisions that a computer architect must make is what basic operations can be performed by the CPU. There are two major camps: the complex instruction set computer (CISC) and the reduced instruction set computer (RISC). A RISC CPU can do just as much as a CISC CPU; however, it may require more instructions to perform the same operation. The tradeoff is that a RISC CPU, because the instructions are fewer and simpler, may be able to execute each instruction faster (i.e., the CPU can have a higher clock speed),

[1]Note that we did not say an instruction or a statement. As we will see, modern CPUs may perform both less than an instruction and more than one instruction in a clock cycle.

allowing it to complete the operation more quickly.

The specific set of instructions that a CPU can perform is called the instruction set. The design of that instruction set relative to the CPU represents the instruction set architecture (ISA). The instructions are usually produced by compilers from programs written in higher-level languages such as Fortran or C. The success of the personal computer has made the Intel x86 ISA the most common ISA, but many others exist, particularly for enterprise and technical computing. We note that while the ISA may be directly executed by the CPU, another possibility is to design the CPU to convert each instruction into a sequence of one or more "micro" instructions. This allows a computer architect to take advantage of simple operations to raise the "core" speed of a CPU, even for an ISA with complex instructions (i.e., a CISC architecture). Thus, even though a CPU may have a clock speed of over 1 GHz, it may need multiple clock cycles to execute a single instruction in the ISA. Hence, simple clock speed comparisons between different architectures are deceptive. Even though one CPU may have a higher clock speed than another, it may also require more clock cycles than the "slower" CPU in order to execute a single instruction.

Programs executed by the CPU are stored in memory. The *program counter* specifies the address in memory of the executing instruction. This instruction is fetched from memory and decoded in the CPU. As each instruction is executed, the PC changes to the address of the next instruction. Control flow in a program (e.g., `if`, `while`, or function all) is implemented by setting the PC to a new address.

One important part of the ISA concerns how memory is accessed. When memory speeds were relatively fast compared with CPU speeds (particularly for complex operations such as floating-point division), the ISA might include instructions that read several items from memory, performed the operation, and stored the result into memory. These were called memory-to-memory operations. However, as CPU speeds increased dramatically relative to memory access speeds, ISAs changed to emphasize a "load-store" architecture. In this approach, all operations are performed by using data in special, very fast locations called *registers* that are part of the CPU. Before a value from memory can be used, it must first be loaded into a register, using an address that has been computed and placed into another register. Operations take operands from registers and put the result back into a register; these are sometimes called register-to-register operations. A separate store operation puts a value back into the memory (generally indirectly by way of a cache hierarchy analogous to the register scheme just described). Load and store operations are often handled by a load/store functional unit, much as floating-point arithmetic is handled by a floating-point unit (FPU).

Over the years, CPUs have provided special features to support various programming models. For example, CISC-style ISAs often include string search instructions and even polynomial evaluation. Some current ISAs support instructions that make it easy to access consecutive elements in memory by updating the register holding the load address; this corresponds closely to the `a=*x++;` statement in the C programming language and to typical Fortran cod-

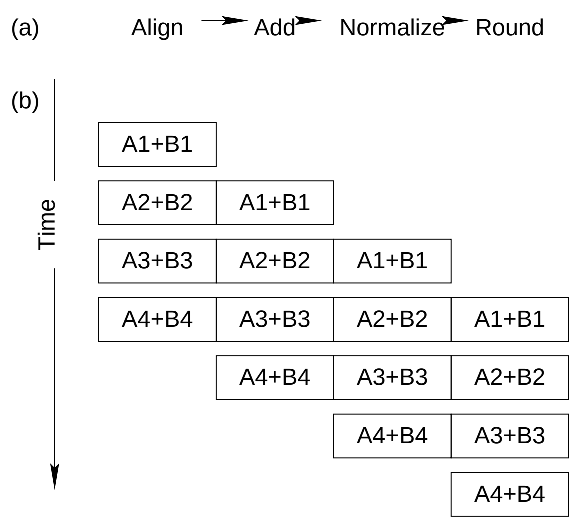

Figure 2.3: Example of a floating-point pipeline. The separate stages in the pipeline are shown in (a). In (b), four pairs of numbers are added in 7 clock cycles. Note that after a 3-cycle delay, one result is returned every cycle. Without pipelining, 16 clock cycles would be required to add four pairs of numbers.

ing practice for loops.

One source of complexity in a CPU is the difference in the complexity of the instructions. Some instructions, such as bitwise logical **or**, are easy to implement in hardware. Others, such as floating-point division, are extremely complicated. Memory references provide a different kind of complexity; as we will see, the CPU often cannot predict when a memory reference will complete. Many different approaches have been taken to address these issues. For example, in the case of floating-point operations, *pipelining* has been used. Like the RISC approach, pipelining breaks a complex operation into separate parts. Unlike the RISC approach, however, each stage in the pipeline can be executed at the same time by the CPU, but on different data. In other words, once a floating-point operation has been started in a clock cycle, even though that operation has not completed, a new floating-point operation can be started in the next clock cycle. It is not unusual for operations to take two to twenty cycles to complete. Figure 2.3 illustrates a pipeline for floating-point addition. Pipelines have been getting deeper (i.e., have more stages) as clock speeds increase. Note also that this hardware approach is very similar to the use of pipelining in algorithms described in Section 2.3.2.

From this discussion, we can already see some of the barriers to achieving higher performance. A clock rate of 1 GHz corresponds to a period of only 1 ns. In 1 ns, light travels only about 1 foot in a vacuum, and less in an electrical circuit. Even in the best case, a single processor running at 10 GHz (three more doublings in CPU performance or, if Moore's law continues to hold, appearing in less than five years) and its memory could be only about one inch across (any larger and a signal could not cross the chip during a single clock cycle); at that size, heat dissipation becomes a major problem (in fact, heat dissipation is already a problem for many CPUs). Approaches such as pipelining (already a kind of parallelism) require that enough operations and operands be available to keep the pipeline full. Other approaches begin to introduce a very fine scale of parallelism, for example by providing multiple *functional units* such as multiple floating-point adders and multipliers. In such cases, however, the program must be rewritten (and/or compiled) to make use of the additional resources. (These enhancements are discussed in Section 2.2.3.)

Once on-chip clock latency is addressed, the designer must face an even more challenging problem: latency to storage, beginning with memory.

2.1.2 Memory

While a computer is running, active data and programs are stored in memory. Memory systems are quite complex, introducing a number of design issues. Among these are the following:

Memory size. Users never have enough computer memory, so the concept of *virtual memory* was introduced to fool programs into thinking that they have large amounts of memory just for their own use.

Memory latency and hierarchy. The time to access memory has not kept pace with CPU clock speeds. Levels or *hierarchies* of memory try to achieve a compromise between performance and cost.

Memory bandwidth. The rate at which memory can be transferred to and from the CPU (or other devices, such as disks) also has not kept up with CPU speeds.

Memory protection. Many architectures include hardware support for memory protection, aimed primarily at preventing application software from modifying (intentionally or inadvertently) either system memory or memory in use by other programs.

