Contents

CHAPTER 1

Parallel Programming Platforms

The past decade has seen tremendous advances in microprocessor technology. Clock rates of processors have increased from about 40 MHz (eg., a MIPS R3000, circa 1988) to over 1.5 GHz (eg., a Pentium 4, circa 2000). At the same time, processors are now capable of executing multiple instructions in the same cycle. The average nymber of cycles per instruction (CPI) of high end processors has improved by roughly an order of magnitude over the past 10 years. All this translates to an increase in the peak floating point operation execution rate (floating point operations per second, or FLOPS) of several orders of magnitude. A variety of other issues have also become important over the same period. Perhaps the most prominent of these is the ability (or lack thereof) of the memory system to feed data to the processor at the desired rate. Significant innovations in architecture and software have addressed the alleviation of bottlenecks posed by the datapath and the memory.

The role of concurrency in accelerating computing elements has been recognized for several decades. However, their role in providing multiplicity of datapaths, increased access to storage elements (both memory and disk), scalable performance, and lower costs is reflected in the wide variety of applications of parallel computing. Desktop machines, engineering workstations, and compute servers with two, four, or even eight processors connected together are becoming common platforms for design applications. Large scale applications in science and engineering rely on larger configurations of parallel computers, often comprising of hundreds of tightly coupled processors. Data intensive platforms such as database or web servers and applications such as transaction processing and data mining often use clusters of workstations that provide high aggregate disk bandwidth. Applications in graphics and visualization use multiple rendering pipes and processing elements to compute and render realistic environments with millions of polygons in real time. Applications requiring guaranteed (high) availability rely on parallel and distributed platforms for redundancy. It is therefore extremely important, from the point of view of cost, performance, and applications, to understand the principles, tools, and techniques for programming the wide variety of parallel platforms currently available. As we proceed through this endaevour, we will also show how the principles of efficient parallel programming help in writing efficient serial code.

In this chapter, we provide a broad overview of hardware trends and platforms with a view to understanding the costs associated with various parallel computing tasks and their impact on parallel algorithm development and programming.

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 effort towards exploiting parallelism as a means of accelerating applications. Afterall, 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 Speed Argument

Back 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 remain nearly constant for at least 10 years." (Moore 1965)

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." (Moore 1975)

He revised his rate of circuit density doubling to eighteen months and projected the rate from 1975 onwards at this reduced rate. This curve came to be dubbed as the "Moore's Law". Formally, Moore's Law states that circuit density or capacity of semiconductors doubles every eighteen months. This empirical relationship has been amazingly resilient over the years both for microprocessors as well as DRAMs. By relating component density directly (linearly) 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 and physical size 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, there are two issues that are critical to sustaining current growth rates in computation speed: (i) there is significant agreement that the clock rates of microprocessors are unlikely to maintain past growth rates; and (ii) in the face of the first observation, how does one translate transistors into useful OPS (operations per second).

The first observation, namely that clock speeds are not expected to grow 15-20 fold in the next decade, is prompted by fundamental physical limitations. Light travels roughly one foot in a nanosecond (one clock tick of a one GHz clock). Assuming a 20 mm die size, we are already within one order of magnitude with respect to signal propagation time across the die. The obvious solutions to this problem would involve either a higher level of device integration or pipelining of signals. The level of device integration is unfortunately impacted by electromagnetic (inductive and capacitive) effects at high frequencies and heat dissipation. Pipelining signals on transmission lines in processors would require fundamental developments in device technology.

With increasing number of components available on the die, the logical recourse to achieving performance is to rely on parallelism – both implicit and explicit. We will briefly discuss implicit parallelism in Section **??** and devote the rest of this text 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% over the past decade, DRAM access times have only improved at the rate of roughly 10% over this interval. Coupled with increases in instructions executed per clock cycle, this gap between processor speed and memory presents a tremendous performance bottleneck. The speed mismatch between a 0.66 to 1.0 ns processor clock and a 50 to 60 ns 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 latency, the net effective bandwidth between DRAM and the processor poses additional problems for sustained computation rates. While advances in bus and memory architectures (page mode and static-column-mode operation), have adressed this issue, a significant bandwidth mismatch still exists between the two ends of the datapath. Cache memories also help in alleviating this bandwidth mismatch by providing higher bandwidth to a small memory that can be used to store repeatedly accessed data items.

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 **??**. Parallel platforms typically yield better memory system performance by (i) larger caches (typically linear in the number of processors) and (ii) higher aggregate bandwidth to the memory system. 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 also 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 computational speed but rather on their 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. These applications typically have constraints relating to location of data and/or resources. 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 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 the infeasibility of alternate (centralized) approaches.

1.2 Implicit Parallelism: Trends in Microprocessor Architectures*

While microprocessor technology has delivered significant improvements in clock speeds over the past decade, there are definite trends indicating that continued improvements will require significant breakthroughs in design and fabrication of devices. In view of this, 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.

In Figure 1.1, we plot the improvement in clock speeds with time over the past 15 years. While this figure paints a very favorable picture, it has some important implications. Fundamental limitations of physics indicate that a 40-fold increment in performance over the next 15 years is unlikely without fundamental breakthroughs in device fabrication and operation. Even if such an increment in clock speed is possible, limitations of memory technology would severely dilute its impact as we shall see in Section 1.3. Consequently, designers have increasingly relied on techniques that enable execution of multiple instructions in a single clock cycle. Indeed, this trend is evident in current generation of microprocessors such as the Intel Pentium IV, Sparc UltraIII, MPIS R12000, PA/RISC 8500, and Power3. In this section, we briefly explore mechanisms used by various processors for supporting multiple instruction execution.

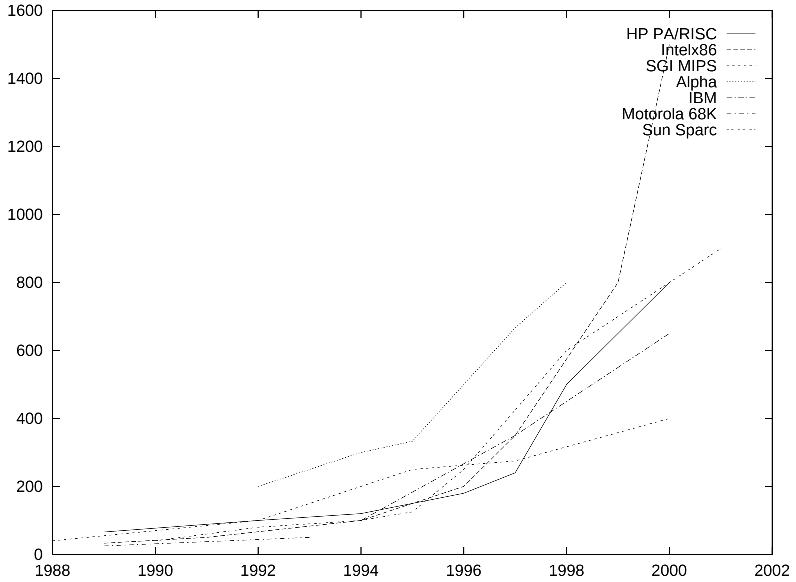


Figure 1.1 Improvements in clock speeds of microprocessors from various vendors.

1.2.1 Pipelining and Superscalar Execution

Processors have long relied on pipelines for improving execution rates. By overlapping various stages in instruction execution (fetch, decode, operand fetch, execute, store), 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 also enables faster clock rates since the tasks are now smaller. For example, the Pentium 4, which operates at 1.5GHz has a 10 stage pipeline. The speed of a single pipeline is ultimately limited by the largest atomic task in the pipeline. Furthermore, in typical instruction traces, every 5th to 6th 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.

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 1.1 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 in Figure 1.2 allows two issues per clock cycle, it is also referred to as two-way superscalar or dual issue execution.

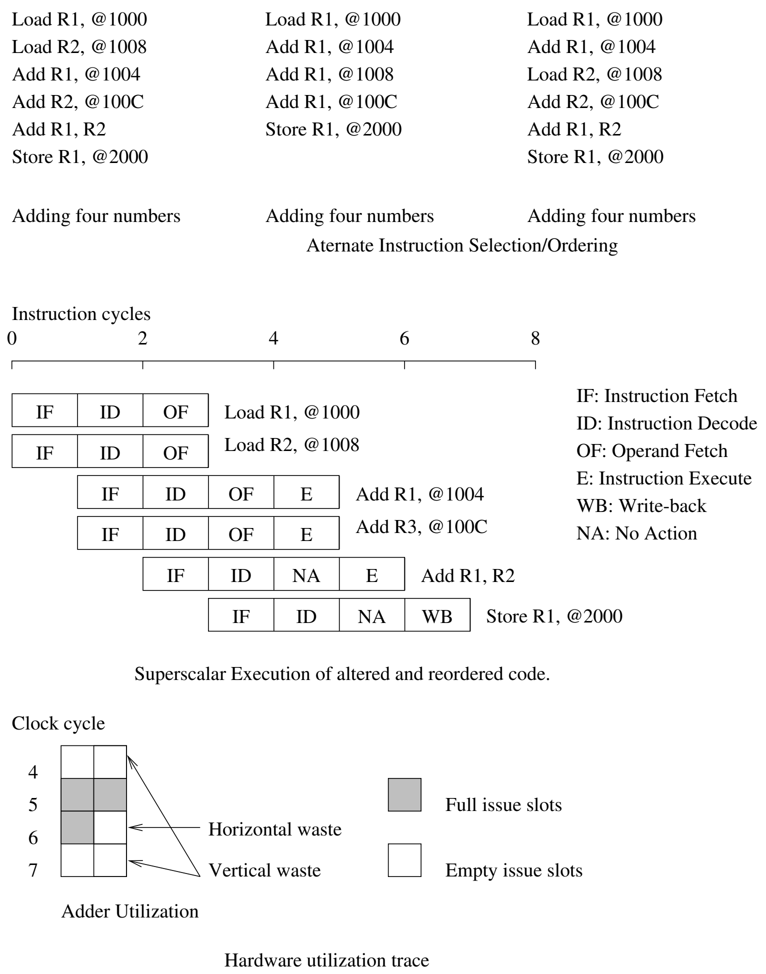


Figure 1.2 Example of a two-way superscalar execution of instructions.

Consider the execution of the first code fragment in Figure 1.2 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 (there is no execution stage for the Load instructions). 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 1.2(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 1.3 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 1.1, instructions in a program are 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 1.2 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: (i) since the resolution is done at run-time, it must be supported in hardware. The complexity of this hardware can be very high; (ii) the amount of instruction level parallelism in a program is often limited and is a sensitive function of coding technique. In the second code fragment, there can be no simultaneous issue; therefore leading to poor resource utilization. Such examples and techniques for alleviating this drawback are illustrated in Problem **??**. The three code fragments in Figure 1.2 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 an average, a branch instruction is encountered between every 5 to 6 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 1.2 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 needed 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 1.2. 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 (add-multiply 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 3 of the 8 available cycles are used for computation. This implies that the code fragment will yield no more than three-eigths 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 four-issue superscalar execution.

1.2.2 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 UltraSPARC). This complexity grows 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 (and 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, 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, superscalar architectures have the advantage of binary compatability. Code compiled for traditional processors can be executed on superscalar processors supporting the same instruction set without any changes. This is not the case for VLIW processors since they require extensive compiler support. One of the problems with the static scheduling of VLIW processors arises from the fact that dynamic process environments are difficult to predict at compile time. For instance, the processor might stall on a data fetch because of a cache miss. 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 an important role 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 4-8 way parallelism.

1.3 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 conventional model of a (serial) random access machine (RAM) assumes that the latency l is constant for any memory word and over all time instants. Unfortunately, this assumption does not hold for the current generation of memory systems, and potentially results in several orders of magnitude degradation in performance. Similarly, the rate at which data can be pumped from the memory to the processor determines the bandwidth of the memory system. This bandwidth is also a sensitive function of the spatial and temporal location of the memory block being accessed.

It is very important to understand the difference between latency and bandwidth since different, often competing, techniques are required for addressing these. If water comes out of the end of a fire hose 2 seconds after the 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. If on the other hand, if we wish to fight bigger fires, we might desire a higher flow rate, necessitating a bigger 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.

The sequence of examples presented here makes many simplifying assumptions, but is designed to effectively convey the role of key memory system compoments. For example, DRAM chips are typically available in narrow configurations (eg., 16M $\times$ 4 or 64M $\times$ 1 bits). These components can be physically organized to give a logical view of a wider memory (for eg., eight 16M $\times$ 4 modules can be stacked sideways to present a 16M $\times$ 32 logical view). This is briefly discussed in Example 1.4. Current DRAM chips (EDO-DRAMs, Synchronous DRAMs) also support several optimizations such as page mode operation. Memories are typically organized into smaller number of rows with a large number of bits in each row. For example, a 64M $\times$ 1 memory may be internally organized into a 8k $\times$ 8k grid of bits. The 13 most significant bits of the address are used to identify the row number. This row is fetched into the row buffer and the 13 least significant bits are used to identify the desired bit within this row. This entire operation may take in the range of 100 ns. However, if a subsequent data item accessed is already present in the row buffer, then the access time for this data item can be significantly lower (in the range of 25 ns). Such optimizations that form the basis for page-mode operation are not considered in the following series of examples.

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

Example 1.2 Effect of Memory Latency on Performance

Consider a processor operating at 1 GHz (1 ns clock) connected to a DRAM with a latency of 100ns (no caches). Assume that the processor has two add-multiply 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. Therefore, assuming that each floating point operation requires at least one data fetch (i.e. we assume that the number of operations on register data only are insignificant), the peak speed of any computation is limited to one floating point operation every 100 ns, or a speed of 10 MFLOPS. This is a very small fraction of the peak processor rating. Clearly, this example motivates the need for effective memory system performance in achieving high computation rates. ∎

1.3.1 Improving Effective Memory Latency Using Caches

Contrary to microprocessor speeds, improvements in DRAM latencies have been much slower (roughly 10%/year versus 40%/year for microprocessors over the past decade). Handling this mismatch in speed 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 at any level is called the cache hit ratio for that level. The performance of memory bound programs is critically impacted by the cache hit ratio.

Example 1.3 Impact of caches on memory system performance

As in the previous example, consider a 1 GHz processor connected to a 100 ns latency DRAM. In this case, we introduce a cache of size 32KB 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 32×32. 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 $n \times n$ matrices takes $2n^3$ operations. For our problem, this corresponds to 64K operations, which can be performed in 16K cycles (or 16μs) at 4 instructions per cycle. The total time for the computation is therefore given by 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 296 MFLOPS. Note that this is a twenty-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 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. In our example, we had $O(n^2)$ data accesses and $O(n^3)$ computation. 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. As a programmer, this assumption has important implications. First, there are applications for which data reuse is limited. Consider the problem of adding two vectors or computing their dot-product. The reader can verify that there are as many memory accesses in this problem as there are FLOPs in the asymptotic sense. This means that the peak speed for such a computation is typically bound by the rate at which data can be moved between DRAM and the cache. The second implication is that for a given computation, different access patterns may lead to markedly different performance (Problem **??**.

1.3.2 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. Let us now relax our simplifying restriction on the size of the memory block and assume that a single memory request returns a block of 4 words. The single unit of 4 words in this case is also referred to as a cache line. Conventional computers typically fetch 2-8 words together into the cache.

Example 1.4 Effect of block size: Computing the dot-product of two vectors
Consider again a memory system with a cache of 32 KB and DRAM of 256 MB with the processor operating at 1 GHz. The latency to cache is one cycle and the latency to DRAM is 100 cycles. Assuming a block size of one word, the processor fetches one word at a time and pays a cost of 100 cycles. For each pair of vector components, the dot-product performs one multiply-add (two FLOPs). Therefore, the algorithm performs one FLOP every 100 cycles for a peak speed of 10 MFLOPS.

