Chapter 2. Parallel Programming Platforms

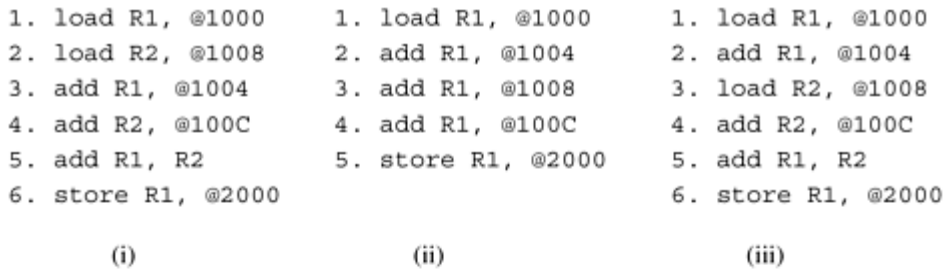
The traditional logical view of a sequential computer consists of a memory connected to a processor via a datapath. All three components – processor, memory, and datapath – present bottlenecks to the overall processing rate of a computer system. A number of architectural innovations over the years have 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.

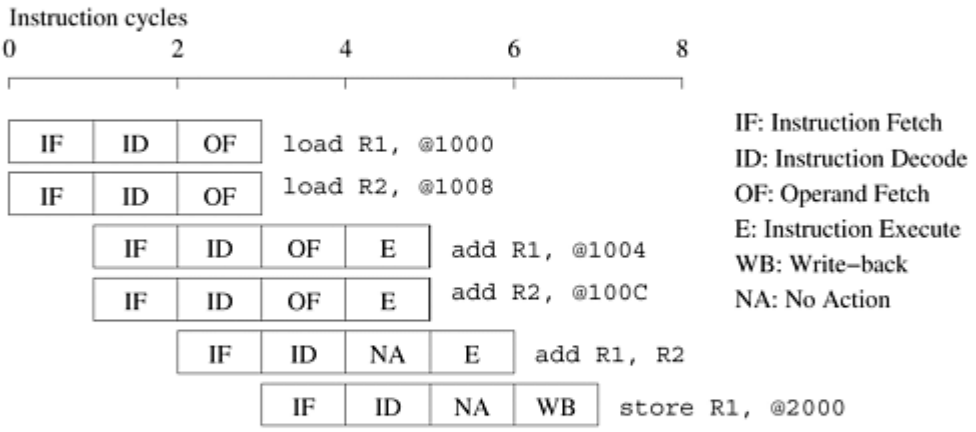
Example 2.1 Superscalar execution

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.

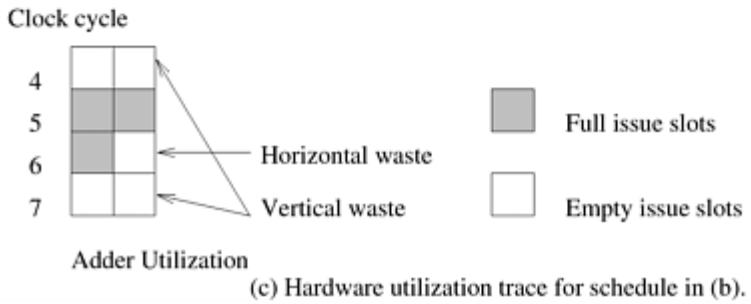
Figure 2.1. Example of a two-way superscalar execution of instructions.



(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 x 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 µs. We know from elementary algorithmics that multiplying two n x n matrices takes $2n^3$ operations. For our problem, this corresponds to 64K operations, which can be performed in 16K cycles (or 16 µs) 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 µs. 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 x 1 + 0.25 x 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 x (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.

