by Hwang + Briggs McGraw Hill, 1984

## 1.4 ARCHITECTURAL CLASSIFICATION SCHEMES

Three computer architectural classification schemes are presented in this section. *Flynn's classification* (1966) is based on the multiplicity of instruction streams and data streams in a computer system. *Feng's scheme* (1972) is based on serial versus parallel processing. *Händler's classification* (1977) is determined by the degree of parallelism and pipelining in various subsystem levels.

### 1.4.1 Multiplicity of Instruction-Data Streams

In general, digital computers may be classified into four categories, according to the multiplicity of instruction and data streams. This scheme for classifying computer organizations was introduced by Michael J. Flynn. The essential computing process is the execution of a sequence of instructions on a set of data. The term *stream* is used here to denote a sequence of items (instructions or data) as executed or operated upon by a single processor. *Instructions* or *data* are defined with respect to a referenced machine. An *instruction stream* is a sequence of instructions as executed by the machine; a *data stream* is a sequence of data including input, partial, or temporary results, called for by the instruction stream.

Computer organizations are characterized by the multiplicity of the hardware provided to service the instruction and data streams. Listed below are Flynn's four machine organizations:

- Single instruction stream-single data stream (SISD)
- Single instruction stream-multiple data stream (SIMD)
- Multiple instruction stream-single data stream (MISD)
- Multiple instruction stream-multiple data stream (MIMD)

These organizational classes are illustrated by the block diagrams in Figure 1.16. The categorization depends on the multiplicity of simultaneous events in the system components. Conceptually, only three types of system components are needed in the illustration. Both instructions and data are fetched from the *memory modules*. Instructions are decoded by the *control unit*, which sends the decoded instruction stream to the *processor units* for execution. Data streams flow between the processors and the memory bidirectionally. Multiple memory modules may be used in the shared memory subsystem. Each instruction stream is generated by an independent control unit. Multiple data streams originate from the subsystem of shared memory modules. I/O facilities are not shown in these simplified block diagrams.

**SISD computer organization** This organization, shown in Figure 1.16*a*, represents most serial computers available today. Instructions are executed sequentially but may be overlapped in their execution stages (pipelining). Most SISD uniprocessor systems are pipelined. An SISD computer may have more than one functional unit in it. All the functional units are under the supervision of one control unit.