Of these, memory latency is the most difficult problem. Memory size, in many ways, is simply a matter of money. Bandwidth can be increased by increasing the number of paths to memory (another use of parallelism) and using techniques such as interleaving (analogous to striping). Latencies are related to physical constraints are harder to reduce. Further, high latencies reduce the effective bandwidth of a given load or store. To see this, consider a memory interconnect that transfers blocks of 32 bytes with a bandwidth of 1 GB/s. In other words, the time to transfer 32 bytes is 32 ns. If the latency of the memory system is also 32 ns (an optimistic figure), the total time to transfer the data is 64 ns, reducing the effective bandwidth from 1 GB/s to 500 MB/s. The most common approach to improving bandwidth in the presence of high latency is to increase the amount of data moved each time, thus amortizing the latency over more data. However, this helps only when all of the data moved is needed by the running program.

An executing program, or *process*, involves an address space and (one or more) program counters. Operating systems manage the time-sharing of a CPU to allow many processes to appear to be running at the same time (we will see that for parallel computers, the processes may in fact be running simultaneously). The operating system, working with the memory system hardware, provides each process with the appearance of a private address space. Most systems further allow the private memory space to appear larger than the available amount of physical memory. This is called a *virtual address space*. Of course, the actual physical memory hardware defines an address space, or *physical address space*. Any memory reference made by a process, for example, with a load or store instruction, must first be translated from the virtual address (the address known to the process) to the physical address. This step is performed by the *translation lookaside buffer* (TLB), which is part of the memory system hardware. In most systems, the TLB can map only a subset of the virtual addresses (it is a kind of address cache); if a virtual address can't be handled by the TLB, the operating system is asked to help out; in such a case, the cost of accessing memory greatly increases. For this reason, some high-performance systems have chosen not to provide virtual addressing.

Decreasing memory latency is a difficult problem. Semiconductor memory comes in two main types: static random access memory (SRAM), in which each bit of memory is stored in a latch made up of transisitors, and dynamic random access memory (DRAM), in which each bit of memory is stored as a charge on a capacitor. SRAM is faster than DRAM but is much less dense (has fewer bits per chip) and requires much greater power (resulting in heat). The difference is so great that virtually all computers use DRAM for the majority of their memory. However, as Figure 2.4 shows, the performance of DRAM memory has not followed the Moore's law curve that CPU clock speeds hav. Instead, the density and price-performance of DRAMs have risen exponentially. The scale of this problem can be seen by comparing the speeds of DRAMs and CPUs. For example, a 1 GHz CPU will execute 60 instructions before a typical (60 ns) DRAM can return a single byte. Hence, in a program that issues a load for a data item that must come from DRAM, at least 60 cycles will pass before the data will be available. In practice, the delay can be longer because there is more involved in providing the data item than just accessing the DRAM.

To work around this performance gap, computer architects have introduced a hierarchy of smaller but faster memories. These are called *cache memories* because they work by caching copies of data from the DRAM memory in faster SRAM memory, closer to the CPU. Because SRAM memory is more expensive and less dense (takes up more die space) and consumes much more power (produces more heat to dissipate) than does DRAM memory, cache memory sizes are small relative to *main memory*. In fact, there is usually a hierarchy of cache memory, starting from level 1 (L1) which is the smallest (and fastest) and is in some architectures on-chip with the CPU. Many systems have two or three levels of cache. A typical size is 16 KB to 128 KB for L1 cache memory to as

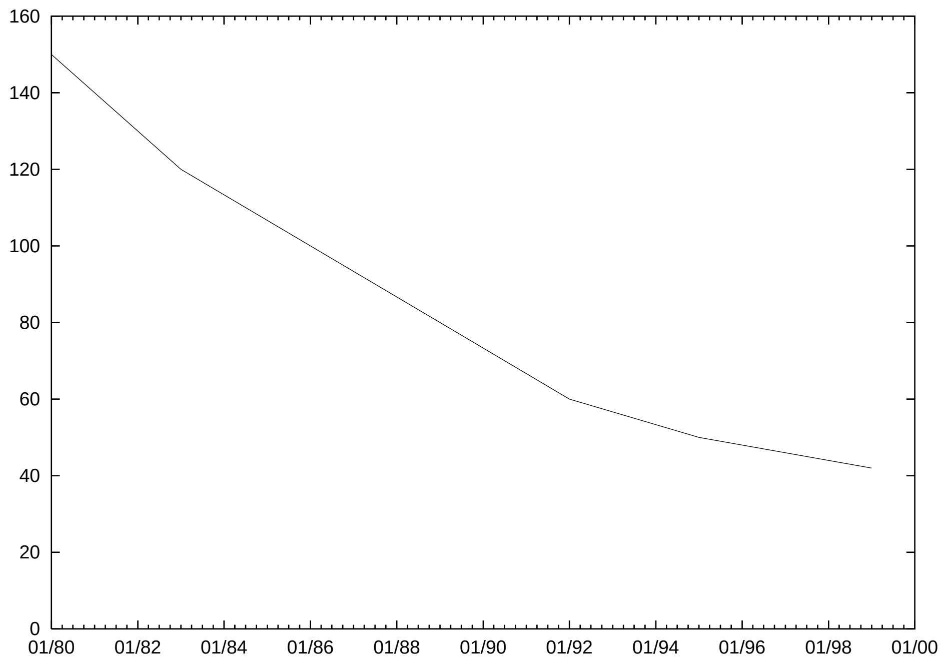

Figure 2.4: DRAM latency versus time. Note that, unlike the CPU times in Figure 2.1, the time axis is linear, and the improvement in performance is little more than a factor of two in ten years.

much as 4 MB to 8 MB for L2 or L3 cache memory. DRAM memory sizes, on the other hand, are 256 MB to 4 GB—a factor of about a thousand larger.

Memory hierarchy brings up another problem. Because the cache memory is so much smaller than the main memory, it often isn't possible for all of the memory used by a process to reside in the L1 or even L2 cache memory. Thus, as a process runs, the memory system hardware must decide which memory locations to copy into cache. If the cache is full and a new memory location is needed, some other item must be removed from the cache and written back to the main memory. The rate at which this happens is called the *cache miss rate*, and one of the primary goals of a memory system architect is to make the miss rate as small as possible. Of course, the rate depends on the behavior of the program, and this in turn depends on the algorithms used by the program. Many different strategies are used to try to achieve low miss rates in a cache while keeping the cache fast and relatively inexpensive. To reduce the miss rate, programs exploit *temporal locality*: reusing the same data within a short span of time, that is, reusing the data before it is removed from the cache to make room for some other data. This process, in turn, requires the algorithm developer and programmer to pay close attention to how data is used in a program.

As just one example, consider the choice of the *cache line size*. Data between cache and main memory usually is transferred in groups of 64, 128, or 256 bytes. This group is called a cache line. Moving a cache line at one time allows the main memory to provide relatively efficient bursts of data (it will be at least 60 ns before we can get the first byte; subsequent consecutive bytes can be delivered without much delay). Thus, programs that access "nearby" memory after the first access will find that the data they need is already in cache. For these programs, a larger line size will improve performance. However, programs that access memory in a less structured way may find that they spend most of their time reading data into cache that is never used. For these programs, a large line size reduces performance compared with a system that uses a shorter cache line.