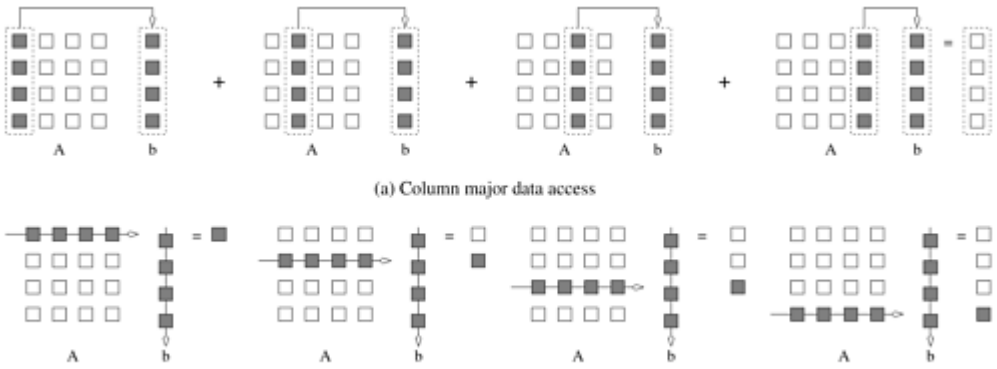
Example 2.5 Impact of strided access

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^{th} 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* x *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. ■

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

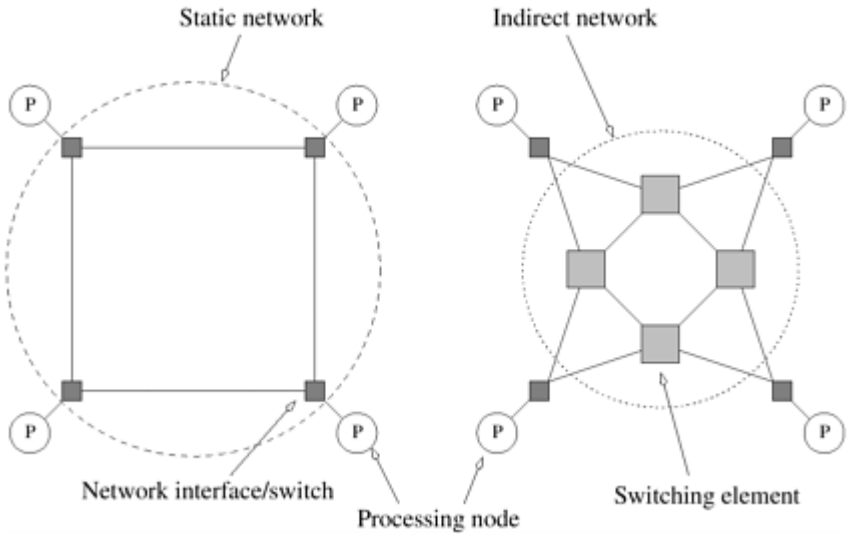
Consider the implementation of an EREW PRAM as a shared-memory computer with p processors and a global memory of m words. The processors are connected to the memory through a set of switches. These switches determine the memory word being accessed by each processor. In an EREW PRAM, each of the p processors 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 $\Theta(mp)$. (See the Appendix for an explanation of the Θ 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.



A single switch in an interconnection network consists of a set of input ports and a set of output ports. Switches provide a range of functionality. The minimal functionality provided by a switch is a mapping from the input to the output ports. The total number of ports on a switch is also called the *degree* of the switch. Switches may also provide support for internal buffering (when the requested output port is busy), routing (to alleviate congestion on the network), and multicast (same output on multiple ports). The mapping from input to output ports can be provided using a variety of mechanisms based on physical crossbars, multi-ported memories, multiplexor-demultiplexors, and multiplexed buses. The cost of a switch is influenced by the cost of the mapping hardware, the peripheral hardware and packaging costs. The mapping hardware typically grows as the square of the degree of the switch, the peripheral hardware linearly as the degree, and the packaging costs linearly as the number of pins.

The connectivity between the nodes and the network is provided by a network interface. The network interface has input and output ports that pipe data into and out of the network. It typically has the responsibility of packetizing data, computing routing information, buffering incoming and outgoing data for matching speeds of network and processing elements, and error checking. The position of the interface between the processing element and the network is also important. While conventional network interfaces hang off the I/O buses, interfaces in tightly coupled parallel machines hang off the memory bus. Since I/O buses are typically slower than memory buses, the latter can support higher bandwidth.

2.4.3 Network Topologies

A wide variety of network topologies have been used in interconnection networks. These topologies try to trade off cost and scalability with performance. While pure topologies have attractive mathematical properties, in practice interconnection networks tend to be combinations or modifications of the pure topologies discussed in this section.