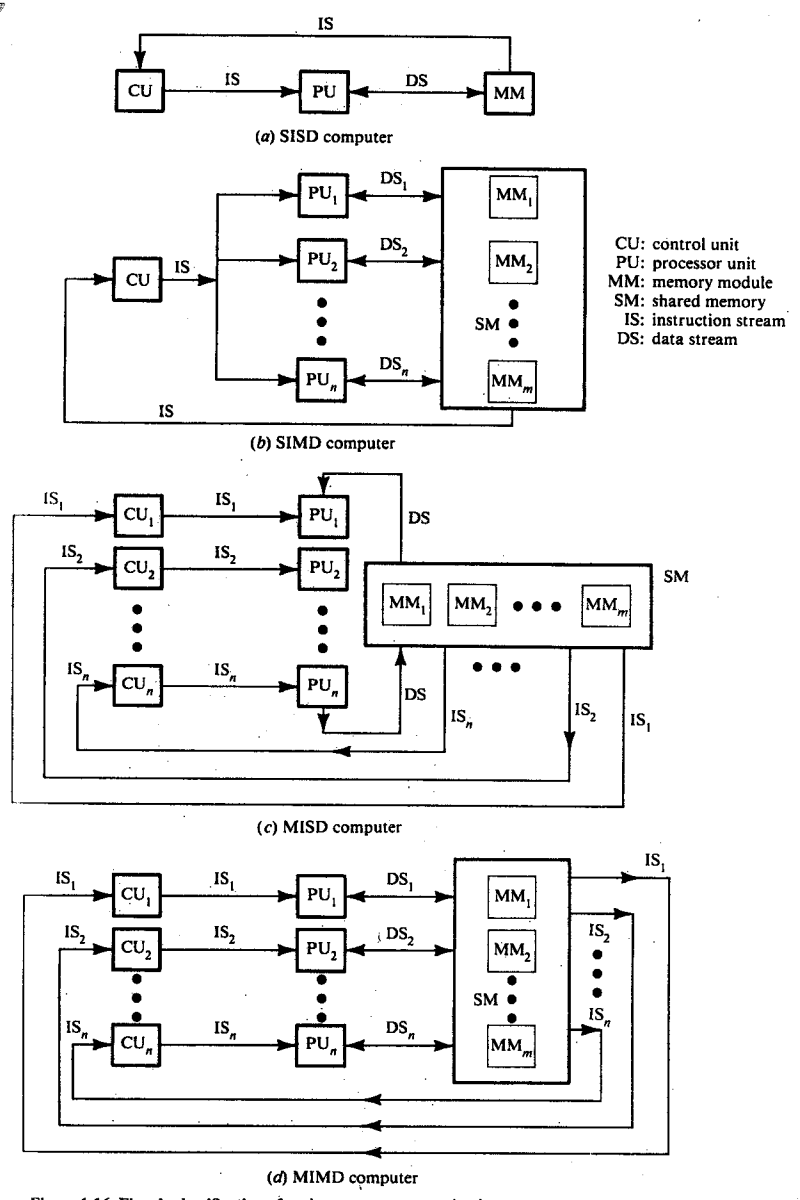


**Figure 1.16 Flynn's classification of various computer organizations.**

**SIMD computer organization** This class corresponds to array processors, introduced in Section 1.3.2. As illustrated in Figure 1.16*b*, there are multiple processing elements supervised by the same control unit. All PEs receive the same instruction broadcast from the control unit but operate on different data sets from distinct data streams. The shared memory subsystem may contain multiple modules. We further divide SIMD machines into *word-slice* versus *bit-slice* modes, to be described in Section 1.4.2.

**MISD computer organization** This organization is conceptually illustrated in Figure 1.16*c*. There are $n$ processor units, each receiving distinct instructions operating over the same data stream and its derivatives. The results (output) of one processor become the input (operands) of the next processor in the macropipe. This structure has received much less attention and has been challenged as impractical by some computer architects. No real embodiment of this class exists.

**MIMD computer organization** Most multiprocessor systems and multiple computer systems can be classified in this category (Figure 1.16*d*). An *intrinsic* MIMD computer implies interactions among the $n$ processors because all memory streams are derived from the same data space shared by all processors. If the $n$ data streams were derived from disjointed subspaces of the shared memories, then we would have the so-called multiple SISD (MSISD) operation, which is nothing but a set of $n$ independent SISD uniprocessor systems. An intrinsic MIMD

**Table 1.3 Flynn's computer system classification**

| Computer class | Computer system models (chapters where the system is quoted or described) |
|---|---|
| SISD (uses one functional unit) | IBM 701 (1); IBM 1620 (1); IBM 7090 (1); PDP VAX11/780 (1). |
| SISD (with multiple functional units) | IBM 360/91 (3); IBM 370/168UP (1); CDC 6600 (1); CDC Star-100 (4); TI-ASC (4); FPS AP-120B (4); FPS-164 (4); IBM 3838 (4); Cray-1 (4); CDC Cyber-205 (4); Fujitsu VP-200 (4); CDC-NASF (4); Fujitsu FACOM-230/75 (4). |
| SIMD (word-slice processing) | Illiac-IV (6); PEPE (1); BSP (6) |
| SIMD (bit-slice processing) | STARAN (1); MPP (6); DAP (1). |
| MIMD (loosely coupled) | IBM 370/168 MP (9); Univac 1100/80 (9); Tandem/16 (9); IBM 3081/3084 (9); C.m* (9) |
| MIMD (tightly coupled) | Burroughs D-825 (9); C.mmp (9); Cray-2 (9). S-1 (9); Cray-X MP (9); Denelcor HEP (9) |



computer is *tightly coupled* if the degree of interactions among the processors is high. Otherwise, we consider them *loosely coupled*. Most commercial MIMD computers are loosely coupled.