Many other issues also remain, with similarly difficult tradeoffs, such as associativity (how main memory addresses are mapped into the cache), replacement policy (what data is ejected to make room for new data), and cache size. Exploiting the fact that memory is loaded in larger units than the natural scalar objects (such as integers, characters, or floating-point numbers) is called *exploiting spatial locality.* Spatial locality also requires temporal locality.

The effective use of cache memory is so important for high-performance applications that algorithms have been developed tailored to the requirements of these memory hierarchies. On the other hand, the most widely used programming models ignore cache memory requirements. Hence, problems remain with the practical programming of these systems for high performance. We will also see in Section 2.2.1 that the use of copies of data in a cache causes problems for parallel systems.

2.1.3 I/O and Networking

Discussions of computers often slight the issues of I/O and networking. I/O, particularly to the disks that store files and swap space for supporting virtual memory, has followed a path similar to that of main memory. That is, densities and sizes have increased enormously (twenty-five years ago, a 40 MB disk was large and expensize; today, a 40 GB disk is a commodity consumer item), but latencies have remained relatively unchanged. Because disks are electromechanical devices, latencies are in the range of milliseconds or a million times greater than CPU speeds. To address this issue, some of the same techniques used for memory have been adopted, particularly the use of caches (typically using DRAM memory) to improve performance.

Networking has changed less. Although Ethernet was introduced twenty-one years ago, only relatively modest improvements in performance were seen for many years, and most of the improvement has been in reduced monetary cost. Fortunately, in the past few years, this situation has started to change. In particular, 100 Mb Ethernet has nearly displaced the original 10 Mb Ethernet, and several Gigabit networking technologies are gaining ground, as are industry efforts, such as Infiniband [12] to accelerate the rate of improvement in network bandwidth. Optical technologies have been in use for some time but are now poised to significantly increase the available bandwidths. Networks, are, however, fundamentally constrained by the speed of light. Latencies can never be less than 3 ns per meter. Another constraint is the way in which the network is used by the software. The approaches that are currently used by most software involve the operating system (OS) in most networking operations, including most data transfers between the main memory and the network. Involving the OS significantly impacts performance; in many cases, data must be moved several times. Recent developments in networking [29, 30] have emphasized transfers that are executed without the involvement of the operating system, variously called "user-mode," "OS bypass," or "scheduled transfer." These combine hardware support with a programming model that allows higher network performance.

2.1.4 Summary

The design of a single-processor computer is a constant struggle against competing constraints. How should resources be allocated? Is it better to use transistors on a CPU chip to provide a larger fast L1 cache, or should they be used to improve the performance of some of the floating-point instructions? Should transistors be used to add more functional units? Should there be more registers, even if the ISA then has to change? Should the L1 cache be made larger at the expense of the L2 cache? Should the memory system be optimized for applications that make regular or irregular memory accesses? There are no easy answers here. The complexity has in fact led to increasingly complex CPU designs that use tens of millions of transistors and that are enormously costly to design and manufacture. Particularly difficult is the mismatch in performance between memory and CPU. This mismatch also causes problems for programmers; see, for example, [13] for a discussion of what should be a simple operation (bit reversal) but whose performance varies widely as a result of the use of caches and TLBs. These difficulties have encouraged computer architects to consider a wide variety of alternative approaches for improving computer system performance. Parallelism is one of the most powerful and most widely used.

2.2 Parallel Architectures

This section presents an overview of parallel architectures, considered as responses to limitations and problems in uniprocessor architectures and to technology opportunities. We start by considering parallelism in the memory systems, since the choices here have the most effect on programming models and algorithms. Parallelism in the CPU is discussed next; after increases in clock rates, this is a source of much of the improvement in sustained performance in microprocessors. For a much more detailed discussion of parallel computer architectures, see [2].

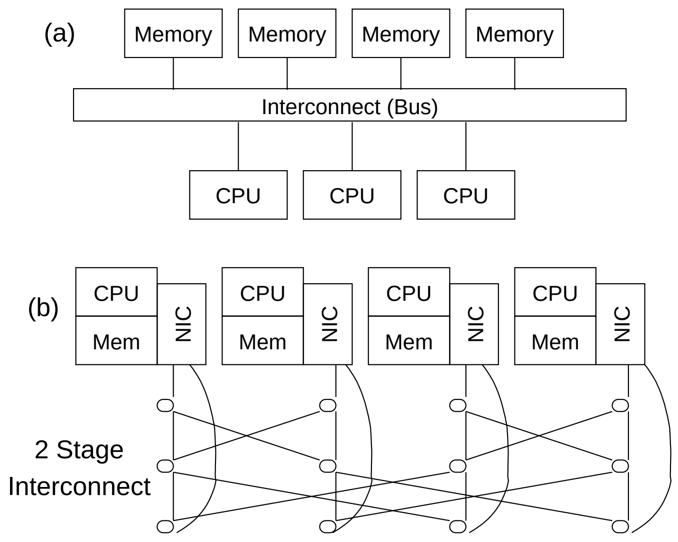

Figure 2.5: Schematic parallel computer organization. A typical shared-memory system is shown in (a) where the interconnect may be either a simple bus or a sophisticated switch. A distributed memory system is shown in (b); this may be either a distributed shared-memory system or a simpler shared-nothing system depending on the capabilities of the network interface (NIC).

2.2.1 Memory Parallelism

One of the easiest ways to improve performance of a computer system is simply to replicate entire computers and add a way for the separate computers to communicate data. This approach is shown schematically in Figure 2.5. This provides an easy way to increase memory bandwidth and aggregate processing power without changing the CPU, allowing parallel computers to take advantage of the huge investment in commodity microprocessor CPUs. The cost is in increased complexity of the software and in the impact that this has on the performance of applications. The major choice here is between distributed memory and shared memory.

Distributed Memory

The simplest approach from the hardware perspective is the *distributed memory*, or *shared nothing*, model. The approach here is to use separate computers connected by a network. The typical programming model consists of separate processes on each computer communicating by sending messages (*message passing*), usually by calling library routines. This is the most classic form of parallel computing, dating back to when the computers were people with calculators and the messages were written on slips of paper [22]. The modern distributed-memory parallel computer started with the work of Seitz [23].

Typical distributed-memory systems include the IBM SP and Beowulf clusters. The major feature that distinguishes between different distributed-memory parallel computers is the network that connects them. Different interconnects are described in Section 2.2.2. While the message-passing programming model has been successful, it emphasizes that the parallel computer is a collection of separate computers.