Bus-Based Networks

A bus-based network is perhaps the simplest network consisting of a shared medium that is common to all the nodes. A bus has the desirable property that the cost of the network scales linearly as the number of nodes, p. This cost is typically associated with bus interfaces. Furthermore, the distance between any two nodes in the network is constant ($O(1)$). Buses are also ideal for broadcasting information among nodes. Since the transmission medium is shared, there is little overhead associated with broadcast compared to point-to-point message transfer. However, the bounded bandwidth of a bus places limitations on the overall performance of the network as the number of nodes increases. Typical bus based machines are limited to dozens of nodes. Sun Enterprise servers and Intel Pentium based shared-bus multiprocessors are examples of such architectures.

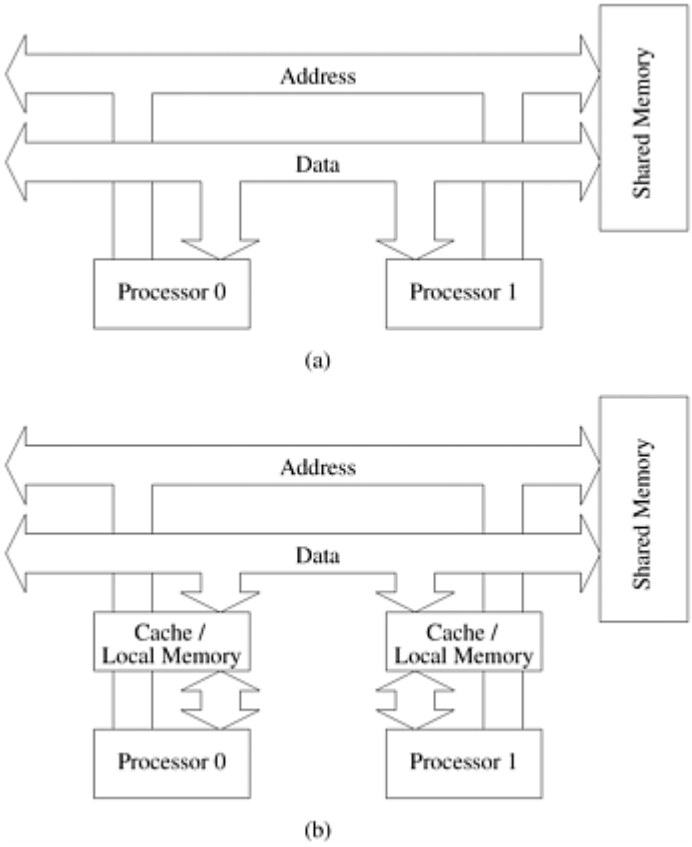
The demands on bus bandwidth can be reduced by making use of the property that in typical programs, a majority of the data accessed is local to the node. For such programs, it is possible to provide a cache for each node. Private data is cached at the node and only remote data is accessed through the bus.

Example 2.12 Reducing shared-bus bandwidth using caches

Figure 2.7(a) illustrates p processors sharing a bus to the memory. Assuming that each processor accesses k data items, and each data access takes time t_{cycle}, the

execution time is lower bounded by t_{cycle} x kp seconds. Now consider the hardware organization of Figure 2.7(b). Let us assume that 50% of the memory accesses (0.5k) are made to local data. This local data resides in the private memory of the processor. We assume that access time to the private memory is identical to the global memory, i.e., t_{cycle}. In this case, the total execution time is lower bounded by 0.5 x t_{cycle} x k + 0.5 x t_{cycle} x kp. Here, the first term results from accesses to local data and the second term from access to shared data. It is easy to see that as p becomes large, the organization of Figure 2.7(b) results in a lower bound that approaches 0.5 x t_{cycle} x kp. This time is a 50% improvement in lower bound on execution time compared to the organization of Figure 2.7(a). ■

Figure 2.7. Bus-based interconnects (a) with no local caches; (b) with local memory/caches.

