

## 1.1 Motivating Parallelism

Development of parallel software has traditionally been thought of as time and effort intensive. This can be largely attributed to the inherent complexity of specifying and coordinating concurrent tasks, a lack of portable algorithms, standardized environments, and software development toolkits. When viewed in the context of the brisk rate of development of microprocessors, one is tempted to question the need for devoting significant effort towards exploiting parallelism as a means of accelerating applications. After all, if it takes two years to develop a parallel application, during which time the underlying hardware and/or software platform has become obsolete, the development effort is clearly wasted. However, there are some unmistakable trends in hardware design, which indicate that uniprocessor (or implicitly parallel) architectures may not be able to sustain the rate of *realizable* performance increments in the future. This is a result of lack of implicit parallelism as well as other bottlenecks such as the datapath and the memory. At the same time, standardized hardware interfaces have reduced the turnaround time from the development of a microprocessor to a parallel machine based on the microprocessor. Furthermore, considerable progress has been made in standardization of programming environments to ensure a longer life-cycle for parallel applications. All of these present compelling arguments in favor of parallel computing platforms.

### 1.1.1 The Computational Power Argument from Transistors to FLOPS

In 1965, Gordon Moore made the following simple observation:

*"The complexity for minimum component costs has increased at a rate of roughly a factor of two per year. Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least 10 years. That means by 1975, the number of components per integrated circuit for minimum cost will be 65,000."*

His reasoning was based on an empirical log-linear relationship between device complexity and time, observed over three data points. He used this to justify that by 1975, devices with as many as 65,000 components would become feasible on a single silicon chip occupying an area of only about one-fourth of a square inch. This projection turned out to be accurate with the fabrication of a 16K CCD memory with about 65,000 components in 1975. In a subsequent paper in 1975, Moore attributed the log-linear relationship to exponential behavior of die sizes, finer minimum dimensions, and "circuit and device cleverness". He went on to state that:

*"There is no room left to squeeze anything out by being clever. Going forward from here we have to depend on the two size factors - bigger dies and finer dimensions."*

He revised his rate of circuit complexity doubling to 18 months and projected from 1975 onwards at this reduced rate. This curve came to be known as "Moore's Law".

Formally, Moore's Law states that circuit complexity doubles every eighteen months. This empirical relationship has been amazingly resilient over the years both for microprocessors as well as for DRAMs. By relating component density and increases in die-size to the computing power of a device, Moore's law has been extrapolated to state that the amount of computing power available at a given cost doubles approximately every 18 months.

The limits of Moore's law have been the subject of extensive debate in the past few years. Staying clear of this debate, the issue of translating transistors into useful OPS (operations per second) is the critical one. It is possible to fabricate devices with very large transistor counts. How we use these transistors to achieve increasing rates of computation is the key architectural challenge. A logical recourse to this is to rely on parallelism ♦ both implicit and explicit. We will briefly discuss implicit parallelism in [Section 2.1](#) and devote the rest of this book to exploiting explicit parallelism.

### **1.1.2 The Memory/Disk Speed Argument**

The overall speed of computation is determined not just by the speed of the processor, but also by the ability of the memory system to feed data to it. While clock rates of high-end processors have increased at roughly 40% per year over the past decade, DRAM access times have only improved at the rate of roughly 10% per year over this interval. Coupled with increases in instructions executed per clock cycle, this gap between processor speed and memory presents a tremendous performance bottleneck. This growing mismatch between processor speed and DRAM latency is typically bridged by a hierarchy of successively faster memory devices called caches that rely on locality of data reference to deliver higher memory system performance. In addition to the latency, the net effective bandwidth between DRAM and the processor poses other problems for sustained computation rates.

The overall performance of the memory system is determined by the fraction of the total memory requests that can be satisfied from the cache. Memory system performance is addressed in greater detail in [Section 2.2](#). Parallel platforms typically yield better memory system performance because they provide (i) larger aggregate caches, and (ii) higher aggregate bandwidth to the memory system (both typically linear in the number of processors). Furthermore, the principles that are at the heart of parallel algorithms, namely locality of data reference, also lend themselves to cache-friendly serial algorithms. This argument can be extended to disks where parallel platforms can be used to achieve high aggregate bandwidth to secondary storage. Here, parallel algorithms yield insights into the development of out-of-core computations. Indeed, some of the fastest growing application areas of parallel computing in data servers (database servers, web servers) rely not so much on their high aggregate computation rates but rather on the ability to pump data out at a faster rate.

### **1.1.3 The Data Communication Argument**

As the networking infrastructure evolves, the vision of using the Internet as one large heterogeneous parallel/distributed computing environment has begun to take

shape. Many applications lend themselves naturally to such computing paradigms. Some of the most impressive applications of massively parallel computing have been in the context of wide-area distributed platforms. The SETI (Search for Extra Terrestrial Intelligence) project utilizes the power of a large number of home computers to analyze electromagnetic signals from outer space. Other such efforts have attempted to factor extremely large integers and to solve large discrete optimization problems.

In many applications there are constraints on the location of data and/or resources across the Internet. An example of such an application is mining of large commercial datasets distributed over a relatively low bandwidth network. In such applications, even if the computing power is available to accomplish the required task without resorting to parallel computing, it is infeasible to collect the data at a central location. In these cases, the motivation for parallelism comes not just from the need for computing resources but also from the infeasibility or undesirability of alternate (centralized) approaches.

## **1.2 Scope of Parallel Computing**

Parallel computing has made a tremendous impact on a variety of areas ranging from computational simulations for scientific and engineering applications to commercial applications in data mining and transaction processing. The cost benefits of parallelism coupled with the performance requirements of applications present compelling arguments in favor of parallel computing. We present a small sample of the diverse applications of parallel computing.

### **1.2.1 Applications in Engineering and Design**

Parallel computing has traditionally been employed with great success in the design of airfoils (optimizing lift, drag, stability), internal combustion engines (optimizing charge distribution, burn), high-speed circuits (layouts for delays and capacitive and inductive effects), and structures (optimizing structural integrity, design parameters, cost, etc.), among others. More recently, design of microelectromechanical and nanoelectromechanical systems (MEMS and NEMS) has attracted significant attention. While most applications in engineering and design pose problems of multiple spatial and temporal scales and coupled physical phenomena, in the case of MEMS/NEMS design these problems are particularly acute. Here, we often deal with a mix of quantum phenomena, molecular dynamics, and stochastic and continuum models with physical processes such as conduction, convection, radiation, and structural mechanics, all in a single system. This presents formidable challenges for geometric modeling, mathematical modeling, and algorithm development, all in the context of parallel computers.

Other applications in engineering and design focus on optimization of a variety of processes. Parallel computers have been used to solve a variety of discrete and continuous optimization problems. Algorithms such as Simplex, Interior Point Method for linear optimization and Branch-and-bound, and Genetic programming for discrete optimization have been efficiently parallelized and are frequently used.

### **1.2.2 Scientific Applications**

The past few years have seen a revolution in high performance scientific computing applications. The sequencing of the human genome by the International Human Genome Sequencing Consortium and Celera, Inc. has opened exciting new frontiers in bioinformatics. Functional and structural characterization of genes and proteins hold the promise of understanding and fundamentally influencing biological processes. Analyzing biological sequences with a view to developing new drugs and cures for diseases and medical conditions requires innovative algorithms as well as large-scale computational power. Indeed, some of the newest parallel computing technologies are targeted specifically towards applications in bioinformatics.

Advances in computational physics and chemistry have focused on understanding processes ranging in scale from quantum phenomena to macromolecular structures. These have resulted in design of new materials, understanding of chemical pathways, and more efficient processes. Applications in astrophysics have explored

the evolution of galaxies, thermonuclear processes, and the analysis of extremely large datasets from telescopes. Weather modeling, mineral prospecting, flood prediction, etc., rely heavily on parallel computers and have very significant impact on day-to-day life.

Bioinformatics and astrophysics also present some of the most challenging problems with respect to analyzing extremely large datasets. Protein and gene databases (such as PDB, SwissProt, and ENTREZ and NDB) along with Sky Survey datasets (such as the Sloan Digital Sky Surveys) represent some of the largest scientific datasets. Effectively analyzing these datasets requires tremendous computational power and holds the key to significant scientific discoveries.