Shared Memory

A more complex approach ties the computers more closely together by placing all of the memory into a single (physical) address space and supporting virtual address spaces across all of the memory. That is, data is available to all of the CPUs through the load and store instructions of the ISA. Because access to the memory is through load and store operations rather than the network operations used in distributed-memory systems, access to remote memory has lower latency and higher bandwidth. These advantages come with a cost, however. The most serious problem is *consistency*. To understand this problem, consider the following simple Fortran program:

```
    a = a + 1
    b = 1
```

In a generic ISA, the part that increments `a` might be translated to

```
    ...
    LOAD R12, %A10  ;  Load a into register
    ADD  R12, #1    ;  Add one to the value in R12
    STORE R12, %A10 ;  Store the result back into A
    ...
```

The important point here is that the single program statement `a=a+1` turns into three separate instructions. Now, recall our discussion of cache memory. In a uniprocessor, the first time the `LOAD` operation occurs, the value is brought into the memory cache. The store operation writes the value from register *back into the cache.* Now, assume that another CPU, executing a program that is using the same address space, executes

```
10  if (b .eq. 0) goto 10
    print *, a
```

What value of `a` does that CPU see? We would like it to see the value of `a` after the increment. But that requires that the value has both been written back to the memory from the cache of the first CPU and read into cache (even if the corresponding cache line had previously been read into memory) on the second CPU. In other words, we want the program to execute as if the cache was not present, that is, as if every load and store operation worked directly on the memory. The copies of the memory in the cache are used only to improve performance of memory operations but do not change the behavior of programs that are accessing the same memory locations. Cache memory systems that accomplish this objective are called *cache coherent.* Ensuring that a memory system is cache coherent requires additional hardware and adds to the complexity of the system. On the other hand, it simplifies the job of the programmer, since the correctness of a program doesn't depend on details of the behavior of the cache. We will see, however, that while cache coherence is necessary, it is not sufficient to provide the programmer with a friendly programming environment.

The complexity of providing cache coherency has led to different designs. One important class is called *uniform memory access* (UMA). In this design, each memory and cache are connected to all of the others; each part observes any memory operation (such as a load from a memory location) and ensures that cache coherence is maintained. Because the time to access a location from memory (not from cache) is independent of the address (and hence particular memory unit), this is called UMA. Early implementations used a *bus*, which is a common signaling layer that each processor and memory were connected to. Because buses are not scalable (all devices on the bus must share a limited amount of communication), higher-performance UMA systems based on completely connected networks have been constructed. Such networks themselves are not scalable (the number of connections for p components grows as p^2), leading to the other class of shared memory designs.

The *nonuniform memory access* (NUMA) approach does not require that all memory be equally "distant" (in terms of access time). Instead, the memory may be connected by a scalable network. Such systems can be more sensitive to the details of data layout but can also scale to much larger numbers of processors. To emphasize that a NUMA system is cache coherent, the term CC-NUMA is often used. The term *distributed shared memory* (DSM) is also often used to emphasize the NUMA characteristics of this approach to building shared-memory hardware. The term *virtual shared memory*, or *virtual distributed shared memory*, is used to describe a system that provides the programmer with a shared-memory programming model built on top of distributed-memory (not DSM) hardware.

Typical UMA systems include the Sun E10000 (up to 64 processors) and SGI Power Challenge (up to 18 processors). Typical CC-NUMA systems include the SGI Origin (typically up to 128 processors, 1024 in special configurations) and the HP Exemplar V2600 (up to 128 processors). The SGI Origin uses an approach called directory-based cache coherency (directory caches, for short) [16] to distribute the information needed to maintain cache coherency across the network that connects the memory to the CPUs.

Memory Consistency and Programming Models

How does the programming model change when several threads or processes share memory? What are the new issues and concerns? Consider a uniprocessor CPU executing a single-user program (a single-threaded, single-process program). Programs execute simply, one statement after the other. Implicit in this is that all statements before the current statement have completed before the current statement is executed. In particular, all stores to and loads from memory issued by previous statements have completed before the current statement begins to execute. In a multiprocessor executing a single program on multiple processors, the notion of "current" statement and "completed before" is unclear. Or rather, it can be defined to be clear, but only at a high cost in performance.

The fundamental observation is by Lamport [14] in an article titled "How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs". From a programmer's perspective, a parallel program should execute as if it were some arbitrary interleaving (but preserving order) of the statements in the program. This requirement is called *sequential consistency.* Unfortunately, while this matches the way most programmers look at their code, it imposes severe constraints on the hardware, in large part because of the high latency of memory accesses relative to the CPU speed.

Because providing sequential consistency limits performance, weaker models have been proposed. One model proposed in the late 1980s, called *processor consistency* [7], matched many of the then-current multiprocessor implementations but (usually) required some explicit action by the programmer to ensure correct program behavior. Programmers who use the thread programming model with thread locks to synchronize accesses to shared data structures satisfy this requirement because the implementation of the lock and unlock calls in the thread library ensures that the correct instructions are issued.

Some programmers prefer to avoid the use of locks, however, because of their relatively high overhead and instead use flag variables to control access to shared data (as we used `a` as the flag variable in the preceding section). *Weak consistency* [4] is appropriate for such programs; like processor consistency, the programmer is required to take special steps to ensure correct operation.

Even weak consistency interferes with some performance optimizations, however. For this reason, *release consistency* [6] was introduced. This form of consistency separates synchronization between two processes or threads into an *acquire* and a *release* step.

The important point for programmers and algorithm developers is that the programming model that is most natural for programmers and that reflects the way we read programs is sequential consistency, and this model is not implemented by parallel computer hardware. Consequently, the programmer cannot rely on programs executing as some interleaved ordering of the statements. The specific consistency model that is implemented by the hardware may require different degrees of additional specification by the programmer. Language design for parallel programming may take the consistency model into account, providing ways for the compiler, not the programmer, to enforce consistency. Unfortunately, most languages (including C, C++, and Fortran) were designed for single threads of control and do not provide any mechanism to enforce consistency.

Note that if memory latency was small, providing sequential consistency would not greatly impact performance. Weaker forms of consistency would not be needed, and Lamport's title [14] would reflect real machines.

Other Approaches

Two other approaches to parallelism in memory are important. In both of these, the CPU is customized to work with the memory system. In single instruction, multiple data (SIMD) parallelism, simplified CPUs are connected to memory. Unlike the previous cases, in the SIMD approach, each CPU executes the *same* instruction in each clock cycle. Such systems are well suited for the *data-parallel* programming model, where data is divided up among memory systems and the same operation is performed on each data element. For example, the Fortran code

```
do i=1, 10000
    a(i) = a(i) + alpha * b(i)
enddo
```

can be converted into a small number of instructions, with each CPU taking a part of the arrays `a` and `b`. While these systems have fallen out of favor as general-purpose computers, they are still important in fields such as signal processing. The most famous general-purpose SIMD system was the Connection Machine (CM-1 and CM-2) [9].

The other major approach is vector computing. This is often not considered parallelism because the CPU has little explicit parallelism, but parallelism is used in the memory system. In vector computing, operations are performed on *vectors*, typically groups of 64 floating-point numbers. A single instruction in a vector computer may cause 64 results to be computed (often with a pipelined floating-point unit), using vectors stored in vector registers. Data transfers from memory to vector registers make use of multiple memory *banks*; the parallelism in the memory supports very high bandwidths between the CPU and the memory. Vector computers often have memory bandwidths that are an order of magnitude or more greater than nonvector computers. We will come back to vector computing in Section 2.2.3 after discussing parallelism in the CPU.