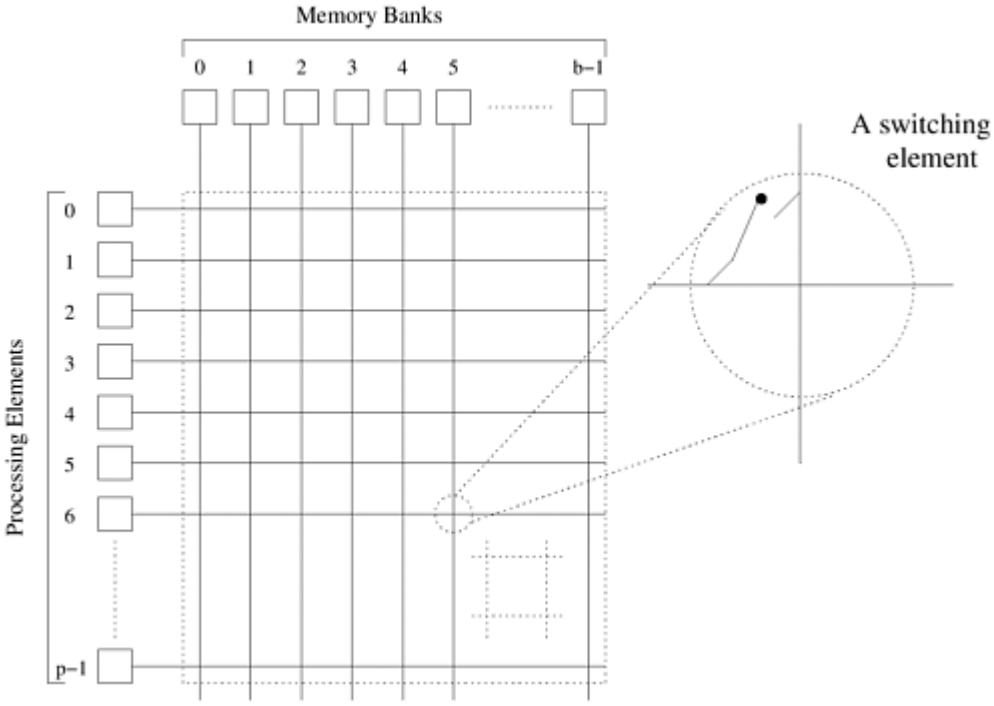


In practice, shared and private data is handled in a more sophisticated manner. This is briefly addressed with cache coherence issues in Section 2.4.6.

Crossbar Networks

A simple way to connect p processors to b memory banks is to use a crossbar network. A crossbar network employs a grid of switches or switching nodes as shown in Figure 2.8. The crossbar network is a non-blocking network in the sense that the connection of a processing node to a memory bank does not block the connection of any other processing nodes to other memory banks.

Figure 2.8. A completely non-blocking crossbar network connecting *p* processors to *b* memory banks.



The total number of switching nodes required to implement such a network is $\Theta(pb)$. It is reasonable to assume that the number of memory banks b is at least p; otherwise, at any given time, there will be some processing nodes that will be unable to access any memory banks. Therefore, as the value of p is increased, the complexity (component count) of the switching network grows as $\Omega(p^2)$. (See the Appendix for an explanation of the Ω notation.) As the number of processing nodes becomes large, this switch complexity is difficult to realize at high data rates. Consequently, crossbar networks are not very scalable in terms of cost.

Multistage Networks

The crossbar interconnection network is scalable in terms of performance but unscalable in terms of cost. Conversely, the shared bus network is scalable in terms of cost but unscalable in terms of performance. An intermediate class of networks called *multistage interconnection networks* lies between these two extremes. It is more scalable than the bus in terms of performance and more scalable than the crossbar in terms of cost.

The general schematic of a multistage network consisting of p processing nodes and b memory banks is shown in Figure 2.9. A commonly used multistage connection network is the *omega network*. This network consists of log p stages, where p is the number of inputs (processing nodes) and also the number of outputs (memory banks). Each stage of the omega network consists of an interconnection pattern that connects p inputs and p outputs; a link exists between input i and output j if the following is true:

Equation 2.1

$$j = \begin{cases} 2i, & 0 \le i \le p/2 - 1 \\ 2i + 1 - p, & p/2 \le i \le p - 1 \end{cases}$$