Now let us consider what happens if the block size is increased to 4 words; ie., the processor can fetch a 4-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 vector elements, consequently, two accesses can fetch four elements of both 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 8 data accesses. This corresponds to a cache hit ratio of 75%. Assuming that the dominant overhead is posed by the cache misses, the averate memory access time contributed by the misses is 25% at 100 ns (or 25 ns/word). Since the dot-product has 1 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 1.4 corresponds to a wide data bus (4 words or 128 bits) connected to multiple memory banks. In practice, it is not feasible to construct such wide buses. Consequently, data items are sent on the bus one after the other. 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 $100 + 3 \times$ bus cycle ns. Assuming a data bus operating at 200 MHz, this adds an additional 15 ns to the cache line access time. This does not change our bound on the execution rate significantly.

Example 1.5 Computing the product of a matrix with a vector

Consider an indentical setup to the one outlined above. With a 4-word cache line, the processor can fetch 4 matrix elements. In an ideal setup, the vector elements can be reused across the entire matrix. Therefore, in the best case, we can execute 8 FLOPS (4 multiply-adds) on each cache line corresponding to the matrix assuming, asymptotically, that the vector is already available. Since this takes 100 cycles (100 ns), the peak computation rate corresponds to 80 MFLOPS. Note that this is twice the computation rate of the dot-product. This increase in computation rate is a result of the data reuse associated with the vector.

We can compute the average memory access time as before to be 1 DRAM access for every 8 data accesses (1 DRAM access gets four elements of the matrix. The corresponding elements of the vector are already in the cache). This corresponds to a cache hit ratio of 87.5% and a mean access time of $0.875 \times 1 + 0.125 \times 100$ or 13.4 ns. The corresponding computation rate is 74.8 MFLOPS. Note that the number of operations per memory word is still one, because we are counting the vector repeatedly to compute our cache hit ratio. ■

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. It is relatively easy to beat the cache mechanism by accessing data in strides where the stride value is suitably selected (Problem **??**). 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 the memory. Compilers can often be relied on to do a good job of restructuring computation to take advantage of spatial locality.

Example 1.6 Impact of Strided Access

Consider the following code fragment:

```
for (i = 0; i < 1000; i++)
        column_sum[i] = 0.0;
        for (j = 0; j < 1000; j++)
                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. For a matrix stored in a row-major order, this corresponds to accessing every 1000^{th} entry. Therefore, it is likely 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. (Problem **??**). ■

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 1.7 Eliminating Strided Access

Consider the following restructuring of the column-sum fragment:

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

In this case, the matrix is traversed in a row-order. 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 tile the iteration spaces. The improved performance of this loop is left as an exercise for the reader (Problem **??**).
■

Example 1.8 More on Strided Access

Consider the problem of multiplying two $4K \times 4K$ single precision matrices on the machine outlined in the previous examples. Note that the 32KB cache is just enough to store two full rows of the matrices (4 bytes per float). Using a simple dot-product formulation in which each element of the result matrix is expressed as the dot product of a row of A with a column of B, we expect to get a performance atleast as good as the dot product, and indeed a lot better since the matrix multiplication performs $O(n^3)$ operations on $O(n^2)$ data for a reuse factor of $O(n)$. However, matrices are typically stored in a row-major (or column major) order. This implies that while accessing rows of A, successive elements will be available on the same cache line. However, while accessing columns of B, three other words are fetched that correspond to next three elements in the same row. However, these entries are never used since they are overwritten before we can use them. Consequently, while elements of A can be reused entirely, we can perform only two FLOPs (one multiply-add) on each cache line fetch. Therefore, we can perform only two FLOPs every 100 cycles (100 ns) for a peak speed of 20 MFLOPS. It is clear from this example that data placement and organizing computation appropriately can make the difference between excellent and very poor performance. ■

Let us try to restructure our computation in the above example to see if we can render this computation processor bound as opposed to memory bound.

Example 1.9 Fixing the matrix-multiplication algorithm

An obvious cause of the poor performance of the previous matrix multiplication algorithm was the absence of spatial locality in the second matrix (the not-so-obvious issue of temporal locality is discussed later in this example). We were wasting three of the four words of the second matrix fetched from memory. An obvious fix to this problem is to compute the elements of the result matrix C four at a time. Using this approach, we can increase our FLOP count to 80 MFLOPS with a simple restructuring of the program. This is left as an exercise for the reader (Problem **??**).

However, it is possible to achieve much higher performance from this problem. This is possible by viewing the matrix multiplication problem as a cube in which each internal point corresponds to a multiply-add operation. Matrix multiplication algorithms traverse this cube in different ways; which induce different partitions on the cube. The data required for computing a partition grows as the surface area of the input faces of the partition and the computation as the volume of the partition. For the algorithms discussed above, we were slicing thin partitions of the cube for which the area and volume were comparable, thus resulting in poor cache performance. To remedy this, we restructure the computation by partitioning the cube into subcubes of size $k \times k \times k$. The data associated with this is $3 \times k^2$ (k^2 data for each of the three matrices) and the computation is k^3. To maximize performance, we would like

$3 \times k^2$ to be equal to $8K$ words, since that is the amount of cache available. This corresponds to $k = 51$. The computation associated with a cube of this dimension is 132651 multiply-add operations or 265302 FLOPs. To perform this computation, we needed to fetch two submatrices of size 51×51. This corresponds to 5202 words or 1301 cache lines. Accessing these cache lines takes 65050 cycles or 130100 ns. Since 265302 FLOPs are performed in 130100 ns, the peak computation rate of this formulation is 2.04 GFLOPS! The restructured algorithm is left to the reader as an exercise (Problem **??**). ■

So the next question is whether we have effectively solved the problems posed by memory latency and bandwidth. Indeed, the answer to this problem is illustrated in Figure **??**. While peak processor rates have grown very fast over the past decade, memory latency and bandwidth has 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/sec 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 an 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 rate of data transfer required by a computation to make it processor bound.

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

- Exploiting spatial locality in applications is critical for amortizing memory latency.
- Certain applications are tolerant to low memory bandwidths to a higher degree than others. Looking at the ratio of the number of operations to number of memory accesses is a good heuristic of anticipated tolerance to memory bandwidth.
- Memory layouts and organizing computation appropriately can make significant differences in overall computation rates.

1.3.3 Alternate Approaches for Improving Memory System Performance

Imagine sitting at your computer browsing the web during peak network traffic hours. One of the techniques we often use in such situations is to open multiple browsers and to browse other pages while the slow pages load. The same principle can be applied to keep the processor busy while slower memory accesses are being performed. This approach, called *multithreading* can be a very effective tool for hiding memory latency. We illustrate this by means of the following example:

Example 1.10 Hiding Latency by Multithreading

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. ■

Example 1.10 illustrates the concept of multithreading; a technique that can be used to hide the latency of a variety of tasks ranging from memory and disk access to I/O and interprocessor communication. Multithreaded processors are capable of maintaining the context of a number of threads of computation with outstanding requests (memory accesses, I/O, or communication request) and execute them as the requests are satisfied. Machines such as the HEP and Tera rely on mulththreaded 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.

A related technique called prefetching can also be useful in hiding memory latency. In a typical program, a data item is loaded and used 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 will have to 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 same registers) with respect to other threads. Many compilers aggresively try to advance loads to mask memory system latency.

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 1.11 Impact of Bandwidth on Multithreaded Programs

Consider a computation running on a machine with a single level cache for which the cache hit ratio at 1KB of cache is 25% and at 32KB of cache is 90%. Consider two cases, one a single threaded execution in which the entire cache is available to the serial context, and a multithreaded execution with 32 threads where each thread has a cache residency of 1KB. 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 10ns since the other words come from the cache. This corresponds to a bandwidth of 400MB/s. In the second case, the bandwidth requirement to DRAM increases to three words every 4 cycles of each thread. Assuming that all threads exhibit similar cache behavior, this corresponds to 0.75 words/ns, or 3 GB/s. ■

Example 1.11 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.

1.4 Parallel Computing Platforms

In the preceding sections, we pointed out various factors that impact the performance of a serial 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. The drive towards efficient exploitation of parallelism has been facilitated by the tremendous advances made in networking technology and software standardization. It is now possible to buy of-the-shelf networks with gigabit links and switches for as little as $1K per port with this figure expected to drop in the future. Using this networking infrastructure, it is possible to construct cost-effective parallel platforms capable of yielding high performance. The key component of the parallel computing puzzle is the ability to program these parallel platforms to achieve efficient processor utilization.

In addition to hardware trends and prices, additional impetus for parallelism is provided by a variety of applications. Transaction processing in databases, web servers, mining large databases, and real time graphics and visualization provide mainstream applications in addition to traditional numerically oriented problems for parallel computing. With all these in mind, we introduce, at a high level, the elements of parallel computing platforms that are critical for performance oriented and portable parallel programming. Note that this is not meant to be an exhaustive treatise on parallel architectures; merely a sufficiently detailed account to facilitate parallel program design. We refer the interested reader to the bibliography for further readings on parallel computing architectures.

To facilitate our discussion of parallel platforms, we explore a dichotomy based on the logical and physical organization of parallel platforms. The logical organization refers to a programmers view of the platform while the physical organization refers to the actual hardware organization of the platform.

1.4.1 The Logical Organization of Parallel Machines

The two critical components of parallel computing from a programmers 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.

1.4.2 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. Within these extremes lie a range of models for specifying the control structure of programs and the corresponsing architectural support for them.

Multiple instances of the same program executing on different data belong to the class of single program multiple data, or SPMD model. It is easy to see that this model has the same expressiveness as the multiple program model since each of the multiple programs can be inserted into one large if-else statement with conditions specified by the task identifiers. This control model is widely used by many parallel platforms and requires minimal architectural support. Examples of such platforms include the Cray T3E, Origin 2000, and IBM SP2. Under many programming paradigms discussed later in this text, clusters of workstations also belong to this class of platforms.

In an alternate model, it is possible to visualize the same instruction being executed by multiple processors on different data items as shown in the following example:

Example 1.12 Parallelism from single instruction on multiple processors.
Consider the following code segment that adds two vectors:

```
for (i = 0; i < 1000; i++)
        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 independent 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. ■

This form of expressing parallelism is historically referred to as single instruction multiple data, or SIMD parallelism. Supporting such a model requires significant architectural support. Instructions in this model are executed in a lock-step manner on different data items. There is a global control unit that despatches the instruction to various processing units. After each instruction execution, the processing units are synchronized and execution proceeds. Some of the earliest parallel computers such as the Illiac IV, MPP, DAP, CM-2, MasPar MP-1 belonged to this class of machines. More recently, variants of this concept have found use in coprocessing units such as the MMX units in Intel processors and DSP chips such as the Sharc. The Intel Pentium 3 with its SSE (Streaming SIMD Extensions) and the Pentium 4 with SSE2 provide a number of instructions that execute a single 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.

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 <stmnt> <elsewhere stmnt>` are used to support selective execution. Conditional execution can be detrimental to the performance of SIMD processors; and therefore must be used with care.

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 larger product development times. Since the underlying serial processors change so rapidly, SIMD computers suffer from fast obsolescence. Characteristics of applications also have a significant impact on computers. While regularly structured computations are well suited to SIMD machines, unstructured applications are more naturally suited to MIMD computers.

1.5 Communication Model of Parallel Platforms

In addition to facilitating specification of parallel tasks, a parallel platform must also support mechanisms for exchange of data between these tasks. There are two primary forms of data exchange between parallel tasks – exchanging messages and accessing a shared data space.

1.5.1 Platforms Supporting Message Passing.

A program's view of a message passing platform consists of several independent computers (processors and their memory) connected via an interconnection network. For this reason, message passing platforms are also called *multicomputers*. Processors in this model can interact only by passing messages. Since each processor has exclusive access to its memory, it is also referred to as *local* or *private* memory. Examples of message-passing parallel platforms include the IBM SP2, Cray T3E, SGI Origin 2000, and workstation clusters.

Note that some of these machines may support multiple logical views – i.e., under different programming models, they may present a message passing or a shared address space view.

1.5.2 Shared-Address-Space Platforms

The shared address space view of a parallel plaform 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 to access global memory is longer than the time to access a local memory, the platform is called a *non-uniform memory access* (NUMA) multicomputer. Figure **??**(a) represents a UMA platform, whereas Figures **??**(b) and (c) represent NUMA platforms. 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.

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 processors. The dichotomy of shared memory versus distributed memory computers pertains to the physical organization of the machine and is discussed in greater detail in Section 1.6. Either of these physical models, shared or distributed memory models, can present a logical view of a message passing or shared address space platform. A distributed memory shared address space computer is identical to a NUMA machine.

Supporting a shared address space in a parallel machine requires considerable hardware support. The basic problem arises because there are potentially multiple copies of data items residing in caches of different processors and memories (either shared or distributed). The difficulty arises in ensuring that a processor accessing a data item gets the most recent copy of the data item and that concurrent accesses conform to some deterministic semantics. This issue of maintaining a coherent view of the memory is central to shared address space machines and is discussed in greater detail in Section 1.6.4.

It is easy to emulate a message-passing architecture containing p processors on a shared-address-space computer with an identical number of processors. 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 sends a message to another processor by writing into the other processor's partition of memory. However, emulating a shared-address-space architecture on a message-passing computer is costly, since accessing another processor's memory requires sending and receiving messages. Shared address space platforms typically provide additional hardware for supporting a common address space.

1.6 Physical Organization of Parallel Platforms

A variety of physical architectures have been used to support the logical machine views outlined in Section 1.4.1. Figure 1.3 illustrates a general high level view of the physical organization of parallel machines. A parallel platform consists of a number of processors, each potentially with its own cache and local memory, connected via an interconnection network to each other, and potentially to a global shared memory. If the processors in this machine do not have any local memory, the architecture is called a shared memory architecture. Examples of such machines include the C.mmp and the NYU Ultracomputer. If on the other hand, it only has local memory and no shared memory, the architecture is called a distributed memory platform. Examples of such machines include the IBM SP2, SGI Origin, Cray T3E, and workstation clusters. If the machine also has hardware to support a coherent view of the memory, it is called a shared address space platform. Examples of such machines include the SGI Origin and the Sun Enterprise servers. A distributed memory machine with support for coherence is also called a distributed shared memory (DSM) platform. Examples of such machines include, as before, the SGI Origin and the Sun Enterprise servers. Other platforms can be built as hybrids of these pure architectures. For example, the coherence mechanism may only support a shared address space over non-overlapping partitions of the machines. Such machines are sometimes also referred to as multiclusters. Examples of such machines include the ASCI ???.

In all of these architectural models, the two critical components particularly relevant from the parallel processing viewpoint are the interconnection network and the coherence mechanisms. We now discuss these aspects in greater detail.

1.6.1 Interconnection Networks for Parallel Computers

One of the most important factors determining the performance of a parallel system is the ability of the processors to communicate with each other. This ability is, to a large extent, determined by the interconnection network which carries data and control information between processors. In this section, we will study various aspects of interconnection networks and how they impact the performance of parallel programs.