### **1.2.3 Commercial Applications**

With the widespread use of the web and associated static and dynamic content, there is increasing emphasis on cost-effective servers capable of providing scalable performance. Parallel platforms ranging from multiprocessors to linux clusters are frequently used as web and database servers. For instance, on heavy volume days, large brokerage houses on Wall Street handle hundreds of thousands of simultaneous user sessions and millions of orders. Platforms such as IBMs SP supercomputers and Sun Ultra HPC servers power these business-critical sites. While not highly visible, some of the largest supercomputing networks are housed on Wall Street.

The availability of large-scale transaction data has also sparked considerable interest in data mining and analysis for optimizing business and marketing decisions. The sheer volume and geographically distributed nature of this data require the use of effective parallel algorithms for such problems as association rule mining, clustering, classification, and time-series analysis.

### **1.2.4 Applications in Computer Systems**

As computer systems become more pervasive and computation spreads over the network, parallel processing issues become engrained into a variety of applications. In computer security, intrusion detection is an outstanding challenge. In the case of network intrusion detection, data is collected at distributed sites and must be analyzed rapidly for signaling intrusion. The infeasibility of collecting this data at a central location for analysis requires effective parallel and distributed algorithms. In the area of cryptography, some of the most spectacular applications of Internet-based parallel computing have focused on factoring extremely large integers.

Embedded systems increasingly rely on distributed control algorithms for accomplishing a variety of tasks. A modern automobile consists of tens of processors communicating to perform complex tasks for optimizing handling and performance. In such systems, traditional parallel and distributed algorithms for leader selection, maximal independent set, etc., are frequently used.

While parallel computing has traditionally confined itself to platforms with well behaved compute and network elements in which faults and errors do not play a

significant role, there are valuable lessons that extend to computations on ad-hoc, mobile, or faulty environments.

## Chapter 2. Parallel Programming Platforms

The traditional logical view of a sequential computer consists of a memory connected to a processor via a datapath. All three components ♦ processor, memory, and datapath ♦ present bottlenecks to the overall processing rate of a computer system. A number of architectural innovations over the years have addressed these bottlenecks. One of the most important innovations is multiplicity ♦ in processing units, datapaths, and memory units. This multiplicity is either entirely hidden from the programmer, as in the case of implicit parallelism, or exposed to the programmer in different forms. In this chapter, we present an overview of important architectural concepts as they relate to parallel processing. The objective is to provide sufficient detail for programmers to be able to write efficient code on a variety of platforms. We develop cost models and abstractions for quantifying the performance of various parallel algorithms, and identify bottlenecks resulting from various programming constructs.

We start our discussion of parallel platforms with an overview of serial and implicitly parallel architectures. This is necessitated by the fact that it is often possible to re-engineer codes to achieve significant speedups (2 x to 5 x unoptimized speed) using simple program transformations. Parallelizing sub-optimal serial codes often has undesirable effects of unreliable speedups and misleading runtimes. For this reason, we advocate optimizing serial performance of codes before attempting parallelization. As we shall demonstrate through this chapter, the tasks of serial and parallel optimization often have very similar characteristics. After discussing serial and implicitly parallel architectures, we devote the rest of this chapter to organization of parallel platforms, underlying cost models for algorithms, and platform abstractions for portable algorithm design. Readers wishing to delve directly into parallel architectures may choose to skip Sections [2.1](#) and [2.2](#).

## **2.1 Implicit Parallelism: Trends in Microprocessor Architectures\***

While microprocessor technology has delivered significant improvements in clock speeds over the past decade, it has also exposed a variety of other performance bottlenecks. To alleviate these bottlenecks, microprocessor designers have explored alternate routes to cost-effective performance gains. In this section, we will outline some of these trends with a view to understanding their limitations and how they impact algorithm and code development. The objective here is not to provide a comprehensive description of processor architectures. There are several excellent texts referenced in the bibliography that address this topic.

Clock speeds of microprocessors have posted impressive gains - two to three orders of magnitude over the past 20 years. However, these increments in clock speed are severely diluted by the limitations of memory technology. At the same time, higher levels of device integration have also resulted in a very large transistor count, raising the obvious issue of how best to utilize them. Consequently, techniques that enable execution of multiple instructions in a single clock cycle have become popular. Indeed, this trend is evident in the current generation of microprocessors such as the Itanium, Sparc Ultra, MIPS, and Power4. In this section, we briefly explore mechanisms used by various processors for supporting multiple instruction execution.

### **2.1.1 Pipelining and Superscalar Execution**

Processors have long relied on pipelines for improving execution rates. By overlapping various stages in instruction execution (fetch, schedule, decode, operand fetch, execute, store, among others), pipelining enables faster execution. The assembly-line analogy works well for understanding pipelines. If the assembly of a car, taking 100 time units, can be broken into 10 pipelined stages of 10 units each, a single assembly line can produce a car every 10 time units! This represents a 10-fold speedup over producing cars entirely serially, one after the other. It is also evident from this example that to increase the speed of a single pipeline, one would break down the tasks into smaller and smaller units, thus lengthening the pipeline and increasing overlap in execution. In the context of processors, this enables faster clock rates since the tasks are now smaller. For example, the Pentium 4, which operates at 2.0 GHz, has a 20 stage pipeline. Note that the speed of a single pipeline is ultimately limited by the largest atomic task in the pipeline. Furthermore, in typical instruction traces, every fifth to sixth instruction is a branch instruction. Long instruction pipelines therefore need effective techniques for predicting branch destinations so that pipelines can be speculatively filled. The penalty of a misprediction increases as the pipelines become deeper since a larger number of instructions need to be flushed. These factors place limitations on the depth of a processor pipeline and the resulting performance gains.

An obvious way to improve instruction execution rate beyond this level is to use multiple pipelines. During each clock cycle, multiple instructions are piped into the processor in parallel. These instructions are executed on multiple functional units. We illustrate this process with the help of an example.



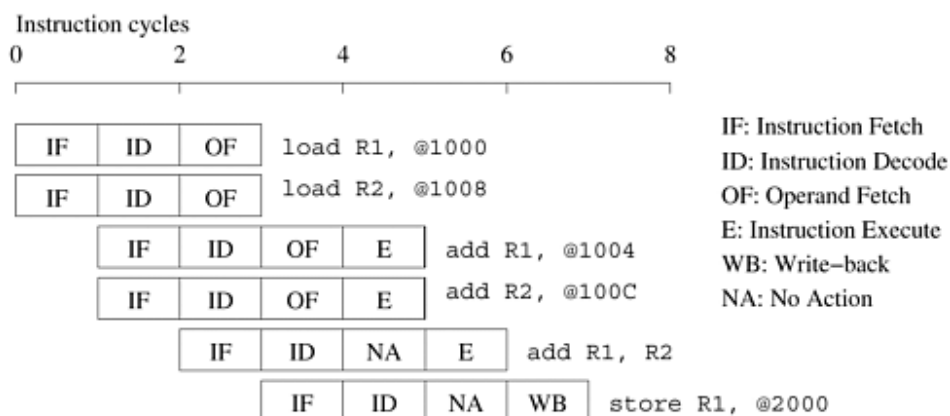
## Example 2.1 Superscalar execution

Consider a processor with two pipelines and the ability to simultaneously issue two instructions. These processors are sometimes also referred to as super-pipelined processors. The ability of a processor to issue multiple instructions in the same cycle is referred to as superscalar execution. Since the architecture illustrated in [Figure 2.1](#) allows two issues per clock cycle, it is also referred to as two-way superscalar or dual issue execution.