The most important vector machine was the Cray 1; most current vector machines are in fact combinations of multiple vector CPUs with shared memory. These systems are often called *parallel vector processors* (PVPs). Systems of this type include the NEC SX-5 and the Cray T90. These systems may not fully support cache coherency, trading some software complexity for faster performance in their memory systems.

Parallel Random Access Memory

A great deal of theoretical work on the complexity of parallel computation has used the parallel random access memory model (PRAM). This is a theoretical model of a shared-memory computer; different varieties of PRAM vary in the details of how memory accesses to the same address are handled. In order to make the theoretical model tractable, memory access times are usually considered constant independent of the CPU performing the (nonconflicting) access; in particular, there are no caches and no factors of one hundred or more difference in access times for different memory locations. While this model is valuable in understanding the limits of parallel algorithms, the PRAM model represents an abstraction that cannot be efficiently implemented in practice.

Limits to Memory System Performance

One interesting limit to memory system performance comes from applying Little's law to memory requests. Little's law is a result from queuing theory; applied to memory requests, it says that if the memory latency that needs to be hidden is L and the rate of requests is r, then the number of simultaneously active requests needed is rL. If this is cast in terms of clock cycles, if the memory latency is 100 cycles and a memory request is issued every cycle, then 100 requests must be active at the same time. The consequences are numerous:

1. The bandwidth of the memory system must support more requests (the number uses the same formula but uses the latency of the interconnect, which may still be around 10 cycles).

2. There must be enough independent work. Some algorithms, particularly those that use recurrence relations, do not have much independent work. This situation places a burden on the algorithm developer and the programmer.

3. The compiler must convert the program into enough independent requests, and there must be enough resources (such as registers) to hold results as they arrive (load) or until they depart (store).

Many current microprocessors allow a small number of outstanding memory operations; only the Cray MTA (discussed below) satisfies the requirements of Little's law for main-memory accesses.

2.2.2 Interconnects

In the preceding section, we described the interaction of memories and CPUs. In this section we say a little more about the interconnection networks that are used to connect components in a computer (parallel or otherwise).

Many types of networks have been used in the past thirty years for constructing parallel systems, ranging from relatively simple buses, to 2D and 3D meshes, to complex hypercube network topologies [15]. Each type of network can be described by its topology, its means of dealing with congestion (e.g., blocking or nonblocking), its approach to message routing, and its bandwidth characteristics.

For a long time, understanding details of the topology was important for programmers and algorithm developers seeking to achieve high performance. This situation is reflected both in the literature and in parallel programming models (e.g., the topology routines in MPI). Recently, networks have improved to the point that for many users, network topology is no longer a major factor in performance. However, some of this apparent "flatness" (uniformity) in the topology comes from greatly increased bandwidth within the network. As network endpoints become faster, network topology may again become an important consideration in algorithms and programming models. Congestion in the network can still be a problem if the network performance doesn't scale with the number of processing nodes. The term *bisection bandwidth* describes the bandwidth of the network across any cut that divides the network into two parts.

Note that there is no best approach. Simple mesh networks, such as those used in the Intel TFLOPS (ASCI Red) system, provide effective scalability for many applications through low latency and high bandwidth, even though a mesh network does not have scalable performance in the sense that the bisection bandwidth of a mesh does not grow proportionally with the number of nodes. It is scalable in terms of the hardware required: there is a constant cost per node for each node added.

When interconnects are viewed as networks between computers, the performance goals have been quite modest. Fast networks of this type typically have latencies of ten microseconds or more (including essential software overheads) and bandwidths on the order of 100 MB/s. Interconnects used to implement shared memory, on the other hand, are designed to operate at memory system speeds and with no extra software overhead. Latencies for these systems are measured in nanoseconds and bandwidths of one to ten gigabytes per second are becoming common.

Early shared-memory systems used a bus to connect memory and processors. A bus provides a single, shared connection that all devices use and is relatively inexpensive to build. The major drawback is that if k devices are using the bus at the same time, under the best of conditions, each gets $1/k$ of the available performance (e.g., bandwidth). Contention between devices on the bus can lower the available bandwidth considerably.

To address this problem, some shared-memory systems have chosen to use networks that connect each processor with each memory system. For small numbers of processors and memories, a direct connection between each processor and memory is possible (requiring p^2 connections for p devices); this is called a *full crossbar*. For larger numbers of processors, a less complete network may be used.

An interesting development is the convergence of the technology used for networking and for shared memory. The *scalable coherent interconnect* (SCI) [11] was an early attempt to provide a memory-oriented view of interconnects and has been used to build CC-NUMA systems from Convex and HP. Building on work both in research and in industry, the VIA [29] and Infiniband [12] industry-standard interconnects allow data to be moved directly from one processor's memory to another along an established circuit. These provide a communication model that is much closer to that used in memory interconnects, and should offer much lower latencies and higher bandwidths than older, message-oriented interconnects.

Systems without hardware cache coherency often provide a way to indicate that all copies of data in a cache should be discarded; this is called *cache invalidation*. Sometimes this is a separate instruction; sometimes it is a side effect of a synchronization instruction such as test-and-set (e.g., Cray SV-1). Software can use this strategy to ensure that programs operate correctly. The cost is that *all* copies of data in the cache are discarded; hence, subsequent operations that reference memory locations stall while the cache is refilled. To avoid this situation, some systems allow individual cache lines to be invalidated rather than the entire cache. However, such an approach requires great care by the software, since the failure to invalidate a line containing data that has been updated by another processor can lead to incorrect and nondeterministic behavior by the program.

For an engaging discussion of the challenges of implementing and programming shared-memory systems, see [21].

2.2.3 CPU Parallelism

Parallelism at the level of the CPU is more difficult to implement than simple replication of CPUs and memory, even when the memory presents a single shared address space. However, modest parallelism in the CPU provides the easiest route to improved performance for the majority of applications because little needs to be done by the programmer to exploit this kind of parallelism.

Superscalar Processing

Look at Figure 2.2 again, and consider the following program fragment:

```
real a, b, c
integer i, j, k
...
a = b * c
i = j + k
```

The values `a`, `b`, `c`, `i`, `j`, and `k` are already in register. These two statements use different functional units (FPU and ALU, respectively) and different register sets (FPR and GPR). A *superscalar* processor can execute both of these statements (each requiring a single register-to-register instruction) in the same clock cycle (more precisely, such a processor will "begin execution" of the two statements, since both may be pipelined). The term superscalar comes from the fact that more than one operation can be performed in a single clock cycle and that performance is achieved on nonvector code. A superscalar processor allows as much parallelism as there are functional units. Because separate instructions are executed in parallel, this is also called *instruction-level parallelism* (ILP). For ILP to be effective, it must be easy for the hardware to find instructions that do not depend on each other and that use different functional units. Consider the following example. If the CPU executes instructions in the order that they appear, then the code sequence on the left will take three cycles and the one on the right only two cycles.

```
    a = b * c          a = b * c
    d = e * f          i = j + k
    i = j + k          d = e * f
    l = m + n          l = m + n
```

Some CPUs will attempt to reorder instructions in the CPU's hardware, an action that is most beneficial to legacy applications that cannot be recompiled. It is often better, however, if the compiler *schedules* the instructions for effective use of ILP; for example, a good code-scheduling compiler would transform the code on the left to the code on the right (but breaking sequential consistency!).