Typical interconnection networks are built using links and switches. A link corresponds to a physical media such as a set of wires or fiber 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

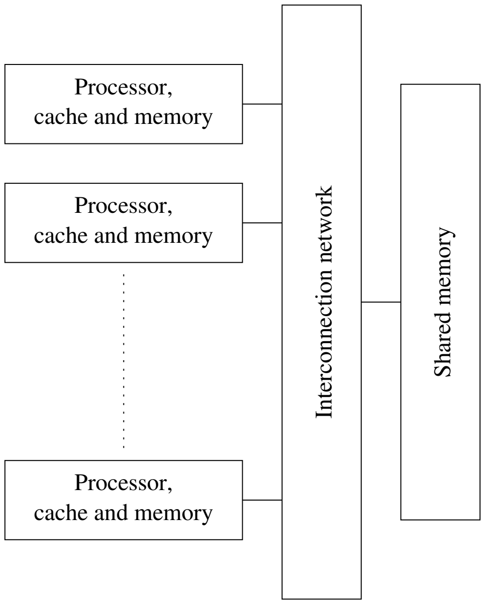


Figure 1.3 High level organization of a general parallel architecture.

coupling and attenuation of signal strength are functions of the length of the link. Consequently, physically tightly coupled systems such as the Cray T3E and SGI Origin 2000 have links that operate at faster rate (bandwidth) than those based on LAN technologies (such as an ATM and Gigabit ethernet).

A 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. In most cases, the number of input ports is identical to the number of output ports. This number 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, multiported 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 in the degree, and the packaging costs linearly in the number of pins.

The connectivity between the processor 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 processor, and error checking. The position of the interface between the processor and the network is also important. While conventional network interfaces hang off the I/O bus of the processor, interfaces in tightly coupled parallel machines hang off the memory bus. Since I/O buses are typically much slower than memory buses, the latter can support higher bandwidth.

1.6.2 Metrics for Interconnection Networks

It is convenient to study interconnection networks as graphs with edges corresponding to links and nodes corresponding to processors, memory modules, or switching elements. Before we discuss specific networks, it is necessary to study the performance and cost metrics that influence design and impact parallel programming.

Performance Metrics The first figure of merit associated with an interconnection network is established by the capacity of a link to carry data. Clearly, it is desirable that the link be able to carry large amounts of data; however, as we had seen, this capacity is limited by physical features such as link length and capacitance. The time for a message to get from a source node to the destination node is influenced by the number of intermediate steps in the path. The distance in terms of number of links between the farthest nodes in the network is referred to as the *diameter* of the network. Networks with small diameters are better than networks with large diameters. As we shall see in Section 1.7, the impact of the diameter can be minimized by using appropriate routing techniques.

At any given point in the execution of a parallel program, multiple processors may be accessing data across the interconnection network. The network must have an ability to carry these messages fast. The total number of links in a network is a measure of the ability of the interconnection network to handle multiple messages. Another metric is the minimum number of links that cross any equipartition of the network. This is referred to as the *bisection-width* of the network. At some point in the computation, it is possible that each node in one half of the machine is communicating with a node in the other half, and vice versa. In this situation, it is desirable to have a bisection-width of $O(p)$ for a graph with p nodes. Many structured and unstructured communication patterns can make use of $O(p)$ bisection-width. Furthermore, to temper the effect of varying communication patterns, many machines use randomized routing. Based on the number of processors participating in the communication operation, bisection-width is critical for performance.

In addition to these graph-based metrics, there are other physical factors that influence the performance of an interconnect. Some topologies require longer wires than others. This influences the nature of the link and the physical layout of the processors. In this chapter, we will limit ourselves to graph-based metrics, since, given the link bandwidth, the impact of the interconnect on performance is largely determined by these.

Cost Metrics A variety of factors influence the cost of a network. These include cost of switching elements, network interfaces and wiring costs. The cost of switching elements can be approximated by the number of switches used in the network. The cost of wiring can be approximated by the the total number of edges (links) in the network. In fact, the cost of wiring is a sensitive function of many other parameters such as whether the underlying network can be laid out in 2 or 3 dimensions, the number of wire crossings, and the length of wires. This strongly influences the network layout as well as width and clock-rate of links. However, these parameters are more difficult to quantify analytically. Consequently, number of switching elements and number of links are commonly used as cost metrics.

1.6.3 Network Topologies

A wide variety of network topologies have been used in interconnection networks. These topologies try to tradeoff 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

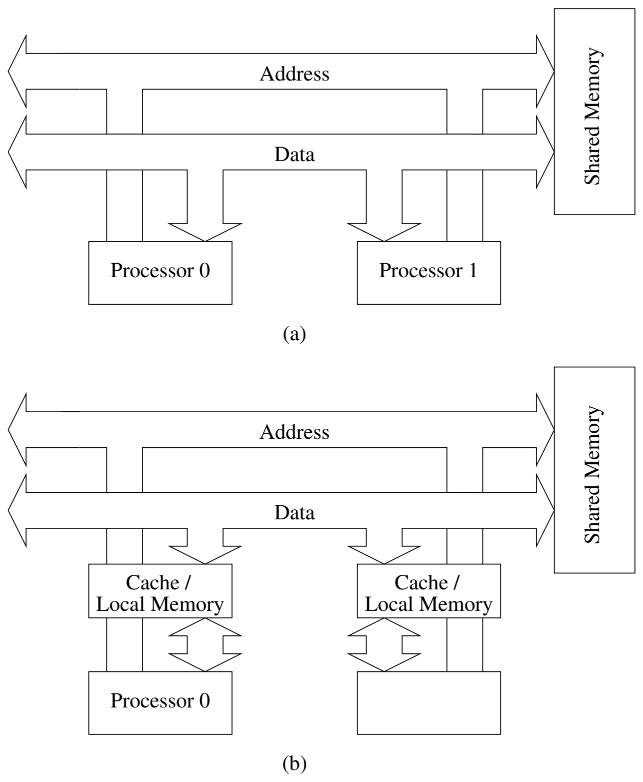


Figure 1.4 Bus-based interconnection networks (a) with no local caches; (b) with local memory/caches.

A bus-based network is perhaps the simplest network consisting of a shared media that is common to all the processors. A bus has the desirable property that the cost of the network scales linearly in 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 processors. Since the transmission media is shared, there is little overhead associated with broadcast compared to point-to-point message transfer. Buses are typically time-multiplexed; i.e., only one transmission can occur at any point of time. Therefore, the bisection width of a bus is $O(1)$. This places limitations on the overall performance of the network as the number of processors increases. Typical bus based machines are limited to dozens of processors.

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 processor. For such programs, it is possible to provide a cache for each processor. Private data is cached at the processor and only remote data is accessed through the bus.

Example 1.13 Reducing shared-bus bandwidth using caches

Figure 1.4(a) illustrates p processors sharing a bus to the memory. Assuming that each processor accesses k data, and each data access takes time t_{cycle}, the execution time is lower bounded by $t_{cycle} \times kp$ seconds. Now consider the hardware organization of Figure 1.4(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 $0.5 \times t_{cycle} \times k + 0.5 \times t_{cycle} \times kp$. 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 1.4(b) results in a 50% improvement in lower bounds on execution time. ■

In practice, shared and private data is handled in a more sophisticated manner. This is briefly addressed with cache coherence issues in Section 1.6.4.

Crossbar Networks

A simple way to connect p processors to b memory banks is to use a crossbar switch. A crossbar switch employs a grid of switching elements as shown in Figure 1.5. The crossbar switching network is a non-blocking network in the sense that the connection of a processor to a memory bank does not block the connection of any other processor to other memory banks.

The total number of switching elements 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 processors that will be unable to access any memory bank. Therefore, as the value of p is increased, the complexity of the switching network grows as

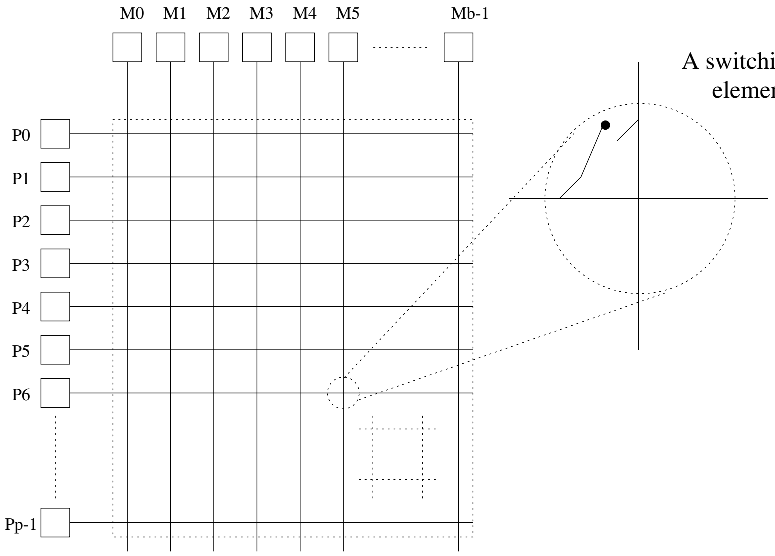


Figure 1.5 A completely non-blocking crossbar switch connecting p processors to b memory banks.

$\Omega(p^2)$. As the number of processors becomes large, this switch complexity is difficult to realize at high data rates. Consequently, crossbar networks are not very scalable in terms of cost. Currently available machines based on a crossbar switched network include the Sun Ultra HPC 1000 and the Fujitsu VPP500.

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 processors and b memory banks is shown in Figure 1.6. A commonly used multistage connection network is the *omega network*.This network consists of log p stages (where p is the number of processors and also the number of 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:

$$j = \begin{cases} 2i, & 0 \leq i \leq p/2 - 1 \\ 2i + 1 - p, & p/2 \leq i \leq p - 1 \end{cases} \tag{1.1}$$

Equation 1.1 represents a left-rotation operation on the binary representation of i to obtain j. This interconnection pattern is called a *perfect shuffle*. Figure 1.7 shows a perfect

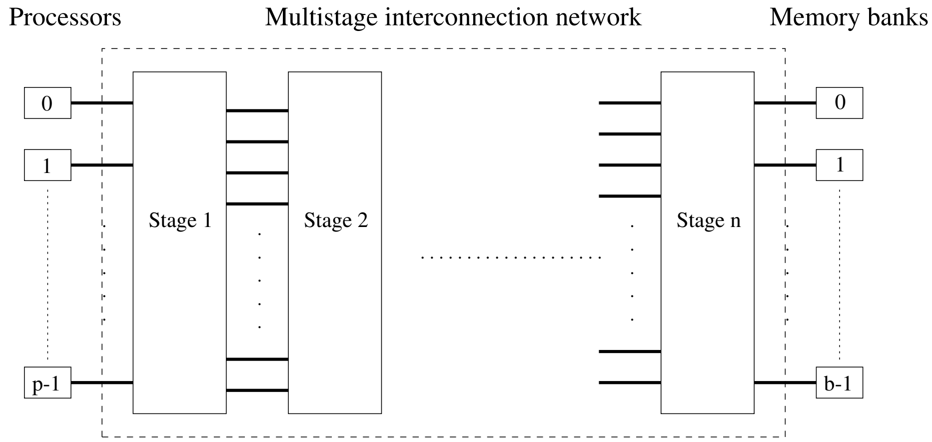


Figure 1.6 The schematic of a typical multistage interconnection network.

shuffle interconnection pattern for eight inputs and outputs. In each stage of an omega network, a perfect shuffle interconnection pattern feeds into a set of $p/2$ switching elements. Each switch is in one of two connection modes. In one mode, the inputs are sent straight through to the outputs, as shown in Figure 1.8(a). This is called the *pass-through* connection. In the other mode, the inputs to the switching element are crossed over and then sent out, as shown in Figure 1.8(b). This is called the *cross-over* connection. An omega network has $p/2 \times \log p$ switching elements, and the cost of such a network grows as $\Theta(p \log p)$. Note that this cost is less than the $\Theta(p^2)$ cost of a complete crossbar switch.

Figure 1.9 shows an omega network for eight processors. Processors form the input nodes of the network and memory banks form the output nodes. Routing messages in an omega network is accomplished by using a simple scheme. Let s and t be the binary representations of the source and destination of the message. The message traverses the link to the first switching element. If the most significant bits of s and t are the same, then the message is routed in pass-through mode by the switch. If these bits are different, then the message is routed through in crossover mode. This scheme is repeated at the next switching stage using the next most significant bit. Traversing $\log p$ stages uses all $\log p$ bits in the binary representations of s and t.

$k - d$ Meshes

The class of $k - d$ meshes refers to the general class of topologies consisting of d dimensions with k processors along each dimension.

The simplest instance of a $k-d$ mesh is a linear array of p processors. Here, k is equal to p and d is one. The bisection-width of the network is one and the diameter of the network is p. A simple extension of the linear array illustrated in Figure 1.10(a) is the ring or a 1-D torus. The ring has a wrap-around connection between the extremeties of the linear

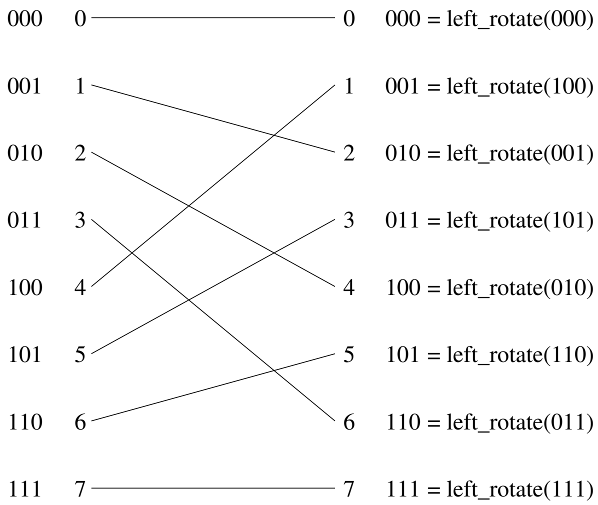


Figure 1.7 A perfect shuffle interconnection for eight inputs and outputs.

Figure 1.8 Two switching configurations of the 2×2 switch: (a) Pass-through; (b) Cross-over.

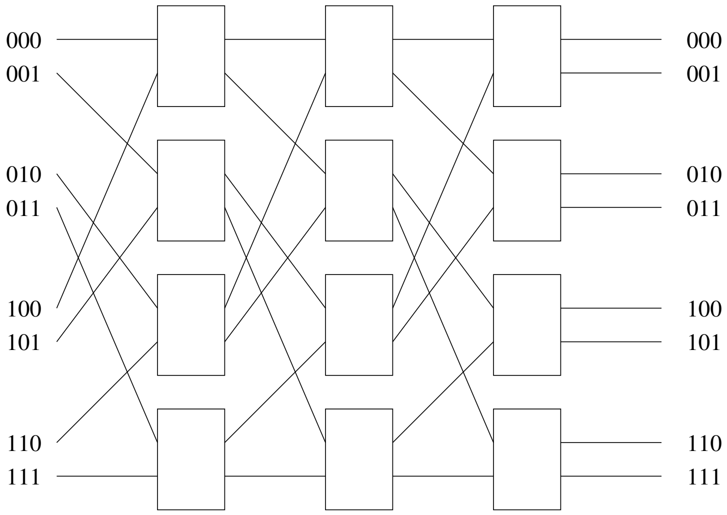
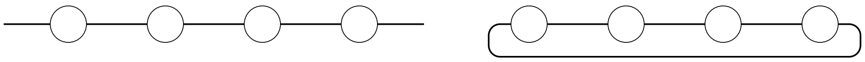


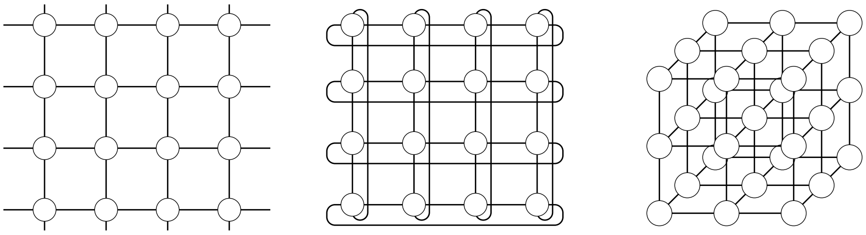
Figure 1.9 A complete omega network connecting eight inputs and eight outputs.