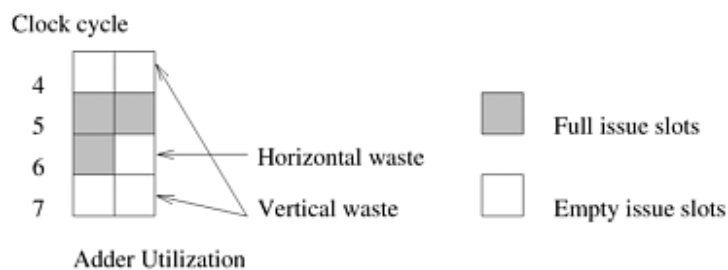
**Figure 2.1. Example of a two-way superscalar execution of instructions.**

|                    |                    |                    |
|--------------------|--------------------|--------------------|
| 1. load R1, @1000  | 1. load R1, @1000  | 1. load R1, @1000  |
| 2. load R2, @1008  | 2. add R1, @1004   | 2. add R1, @1004   |
| 3. add R1, @1004   | 3. add R1, @1008   | 3. load R2, @1008  |
| 4. add R2, @100C   | 4. add R1, @100C   | 4. add R2, @100C   |
| 5. add R1, R2      | 5. store R1, @2000 | 5. add R1, R2      |
| 6. store R1, @2000 |                    | 6. store R1, @2000 |
| (i)                | (ii)               | (iii)              |

(a) Three different code fragments for adding a list of four numbers.



(b) Execution schedule for code fragment (i) above.



(c) Hardware utilization trace for schedule in (b).

Consider the execution of the first code fragment in [Figure 2.1](#) for adding four numbers. The first and second instructions are independent and therefore can be issued concurrently. This is illustrated in the simultaneous issue of the instructions `load R1, @1000` and `load R2, @1008` at  $t = 0$ . The instructions are fetched, decoded, and the operands are fetched. The next two instructions, `add R1, @1004` and `add R2, @100C` are also mutually independent, although they must be executed after the first two instructions. Consequently, they can be issued concurrently at  $t = 1$  since the processors are pipelined. These instructions terminate at  $t = 5$ . The next two instructions, `add R1, R2` and `store R1, @2000` cannot be executed concurrently since the result of the former (contents of register `R1`) is used by the latter. Therefore, only the `add` instruction is issued at  $t = 2$  and the `store` instruction at  $t = 3$ . Note that the instruction `add R1, R2` can be executed only after the previous two instructions have been executed. The instruction schedule is illustrated in [Figure 2.1\(b\)](#). The schedule assumes that each memory access takes a single cycle. In reality, this may not be the case. The implications of this assumption are discussed in [Section 2.2](#) on memory system performance. ■

In principle, superscalar execution seems natural, even simple. However, a number of issues need to be resolved. First, as illustrated in [Example 2.1](#), instructions in a program may be related to each other. The results of an instruction may be required for subsequent instructions. This is referred to as **true data dependency**. For instance, consider the second code fragment in [Figure 2.1](#) for adding four numbers. There is a true data dependency between `load R1, @1000` and `add R1, @1004`, and similarly between subsequent instructions. Dependencies of this type must be resolved before simultaneous issue of instructions. This has two implications. First, since the resolution is done at runtime, it must be supported in hardware. The complexity of this hardware can be high. Second, the amount of instruction level parallelism in a program is often limited and is a function of coding technique. In the second code fragment, there can be no simultaneous issue, leading to poor resource utilization. The three code fragments in [Figure 2.1\(a\)](#) also illustrate that in many cases it is possible to extract more parallelism by reordering the instructions and by altering the code. Notice that in this example the code reorganization corresponds to exposing parallelism in a form that can be used by the instruction issue mechanism.

Another source of dependency between instructions results from the finite resources shared by various pipelines. As an example, consider the co-scheduling of two floating point operations on a dual issue machine with a single floating point unit. Although there might be no data dependencies between the instructions, they cannot be scheduled together since both need the floating point unit. This form of dependency in which two instructions compete for a single processor resource is referred to as **resource dependency**.

The flow of control through a program enforces a third form of dependency between instructions. Consider the execution of a conditional branch instruction. Since the branch destination is known only at the point of execution, scheduling instructions *a priori* across branches may lead to errors. These dependencies are referred to as **branch dependencies** or **procedural dependencies** and are typically handled by speculatively scheduling across branches and rolling back in case of errors. Studies of typical traces have shown that on average, a branch instruction is encountered

between every five to six instructions. Therefore, just as in populating instruction pipelines, accurate branch prediction is critical for efficient superscalar execution.

The ability of a processor to detect and schedule concurrent instructions is critical to superscalar performance. For instance, consider the third code fragment in [Figure 2.1](#) which also computes the sum of four numbers. The reader will note that this is merely a semantically equivalent reordering of the first code fragment. However, in this case, there is a data dependency between the first two instructions `load R1, @1000` and `add R1, @1004`. Therefore, these instructions cannot be issued simultaneously. However, if the processor had the ability to look ahead, it would realize that it is possible to schedule the third instruction `load R2, @1008` with the first instruction. In the next issue cycle, instructions two and four can be scheduled, and so on. In this way, the same execution schedule can be derived for the first and third code fragments. However, the processor needs the ability to issue instructions **out-of-order** to accomplish desired reordering. The parallelism available in **in-order** issue of instructions can be highly limited as illustrated by this example. Most current microprocessors are capable of out-of-order issue and completion. This model, also referred to as **dynamic instruction issue**, exploits maximum instruction level parallelism. The processor uses a window of instructions from which it selects instructions for simultaneous issue. This window corresponds to the look-ahead of the scheduler.

The performance of superscalar architectures is limited by the available instruction level parallelism. Consider the example in [Figure 2.1](#). For simplicity of discussion, let us ignore the pipelining aspects of the example and focus on the execution aspects of the program. Assuming two execution units (multiply-add units), the figure illustrates that there are several zero-issue cycles (cycles in which the floating point unit is idle). These are essentially wasted cycles from the point of view of the execution unit. If, during a particular cycle, no instructions are issued on the execution units, it is referred to as **vertical waste**; if only part of the execution units are used during a cycle, it is termed **horizontal waste**. In the example, we have two cycles of vertical waste and one cycle with horizontal waste. In all, only three of the eight available cycles are used for computation. This implies that the code fragment will yield no more than three-eighths of the peak rated FLOP count of the processor. Often, due to limited parallelism, resource dependencies, or the inability of a processor to extract parallelism, the resources of superscalar processors are heavily under-utilized. Current microprocessors typically support up to four-issue superscalar execution.

## 2.1.2 Very Long Instruction Word Processors

The parallelism extracted by superscalar processors is often limited by the instruction look-ahead. The hardware logic for dynamic dependency analysis is typically in the range of 5-10% of the total logic on conventional microprocessors (about 5% on the four-way superscalar Sun UltraSPARC). This complexity grows roughly quadratically with the number of issues and can become a bottleneck. An alternate concept for exploiting instruction-level parallelism used in very long instruction word (VLIW) processors relies on the compiler to resolve dependencies and resource availability at compile time. Instructions that can be executed concurrently are packed into groups and parceled off to the processor as a single long instruction word (thus the name) to be executed on multiple functional units at the same time.

The VLIW concept, first used in Multiflow Trace (circa 1984) and subsequently as a variant in the Intel IA64 architecture, has both advantages and disadvantages compared to superscalar processors. Since scheduling is done in software, the decoding and instruction issue mechanisms are simpler in VLIW processors. The compiler has a larger context from which to select instructions and can use a variety of transformations to optimize parallelism when compared to a hardware issue unit. Additional parallel instructions are typically made available to the compiler to control parallel execution. However, compilers do not have the dynamic program state (e.g., the branch history buffer) available to make scheduling decisions. This reduces the accuracy of branch and memory prediction, but allows the use of more sophisticated static prediction schemes. Other runtime situations such as stalls on data fetch because of cache misses are extremely difficult to predict accurately. This limits the scope and performance of static compiler-based scheduling.

Finally, the performance of VLIW processors is very sensitive to the compilers' ability to detect data and resource dependencies and read and write hazards, and to schedule instructions for maximum parallelism. Loop unrolling, branch prediction and speculative execution all play important roles in the performance of VLIW processors. While superscalar and VLIW processors have been successful in exploiting implicit parallelism, they are generally limited to smaller scales of concurrency in the range of four- to eight-way parallelism.