One major drawback of ILP, then, is that the hardware must rediscover what a scheduling compiler already knows about the instructions that can be executed in the same clock cycle.

Explicitly Parallel Instructions

Another approach is for the instruction set to encode the use of each part of the CPU. That is, each instruction contains explicit subinstructions for each of the different functional units in the CPU. Since each instruction must explicitly specify more details about what happens in each clock cycle, instructions result that are longer than in other ISAs. In fact, they are usually referred to as very long instruction word (VLIW) ISAs. VLIW systems usually rely on the compiler to schedule each functional unit. One of the earliest commercial VLIW machines was the Multiflow Trace. The Intel IA64 ISA is a descendent of this approach; the term EPIC (explicitly parallel instruction computing) is used for the Intel variety. EPIC does relax some of the restrictions of VLIW but still relies on the compiler to express most of the parallelism.

SIMD and Vectors

One approach to parallelism is to apply the same operation to several different data values, using multiple functional units. For example, a single instruction might cause four values to be added to four others, using four separate adders. We have seen this SIMD style of parallelism before, when applied to separate memory units. The SIMD approach is used in some current processors for special operations. For example, the Pentium III includes a small set of SIMD-style instructions for single-precision floating-point and related data move operations. These are designed for use in graphics transformations that involve matrix-vector multiplication by 4×4 matrices.

Vector computers use similar techniques in the CPU to achieve greater performance. A vector computer can apply the same operation to a collection of data called a vector; this is usually either successive words in memory or words separated by a constant offset or *stride*. Early systems such as the CDC Star 100 and Cyber 205 were vector memory-to-memory architectures where vectors could be nearly any length. Since the Cray 1, most vector computers have used vector registers, typically limiting vectors to 64 elements. The big advantage of vector computing comes from the regular memory access that a vector represents. Through the use of pipelining and other techniques such as chaining, a vector computer can completely hide the memory latency by overlapping the access to the next vector with operations on a current vector.

Vector computing is related to VLIW or explicitly parallel computing in the sense that each instruction can specify a large amount of work and that advanced compilers are needed to take advantage of the hardware. Vectors are less flexible than the VLIW or EPIC approach but, because of the greater regularity, can sustain higher performance on applications that can be expressed in terms of vectors.

Multithreading

Parallelism in the CPU involves executing multiple sets of instructions. Any one of these sets, along with the related virtual address space and any state, is called a *thread*. Threads are most familiar as a software model (see Chapter 10), but they are also a hardware model. In the usual hardware model, a thread has no explicit dependencies with instructions in any other thread, although there may be implicit dependencies through operations on the same memory address. The critical issues are (1) How many threads issue operations in each clock cycle? and (2) How many clock cycles does it take to switch between different threads?

Simultaneous multithreading (SMT) [27] allows many threads to issue instructions in each clock cycle. For example, if there are four threads and four functional units, then as long as each functional unit is needed by some thread in each clock cycle, all functional units can be kept busy every cycle, providing maximum use of the CPU hardware. The compiler or programmer must divide the program into separately executing threads. The SMT approach is starting to show up in CPU designs including versions of the Compaq Alpha and IBM Power processors.

Fine-grained multithreading uses a single thread at a time but allows the CPU to change threads in a single clock cycle. Thus, a thread that must wait for a slow operation (anything from a floating point addition to a load from main memory) can be "set aside," allowing other threads to run. Since a load from main memory may take 100 cycles or more, the benefit of this approach for hiding memory latency is apparent. The drawback when used to hide memory latency can be seen by applying Little's law. Large numbers of threads must be provided for this approach to succeed in completely hiding the latency of main (rather than cache) memory. The Cray MTA is the only commercial architecture to offer enough threads for this purpose.

All of these techniques can be combined. For example, fine-grained multithreading can be combined with superscaler ILP or explicit parallelism. SMT can restrict groups of threads to particular functional units in order to simplify the processor design, particular in processors with multiple FPUs and ALUs.

2.2.4 I/O and Networks

Just as in the uniprocessor case, I/O and networking have not received the same degree of attention as have CPU and memory performance. Fortunately, the lower performance levels of I/O and networking devices relative to CPU and memory allow a simpler and less expensive architecture. On the other hand, lower performance puts tremendous strain on the architect trying to maintain balance in the system. A common I/O solution for parallel computers, particularly clusters, is not a parallel file system but rather a conventional file system, accessed by multiple processors.

Recall that data caches are often used to improve the performance of I/O systems in uniprocessors. As we have seen, it is important to maintain consistency between the different caches and between caches and memory if correct data is to be provided to programs. Unfortunately, particularly for networked file systems such as NFS, maintaining cache consistency seriously degrades performance. As a result, such file systems allow the system administrator to trade performance against cache coherence. For environments where most applications are not parallel or do not have multiple processes accessing the same file at the same time, cache-coherence is usually sacrificed in the name of speed.

The Redundant arrays of inexpensive disks (RAID) approach is an example of the benefits of parallelism in I/O. RAID was first proposed in 1988 [19] with five different levels representing different uses of multiple disks to provide fault tolerance (disks, being mechanical, fail more often than entirely electronic components) while maintaining a balance between read rates, write rates, and efficient use of storage. The RAID approach has since been generalized to additional levels. Both hardware (RAID managed by hardware, presenting the appearance of a single but faster and/or more reliable disk) and software (separate disks managed by software) versions exist.

Parallel I/O can also be achieved by using arrays of disks arranged in patterns different from those described by the various RAID levels. Chapter 11 describes parallel I/O from the programmer's standpoint. A more detailed discussion of parallel I/O can be found in [17].

The simplest form of parallelism in networks is the use of multiple paths, each carrying part of the traffic. Networks within a computer system often achieve parallelism by simply using separate wires for each bit. Less tightly coupled systems, such as Beowulf clusters, sometimes use a technique called *channel bonding*, which uses multiple network paths, each carrying part of the message. GridFTP [1] is an example of software that exploits the ability of the Internet to route data over separate paths to avoid congestion in the network.

A more complex form of parallelism is the use of different electrical or optical frequencies to concurrently place several messages on the same wire or fiber. This approach is rarely used within a computer system because of the added cost and complexity, but it is used extensively in long-distance networks. New techniques for optical fibers, such as dense wavelength division multiplexing (DWDM), will allow a hundred or more signals to share the same optical fiber, greatly increasing bandwidth.

2.2.5 Support for Programming Models

Special operations are needed to allow processes and threads that share the same address space to coordinate their actions. For example, one thread may need to keep others from reading a location in memory until it is done modifying that location. Such protection is often provided by *locks*: any thread that wants to access the particular data must first acquire the lock, releasing the lock when it is done. A lock, however, is not easy to implement with just load and store operations (though it can be done). Instead, many systems provide compound instructions that can be used to implement locks, such as test-and-set or fetch-and-increment. RISC systems often provide a "split" compound instruction that can be used to build up operations such as fetch-and-increment based on storing a result after reading from the same address only if no other thread or process has accessed the same location since the load.