(a) (b)

Figure 1.10 Linear arrays (a) with no wraparound links; (b) with wraparound link.

array. This reduces the diameter of the network to $p/2$ and increases the bisection-width to two. The low bisection-width of this family renders them unsuitable for large scale parallel processing; however their low cost (linear in number of processors) and simplicity makes them attractive for local area networks (LANs). Instances of LANs based on rings include the Fiber Digital Data Interface (FDDI) and Scalable Coherent Interface (SCI).



(a) (b) (c)

Figure 1.11 Two and three dimensional meshes (a) 2-D mesh with no wraparound; (b) 2-D mesh with wraparound link (2-D torus); and (c) a 3-D mesh with no wraparound.

The two-dimensional mesh illustrated in Figure 1.11(a) is an extension of the linear array to two-dimensions. Each dimension has $\sqrt{p}$ processors with a processor identified by a two-tuple (i, j). The 2-D mesh has a diameter of $2\sqrt{p}$ between the diagonal extremeties and a bisection-width of $\sqrt{p}$. The 2-D mesh has the property that it can be laid out in 2-D space, making it attractive from the wiring point of view. Furthermore, a variety of regularly structured computations map very naturally to a 2-D mesh. For this reason, 2-D meshes were often used as interconnects in parallel machines. Two dimensional meshes can be augmented with wraparound links to form two dimensional torii illustrated in Figure 1.11(b). The bisection-width of a 2-D mesh can be increased, and the diameter reduced by increasing the dimensionality. The three-dimensional cube illustrated in Figure 1.11(c) has a bisection-width that scales as $p^{2/3}$ and diameter of $3p^{1/3}$. It is possible to show that a 3-D mesh can be laid out in space that scales linearly in the number of nodes. A variety of 3-D simulations have bandwidth requirements that scale as $p^{2/3}$. A 3-D topology is particularly ideal for such simulations. The SGI-Cray T3D and T3E are 3-D torii.

Just as the mesh forms one extreme of the $k-d$ mesh family, the other extreme is formed by an interesting topology called the hypercube. The hypercube topology has two processors along each dimension and $\log p$ dimensions. The construction of a hypercube is

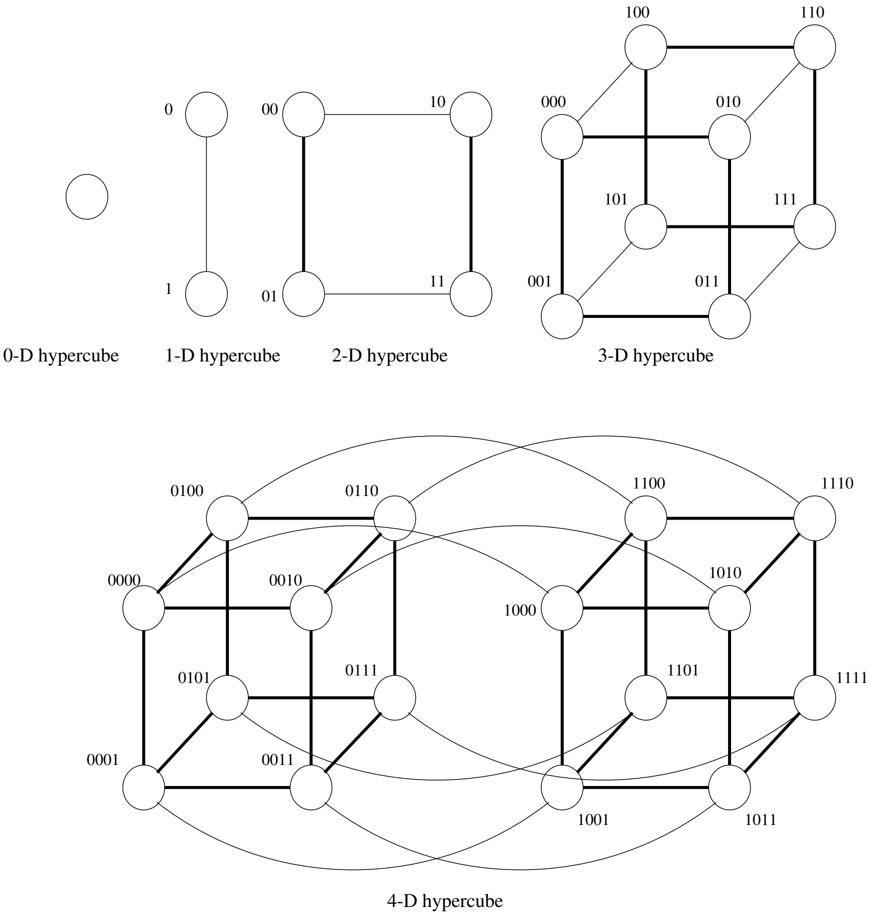


Figure 1.12 Construction of hypercubes from hypercubes of lower dimension.

illustrated in Figure 1.12. A zero-dimensional hypercube consists of 2^0, i.e., one processor. A one-dimensional hypercube is constructed from two zero-dimensional hypercubes by connecting them. A two-dimensional hypercube of four processors is constructed from two one-dimensional hypercubes by connecting corresponding processors. In general a d-dimensional hypercube is constructed by connecting corresponding processors of two $d - 1$ dimensional hypercubes. Figure 1.12 illustrates this for up to 16 processors in a 4-D hypercube.

Hypercubes have several properties that make them attractive from a programming perspective. Perhaps, the most important of these is the fact that they have bisection-width linear in the number of processors. Specifically, the bisection-width of a p processor hypercube is $p/2$. Other members of the $k - d$ mesh family can be effectively embedded into hypercubes. The diameter of the network is $\log p$. Because of all of these properties, hypercubes have traditionally been used to implement parallel computers from the days of the Cosmic Cube and nCube 1 to the SGI Origin 2000, which is based on a variant of the hypercube.

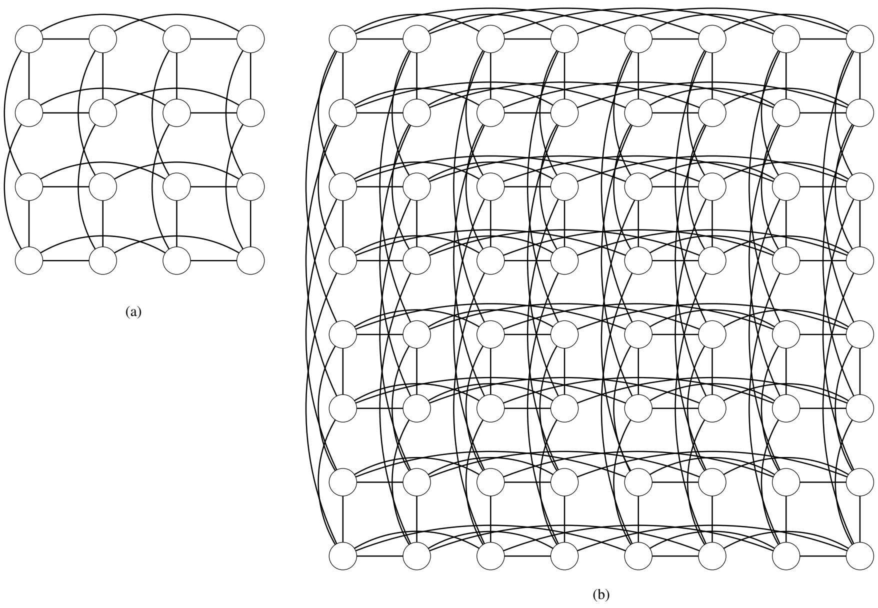


Figure 1.13 Embedding a higher dimensional $k-d$ cube into a lower dimensional cube with fatter links.

While higher dimensional $k - d$ meshes have attractive performance properties, they pose difficulties for wiring with short links. One way of addressing this issue is to consider the embeddings of higher dimensional network into a lower dimensional one. Figure 1.13(a) illustrates the embedding of a 16 processor hypercube into a mesh. It is easy to see from the embedding that each link in the mesh corresponds to a maximum of 2 links in the hypercube. In general, the embedding of a p processor hypercube into a p processor mesh results in a maximum of $\sqrt{p}/2$ hypercube links mapped to a mesh link. This can be derived simply as the ratio of bisection-widths. We will henceforth call such a mesh a fat-mesh. A fat-mesh has the same bisection-width as a hypercube; however it has a higher diameter. As we shall see in Section 1.7, by using appropriate message routing techniques, the effect of node distance can be minimized. In these cases, a fat-mesh provides an attractive alternative to higher dimensional meshes because of its superior layout properties.

In summary, the general class of $k - d$ meshes is characterized by a bisection width of $p^{(d-1)/d}$, a diameter of kd, and link and switch costs of pd.

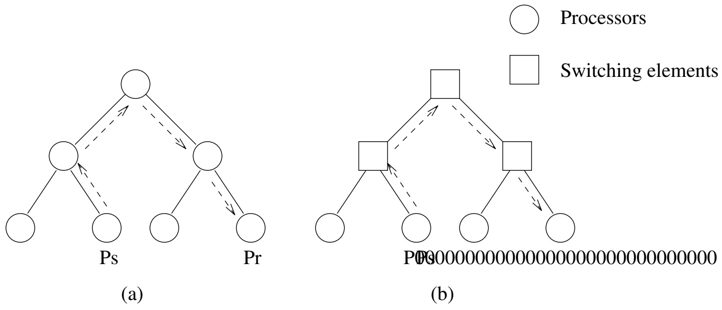


Figure 1.14 Complete binary tree networks and message routing in them.

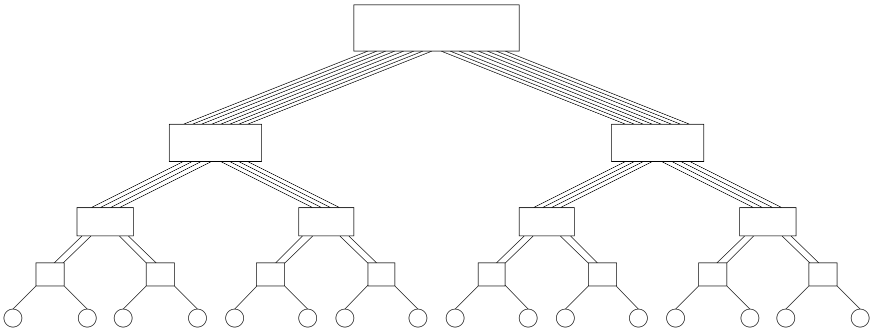


Figure 1.15 A fat tree network of 16 processors.

Tree-Based Networks

A *tree network* is one in which there is only one path between any pair of processors. Both linear arrays and star-connected networks are special cases of tree networks. Figure 1.14 shows networks based on complete binary trees. Static tree networks have a processor at each node of the tree (Figure 1.14(a)). Tree networks also have a dynamic counterpart. In a dynamic tree network, nodes at intermediate levels are switching elements and the leaf nodes are processors (Figure 1.14(b)).

To route a message in a tree, the source processor sends the message up the tree until it reaches the processor or the switch at the root of the smallest subtree containing both the source and destination processors. Then the message is sent down the tree toward the destination processor. This is illustrated in Figure 1.14.

Tree networks suffer from a communication bottleneck at higher levels of the tree. For example, when many processors in the left subtree of a node communicate with processors in the right subtree, the root node has to handle all the messages. This problem can be alleviated by increasing the number of communication links between processors that are closer to the root. This network, illustrated in Figure 1.15, is called a *fat tree*.

1.6.4 Cache Coherence in Multiprocessor Systems

The problem of keeping caches in multiprocessor systems coherent is significantly more complex than uniprocessor systems. This is because there are potentially multiple copies of data items and multiple agents modifying these copies. Consider a simple scenario illustrated in Figure 1.16. Two processors P_0 and P_1 are connected over a shared bus to a globally accessible memory. Both processors load the same variable. There are now three copies of the variable. The coherence mechanism must now ensure that all operations performed on these copies are serializable (i.e. there exists some serial order which corresponds to the parallel schedule). When a processor changes the value of its copy of the variable, one of two things must happen: the other copies must be invalidated; or, the other copies must be updated. Failing this, other processors may potentially work with incorrect values of the variable. These two protocols are referred to as *invalidation* and *update* protocols and are illustrated in Figure 1.16(a) and (b).

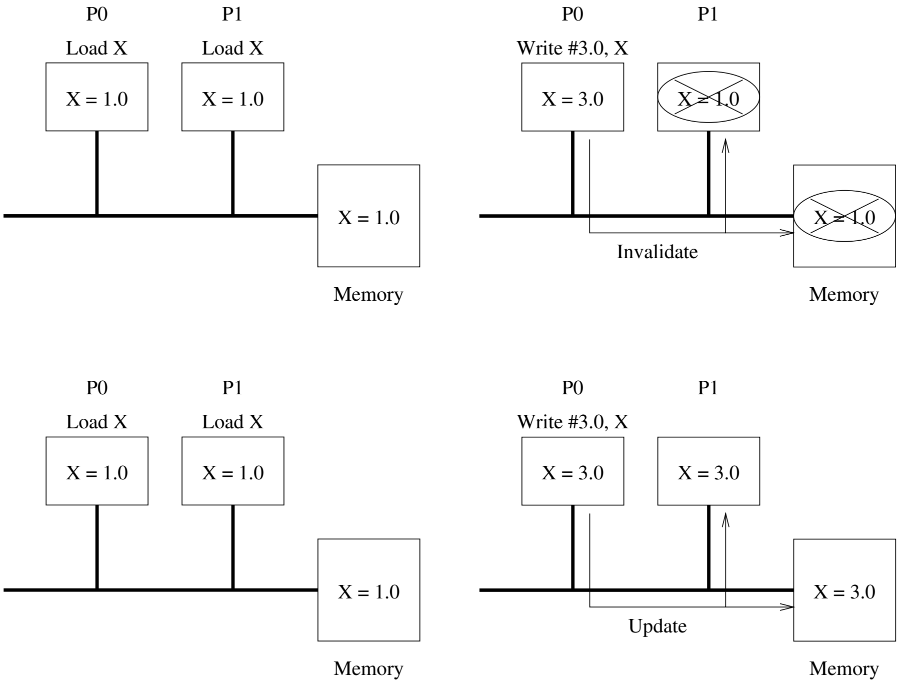


Figure 1.16 Cache coherence in multiprocessor systems: (a) Invalidate protocol; (b) Update protocol for shared variables.

Performance of Update and Invalidate Protocols Let us briefly look at the advantages and disadvantages of these two protocols. Update protocols may require significantly higher bandwidth from the interconnection network. This is because of a variety of reasons:

- A processor may access a data item, use it and then the data item may just reside in the cache although the processor does not need it anymore. An update protocol will update all such copies of the data. On the other hand, an invalidate protocol will invalidate this data on the first update and then subsequent updates need not be performed on this copy.

- Consider a case in which the shared variable X in Figure 1.16 is repeatedly incremented by processor P_0. In an invalidate protocol, all other copies are invalidated, and subsequent increment operations can be performed locally. On the other hand, in an update protocol, every increment leads to an update. Clearly, in such circumstances invalidate protocols perform better. However, if processor P_1 needs the updated value of variable X soon after P_0 writes it, then the update protocol is superior. This is because the invalidate operation is essentially wasted.

- Updates are performed on single data items and not entire cache-lines. Typically the memory system does not handle these very well.

Another important factor affecting the performance of these protocols is *false sharing*. False sharing refers to the situation in which different processors update different parts of of the same cache-line. Thus, although the updates are not performed on shared variables, the system does not detect this. In an invalidate protocol, when a processor updates its part of the cache-line, the other copies of this line are invalidated. When other processors try to update their parts of the cache-line, the line must actually be fetches from the remote processor. It is easy to see that false-sharing can cause a cache-line to be ping-ponged between various processors. In an update protocol, the situation is better since all reads can be performed locally, and the writes must be written-through.