## 2.2 Limitations of Memory System Performance\*

The effective performance of a program on a computer relies not just on the speed of the processor but also on the ability of the memory system to feed data to the processor. At the logical level, a memory system, possibly consisting of multiple levels of caches, takes in a request for a memory word and returns a block of data of size  $b$  containing the requested word after  $l$  nanoseconds. Here,  $l$  is referred to as the **latency** of the memory. The rate at which data can be pumped from the memory to the processor determines the **bandwidth** of the memory system.

It is very important to understand the difference between latency and bandwidth since different, often competing, techniques are required for addressing these. As an analogy, if water comes out of the end of a fire hose 2 seconds after a hydrant is turned on, then the latency of the system is 2 seconds. Once the flow starts, if the hose pumps water at 1 gallon/second then the 'bandwidth' of the hose is 1 gallon/second. If we need to put out a fire immediately, we might desire a lower latency. This would typically require higher water pressure from the hydrant. On the other hand, if we wish to fight bigger fires, we might desire a higher flow rate, necessitating a wider hose and hydrant. As we shall see here, this analogy works well for memory systems as well. Latency and bandwidth both play critical roles in determining memory system performance. We examine these separately in greater detail using a few examples.

To study the effect of memory system latency, we assume in the following examples that a memory block consists of one word. We later relax this assumption while examining the role of memory bandwidth. Since we are primarily interested in maximum achievable performance, we also assume the best case cache-replacement policy. We refer the reader to the bibliography for a detailed discussion of memory system design.

### Example 2.2 Effect of memory latency on performance

Consider a processor operating at 1 GHz (1 ns clock) connected to a DRAM with a latency of 100 ns (no caches). Assume that the processor has two multiply-add units and is capable of executing four instructions in each cycle of 1 ns. The peak processor rating is therefore 4 GFLOPS. Since the memory latency is equal to 100 cycles and block size is one word, every time a memory request is made, the processor must wait 100 cycles before it can process the data. Consider the problem of computing the dot-product of two vectors on such a platform. A dot-product computation performs one multiply-add on a single pair of vector elements, i.e., each floating point operation requires one data fetch. It is easy to see that the peak speed of this computation is limited to one floating point operation every 100 ns, or a speed of 10 MFLOPS, a very small fraction of the peak processor rating. This example highlights the need for effective memory system performance in achieving high computation rates. ■

## 2.2.1 Improving Effective Memory Latency Using Caches

Handling the mismatch in processor and DRAM speeds has motivated a number of architectural innovations in memory system design. One such innovation addresses the speed mismatch by placing a smaller and faster memory between the processor and the DRAM. This memory, referred to as the cache, acts as a low-latency high-bandwidth storage. The data needed by the processor is first fetched into the cache. All subsequent accesses to data items residing in the cache are serviced by the cache. Thus, in principle, if a piece of data is repeatedly used, the effective latency of this memory system can be reduced by the cache. The fraction of data references satisfied by the cache is called the cache **hit ratio** of the computation on the system. The effective computation rate of many applications is bounded not by the processing rate of the CPU, but by the rate at which data can be pumped into the CPU. Such computations are referred to as being **memory bound**. The performance of memory bound programs is critically impacted by the cache hit ratio.

### Example 2.3 Impact of caches on memory system performance

As in the previous example, consider a 1 GHz processor with a 100 ns latency DRAM. In this case, we introduce a cache of size 32 KB with a latency of 1 ns or one cycle (typically on the processor itself). We use this setup to multiply two matrices  $A$  and  $B$  of dimensions  $32 \times 32$ . We have carefully chosen these numbers so that the cache is large enough to store matrices  $A$  and  $B$ , as well as the result matrix  $C$ . Once again, we assume an ideal cache placement strategy in which none of the data items are overwritten by others. Fetching the two matrices into the cache corresponds to fetching 2K words, which takes approximately 200  $\mu\text{s}$ . We know from elementary algorithmics that multiplying two  $n \times n$  matrices takes  $2n^3$  operations. For our problem, this corresponds to 64K operations, which can be performed in 16K cycles (or 16  $\mu\text{s}$ ) at four instructions per cycle. The total time for the computation is therefore approximately the sum of time for load/store operations and the time for the computation itself, i.e., 200+16  $\mu\text{s}$ . This corresponds to a peak computation rate of 64K/216 or 303 MFLOPS. Note that this is a thirty-fold improvement over the previous example, although it is still less than 10% of the peak processor performance. We see in this example that by placing a small cache memory, we are able to improve processor utilization considerably. ■

The improvement in performance resulting from the presence of the cache is based on the assumption that there is repeated reference to the same data item. This notion of repeated reference to a data item in a small time window is called **temporal locality** of reference. In our example, we had  $O(n^2)$  data accesses and  $O(n^3)$  computation. (See the Appendix for an explanation of the  $O$  notation.) Data reuse is critical for cache performance because if each data item is used only once, it would still have to be fetched once per use from the DRAM, and therefore the DRAM latency would be paid for each operation.

## 2.2.2 Impact of Memory Bandwidth

Memory bandwidth refers to the rate at which data can be moved between the processor and memory. It is determined by the bandwidth of the memory bus as well as the memory units. One commonly used technique to improve memory bandwidth is to increase the size of the memory blocks. For an illustration, let us relax our simplifying restriction on the size of the memory block and assume that a single memory request returns a contiguous block of four words. The single unit of four words in this case is also referred to as a **cache line**. Conventional computers typically fetch two to eight words together into the cache. We will see how this helps the performance of applications for which data reuse is limited.

### Example 2.4 Effect of block size: dot-product of two vectors

Consider again a memory system with a single cycle cache and 100 cycle latency DRAM with the processor operating at 1 GHz. If the block size is one word, the processor takes 100 cycles to fetch each word. For each pair of words, the dot-product performs one multiply-add, i.e., two FLOPs. Therefore, the algorithm performs one FLOP every 100 cycles for a peak speed of 10 MFLOPS as illustrated in [Example 2.2](#).

Now let us consider what happens if the block size is increased to four words, i.e., the processor can fetch a four-word cache line every 100 cycles. Assuming that the vectors are laid out linearly in memory, eight FLOPs (four multiply-adds) can be performed in 200 cycles. This is because a single memory access fetches four consecutive words in the vector. Therefore, two accesses can fetch four elements of each of the vectors. This corresponds to a FLOP every 25 ns, for a peak speed of 40 MFLOPS. Note that increasing the block size from one to four words did not change the latency of the memory system. However, it increased the bandwidth four-fold. In this case, the increased bandwidth of the memory system enabled us to accelerate the dot-product algorithm which has no data reuse at all.

Another way of quickly estimating performance bounds is to estimate the cache hit ratio, using it to compute mean access time per word, and relating this to the FLOP rate via the underlying algorithm. For example, in this example, there are two DRAM accesses (cache misses) for every eight data accesses required by the algorithm. This corresponds to a cache hit ratio of 75%. Assuming that the dominant overhead is posed by the cache misses, the average memory access time contributed by the misses is 25% at 100 ns (or 25 ns/word). Since the dot-product has one operation/word, this corresponds to a computation rate of 40 MFLOPS as before. A more accurate estimate of this rate would compute the average memory access time as  $0.75 \times 1 + 0.25 \times 100$  or 25.75 ns/word. The corresponding computation rate is 38.8 MFLOPS. ■

Physically, the scenario illustrated in [Example 2.4](#) corresponds to a wide data bus (4 words or 128 bits) connected to multiple memory banks. In practice, such wide buses are expensive to construct. In a more practical system, consecutive words are sent on the memory bus on subsequent bus cycles after the first word is retrieved.



For example, with a 32 bit data bus, the first word is put on the bus after 100 ns (the associated latency) and one word is put on each subsequent bus cycle. This changes our calculations above slightly since the entire cache line becomes available only after  $100 + 3 \times (\text{memory bus cycle})$  ns. Assuming a data bus operating at 200 MHz, this adds 15 ns to the cache line access time. This does not change our bound on the execution rate significantly.