Because rapid synchronization is necessary to support fine-grained parallelism, some systems (particularly PVPs) use special registers that all CPUs can access. Other systems have provided extremely fast *barriers*: no process can leave a barrier until all have entered the barrier. In a system with a fast barrier, a parallel system can be viewed as sequentially consistent where an "operation" is defined as the group of instructions between two barriers. This provides an effective programming model for some applications.

In distributed-memory machines, processes share no data and typically communicate through messages. In shared-memory machines, processes directly access data. There is a middle ground: remote memory access (RMA). This is similar to the network-connected distributed-memory system except that additional hardware provides put and get operations to store to or load from memory in another node. The result is still a distributed-memory machine, but one with very fast data transfers. Examples are the Compaq AlphaServer SC, Cray T3D and T3E, NEC Cenju 4, and Hitachi SR8000.

2.2.6 Summary

Parallelism is a powerful approach to improving the performance of a computer system. All systems employ some degree of parallelism, even if it is only parallel data paths between the memory and the CPU. Parallelism is particularly good at solving problems related to bandwidth or throughput; it is less effective at dealing with latency or startup costs. However, the ability to switch between tasks provides one way to hide latency as long as enough independent tasks can be found. Parallelism does not come free, however. The effects of memory latency are particularly painful, forcing complex consistency models on the programmer and difficult design constraints on the hardware designer.

In the continuing quest for ever greater performance, today's parallel computers often combine many of the approaches discussed here. One of the most popular is distributed-memory clusters of nodes, where each node is a shared-memory processor, typically with 2 to 16 processors, though some clusters have SMP nodes with as many as 128 processors. Another important class of machines is the parallel vector processors; these use vector-style CPU parallelism combined with shared memory.

We emphasize that hardware models and software (or programming) models are essentially disjoint; shared-memory hardware provides excellent message-passing support, and distributed-memory hardware can (at sometimes substantial cost) support a shared-memory programming model.

We close this section with a brief mention of taxonomies of parallel computers. A taxonomy of parallel computers provides a way to identify the important features of a system. Flynn [5] introduced the best known taxonomy that defines four different types of computer based on whether there are multiple data streams and/or multiple instruction streams. A conventional uniprocessor has a single instruction stream and a single data stream and is denoted SISD. Most of the parallel computers that we have described in this section have both multiple data and multiple instruction streams (because they have many memories and CPUs); these are called MIMD. The single instruction but multiple data parallel computer, or SIMD, has already been mentioned. The fourth possibility is the multiple instruction, single data, or MISD; this category is not used. A standard taxonomy for MIMD architectures has not yet emerged, but it is likely to be based on whether the memory is shared or distributed and, if it is shared, whether it is cache coherent and how access time varies. Many of the terms used to describe these alternatives have been discussed above, including UMA, CC-NUMA, and DSM.

The term *single program, multiple data* (SPMD) is inspired by Flynn's taxonomy. Because the single program has branches and other control-flow constructs, SPMD is a subset of MIMD, not a subset of SIMD programs. Using a single program, however, does provide an important simplification, and most parallel programs in technical and scientific computing are SPMD.

2.3 Today's Parallel Systems

Most systems today are hybrids, combining different technologies to provide the greatest possible computing power within the limits of cost, complexity, and usability. Because most systems are hybrids, there is no unique taxonomy. This section divides parallel systems in terms of their user communities: parallel vector processors, shared memory, and distributed memory. See [28] for a review of current supercomputers, including large-scale parallel systems.

This section covers only those systems typically used for scientific computing. Parallelism is widely used in commercial computing for applications such as databases and Web servers. Special architectures and hardware have been developed to support these applications, including special hardware support for synchronization and fault tolerance.

2.3.1 Parallel Vector Processors

Parallel vector processors represent one of the most powerful classes of parallel computer, combining impressive per processor performance with parallelism. As late as 1996, the top machines on the Top 500 list of supercomputers were parallel vector processors [26], and since then only massively parallel systems with thousands of processors are faster.

The fastest of these machines may not provide full cache coherency in hardware; instead, they may require some support from the software to maintain a consistent view of memory. Machines in this category include the NEC SX-5 and Cray SV1. This is an example of the sort of tradeoff of performance versus cost and complexity that continues to face architects of parallel systems.

A distinguishing feature of vector processors and parallel vector processors is the high memory bandwidth, often 4–16 bytes per floating-point operation. This is reflected in the high sustained performance achieved on these machines for many scientific applications.

2.3.2 Shared Memory

Shared-memory systems are becoming common, even for desktop systems. Most vendors include shared-memory systems among their offerings, including Compaq, HP, IBM, SGI, and Sun and many personal computer vendors. Most of these systems have between 2 and 16 processors, with a few providing up to 128 processors. Both UMA and CC-NUMA designs are common.

In contrast to PVPs, these systems usually have quite modest memory bandwidths. At the low end, in fact, the same aggregate memory bandwidth may be provided to systems with 1 to 4 or even 16 processors. As a result, some of these systems are often starved for memory bandwidth. This can be a problem for applications that do not fit in cache. Applications that are memory access bound can even slow down as processors are added in such systems. Of course, not all systems are underpowered, and the memory performance of even the low end systems has been improving rapidly.

2.3.3 Distributed Memory

Distributed-memory systems are the most common because they are the easiest to assemble. Systems from Intel, particularly the Paragon and the 512-processor Delta, were important in demonstrating that applications could make effective use of large numbers of processors. Perhaps the most successful commercial distributed-memory system is the IBM SP family. SP systems combine various versions of the successful RS6000 workstation and server nodes with different interconnects to provide a wide variety of parallel systems, from 8 processors to the 8192-processor ASCI White system.

As mentioned above, some distributed-memory systems have been built with special-purpose hardware that provides remote memory operations such as put and get. The most successful of these are the Cray T3D and T3E systems.

Many groups have exploited the low cost and relatively high performance of commodity microprocessors to build clusters of personal computers or workstations. Early versions of these were built from desktop workstations and were sometimes referred to as NOWs, for networks of workstations. The continued improvement in performance of personal computers, combined with the emergence of open source (and free) versions of the Unix operation system, gave rise to clusters of machines. These systems are now widely know as Beowulfs or Beowulf clusters, from a project begun by Thomas Sterling and Donald Becker at NASA [24, 25]. They are real parallel machines; as of 2000, two of the top 100 supercomputer systems were built from commodity parts.

2.4 Future Directions for Parallel Architectures

In some ways, the future of parallel architectures, at least for the next five years, is clear. Most parallel machines will be hybrids, combining nodes containing a modest number of commodity CPUs sharing memory in a distributed-memory system. Many users will have only one shared-memory node; for them, shared-memory programming models will be adequate. In the longer term, the picture is much hazier. Many challenges will be difficult to overcome. Principal among these are memory latency and the limits imposed by the speed of light. Heat dissipation is also becoming a major problem for commodity CPUs. One major contributor to the increase in clock speeds for CPUs has been a corresponding decrease in the size of the features on the CPU chip. These feature sizes are approaching the size of a single atom, beyond which no further decrease is possible.