The tradeoffs between invalidation and update schemes is the classic tradeoff between communication overheads (updates) and idling (stalling in invalidates). Hybrid schemes based on these two have also been proposed. However, none of these schemes have been conclusively shown to be better than the others. The rest of our discussion of multiprocessor cache systems assumes invalidate protocols.

Snoopy Cache Systems

Snoopy cache systems are typically associated with multiprocessor systems based on broadcast interconnection networks such as a bus or a ring. In such systems, all processors snoop on (monitor) the bus for transactions. This allows the processor to make state transitions for its cache-blocks. Figure 1.17 illustrates a typical snoopy bus based system. Each processor's cache has a set of tag bits associated with it that determine the state of the cache blocks. These tags are updated according to the state diagram associated with the coherence protocol. Often, systems maintain two sets of duplicate tags: the snoop tags and the local tags. The snoop tags are updated by the snoop hardware based on bus transactions. The local tags are updated by local operations performed by the processor. Having two sets of tags reduces contention on tag memory on updates.

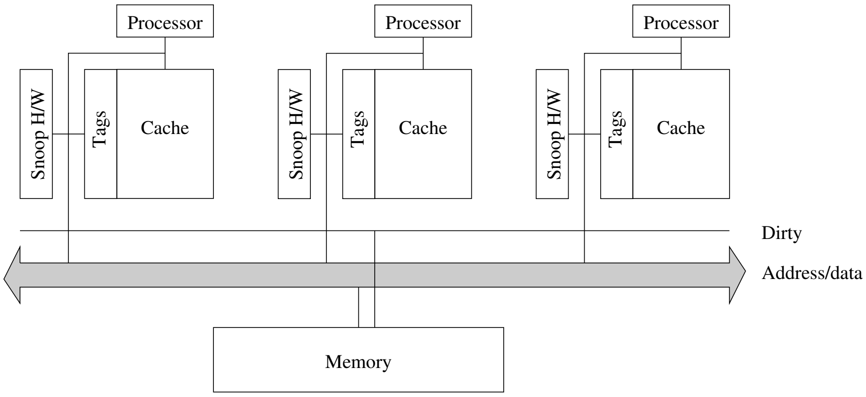


Figure 1.17 A simple snoopy bus based cache coherence system.

Memory States in Snoopy Protocols Let us revisit the example in Figure 1.16. Initially the variable X resides in the global memory. The first step executed by both processors is a load operation on this variable. At this point, the state of the variable is said to be *shared*, since it is shared by multiple processors. When processor P_0 executes a store on this variable, it marks all other copies of this variable as *invalid*. It must also mark its own copy as modified or *dirty*. This is done to ensure that all subsequence accesses to this variable at other processors will be serviced by processor P_0 and not from the memory. Let us say after this processor P_1 executes another load operation on X. Processor P_1 attempts to fetch this variable and since this was marked dirty by processor P_0, it services the request. This is typically done as follows: the snoop hardware on processor P_0 detects a read on the cache line and determines that the line is dirty. It asserts the dirty line that enables it to take control of the bus. Once this has been done the request can be serviced from processor P_0's cache and not from the main memory. The copies of this variable at processor P_1 and the global memory are updated and the variable re-enters the shared state. Thus, in the simplest model, there are three states *shared*, *invalid*, and *dirty* that a cache line goes through.

The complete state diagram of a simple three state protocol is illustrated in Figure 1.18. The solid lines depict processor actions and the dashed lines coherence transactions. For example, when a processor executes a read on an invalid block, the block is fetched and a transition is made from invalid to shared. Similarly, if a processor does a write on a shared block, the coherence protocol propagates a C_write (a coherence write) on the block. This triggers a transition from shared to invalid at all the other blocks.

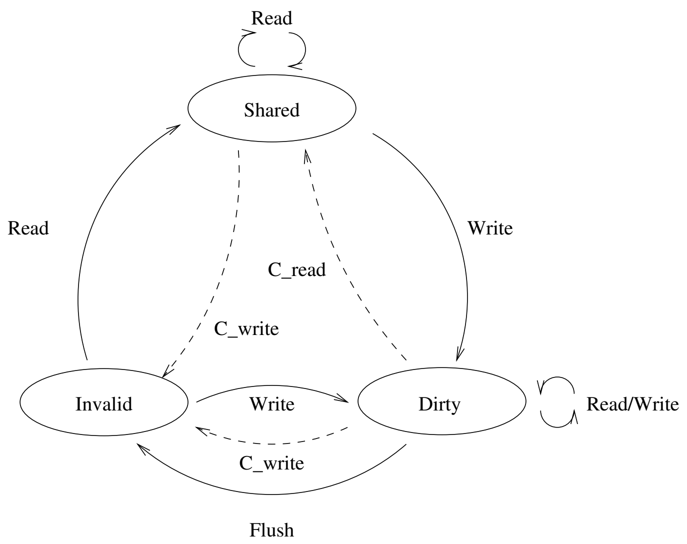


Figure 1.18 State diagram of a simple three state coherence protocol.

There are other coherence protocols that may allow a larger number of states. For instance, the *Goodman Write-Once* protocol proposed in the early eighties has four states: *invalid*, *valid* (identical to the shared state in the simple model), *reserved* (cache-block has been written once; block is not shared with any other cache, but is consistent with memory), and *dirty* (block is not shared with any other cache, and is not consistent with memory). A read-miss or an invalid block transitions to a valid block on a processor read. A block goes from valid to reserved on the first write. This write is performed as a write-through (i.e. the memory is updated on the write). A reserved block becomes dirty on a write. All subsequent reads are serviced by the processor holding the dirty block.

A number of other protocols such as the *Berkeley protocol*, *Synapse protocol*, and *Firefly protocol* have been proposed and tested.

Drawbacks of Snoopy Caches Snoopy protocols have been extensively studied and used in commercial systems. This is largely because of their simplicity and the fact that existing bus bases can be upgraded to accommodate snoopy protocols. However, snoopy protocols do not scale well to larger number of processors. First, snoopy protocols are intimately tied to multicomputers based on broadcast networks such as buses. This is because all processors must snoop all the messages. The traffic corresponding to this grows as the square of the number of processors. This is difficult to realize in real architectures. Therefore, snoopy cache protocols are restricted to 32 processors or less.

Directory Based Systems

Directory-based coherence protocols improve the scalability of snoopy protocols. Consider a simple system in which the global memory is augmented with a directory that maintains a bitmap representing cache-blocks and the processors at which they are cached. These bitmap entries are sometimes referred to as the *presence bits*. Let us examine the performance of such a system in detail. As before, we assume a three-state protocol with the states labeled *invalid*, *valid*, and *shared*. The key to the performance of directory based schemes is the simple observation that only processors that hold a particular block (or are reading it) participate in the state transitions due to coherence operations. Note that there may be other state transitions triggered by processor read / write / flush but these transitions can be handled locally with the operation reflected in the presence bits and state in the directory.

Revisiting the code segment in Figure 1.17, when processors P_0 and $P-1$ access the block corresponding to variable x, the state of the block is changed to shared, and the presence bits updated to indicate that processors P_0 and P_1 have it. When P_0 executes a store on the variable, the state in the directory is changed to dirty and the presence bits of P_1 are reset. All subsequent operations on this variable performed at processor P_0 can proceed locally. If another processor reads the value, the directory notices the dirty tag and uses the presence bits to direct the request to the appropriate processor. Processor P_0 updates the block in the memory, and sends it to the requesting processor. The presence bits are modified to reflect this and the state transitions to shared.

Even though directory based caches do not require broadcasts, they suffer from drawbacks. The directory scheme with a centralized directory is a source of contention from various processors that might be trying to update the directory simultaneously. The amount of memory required to store the directory may itself become a bottleneck as the number of processors increases. Furthermore, the directory scheme does require communications for maintaining state and presence information. All of these can limit the applicability of these schemes.

Distributed Directory Schemes In the previous discussion, we assumed a centralized location that holds all the memory blocks and their corresponding directory entries. Such architectures do not scale well because of the required bandwidth from such a memory. In scalable architectures, the memory is physically distributed across processors. The corresponding presence bits of the blocks are also distributed. The architecture of such a system is illustrated in Figure 1.19(a). A number of processing elements are connected via a low-latency interconnection network. The total memory available is partitioned into p equal chunks, each of which is physically connected to a processor. When a processor reads a block of memory from the remote processor, a request is sent to the processor. The presence bits and the state of the memory block at the host processor are updated and the block is sent to the requesting processor. When a processor performs a write, invalidations are propagated to all other copies of the system. The cached block is itself marked dirty. All subsequent operations on the block can be performed locally. On a subsequent read

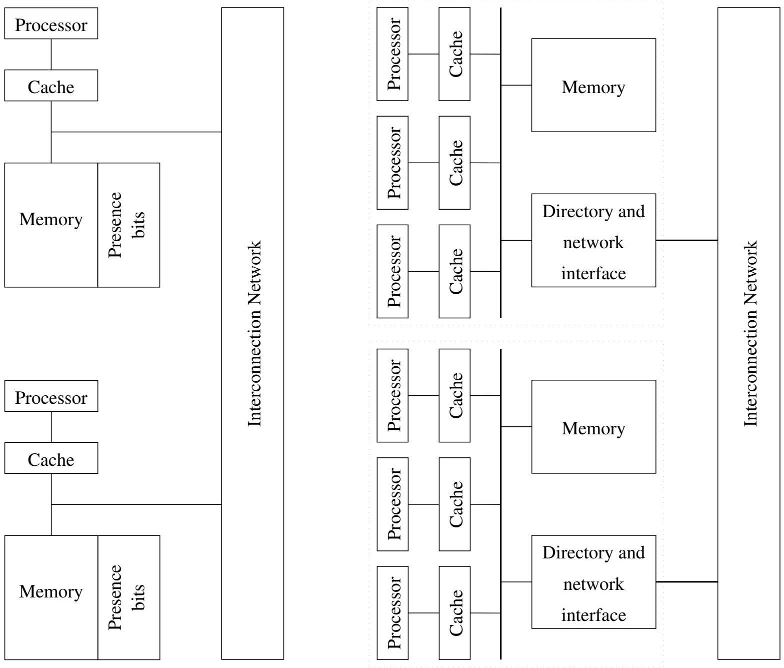


Figure 1.19 Architecture of typical distributed directories: (a) distributed directory; (b) Hierarchical directory.

from another processor, the host processor recognizes the dirty block and determines the destination by looking up the presence bits. The read request is serviced by the processor containing the dirty block. The host processor's block is also updated. This is the strongest model of cache consistency and is also referred to as *sequential consistency*. Weaker models of consistency have also been proposed and used extensively in shared address space machines.

Hierarchical Directory Schemes Although the simple distributed directory protocol described above scales to larger number of processors, the memory required for storing the directory increases quadratically with number of processors. This becomes a limiting factor for larger number of processors. Consider an alternate structure illustrated in Figure 1.19(b). The architecture consists of a set of tightly coupled clusters connected via an interconnection network. Each cluster has a single directory whose presence bits point to clusters as opposed to processors. In the example illustrated, the clusters themselves operate using snoopy protocols to keep caches coherent within clusters. When a processor reads a memory block across the cluster, the status of the block is marked as shared and the presence bits updated to indicate that the block is shared by (some processor at) specified cluster. When a write is performed, the presence bits indicate the clusters to which the invalidation must be propagated. Such directory structures are also referred to as *hierarchical directory structures*. Indeed, alternative hierarchical directory structures have been proposed and built. It is easy to see that the memory requirements of directories is significantly reduced using such schemes.

1.7 Communication costs in parallel machines

One of the major overheads in the execution of parallel programs arises from communication of information between processors. The cost of communication is dependent on a variety of features that include the network topology, handling and routing, and software protocols. Handling messages refers to how messages are passed on from one node to the next. Routing refers to the process of selecting a path for a message with a view to minimizing routing time. A variety of software protocols exist for different kinds of message transfers. These protocols have a critical impact on the associated communication time. All of these issues form the focus of our discussion in this section.

1.7.1 Message Handling in Parallel Computers

Various mechanisms for handling message transfers between processors have been used. These include store-and-forward routing, packet routing, and cut-through routing. In developing an analytical model for these message handling mechanisms, we first look at the components of communication overheads. The principal parameters that determine the communication latency are as follows:

1. *Startup time* (t_s): The startup time is the time required to handle a message at the sending processor. This includes the time to prepare the message (adding header, trailer, and error correction information), the time to execute the routing algorithm, and the time to establish an interface between the local processor and the router. This delay is incurred only once for a single message transfer.

2. *Per-hop time* (t_h): After a message leaves a processor, it takes a finite amount of time to reach the next processor in its path. The time taken by the header of a message to travel between two directly-connected processors in the network is called the per-hop time. It is also known as *node latency*. The per-hop time is directly related to the latency within the routing switch for determining which output buffer or channel the message should be forwarded to.

3. *Per-word transfer time* (t_w): If the channel bandwidth is r words per second, then each word takes time $t_w = 1/r$ to traverse the link. This time is called the per-word transfer time.

We now discuss two switching techniques that are frequently used in parallel computers. The routing techniques that use them are called store-and-forward routing and cut-through routing.

1.7.2 Store-and-Forward Routing

In store-and-forward routing, when a message is traversing a path with multiple links, each intermediate processor on the path forwards the message to the next processor after it has received and stored the entire message. Figure 1.20(a) shows the communication of a message through a store-and-forward network.

Suppose that a message of size m is being transmitted through such a network. Assume that it traverses l links. At each link, the message incurs a cost t_h for the header and mt_w for the rest of the message to traverse the link. Since there are l such links, the total time is $(t_h + mt_w)l$. Therefore, for store-and-forward routing, the total communication cost for a message of size m words to traverse l communication links is

$$t_{comm} = t_s + (mt_w + t_h)l. \tag{1.2}$$

In current parallel computers, the per-hop time t_h is quite small. For most parallel algorithms, it is less than mt_w even for small values of m and thus can be ignored. For parallel algorithms using store-and-forward routing discussed in this book, we simplify the time given by Equation 1.2 to

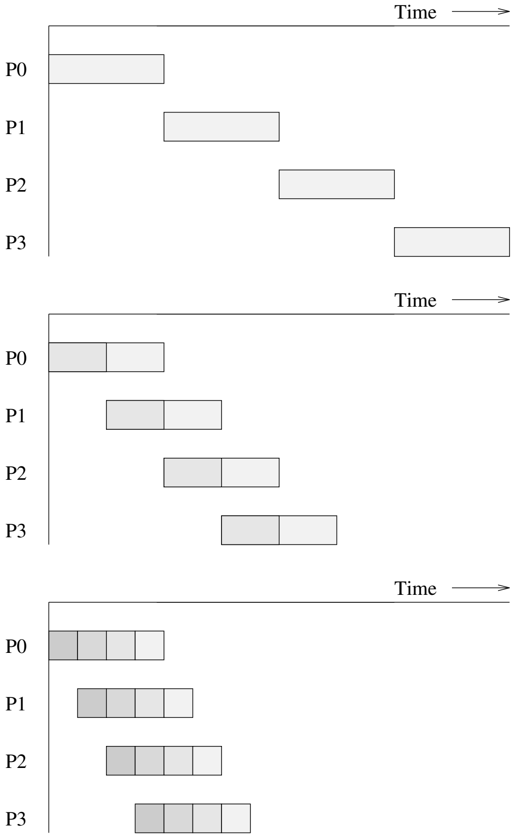
$$t_{comm} = t_s + mt_w l.$$

1.7.3 Packet Routing

Store-and-forward routing makes poor use of communication resources. A message is sent from one processor to the next only after the entire message has been received (Figure 1.20(a)). Consider the scenario shown in Figure 1.20(b), in which the original message is broken into two equally sized parts before it is sent. In this case, an intermediate processor waits for only half of the original message to arrive before passing it on. The increased utilization of communication resources and reduced communication time is apparent from Figure 1.20(b). Figure 1.20(c) goes a step further and breaks the message into four parts.