The above examples clearly illustrate how increased bandwidth results in higher peak computation rates. They also make certain assumptions that have significance for the programmer. The data layouts were assumed to be such that consecutive data words in memory were used by successive instructions. In other words, if we take a computation-centric view, there is a ***spatial locality*** of memory access. If we take a data-layout centric point of view, the computation is ordered so that successive computations require contiguous data. If the computation (or access pattern) does not have spatial locality, then effective bandwidth can be much smaller than the peak bandwidth.

An example of such an access pattern is in reading a dense matrix column-wise when the matrix has been stored in a row-major fashion in memory. Compilers can often be relied on to do a good job of restructuring computation to take advantage of spatial locality.

### Example 2.5 Impact of strided access

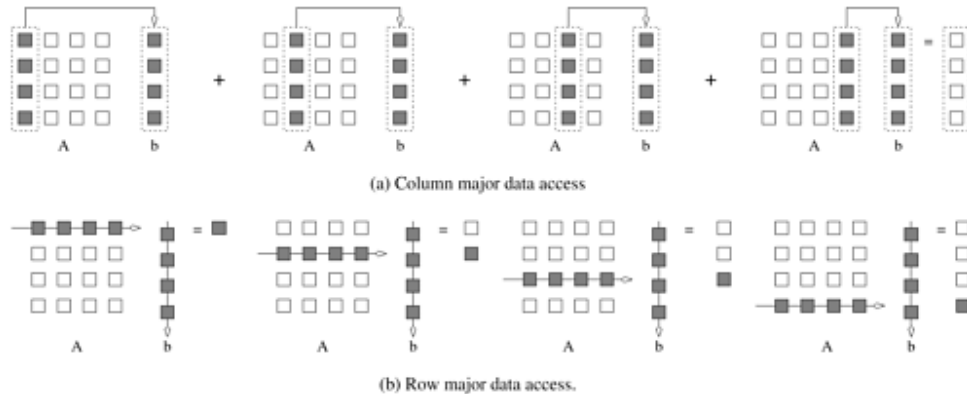
Consider the following code fragment:

```
1  for (i = 0; i < 1000; i++)
2      column_sum[i] = 0.0;
3      for (j = 0; j < 1000; j++)
4          column_sum[i] += b[j][i];
```

The code fragment sums columns of the matrix **b** into a vector **column\_sum**. There are two observations that can be made: (i) the vector **column\_sum** is small and easily fits into the cache; and (ii) the matrix **b** is accessed in a column order as illustrated in [Figure 2.2\(a\)](#). For a matrix of size 1000 x 1000, stored in a row-major order, this corresponds to accessing every 1000<sup>th</sup> entry. Therefore, it is likely that only one word in each cache line fetched from memory will be used. Consequently, the code fragment as written above is likely to yield poor performance. ■

**Figure 2.2. Multiplying a matrix with a vector: (a) multiplying column-by-column, keeping a running sum; (b) computing each element of the result as a dot product of a row of the matrix with the vector.**





The above example illustrates problems with strided access (with strides greater than one). The lack of spatial locality in computation causes poor memory system performance. Often it is possible to restructure the computation to remove strided access. In the case of our example, a simple rewrite of the loops is possible as follows:

### Example 2.6 Eliminating strided access

Consider the following restructuring of the column-sum fragment:

```

1  for (i = 0; i < 1000; i++)
2      column_sum[i] = 0.0;
3  for (j = 0; j < 1000; j++)
4      for (i = 0; i < 1000; i++)
5          column_sum[i] += b[j][i];

```

In this case, the matrix is traversed in a row-order as illustrated in [Figure 2.2\(b\)](#). However, the reader will note that this code fragment relies on the fact that the vector `column_sum` can be retained in the cache through the loops. Indeed, for this particular example, our assumption is reasonable. If the vector is larger, we would have to break the iteration space into blocks and compute the product one block at a time. This concept is also called **tiling** an iteration space. The improved performance of this loop is left as an exercise for the reader. ■

So the next question is whether we have effectively solved the problems posed by memory latency and bandwidth. While peak processor rates have grown significantly over the past decades, memory latency and bandwidth have not kept pace with this increase. Consequently, for typical computers, the ratio of peak FLOPS rate to peak memory bandwidth is anywhere between 1 MFLOPS/MBs (the ratio signifies FLOPS per megabyte/second of bandwidth) to 100 MFLOPS/MBs. The lower figure typically corresponds to large scale vector supercomputers and the higher figure to fast microprocessor based computers. This figure is very revealing in that it tells us that


on average, a word must be reused 100 times after being fetched into the full bandwidth storage (typically L1 cache) to be able to achieve full processor utilization. Here, we define full-bandwidth as the rate of data transfer required by a computation to make it processor bound.

The series of examples presented in this section illustrate the following concepts:

- Exploiting spatial and temporal locality in applications is critical for amortizing memory latency and increasing effective memory bandwidth.
- Certain applications have inherently greater temporal locality than others, and thus have greater tolerance to low memory bandwidth. The ratio of the number of operations to number of memory accesses is a good indicator of anticipated tolerance to memory bandwidth.
- Memory layouts and organizing computation appropriately can make a significant impact on the spatial and temporal locality.

## 2.2.3 Alternate Approaches for Hiding Memory Latency

Imagine sitting at your computer browsing the web during peak network traffic hours. The lack of response from your browser can be alleviated using one of three simple approaches:

(i) we anticipate which pages we are going to browse ahead of time and issue requests for them in advance; (ii) we open multiple browsers and access different pages in each browser, thus while we are waiting for one page to load, we could be reading others; or (iii) we access a whole bunch of pages in one go  amortizing the latency across various accesses. The first approach is called ***prefetching***, the second ***multithreading***, and the third one corresponds to spatial locality in accessing memory words. Of these three approaches, spatial locality of memory accesses has been discussed before. We focus on prefetching and multithreading as techniques for latency hiding in this section.

### Multithreading for Latency Hiding

A thread is a single stream of control in the flow of a program. We illustrate threads with a simple example:

#### Example 2.7 Threaded execution of matrix multiplication

Consider the following code segment for multiplying an  $n \times n$  matrix **a** by a vector **b** to get vector **c**.

```
1  for(i=0;i<n;i++)
2      c[i] = dot_product(get_row(a, i), b);
```

This code computes each element of **c** as the dot product of the corresponding row of **a** with the vector **b**. Notice that each dot-product is independent of the other, and therefore represents a concurrent unit of execution. We can safely rewrite the above code segment as:

```
1 for(i=0;i<n;i++)
2   c[i] = create_thread(dot_product, get_row(a, i), b);
```

The only difference between the two code segments is that we have explicitly specified each instance of the dot-product computation as being a thread. (As we shall learn in [Chapter 7](#), there are a number of APIs for specifying threads. We have simply chosen an intuitive name for a function to create threads.) Now, consider the execution of each instance of the function **dot\_product**. The first instance of this function accesses a pair of vector elements and waits for them. In the meantime, the second instance of this function can access two other vector elements in the next cycle, and so on. After *l* units of time, where *l* is the latency of the memory system, the first function instance gets the requested data from memory and can perform the required computation. In the next cycle, the data items for the next function instance arrive, and so on. In this way, in every clock cycle, we can perform a computation. ■

The execution schedule in [Example 2.7](#) is predicated upon two assumptions: the memory system is capable of servicing multiple outstanding requests, and the processor is capable of switching threads at every cycle. In addition, it also requires the program to have an explicit specification of concurrency in the form of threads. Multithreaded processors are capable of maintaining the context of a number of threads of computation with outstanding requests (memory accesses, I/O, or communication requests) and execute them as the requests are satisfied. Machines such as the HEP and Tera rely on multithreaded processors that can switch the context of execution in every cycle. Consequently, they are able to hide latency effectively, provided there is enough concurrency (threads) to keep the processor from idling. The tradeoffs between concurrency and latency will be a recurring theme through many chapters of this text.

## Prefetching for Latency Hiding

In a typical program, a data item is loaded and used by a processor in a small time window. If the load results in a cache miss, then the use stalls. A simple solution to this problem is to advance the load operation so that even if there is a cache miss, the data is likely to have arrived by the time it is used. However, if the data item has been overwritten between load and use, a fresh load is issued. Note that this is no worse than the situation in which the load had not been advanced. A careful examination of this technique reveals that prefetching works for much the same reason as multithreading. In advancing the loads, we are trying to identify independent threads of execution that have no resource dependency (i.e., use the same registers) with respect to other threads. Many compilers aggressively try to advance loads to mask memory system latency.