While these challenges may seem daunting, they offer an important opportunity to computer architects and software scientists—an opportunity to take a step that is more than just evolutionary.

Exotic Parallel Architectures

As we have discussed above, one of the major problems in designing any computer is providing a high-bandwidth, low-latency path between the CPU and memory. Some of this cost comes from the way DRAMs operate: data is stored in rows; when an item is needed, the entire row is read and the particular bit is extracted; the other bits in the row are discarded. This simplifies the construction of the DRAM (separate wires are not needed to get to each bit), but it throws away significant bandwidth. Observing that DRAM densities are increasing at a rate even faster than the rate at which commodity software demands memory, several researchers have explored combining the CPU and memory on the same chip and using the entire DRAM row rather than a single bit at a time. In fact, an early commercial version of this approach, the Mitsubishi M32000D3 processor, used a conventional, cache-oriented RISC processor combined with memory and organized so that a row of the memory was a cache line, allowing for enormous (for the time) bandwidth in memory-cache transfers. Several different architectures the exploit processors and memory in the same chip are currently being explored [3, 20], including approaches that consider vector-like architectures and approaches that place multiple processors on the same chip. Other architects are looking a parallel systems built from such chips; the IBM Blue Gene [10] project expects to have a million processor system (with around 32 processors per node).

Superconducting elements promise clock speeds of 100 GHz or more. Of course, such advances will only exacerbate the problem of the mismatch between CPU and memory speeds. Designs for CPUs of this kind often rely on hardware multithreading techniques to reduce the impact of high memory latencies.

Computing based on biological elements often seeks to make use of parallelism by using molecules as processing elements. Quantum computing, particularly quantum computing based on exploiting the superposition principle, is a fundamentally different kind of parallelism.

2.5 Conclusion

Parallel architecture continues to be an active and exciting area of research. Most systems now have some parallelism, and the trends point to increasing amounts of parallelism at all levels, from 2 to 16 processors on the desktop to tens to hundreds of thousands for the highest-performance systems.

Access to memory continues to be a major issue; hiding memory latency is one area where parallelism doesn't provide a (relatively) simple solution. The architectural solutions to this problem have included deep memory hierarchies (allowing the use of low-latency memory close to the processor), vector operations (providing a simple and efficient "prefetch" approach), and fine-grained multithreading (enabling other work to continue while waiting on memory). In practice, none of these approaches completely eliminates the problem of memory latency. The use of low-latency memories, such as caches, suffers when the data does not fit in the cache. Vector operations require a significant amount of regularity in the operations that may not fit the best (often adaptive) algorithms, and multithreading relies on identifying enough independent threads. Because of this, parallel programming models and algorithms have been developed that allow the computational scientist to make good use of parallel systems. That is the subject of the rest of this book.

Bibliography

[1] W. Allcock, J. Bester, J. Bresnahan, A. Chervenak, Lee Liming, and Steven Tuecke. GridFTP: Protocol extensions to FTP for the Grid, March 2001. http://www-fp.mcs.anl.gov/dsl/GridFTP-Protocol-RFC-Draft.pdf.

[2] David E. Culler, Jaswinder Pal Singh, and Anoop Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers, San Francisco, 1999.

[3] Diva (Data IntensiVe Architecture) home page. `http://www.isi.edu/asd/diva/`.

[4] Michael Dubois, Christopher Scheurich, and Faye Briggs. Memory access buffering in multiprocessors. In *Proc. 13th Annual International Symposium on Computer Architecture, Computer Architecture News*, pages 434–442. ACM Press, June 1986.

[5] M. Flynn. Very high speed computing. *Proceedings of the IEEE*, pages 1901–1909, 1966.

[6] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Philip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proc. 17th Annual Symposium on Computer Architecture (17th ISCA'90), Computer Architecture News*, pages 15–26, Seattle, June 1990. ACM Press.

[7] J. R. Goodman. Cache consistency and sequential consistency. Technical Report 61, IEEE Scalable Coherence Interface Working Group, March 1989.

[8] John L. Hennessy and David A. Patterson. *Computer Architecture A Quantative Approach*. Morgan Kaufman, San Mateo, CA, 1990.

[9] W. Daniel Hillis. *The Connection Machine*. Series in Artificial Inteligence. MIT Press, Cambridge, MA, 1985.

[10] Blue Gene. `http://www.research.ibm.com/bluegene/`.

[11] *1596-1992 IEEE Standard for Scalable Coherent Interface (SCI)*, 1992.

[12] Infiniband trade association. `http://www.infinibandta.com`.

[13] Alan H. Karp. Bit reversal on uniprocessors. *SIAM Review*, 38(1):1–26, March 1996.

[14] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[15] F. T. Leighton. *Introduction to Parallel Architectures: Arrays, Trees, Hypercubes.* Morgan Kaufmann Publishers, 1992.

[16] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica S. Lam. The Stanford Dash multiprocessor. *Computer*, 25(3):63–79, March 1992.

[17] John M. May. *Parallel I/O for High Performance Computing.* Morgan Kaufmann Publishers, Los Altos, CA, 2000.

[18] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics Magazine*, 38(8):114–117, April 1965.

[19] D. Patterson, G. Gibson, and R. Katz. A Case for Redundant Arrays of Inexpensive Disks. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 109–116. ACM Press, June 1988.

[20] David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick. A case for intelligent RAM. *IEEE Micro*, 17(2):34–44, March/April 1997.

[21] G. F. Pfister. *In Search of Clusters.* Prentice–Hall, Inc., 2 edition, 1998.

[22] Louis Frye Richarson. *Weather Prediction by Numerical Process.* Cambridge University Press, 1922.

[23] C. L. Seitz. The cosmic cube. *Communications of the ACM*, 28(1):22–33, January 1985.

[24] T. Sterling, D. Savarese, D. J. Becker, J. E. Dorband, U. A. Ranawake, and C. V. Packer. BEOWULF : A parallel workstation for scientific computation. In *International Conference on Parallel Processing, Vol. 1: Architecture*, pages 11–14, Boca Raton, FL, August 1995. CRC Press.

[25] Thomas L. Sterling, John Salmon, Donald J. Becker, and Daniel F. Savarese. *How to Build a Beowulf.* MIT Press, 1999.

[26] Top 500 supercomputers, November 1996. `http://www.top500.org/lists/1996/11/`.

[27] Dean M. Tullsen, Susan Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proc. 22nd Annual International Symposium on Computer Architecture*, pages 392–403. ACM Press, June 1995.

[28] Aad J. van der Steen and Jack J. Dongarra. Overview of recent supercomputers, 2000. `http://www.phys.uu.nl/~steen/web00/overview00.html`.

[29] VI Architecture. `http://www.viarch.org`.

[30] Thorsten von Eicken, Anindya Basu, Vineet Buch, and Werner Vogels. U-Net: A user-level network interface for parallel and distributed computing. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, pages 40–53. ACM Press, December 1995.