This illustrates the principle of packet switching. The original message is broken into packets and transferred over the network. Each packet contains routing, sequencing, and error checking information. Packets may be received in any particular order. Therefore, they must contain sequencing information. Since different packets do not necessarily traverse the same path, they much contain routing information.

Consider the transfer of a m word message through the network. The time taken for programming network interfaces, computing routing information etc. takes a time that is independent of the message length. This is aggregated into the startup time t_s of the message transfer. The message is broken into packets and packets are assembled with their error, routing, and sequencing fields. This takes a time proportional to the length of the message. We denote this time by $t_{w1}m$. If the first packet traverses l links, then this packet arrives at the destination processor after a time of $t_h l$. Here, t_h is the link delay. Let us assume that every other packet traverses the same number of links (this is merely a simplification of the actual scenario; in real situations, this is determined by the routing



(a) A single message sent over a store-and-forward network

(b) The same message broken into two parts and sent over the network.

(c) The same message broken into four parts and sent over the network.

Figure 1.20 Passing a message from processor P_0 to P_4 (a) through a store-and-forward communication network; (b) and (c) extending the concept to cut-through routing. The shaded regions represent the time that the message is in transit. The startup time associated with this message transfer is assumed to be zero.

algorithm). If the packet size is given by $r + s$ where r is the original message and s is the additional information, and the network has an effective bandwidth of t_{w2}, then the destination will receive an additional packet every $t_{w2}(r + s)$ seconds. Since there are m/r packets, the total communication time is given by:

$$
\begin{aligned}
t_{comm} &= t_s + t_{w1}m + t_h l + \frac{m}{r} t_{w2}(r + s) \\
&= t_s + t_{w1}m + t_h l + t_{w2}m + t_{w2}\frac{s}{r}m \\
&= t_s + t_h l + t_w m
\end{aligned}
$$

where

$$t_w = t_{w1} + t_{w2}(1 + \frac{s}{r})$$

Packet routing is suited to networks with a highly dynamic environment and higher error rates. These are typical of LANs and WANs.

1.7.4 Cut-Through Routing

In tightly coupled parallel platforms, the restrictions on message transfers can be tightened to reduce the overheads associated with packet switching. First, by forcing in-sequence delivery, sequencing information can be eliminated. By associating error information at message level rather than packet level, the overhead associated with error detection and correstion can be reduced. Furthermore, since error rates in interconnection networks for parallel machines are extremely low, lean error detection mechanisms can be used instead of expensive error correction schemes. Finally, by forcing all packets to take the same path, we can eliminate the overhead of transmitting routing information with each packet.

This routing scheme is called the cut-through routing scheme. The message is broken into fixed size units called flow control digits or flits. Since flits do not contain the overheads of packets, they can be much smaller than packets. A tracer is first sent from the source to the destination node to establish a connection. Once a connection has been established, the flits are sent one after the other. All flits follow the same path in a dovetailed fashion. An intermediate processor does not wait for the entire message to arrive before forwarding it. As soon as a flit is received at an intermediate processor, the flit is passed on to the next processor. Unlike store-and-forward routing, it is no longer necessary to have buffer space at each intermediate processor to store the entire message. Therefore, wormhole routing uses less memory and memory bandwidth at intermediate processors and is faster.

While traversing the network, if a message needs to use a link that is currently in use, then the message is blocked. This may lead to deadlock. Figure 1.21 illustrates deadlock in a wormhole routing network. The destinations of messages 0, 1, 2, and 3 are A, B, C, and D, respectively. A flit from message 0 occupies the link CB (and the associated buffers). However, since link BA is occupied by a flit from message 3, the flit from message 0 is blocked. Similarly, the flit from message 3 is blocked since link AD is in use. We can see that no messages can progress in the network and the network is deadlocked. Deadlocks can be avoided in wormhole networks by using appropriate routing techniques. Both E-cube and XY-routing algorithms are deadlock-free routing schemes.

Consider a message that is traversing such a network. If the message traverses l links, and t_h is the per-hop time, then the header of the message takes lt_h time to reach the destination. If the message is m words long, then the entire message will arrive in time mt_w after the arrival of the header of the message. Therefore, the total communication time for cut-through routing is given by

$$t_{comm} = t_s + lt_h + mt_w \tag{1.3}$$

This time is an improvement over store-and-forward routing. Disregarding the startup time, the cost to send a message of size m over l hops is only $\Theta(m + l)$ for cut-through

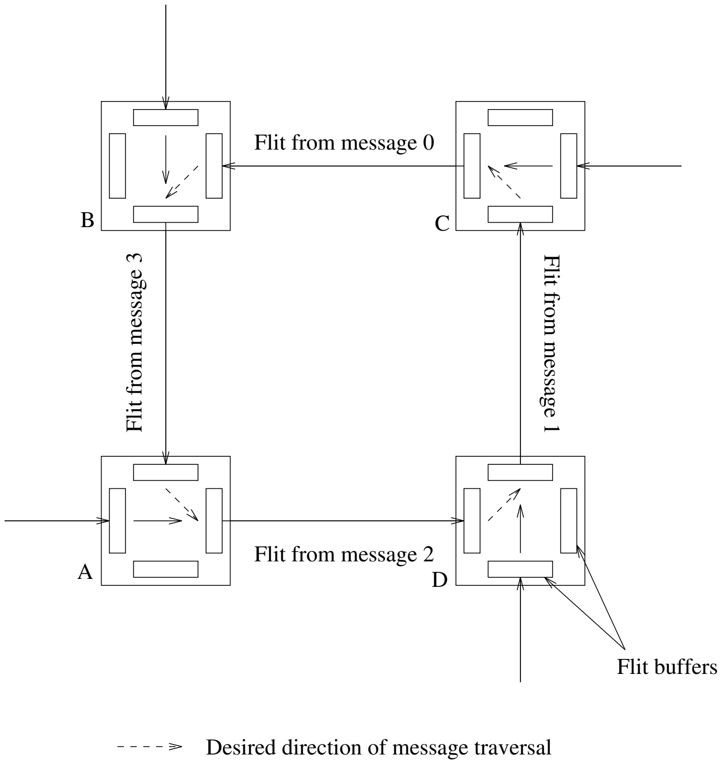


Figure 1.21 An example of deadlock in a wormhole-routing network.

routing, whereas in store-and-forward routing, it is $\Theta(ml)$. Note that if the communication is between nearest neighbors (that is, $l = 1$), or if the message size is small, then the communication time is similar for store-and-forward and cut-through routing schemes.

Most current parallel computers and many networks support wormhole routing. The size of a flit is determined by a variety of network parameters. The control circuitry must operate at the flit rate. Therefore, if we select a very small flit size, for a give link bandwidth, the flit rate becomes large. Therefore, the control circuitry must operate at a very high speed. This poses considerable challenges for designing routers. On the other hand, as flit sizes become large, internal buffer sizes increase, so does the latency of message transfer. Both of these are undesirable. Flit sizes in recent wormhole-based interconnection networks range from four bits (CM-5), one byte in the J-Machine, two bytes in Cray T3D and Paragon, eight bytes in the Cray T3E, sixteen bytes in the SGI Spider chip, and thirty two bytes in the RCube router. In many parallel programming paradigms relying predominantly on short messages (such as cache lines), the latency of messages is critical. For these, it is unreasonable for a long message passing on a link to hold up a short message. Such scenarios are addressed in routers using multilane wormhole routing. In multilane wormhole routing, a single physical channel is split into a number of virtual channels.

Messaging constants t_s, t_w, and t_h are determined by hardware characteristics, software layers, and programming paradigms. Certain paradigms such as message passing are best served by variable length messages, others by fixed length short messages. While effective bandwidth may be critical for the former, reducing latency is more important for the latter. Messaging layers for these paradigms are tuned to reflect these requirements.

The convergence of routers traditionally used in parallel computers and those in LANs have very interesting implications for both areas. While issues of low latency and high throughput are critical in parallel computing, LAN switches must address additional problems associated with long links. However, with widespread availability of ATMs and Myrinets, network of workstations are becoming an increasingly attractive platform for parallel programming.

1.8 Communication Costs in Non-Cache Coherent Shared Address Machines

Cache-coherent shared address space computers involve significant hardware and performance overhead for implementing the directories and snoopy caches. Furthermore, the availability of low-latency messaging support for short (cache line) messages is critical to their performance. A weaker model for parallel computers provides a global addressing mechanism. However, each processor's memory is entirely private to it. There is no memory that is physically or logically shared between processors. The global memory map makes it possible to access variables at remote processors directly. An example of such a machine is the Cray T3D programmed using the Shmem library.

As we shall subsequently see, non-cache coherent shared address space machines are a good compromise between cache coherent SMPs and message passing architectures in terms of price and performance. However, programming these machines requires considerable effort to maintain coherence of multiple copies of a data item that may be present. We illustrate this using a very simple example. Assume that there are 32 processors, each with a single integer. We are trying to write a code segment that adds each processor's local integer to a variable named `sum` at processor P_0. The program segment is as follows:

```
get (temp, sum, 0);
temp += my_integer;
put (temp, sum, 0);
```

In this segment, we use a simplified syntax for get and put. The get statement gets the value of `sum` from processor P_0 and stores it into a local variable `temp`. The put statement puts it back after the value of `my_integer` has been added to it. As the reader might have guessed, this program segment cannot be expected to yield the right answer (all the time). This is because multiple processors may be getting and putting the value of sum at the same time. Therefore, the programmer must ensure that the variable `sum` at processor P_0 is locked when it has been obtained by another processor. This is done in the following code sample:

```
            lock (sum);
            get (temp, sum, 0);
            temp += my_integer;
            put (temp, sum, 0);
            unlock (sum);
```

This example also illustrates the difference between non-cache coherent and cache coherent machines. In non-cache coherent machines, all memory is private. However, all variables declared statically have a memory map available at other processors. This lets the processors put and get these variables directly. These primitives are also referred to as one-way communication primitives, since they involve only the processors putting or getting data. In contrast, cache-coherent parallel computers allow programmers to declare global and local variables. The same code written for a cache-coherent machine would declare `sum` as a global variable and `my_integer` as local. The code segment for the cache-coherent machine would be:

```
            lock (sum);
            sum += my_integer;
            unlock (sum);
```

Notice that the put and get operations are implicitly handled by the coherence hardware but have to be explicitly programmed in non-cache coherent machines.

One-way communication primitives require additional hardware for detecting memory inconsistencies. Consider a case in which processor P_1 has a variable `global_count`. At a certain stage in the computation, this variable is loaded into the cache of processor P_1. While this variable is in the cache, it is overwritten (via a put) by processor P_0. The system must be able to detect this and invalidate or update the cache line.

While it may appear that the cost model for one-way communication is similar to the message passing model, there are certain critical differences. Unlike message passing, one way protocols do not buffer data. Therefore, a get executed on a remote piece of data establishes hand-shaking, programs routers and switches and performs the transfer. Similarly, a put operation simply puts the data into a remote location without copying it into intermediate buffers. This implies that the per-word cost of one-way communication is typically less than that of buffered message passing. The analytical model of Equation 1.3 is still valid for these promitives.

1.9 Case Studies

We now study high-level architectures of some of the current parallel computing platforms.

The Cray T3E Network

The Cray T3E employs a three-dimensional torus topology with a single processor per node (the Cray T3D uses a similar topology with two processors per node and different link characteristics). Each node contains a router which has a processor interface and six full-duplex links (one for each direction of the cube). In addition, the router also supports an I/O port and a central arbiter. The router operates at 75 MHz, delivering a single flit every clock cycle. A flit consists of 5 14 bit phits that together carry 64 bits of data and 6 bits of additional information. This corresponds to a total link bandwidth of 75 * 8 = 600 MB/s. This raw bandwidth reduces to about 500 MB/s after the protocol overheads. In contrast, the T3D switches at 150 MHz carrying 16 bits per clock cycle. This corresponds to a peak link bandwidth of 300 MB/s. The effective congested per processor bandwidth is about 75 MB/s because of protocol overheads and the fact that two processors share the link.

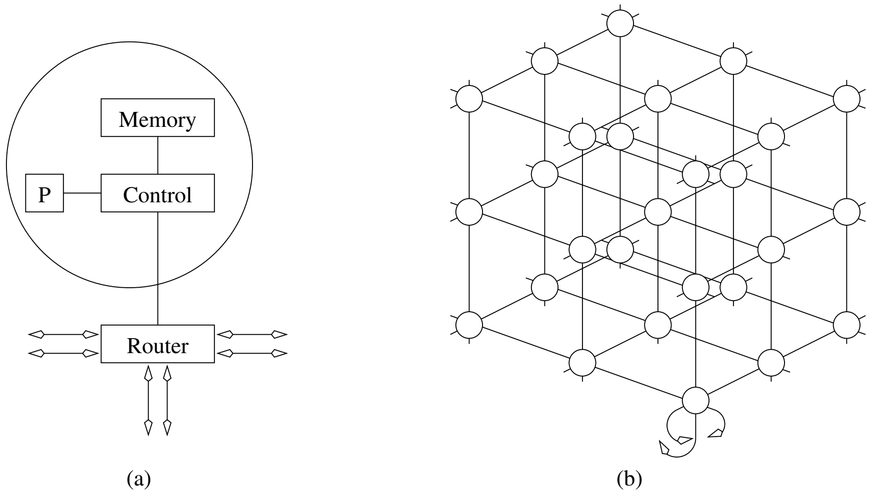


Figure 1.22 Interconnection network of the Cray T3E: (a) node architecture; (b) network topology.

Messages in the T3D are routed using the e-cube or dimension ordering. The T3E relies on a direction-ordered routing as opposed to dimension-ordered routing. Whereas dimension-ordered routing passes the message in the X dimension followed by the Y and Z dimensions, direction-ordered routing passes messages along six directions: X, Y, Z, -X, -Y, -Z. This adds a measure of fault tolerance to the routing.

While the T3E interconnect aims to provide low latency, the primary focus of network design is on supporting sustained bandwidths across a range of applications. The pin-to-pin latency from the router to the next router varies between 40 ns (passthrough) to 80 ns (turn). The effective link bandwidth approaches to within 95% of the peak rated link bandwidth. The cross-section bandwidth of the network varies with the aspect ratio of the cube; which in turn is a function of the number of processors. The application level latency and bandwidth are functions of the underlying communication library. Cray's Shmem library yields a letency of 2.4 and 1.5 μs on shmem_put and shmem_get respectively. The corresponding bandwidth figures for an uncongested network were observed to be 320 MB/s. Using message passing interface (MPI) the latency is 18 μs and the bandwidth varies between 140 MB/s for a 16KB message and 305 MB/s for a 2MB message (the numbers are for Cray's version of MPI).

An important operation in many parallel programs is the barrier (synchronization) operation. Many machines provide additional hardware resources for fast barrier implementation. This includes the CM5 and the Cray T3D. While this results in fast barriers, the associated hardware cost is high. The Cray T3E takes an alternate approach by providing a set of barrier registers. A processor writes to its barrier register when it wants to participate in a barrier. A virtual Eureka network is embedded into the torus based on the participating processors. The barrier is accomplished over this virtual Eureka network using high priority packets. These packets are typically handled by interrupt or polling of local registers. In this way, the T3E provides fast barriers without incurring the extra cost of a dedicated network. In real applications, it is possible to observe barrier times ranging from 3 - 7 μs. This number is largely independent of the number of processors.