### Example 2.8 Hiding latency by prefetching

Consider the problem of adding two vectors **a** and **b** using a single **for** loop. In the first iteration of the loop, the processor requests **a[0]** and **b[0]**. Since these are not in the cache, the processor must pay the memory latency. While these requests are being serviced, the processor also requests **a[1]** and **b[1]**. Assuming that each request is generated in one cycle (1 ns) and memory requests are satisfied in 100 ns, after 100 such requests the first set of data items is returned by the memory system. Subsequently, one pair of vector components will be returned every cycle. In this way, in each subsequent cycle, one addition can be performed and processor cycles are not wasted. ■

## 2.2.4 Tradeoffs of Multithreading and Prefetching

While it might seem that multithreading and prefetching solve all the problems related to memory system performance, they are critically impacted by the memory bandwidth.

### Example 2.9 Impact of bandwidth on multithreaded programs

Consider a computation running on a machine with a 1 GHz clock, 4-word cache line, single cycle access to the cache, and 100 ns latency to DRAM. The computation has a cache hit ratio at 1 KB of 25% and at 32 KB of 90%. Consider two cases: first, a single threaded execution in which the entire cache is available to the serial context, and second, a multithreaded execution with 32 threads where each thread has a cache residency of 1 KB. If the computation makes one data request in every cycle of 1 ns, in the first case the bandwidth requirement to DRAM is one word every 10 ns since the other words come from the cache (90% cache hit ratio). This corresponds to a bandwidth of 400 MB/s. In the second case, the bandwidth requirement to DRAM increases to three words every four cycles of each thread (25% cache hit ratio). Assuming that all threads exhibit similar cache behavior, this corresponds to 0.75 words/ns, or 3 GB/s. ■

[Example 2.9](#) illustrates a very important issue, namely that the bandwidth requirements of a multithreaded system may increase very significantly because of the smaller cache residency of each thread. In the example, while a sustained DRAM bandwidth of 400 MB/s is reasonable, 3.0 GB/s is more than most systems currently offer. At this point, multithreaded systems become bandwidth bound instead of latency bound. It is important to realize that multithreading and prefetching only address the latency problem and may often exacerbate the bandwidth problem.

Another issue relates to the additional hardware resources required to effectively use prefetching and multithreading. Consider a situation in which we have advanced

10 loads into registers. These loads require 10 registers to be free for the duration. If an intervening instruction overwrites the registers, we would have to load the data again. This would not increase the latency of the fetch any more than the case in which there was no prefetching. However, now we are fetching the same data item twice, resulting in doubling of the bandwidth requirement from the memory system. This situation is similar to the one due to cache constraints as illustrated in [Example 2.9](#). It can be alleviated by supporting prefetching and multithreading with larger register files and caches.

## 2.3 Dichotomy of Parallel Computing Platforms

In the preceding sections, we pointed out various factors that impact the performance of a serial or implicitly parallel program. The increasing gap in peak and sustainable performance of current microprocessors, the impact of memory system performance, and the distributed nature of many problems present overarching motivations for parallelism. We now introduce, at a high level, the elements of parallel computing platforms that are critical for performance oriented and portable parallel programming. To facilitate our discussion of parallel platforms, we first explore a dichotomy based on the logical and physical organization of parallel platforms. The logical organization refers to a programmer's view of the platform while the physical organization refers to the actual hardware organization of the platform. The two critical components of parallel computing from a programmer's perspective are ways of expressing parallel tasks and mechanisms for specifying interaction between these tasks. The former is sometimes also referred to as the control structure and the latter as the communication model.

### 2.3.1 Control Structure of Parallel Platforms

Parallel tasks can be specified at various levels of granularity. At one extreme, each program in a set of programs can be viewed as one parallel task. At the other extreme, individual instructions within a program can be viewed as parallel tasks. Between these extremes lie a range of models for specifying the control structure of programs and the corresponding architectural support for them.

#### Example 2.10 Parallelism from single instruction on multiple processors

Consider the following code segment that adds two vectors:

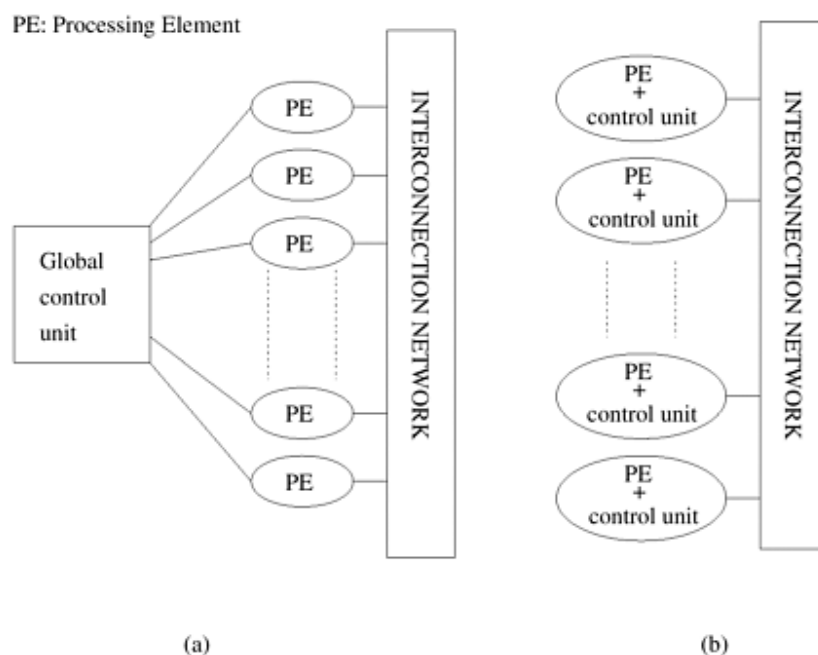
```
1  for (i = 0; i < 1000; i++)  
2      c[i] = a[i] + b[i];
```

In this example, various iterations of the loop are independent of each other; i.e., `c[0] = a[0] + b[0]; c[1] = a[1] + b[1];`, etc., can all be executed independently of each other. Consequently, if there is a mechanism for executing the same instruction, in this case `add` on all the processors with appropriate data, we could execute this loop much faster. ■

Processing units in parallel computers either operate under the centralized control of a single control unit or work independently. In architectures referred to as **single instruction stream, multiple data stream** (SIMD), a single control unit dispatches instructions to each processing unit. [Figure 2.3\(a\)](#) illustrates a typical SIMD architecture. In an SIMD parallel computer, the same instruction is executed synchronously by all processing units. In [Example 2.10](#), the `add` instruction is

dispatched to all processors and executed concurrently by them. Some of the earliest parallel computers such as the Illiac IV, MPP, DAP, CM-2, and MasPar MP-1 belonged to this class of machines. More recently, variants of this concept have found use in co-processing units such as the MMX units in Intel processors and DSP chips such as the Sharc. The Intel Pentium processor with its SSE (Streaming SIMD Extensions) provides a number of instructions that execute the same instruction on multiple data items. These architectural enhancements rely on the highly structured (regular) nature of the underlying computations, for example in image processing and graphics, to deliver improved performance.

**Figure 2.3. A typical SIMD architecture (a) and a typical MIMD architecture (b).**



While the SIMD concept works well for structured computations on parallel data structures such as arrays, often it is necessary to selectively turn off operations on certain data items. For this reason, most SIMD programming paradigms allow for an "activity mask". This is a binary mask associated with each data item and operation that specifies whether it should participate in the operation or not. Primitives such as **where (condition) then <stmt> <elsewhere stmt>** are used to support selective execution. Conditional execution can be detrimental to the performance of SIMD processors and therefore must be used with care.

In contrast to SIMD architectures, computers in which each processing element is capable of executing a different program independent of the other processing elements are called **multiple instruction stream, multiple data stream** (MIMD) computers. [Figure 2.3\(b\)](#) depicts a typical MIMD computer. A simple variant of this model, called the **single program multiple data** (SPMD) model, relies on multiple instances of the same program executing on different data. It is easy to see that the SPMD model has the same expressiveness as the MIMD model since each of the multiple programs can be inserted into one large **if-else** block with conditions

specified by the task identifiers. The SPMD model is widely used by many parallel platforms and requires minimal architectural support. Examples of such platforms include the Sun Ultra Servers, multiprocessor PCs, workstation clusters, and the IBM SP.

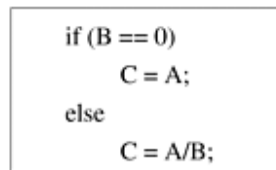
SIMD computers require less hardware than MIMD computers because they have only one global control unit. Furthermore, SIMD computers require less memory because only one copy of the program needs to be stored. In contrast, MIMD computers store the program and operating system at each processor. However, the relative unpopularity of SIMD processors as general purpose compute engines can be attributed to their specialized hardware architectures, economic factors, design constraints, product life-cycle, and application characteristics. In contrast, platforms supporting the SPMD paradigm can be built from inexpensive off-the-shelf components with relatively little effort in a short amount of time. SIMD computers require extensive design effort resulting in longer product development times. Since the underlying serial processors change so rapidly, SIMD computers suffer from fast obsolescence. The irregular nature of many applications also makes SIMD architectures less suitable. [Example 2.11](#) illustrates a case in which SIMD architectures yield poor resource utilization in the case of conditional execution.

### **Example 2.11 Execution of conditional statements on a SIMD architecture**

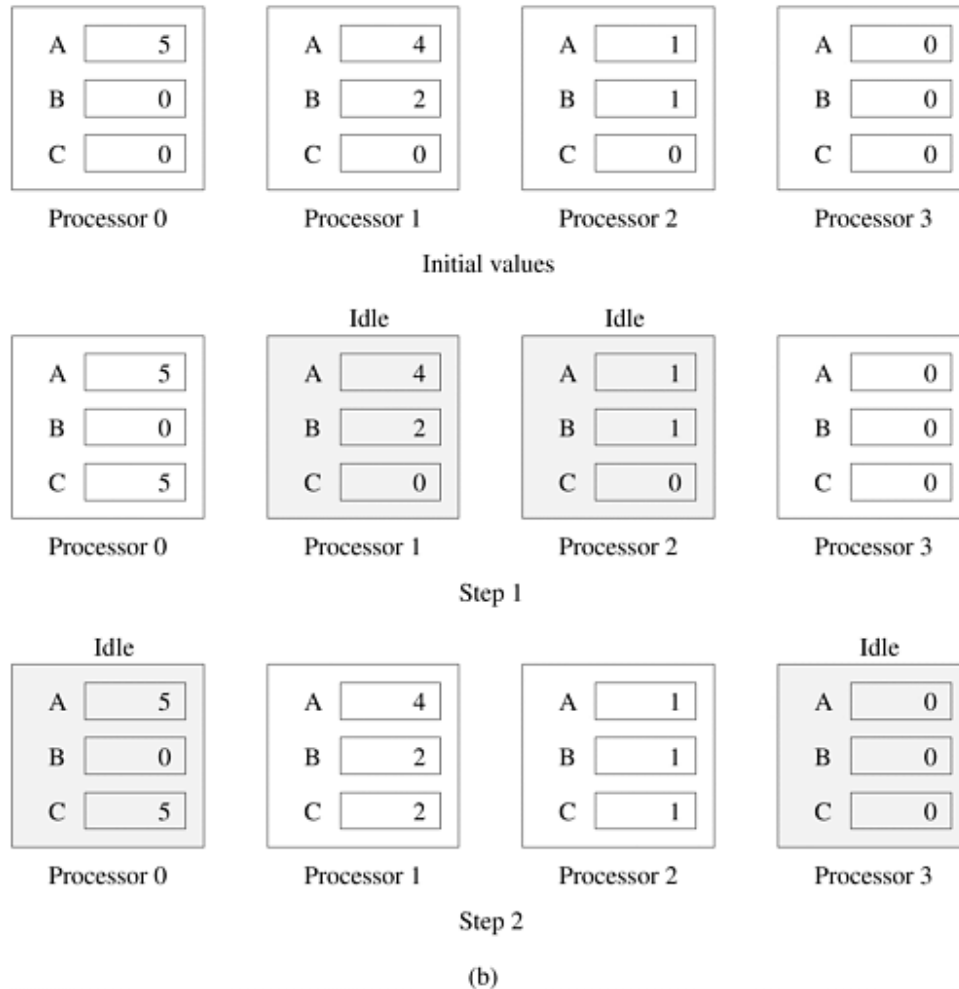
Consider the execution of a conditional statement illustrated in [Figure 2.4](#). The conditional statement in [Figure 2.4\(a\)](#) is executed in two steps. In the first step, all processors that have  $B$  equal to zero execute the instruction  $C = A$ . All other processors are idle. In the second step, the 'else' part of the instruction ( $C = A/B$ ) is executed. The processors that were active in the first step now become idle. This illustrates one of the drawbacks of SIMD architectures. ■

**Figure 2.4. Executing a conditional statement on an SIMD computer with four processors: (a) the conditional statement; (b) the execution of the statement in two steps.**





(a)



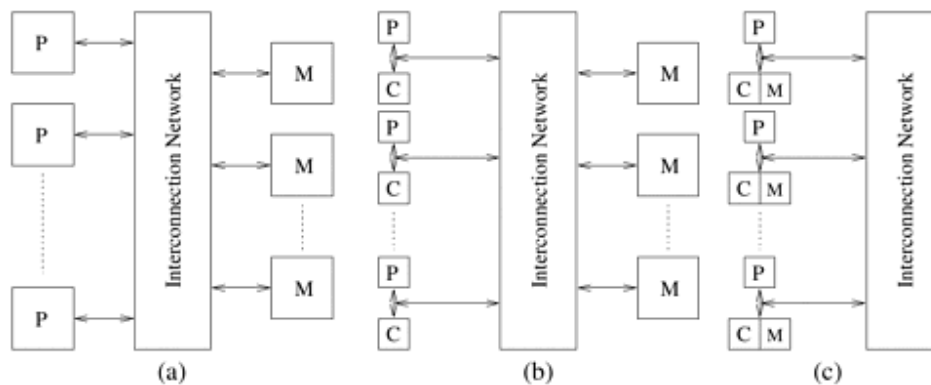
## 2.3.2 Communication Model of Parallel Platforms

There are two primary forms of data exchange between parallel tasks  $\blacklozenge$  accessing a shared data space and exchanging messages.

## Shared-Address-Space Platforms

The "shared-address-space" view of a parallel platform supports a common data space that is accessible to all processors. Processors interact by modifying data objects stored in this shared-address-space. Shared-address-space platforms supporting SPMD programming are also referred to as **multiprocessors**. Memory in shared-address-space platforms can be local (exclusive to a processor) or global (common to all processors). If the time taken by a processor to access any memory word in the system (global or local) is identical, the platform is classified as a **uniform memory access** (UMA) multicomputer. On the other hand, if the time taken to access certain memory words is longer than others, the platform is called a **non-uniform memory access** (NUMA) multicomputer. Figures 2.5(a) and (b) illustrate UMA platforms, whereas Figure 2.5(c) illustrates a NUMA platform. An interesting case is illustrated in Figure 2.5(b). Here, it is faster to access a memory word in cache than a location in memory. However, we still classify this as a UMA architecture. The reason for this is that all current microprocessors have cache hierarchies. Consequently, even a uniprocessor would not be termed UMA if cache access times are considered. For this reason, we define NUMA and UMA architectures only in terms of memory access times and not cache access times. Machines such as the SGI Origin 2000 and Sun Ultra HPC servers belong to the class of NUMA multiprocessors. The distinction between UMA and NUMA platforms is important. If accessing local memory is cheaper than accessing global memory, algorithms must build locality and structure data and computation accordingly.