The SGI Origin 2000

The key component of the Origin 2000 interconnection network is the CrayLink router. The router is used to provide connectivity between nodes. Each node in the Origin consists of two R10000 processors, the memory and a crossbow switch for graphics and I/O support. Two such nodes can be connected to each router. The remaining router ports can be used to provide connectivity to other such routers. Figure 1.23 illustrates topologies of machines of varying sizes. Four nodes can be connected using two routes; eight nodes can be connected in a square topology; and so on. For 32 nodes or less, routers have free ports that can be used to provide dense connectivity using Xpress links. It is easy to see that we can construct 4-D hypercubes of routers while reserving two ports for nodes. This corresponds to 64 processors. Configurations larger than this have to resort to additional levels of routers. These configurations are constructed using aggregates of 8 routers (16 nodes). This leaves one free port at each router that can be used to provide connectivity to other such aggregates using a meta-router. For a fully configured 1024 processor (512) nodes, we have 32 aggregates. The connectivity among these is provided by a network of 32 routers connected in a 5-D hypercube. In this manner, the Origin interconnect supports various levels of connectivity.

The router provides connectivity between six pairs of links; each operating at 1.56 GB/s. For a 8 node configuration, this corresponds to a cross-section bandwidth of 3.12 GB/s and a per-processor cross-section bandwidth of 390 MB/s. Flexible topologies are supported using a table-lookup based routing. Thus, by appropriately initializing these tables, it is possible to provide a wide range of topologies and fault tolerance. The pip-to-pin latency of the network is approximately 41 ns. However, the actual latency of reads and writes depends on a variety of factors. For instance, if the data item being read is local to a processor and not cached elsewhere, the latency is approximately 360 ns. This increases to approximately 1100 ns for data that is cached (and dirty) elsewhere. The actual time varies between 1100 and 1200 ns depending on the distance between the nodes. The latency-bandwidth numbers of the message passing model depend on the message passing library and release.

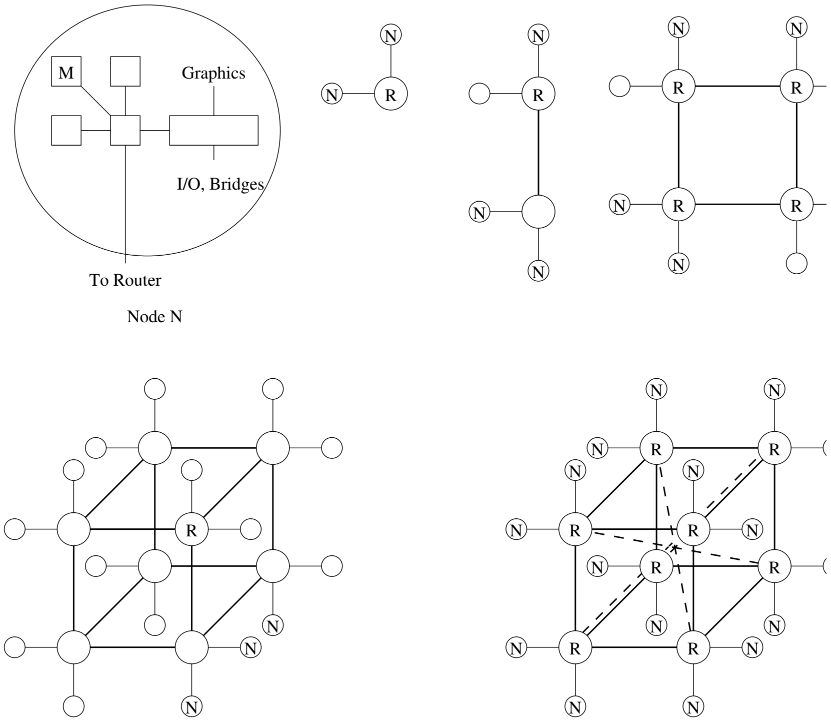


Figure 1.23 Interconnection topologies for the Origin 2000 with various numbers of processors.

The Sun Gigaplane XB

The Gigaplane XB interconnect provides connectivity between up to 16 system boards with 4 processors each for a total of 64 processors. The two level interconnect provides intra- and inter-board connectivity using different interconnection networks. The intra-board connectivity is provided by a fast bus. The data bus is 36 byte wide error correcting bus capable of delivering a 64 byte data block every other clock cycle. The bus operates at 83.3 MHz giving it a total bandwidth of 2.67GB/s. This intra-board network is used for providing a coherent shared address space across the processors. Coherence is based on a snoopy protocol. The address bus is snooped every other cycle (since data is transmitted in 64 bit blocks every two cycles) for a total of 41.7 million snoops per second. This interconnect is used in Sun servers such as the Ultra Enterprise 3000-6000.

The HPC Enterprise 10000 (Starfire) systems provide a scalable connectivity among 16 processor boards. Since each board has 4 processors, ideally, the bus bandwidth must be raised by a factor of 4. However the cache line on these systems is only 64 bytes long. Since the bus is already 36 bytes wide, it is not possible to increase the data bandwidth by a factor of 4 by increasing the width of the databus. The Gigaplane XB uses a crossbar interconnect to provide inter-board connectivity. Each link of the crossbar is 18 bytes wide

(16 + 2). The links operate at 83.3 MHz. For a 16×16 crossbar, this corresponds to a total data rate of 21.3 GB/s. The interconnect also uses four interleaved buses that snoop every other cycle. At 83.3 MHz, this corresponds to a snooping rate of 166.6 million per second. Since the cache line size is 64 bytes, the peak snooping rate (in terms of data) is given by 166.6×64 MB/s or 10.66 GB/s. This is 50data transfer rate sustainable by the crossbar. Thus, when larger configurations (i.e. 16 boards) are connected on the crossbar, the data transfer is limited by the address bus, and on smaller configurations, it is limited by the crossbar. It is easy to see that the balance between address and data rates is achieved at 13 boards.

From a programmer's perspective, the critical factors are the latency, bandwidth and the scaling with number of processors and traffic load. The Gigaplane XB provides an average latency of 550 ns on cache miss across the switch. This is higher than the 270 ns latency for a single processor and 300 ns for the bus-based interconnect. The additional latency results from the glue logic and the need for extra wiring. The crossbar also provides linearly scaling bandwidth with number of processors. It is experimentally shown to scale up to almost 10 GB/s. This corresponds to about 97% of the peak bandwidth of the switch. The effective per-processor switch bandwidth is 156 MB/s.

The Gigaplane XB switch is used in the Ultra Enterprise 10000 series of parallel processors. These find extensive application in database transaction processing as well as scientific computing.

Cluster Computing Platforms

A number of off-the-chelf LAN switches can be used to provide fast connectivity between computers, with a view to facilitating a parallel platform.

Myrinet Myrinet is a high speed switch-based local area network (LAN) providing low-latency, high bandwidth connectivity between workstations. The network consists of host computers and multiple-port switching nodes. The switching nodes can be connected to other switching nodes or to the single-port host interface via full duplex links. This is illustrated in Figure 1.24. Links are capable of supporting communication at the rate of 640 mb/s in each direction for an aggregate link bandwidth of 1.28 gb/s.

Myrinet supports flexible network topologies; however, large networks are typically constructed by connecting network elements in a regular topology. This facilitates fast message routing through the network. This also makes Myrinet more suitable than shared-media networks such as Ethernet and FDDI that can support fixed bandwidth, irrespective of the number of processors. By connecting switches appropriately, it is possible to construct Myrinets that can deliver scalable performance.

Individual switches in the Myrinet are non-blocking pipelined crossbar switches that support message transfer using wormhole routing. This helps in reducing latency and provides multiplicity of data paths through the network. The switch latency (message-in to message-out time) is about 550 ns. Currently, switches with up to 16 ports are available. In

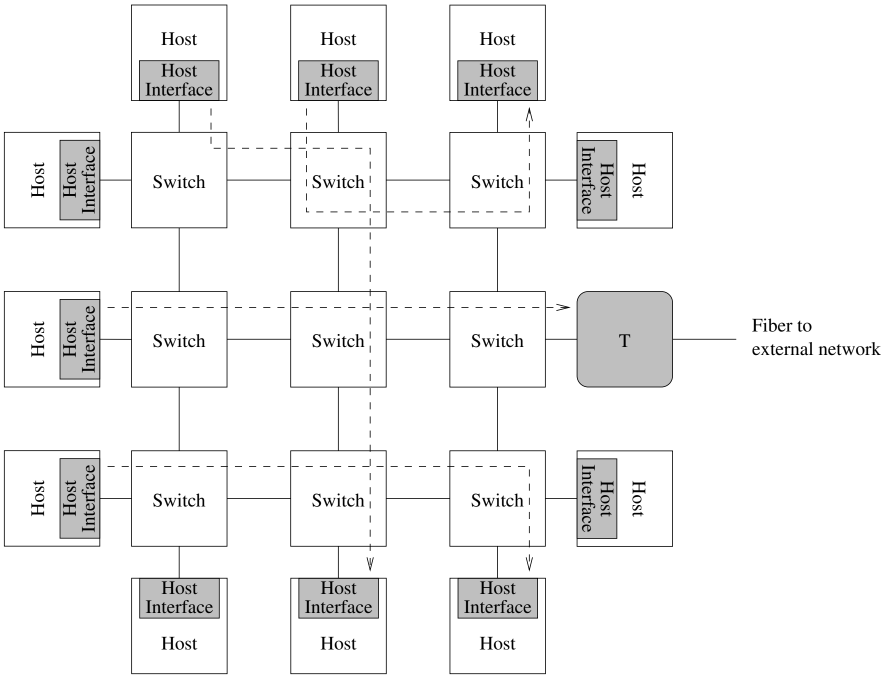


Figure 1.24 A typical connection pattern for switches and hosts in a Myrinet. The figure also illustrates routing of messages between hosts. At any point of time, multiple pairs of processors with non-conflicting paths may be communicating with each other.

the event of a conflict at a switch, one of the messages is blocked. Message transfer through the network is typically dimension ordered to avoid deadlocks. Figure 1.24 illustrates message transfer through a mesh-structured Myrinet. Messages traverse in the x direction first and then in the y direction. The head of the message contains routing information. As the message arrives at an intermediate switch, the switch determines the next switch / host in the path.

Asynchronous Transfer Mode (ATM) Networks Another switched network that is gaining widespread popularity is the ATM network. Unlike the Myrinet which had its origins in massively parallel computers, ATMs have their origins in multimedia networks intended for providing integrated services for voice, video and data (Broad-band Integrated Services Digital Networks or BISDN). This gives ATMs the advantage of an enormous customer base, standardization, and development effort. Due to these reasons, it is highly likely that ATMs will play a major role in providing cluster platforms for parallel computing.

Each host workstation has an ATM adapter card that interfaces to a multi-ported ATM switch. A switch can be connected to other hosts and switches. ATM is based on fast packet switching of small data units called cells. Each cell is 53 bytes long: a 5 byte header for routing and control, and 48 bytes data. Data transfer through an ATM is identical to a circuit switched network. The data-path, also referred to as a *virtual channel* is established prior to transmission. Once the data-path has been established, the intermediate switching elements pass all incoming cells to the specified outgoing link until such time as an alternate data-path is established. Each cell has a 16 bit virtual channel identifier which is used by the switches to determine the route.

A variety of ATM switches are currently available on the market from various vendors. Non-blocking switches supporting up to 16 ATM ports at 155 Mb/s are commonplace (Cisco/NEC HyperSwitch, Bay Networks). The state of the art ATM switch developed at IBM Zurich is capable of supporting 16 ports at 300 - 400 Mb/s for an aggregate switch bandwidth of 6.4 Gb/s. The entire IBM switch is available on a single chip. Many of the ATM switches also support multicast and broadcast operations. These are very useful for implementing collective communication operations when using ATM networks are parallel platforms.

The ATM adapter cards on the host workstations have the responsibility of packaging and assembly of messages, buffering cells, and fault detection and recovery. These adapter cards are typically built around fast RISC processors such as the i960 and contain DMA engines and buffer memory. Much like the Myrinets, user processes can access the network either using a TCP/IP protocol or faster Application Program Interfaces (APIs). Conventional ATM APIs are capable of delivering significantly higher performance than the TCP/IP protocols. The issues involved are identical to those discussed in Myrinet.

FiberChannel The FiberChannel standard was conceived to provide high speed connectivity among workstations and supercomputers. Channels are particularly suited for such applications. This is because channels perform only simple error detection in hardware. More extensive error detection and correction are left to higher level protocols. Other functions such as data retransmission on encountering a busy resource are also handled by the channel as opposed to the software. Because of such high dependence on hardware, channels are faster than networks for buffer-to-buffer data transfers. To achieve this high performance, channels typically operate in a regular predetermined topology.

The basic unit of transmission is referred to as a *frame*. Frames can be data frames or link control frames. The link control frames can be acknowledgements, busy, and reject frames. A frame contains the data, destination and source addresses, and link control information. The packaging of data into frames and assembly at the receiving end is done by the Fibre Channel interface controller. A single frame contains up to 2112 bytes of data. A set of one or more frames transmitted from one host to another is called a sequence.

As is the case with other switched networks, Fibre Channel relies on an active interconnection switch referred to as a *fabric*. A Fibre Channel port is responsible for delivering data and the destination address to the fabric. The fabric is responsible for conveying the data to the receive buffer. The host workstation is connected via a node-port (N_port) to the fabric (its F_port). The fabric itself can have a flexible topology, influenced by performance and cost issues. Fibre Channel supports different data transmission rates (133 Mb/s, 266 Mb/s, 530 Mb/s, and 1 Gb/s) and transmission media (optical, electronic).

Many vendors offer Fiber Channel components. The fabric is perhaps one of the most important components. Many vendors offer fabrics with varying aggregate bandwidths. Ancor communications offers fabrics supporting 8 - 64 ports, each capable of 266 Mb/s to 1062 Mb/s (Ancor FCS266). The IBM 7319 can support 8 ports, each capable of up to 266 Mb/s and can be expanded to 16 ports.

Other Networks Many other networks have gained popularity as interconnects for parallel platforms. These include FiberChannel, HiPPI, and mode recently, Gigabit EtherNets. While each of these differ in their physical layers and the protocols, using these in a parallel platform relies on a common API based on either message passing or a shared address space. Their performance aspects vary rapidly in view of evolving hardware and software technology.

1.10 Bibliographic Remarks

Several textbooks discuss parallel architectures and interconnection networks [**?**, **?**, **?**, **?**, **?**, **?**, **?**]. The classification of parallel computers as SISD, SIMD, and MIMD was introduced by Flynn [**?**]. He also proposed the MISD (multiple instruction stream, single data stream) model. MISD is less natural than the other classes, although it can be viewed as a model for pipelining. Gordon Bell [**?**] classifies MIMD parallel computers based on the physical and logical memory organization. Ni [**?**] provides a layered classification of parallel computers based on hardware architecture, address space, communication model, language, programming environment, and applications.

Interconnection networks have been an area of active interest for decades. Feng [**?**] provides a tutorial on static and dynamic interconnection networks. The perfect shuffle interconnection pattern was introduced by Stone [**?**]. Omega networks were introduced by Lawrie [**?**]. Other multistage networks have also been proposed. These include the Flip network [**?**] and the Baseline network [**?**]. Mesh of trees and pyramidal mesh are discussed by Leighton [**?**]. Leighton [**?**] provides a detailed discussion of many such networks.