**Figure 2.5. Typical shared-address-space architectures: (a) Uniform-memory-access shared-address-space computer; (b) Uniform-memory-access shared-address-space computer with caches and memories; (c) Non-uniform-memory-access shared-address-space computer with local memory only.**



The presence of a global memory space makes programming such platforms much easier. All read-only interactions are invisible to the programmer, as they are coded no differently than in a serial program. This greatly eases the burden of writing parallel programs. Read/write interactions are, however, harder to program than the read-only interactions, as these operations require mutual exclusion for concurrent accesses. Shared-address-space programming paradigms such as threads (POSIX,

NT) and directives (OpenMP) therefore support synchronization using **locks** and related mechanisms.

The presence of caches on processors also raises the issue of multiple copies of a single memory word being manipulated by two or more processors at the same time. Supporting a shared-address-space in this context involves two major tasks: providing an address translation mechanism that locates a memory word in the system, and ensuring that concurrent operations on multiple copies of the same memory word have well-defined semantics. The latter is also referred to as the **cache coherence** mechanism. This mechanism and its implementation are discussed in greater detail in [Section 2.4.6](#). Supporting cache coherence requires considerable hardware support. Consequently, some shared-address-space machines only support an address translation mechanism and leave the task of ensuring coherence to the programmer. The native programming model for such platforms consists of primitives such as **get** and **put**. These primitives allow a processor to get (and put) variables stored at a remote processor. However, if one of the copies of this variable is changed, the other copies are not automatically updated or invalidated.

It is important to note the difference between two commonly used and often misunderstood terms ♦ shared-address-space and shared-memory computers. The term shared-memory computer is historically used for architectures in which the memory is physically shared among various processors, i.e., each processor has equal access to any memory segment. This is identical to the UMA model we just discussed. This is in contrast to a distributed-memory computer, in which different segments of the memory are physically associated with different processing elements. The dichotomy of shared- versus distributed-memory computers pertains to the physical organization of the machine and is discussed in greater detail in [Section 2.4](#). Either of these physical models, shared or distributed memory, can present the logical view of a disjoint or shared-address-space platform. A distributed-memory shared-address-space computer is identical to a NUMA machine.

## Message-Passing Platforms

The logical machine view of a message-passing platform consists of  $p$  processing nodes, each with its own exclusive address space. Each of these processing nodes can either be single processors or a shared-address-space multiprocessor ♦ a trend that is fast gaining momentum in modern message-passing parallel computers. Instances of such a view come naturally from clustered workstations and non-shared-address-space multicomputers. On such platforms, interactions between processes running on different nodes must be accomplished using messages, hence the name **message passing**. This exchange of messages is used to transfer data, work, and to synchronize actions among the processes. In its most general form, message-passing paradigms support execution of a different program on each of the  $p$  nodes.

Since interactions are accomplished by sending and receiving messages, the basic operations in this programming paradigm are **send** and **receive** (the corresponding calls may differ across APIs but the semantics are largely identical). In addition, since the send and receive operations must specify target addresses, there must be a mechanism to assign a unique identification or ID to each of the multiple processes executing a parallel program. This ID is typically made available to the program

using a function such as `whoami`, which returns to a calling process its ID. There is one other function that is typically needed to complete the basic set of message-passing operations  $\blacklozenge$  `numprocs`, which specifies the number of processes participating in the ensemble. With these four basic operations, it is possible to write any message-passing program. Different message-passing APIs, such as the Message Passing Interface (MPI) and Parallel Virtual Machine (PVM), support these basic operations and a variety of higher level functionality under different function names. Examples of parallel platforms that support the message-passing paradigm include the IBM SP, SGI Origin 2000, and workstation clusters.

It is easy to emulate a message-passing architecture containing  $p$  nodes on a shared-address-space computer with an identical number of nodes. Assuming uniprocessor nodes, this can be done by partitioning the shared-address-space into  $p$  disjoint parts and assigning one such partition exclusively to each processor. A processor can then "send" or "receive" messages by writing to or reading from another processor's partition while using appropriate synchronization primitives to inform its communication partner when it has finished reading or writing the data. However, emulating a shared-address-space architecture on a message-passing computer is costly, since accessing another node's memory requires sending and receiving messages.

## 2.4 Physical Organization of Parallel Platforms

In this section, we discuss the physical architecture of parallel machines. We start with an ideal architecture, outline practical difficulties associated with realizing this model, and discuss some conventional architectures.

### 2.4.1 Architecture of an Ideal Parallel Computer

A natural extension of the serial model of computation (the Random Access Machine, or RAM) consists of  $p$  processors and a global memory of unbounded size that is uniformly accessible to all processors. All processors access the same address space. Processors share a common clock but may execute different instructions in each cycle. This ideal model is also referred to as a **parallel random access machine (PRAM)**. Since PRAMs allow concurrent access to various memory locations, depending on how simultaneous memory accesses are handled, PRAMs can be divided into four subclasses.

1. **Exclusive-read, exclusive-write (EREW) PRAM.** In this class, access to a memory location is exclusive. No concurrent read or write operations are allowed. This is the weakest PRAM model, affording minimum concurrency in memory access.
2. **Concurrent-read, exclusive-write (CREW) PRAM.** In this class, multiple read accesses to a memory location are allowed. However, multiple write accesses to a memory location are serialized.
3. **Exclusive-read, concurrent-write (ERCW) PRAM.** Multiple write accesses are allowed to a memory location, but multiple read accesses are serialized.
4. **Concurrent-read, concurrent-write (CRCW) PRAM.** This class allows multiple read and write accesses to a common memory location. This is the most powerful PRAM model.

Allowing concurrent read access does not create any semantic discrepancies in the program. However, concurrent write access to a memory location requires arbitration. Several protocols are used to resolve concurrent writes. The most frequently used protocols are as follows:

- **Common**, in which the concurrent write is allowed if all the values that the processors are attempting to write are identical.
- **Arbitrary**, in which an arbitrary processor is allowed to proceed with the write operation and the rest fail.
- **Priority**, in which all processors are organized into a predefined prioritized list, and the processor with the highest priority succeeds and the rest fail.

- **Sum**, in which the sum of all the quantities is written (the sum-based write conflict resolution model can be extended to any associative operator defined on the quantities being written).

## Architectural Complexity of the Ideal Model

Consider the implementation of an EREW PRAM as a shared-memory computer with  $p$  processors and a global memory of  $m$  words. The processors are connected to the memory through a set of switches. These switches determine the memory word being accessed by each processor. In an EREW PRAM, each of the  $p$  processors in the ensemble can access any of the memory words, provided that a word is not accessed by more than one processor simultaneously. To ensure such connectivity, the total number of switches must be  $Q(mp)$ . (See the Appendix for an explanation of the  $Q$  notation.) For a reasonable memory size, constructing a switching network of this complexity is very expensive. Thus, PRAM models of computation are impossible to realize in practice.

### 2.4.2 Interconnection Networks for Parallel Computers

Interconnection networks provide mechanisms for data transfer between processing nodes or between processors and memory modules. A blackbox view of an interconnection network consists of  $n$  inputs and  $m$  outputs. The outputs may or may not be distinct from the inputs. Typical interconnection networks are built using links and switches. A link corresponds to physical media such as a set of wires or fibers capable of carrying information. A variety of factors influence link characteristics. For links based on conducting media, the capacitive coupling between wires limits the speed of signal propagation. This capacitive coupling and attenuation of signal strength are functions of the length of the link.

Interconnection networks can be classified as **static** or **dynamic**. Static networks consist of point-to-point communication links among processing nodes and are also referred to as **direct** networks. Dynamic networks, on the other hand, are built using switches and communication links. Communication links are connected to one another dynamically by the switches to establish paths among processing nodes and memory banks. Dynamic networks are also referred to as **indirect** networks. [Figure 2.6\(a\)](#) illustrates a simple static network of four processing elements or nodes. Each processing node is connected via a network interface to two other nodes in a mesh configuration. [Figure 2.6\(b\)](#) illustrates a dynamic network of four nodes connected via a network of switches to other nodes.

**Figure 2.6. Classification of interconnection networks: (a) a static network; and (b) a dynamic network.**

not in syllabus X