The C.mmp was an early research prototype MIMD shared-address-space parallel computer based on the Crossbar switch [**?**]. The Fujitsu VPP 500 and the Cray Y-MP are more recent examples of crossbar-based parallel computers. Several parallel computers were based on multistage interconnection networks including the BBN Butterfly [**?**], the NYU Ultracomputer [**?**], and the IBM RP-3 [**?**]. The Stanford Dash [**?**] and the KSR-1 [**?**] are NUMA shared-address-space computers.

The Cosmic Cube [**?**] was among the first message-passing parallel computer based on a hypercube-connected network. Commercially available hypercube-based computers include the nCUBE 2 [**?**], the Intel iPSC-1, iPSC-2, and iPSC/860. Saad and Shultz [**?**, **?**] derive interesting properties of the hypercube-connected network and a variety of other static networks [**?**]. Many message-passing computers are based on the mesh network. The Intel Paragon XP/S [**?**] and the Mosaic C [**?**] are two-dimensional mesh-based computers. The MIT J-Machine [**?**] is based on a three-dimensional mesh network. The performance of mesh-connected computers can be improved by augmenting the mesh network with broadcast buses [**?**]. The reconfigurable mesh architecture (Figure **??**) was introduced by Miller et al. [**?**]. Other examples of reconfigurable meshes include the TRAC and PCHIP.

The DADO parallel computer was based on a tree network [**?**]. It used a complete binary tree of depth ten. Leiserson [**?**] introduced the fat-tree interconnection network and proved several interesting characteristics of it. He showed that for a given volume of hardware, no network has much better performance than a fat tree. The Thinking Machines CM-5 [**?**] parallel computer is based on a fat tree interconnection network.

The Illiac IV [**?**] was among the first SIMD parallel computers. Other SIMD computers include the Goodyear MPP [**?**], the DAP 610, and more recently the CM-2 [**?**], MasPar MP-1, and MasPar MP-2 [**?**]. The CM-5 and DADO incorporate both SIMD and MIMD features. Both are MIMD computers but have extra hardware for fast synchronization, which enables them to operate in SIMD mode. The CM-5 has a control network to augment the data network. The control network provides such functions as broadcasting, reduction, combining, and other global operations.

Leighton [**?**] and Ranka and Sahni [**?**] discuss embedding one interconnection network into another. Gray codes, used in embedding linear array and mesh topologies, are discussed by Reingold [**?**]. Ranka and Sahni [**?**] discuss the concepts of congestion, dilation, and expansion. Our discussion of embeddings in Section **??** is influenced by Ranka and Sahni's discussion [**?**].

A comprehensive survey of cut-through routing techniques is provided by Ni and McKinley [**?**]. The wormhole routing technique was proposed by Dally and Seitz [**?**]. A related technique called *v*irtual cut-through, in which communication buffers are provided at intermediate processors, was described by Kermani and Kleinrock [**?**]. Dally and Seitz [**?**] discuss deadlock-free wormhole routing based on channel dependence graphs. Deterministic routing schemes based on dimension ordering are often used to avoid deadlocks. Cut-through routing has been used in several parallel computers including the nCUBE 2, Paragon XP/S, and J-Machine. Our discussion of wormhole routing is based on the paper by by Ni and McKinley [**?**]. The E-cube routing scheme for hypercubes was proposed by [**?**].

Dally [**?**] discusses the cost-performance tradeoffs of networks for message-passing computers. By using the bisection bandwidth of a network as a measure of the cost of the network, he showed that low-dimensional networks (such as two-dimensional meshes) are more cost-effective than high-dimensional networks (such as hypercubes) [**?**, **?**, **?**]. Kreeger and Vempaty [**?**] derive the bandwidth equalization factor for a mesh with respect to a hypercube-connected computer for all-to-all personalized communication (Section **??**). Gupta and Kumar [**?**] analyze the cost-performance tradeoffs of FFT computations on mesh and hypercube networks for two different cost criteria.

The properties of PRAMs have been studied extensively [**?**, **?**, **?**, **?**, **?**]. Books by Akl

[?], Gibbons [?], and Jaja [?] address PRAM algorithms. Our discussion of PRAM is based upon the book by Jaja [?]. A number of processor networks have been proposed to simulate PRAM models [?, ?, ?, ?, ?, ?, ?]. Mehlhorn and Vishkin [?] propose the ***m***odule parallel computer (MPC) to simulate PRAM models. The MPC is a message-passing parallel computer composed of p processors, each with a fixed amount of memory and connected by a completely-connected network. The MPC is capable of probabilistically simulating T steps of a PRAM in $T \log p$ steps if the total memory is increased by a factor of $\log p$. The main drawback of the MPC model is that a completely-connected network is difficult to construct for a large number of processors. Alt et al. [?] propose another model called the ***b***ounded-degree network (BDN). In this network, each processor is connected to a fixed number of other processors. Karlin and Upfal [?] describe an $O(T \log p)$ time probabilistic simulation of a PRAM on a BDN. Hornick and Preparata [?] propose a bipartite network that connects sets of processors and memory pools. They investigate both the message-passing MPC and BDN based on a mesh of trees.

Many modifications of the PRAM model have been proposed that attempt to bring it closer to practical parallel computers. Aggarwal, Chandra, and Snir [?, ?] propose the LPRAM (local-memory PRAM) model and the BPRAM (block PRAM) model [?]. They also introduce a hierarchical memory model of computation [?]. In this model, memory units at different levels are accessed in different times. Parallel algorithms for this model induce locality by bringing data into faster memory units before using them and returning them to the slower memory units. Other PRAM models such as phase PRAM [?], XPRAM [?], and the delay model [?] have also been proposed. Many researchers have investigated abstract universal models for parallel computers [?, ?, ?].

Problems

1.1 Consider a memory system with a level 1 cache of 32 KB and DRAM of 512MB with the processor operating at 500 MHz. The latency to L1 cache is 1 cycle and the latency to DRAM is 50 cycles. In each memory cycle, the processor fetches 4 words (cache line size is 4 words). What is the peak achievable performance of a dot product of two vectors? Note: Where necessary, assume an optimal cache placement policy.

```
/* dot product loop */
for (i = 0; i < dim; i++)
        dot_prod += a[i] * b[i];
```

1.2 Now consider the problem of multiplying a dense matrix with a vector using a two-loop dot-product formulation. The matrix is of dimension $4K \times 4K$. (Each row of the matrix takes 16KB of storage.) What is the peak achievable performance of this technique using a two-loop dot-product based matrix-vector product?

```
/* matrix-vector product loop */
for (i = 0; i < dim; i++)
        for (j = 0; i < dim; j++)
                c[i] += a[i][j] * b[j];
```

1.3 Extending this further, consider the problem of multiplying two dense matrices of dimension $4K \times 4K$. What is the peak achievable performance using a three-loop dot-product based formulation? (Assume that matrices are laid out in a row-major fashion).

```
/* matrix-matrix product loop */
for (i = 0; i < dim; i++)
        for (j = 0; i < dim; j++)
                for (k = 0; k < dim; k++)
                        c[i][j] += a[i][k] * b[k][j];
```

1.4 Restructure the matrix multiplication algorithm to achieve better cache performance.

The most obvious cause of the poor performance of matrix multiplication was the absence of spatial locality. We were wasting three of the four words fetched from memory. To fix this problem, we compute the elements of the result matrix four at a time. Using this approach, we can increase our FLOP count to 80 MFLOPS with a simple restructuring of the program.

However, it is possible to achieve much higher performance from this problem. This is possible by viewing the matrix multiplication problem as a cube in which each internal lattice point corresponds to a multiply-add operation. Matrix multiplication algorithms traverse this cube in different ways; which induce different partitioning of the cube. The data required for computing a partition grows as the surface area of the input faces of the partition and the computation as the volume of the partition. For the algorithms discussed above, we were slicing thin partitions of the cube for which the area and volume were comparable (thus achieving poor cache performance). To remedy this, we restructure the computation by partitioning the cube into subcubes of size $k \times k \times k$. The data associated with this is $3 \times k^2$ (k^2 data for each of the three matrices) and the computation is k^3. To maximize performance, we would like $3 \times k^2$ to be equal to $8K$ since thats the amount of cache available. This corresponds to $k = 51$. The computation associated with a cube of this dimension is 132651 multiply-add operations or 265302 FLOPs. To perform this computation, we needed to fetch two submatrices of size 51×51. This corresponds to 5202 words or 1301 cache lines. Accessing these cache lines takes 65050 cycles or 130100 ns. Since 265302 FLOPs are performed in 130100 ns, the peak computation rate of this formulation is 2.04 GFLOPS. Since this exceeds the processor performance, the computation rate is infact bound by the number processor performance of 1 GFLOPS. Code this example and plot the performance as a function of k. (Code on any conventional microprocessor. Make sure you note the clock speed, the microprocessor and the cache available at each level).

1.5 Consider an SMP with a distributed shared address space. Consider a simple cost model in which it takes 10 ns to access local cache, 100 ns to access local memory, and 400 ns to access remote memory. A parallel program is running on this machine. The program is perfectly load balanced with 80% of all accesses going to local cache, 10% to local memory, and 10% to remote memory. What is the effective memory access time for this computation? If the computation is memory bounded, what is the peak computation rate?

Now consider the same computation running on 1 processor. Here, the processor hits the cache 70% of the times and local memory 30% of the time. What is the effective peak computation rate for 1 processor? What is the efficiency is this computation?

Note: Notice that the cache hit for multiple processors is higher than that for 1 processor. This is typically because the cache available on multiprocessors is larger than single processor systems. For example, 2 R10000 processors will have 64KB L1 cache and a single R10000 only has 32KB of cache. As we noted in class (and you may extrapolate from this question), this is often a source of superlinearity (speedup of greater than p on p processors).

1.6 Consider the omega network described in Section 1.6.3. As shown there, this network is a blocking network (that is, a processor that uses the network to access a memory location might prevent another processor from accessing another memory location). Consider an omega network that connects p processors. Define a function f that maps $P = [0, 1, \ldots, p-1]$ onto a permutation P' of P (that is, $P'[i] = f(P[i])$ and $P'[i] \in P$ for all $0 \le i < p$). Think of this function as mapping communication requests by the processors so that processor $P[i]$ requests communication with processor $P'[i]$.

1. How many distinct permutation functions exist?
2. How many of these functions result in non-blocking communication?
3. What is the probability that an arbitrary function will result in non-blocking communication?

1.7 Given the labeling scheme described in Section **??**, how many distinct labelings exist for a d-dimensional hypercube?

1.8 A cycle in a graph is defined as a path originating and terminating at the same node. The length of a cycle is the number of edges in the cycle. Show that there are no odd-length cycles in a d-dimensional hypercube.

1.9 The labels in a d-dimensional hypercube use d bits. Fixing any k of these bits, show that processors whose labels differ in the remaining $d-k$ bit positions form a $(d-k)$-dimensional subcube composed of $2^{(d-k)}$ processors.

1.10 Let A and B be two processors in a d-dimensional hypercube. Define $H(A, B)$ to be the Hamming distance between A and B, and $P(A, B)$ to be the number of distinct paths connecting A and B. These paths are called ***p***arallel paths and have no common processors other than A and B. Prove the following:

1. The minimum distance in terms of communication links between A and B is given by $H(A, B)$.
2. The total number of parallel paths between any two processors is $P(A, B) = d$.
3. The number of parallel paths between A and B of length $H(A, B)$ is $P_{length=H(A,B)}(A, B) = H(A, B)$.

4. The length of the remaining $d - H(A, B)$ parallel paths is $H(A, B) + 2$.

1.11 In the informal derivation of the bisection width of a hypercube (Section **??**), we used the construction of a hypercube to show that a d-dimensional hypercube is formed from two $(d-1)$-dimensional hypercubes. We argued that because corresponding processors in each of these subcubes have a direct communication link, there are $2^d - 1$ links across the partition. However, it is possible to partition a hypercube into two parts such that neither of the partitions is a hypercube. Show that any such partitions will have more than $2^d - 1$ direct links between them.

1.12 **[?]** Extend the two-dimensional mesh of trees (Problem **??**) to d dimensions to construct a $p^{1/d} \times p^{1/d} \times \cdots \times p^{1/d}$ mesh of trees. We can do this by fixing grid positions in all dimensions to different values and imposing a complete binary tree on the one dimension that is being varied.
Derive the total number of switching elements in a $p^{1/d} \times p^{1/d} \times \cdots \times p^{1/d}$ mesh of trees. Calculate the diameter, bisection width, and wiring cost in terms of the total number of wires. What are the advantages and disadvantages of a mesh of trees as compared to a wraparound mesh?

1.13 **[?]** One of the drawbacks of a hypercube-connected network is that different wires in the network are of different lengths. This implies that data takes different times to traverse different communication links. It appears that two-dimensional mesh networks with wraparound connections suffer from this drawback too. However, it is possible to fabricate a two-dimensional wraparound mesh using wires of fixed length. Illustrate this layout by drawing such a 4×4 wraparound mesh.

1.14 Show how to embed a p-processor three-dimensional mesh into a p-processor hypercube. What are the allowable values of p for your embedding?

1.15 Show how to embed a p-processor mesh of trees into a p-processor hypercube. Is your embedding different from the mesh embedding discussed in Section **??**?

1.16 Consider a complete binary tree of $2^d - 1$ nodes in which each node is a processor. What is the minimum-dilation mapping of such a tree onto a d-dimensional hypercube?

1.17 The concept of a ***m***inimum congestion mapping is very useful. Consider two parallel computers with different interconnection networks such that a congestion-r mapping of the first into the second exists. Ignoring the dilation of the mapping, if each communication link in the second computer is more than r times faster than the first computer, the second computer is strictly superior to the first.
Now consider mapping a d-dimensional hypercube onto a 2^d-processor mesh. Ignoring the dilation of the mapping, what is the minimum-congestion mapping of the hypercube onto the mesh? Use this result to determine whether a 1024-processor mesh with communication links operating at 25 million bytes per second is strictly better than a 1024-processor hypercube (whose processors are identical to those used in the mesh) with communication links operating at two million bytes

per second.

1.18 Derive the diameter, number of links, and bisection width of a k-ary d-cube with p processors. Define l_{av} to be the average distance between any two processors in the network. Derive l_{av} for a k-ary d-cube.

1.19 Consider a hypercube network of p processors. Assume that the channel width of each communication link is one. The channel width of the links in a k-ary d-cube (for $d < \log p$) can be increased by equating the cost of the this network with that of a hypercube. Two distinct measures can be used to evaluate the cost of a network.

1. The cost can expressed in terms of the total number of wires in the network (the total number of wires is a product of the number of communication links and the channel width).
2. The bisection bandwidth can be used as a measure of cost.

Using each of these cost metrics and equating the cost of a k-ary d-cube with a hypercube, what is the channel width of a k-ary d-cube with an identical number of processors, channel rate, and cost?

1.20 The results from Problems 1.18 and 1.19 can be used in a cost-performance analysis of static interconnection networks. Consider a k-ary d-cube network of p processors with cut-through routing. Assume a hypercube-connected network of p processors with channel width one. The channel width of other networks in the family is scaled up so that their cost is identical to that of the hypercube. Let s and s' be the scaling factors for the channel width derived by equating the costs specified by the two cost metrics in Problem 1.19.
For each of the two scaling factors s and s', express the average communication time between any two processors as a function of the dimensionality (d) of a k-ary d-cube and the number of processors. Plot the communication time as a function of the dimensionality for $p = 256, 512$, and 1024, message size $m = 512$ bytes, $t_s = 50.0\mu s$, and $t_h = t_w = 0.5\mu s$ (for the hypercube). For these values of p and m, what is the dimensionality of the network that yields the best performance for a given cost?

1.21 Repeat Problem 1.20 for a k-ary d-cube with store-and-forward routing.