In Table 1.3, we have listed several system models under each of the three existing computer organizations. Some of these machines will be studied in subsequent chapters. Readers should check the quoted chapters for details or references related to the specific machines.

### 1.4.2 Serial Versus Parallel Processing

Tse-yun Feng has suggested the use of the *degree* of parallelism to classify various computer architectures. The maximum number of binary digits (bits) that can be processed within a unit time by a computer system is called the *maximum parallelism degree* $P$. Let $P_i$ be the number of bits that can be processed within the $i$th processor cycle (or the $i$th clock period). Consider $T$ processor cycles indexed by $i = 1, 2, \ldots, T$. The average parallelism degree, $P_a$ is defined by

$$P_a = \frac{\sum_{i=1}^{T} P_i}{T} \tag{1.10}$$

In general, $P_i \leq P$. Thus, we define the *utilization rate* $\mu$ of a computer system within $T$ cycles by

$$\mu = \frac{P_a}{P} = \frac{\sum_{i=1}^{T} P_i}{T \cdot P} \tag{1.11}$$

If the computing power of the processor is fully utilized (or the parallelism is fully exploited), then we have $P_i = P$ for all $i$ and $\mu = 1$ for 100 percent utilization. The utilization rate depends on the application program being executed.

Figure 1.17 demonstrates the classification of computers by their maximum parallelism degrees. The horizontal axis shows the word length $n$. The vertical axis corresponds to the bit-slice length $m$. Both length measures are in terms of the number of bits contained in a word or in a bit slice. A *bit slice* is a string of bits, one from each of the words at the same vertical bit position. For example, the TI-ASC has a word length of 64 and four arithmetic pipelines. Each pipe has eight pipeline stages. Thus there are $8 \times 4 = 32$ bits per each bit slice in the four pipes. TI-ASC is represented as (64, 32). The maximum parallelism degree $P(C)$ of a given computer system $C$ is represented by the product of the word length $w$ and the bit-slice length $m$; that is,

$$P(C) = n \cdot m \tag{1.12}$$

The pair $(n, m)$ corresponds to a point in the computer space shown by the coordinate system in Figure 1.17. The $P(C)$ is equal to the area of the rectangle defined by the integers $n$ and $m$.

Computer Architecture: A Quantitative Approach, 4th Edition, 2007
by Hennessy and Patterson, Morgan Kaufmann



Parallelism can also be exploited at the level of detailed digital design. For example, set-associative caches use multiple banks of memory that are typically searched in parallel to find a desired item. Modern ALUs use carry-lookahead, which uses parallelism to speed the process of computing sums from linear to logarithmic in the number of bits per operand.

## Principle of Locality

Important fundamental observations have come from properties of programs. The most important program property that we regularly exploit is the *principle of locality:* Programs tend to reuse data and instructions they have used recently. A widely held rule of thumb is that a program spends 90% of its execution time in only 10% of the code. An implication of locality is that we can predict with reasonable accuracy what instructions and data a program will use in the near future based on its accesses in the recent past. The principle of locality also applies to data accesses, though not as strongly as to code accesses.

Two different types of locality have been observed. *Temporal locality* states that recently accessed items are likely to be accessed in the near future. *Spatial locality* says that items whose addresses are near one another tend to be referenced close together in time. We will see these principles applied in Chapter 5.

## Focus on the Common Case

Perhaps the most important and pervasive principle of computer design is to focus on the common case: In making a design trade-off, favor the frequent case over the infrequent case. This principle applies when determining how to spend resources, since the impact of the improvement is higher if the occurrence is frequent.

Focusing on the common case works for power as well as for resource allocation and performance. The instruction fetch and decode unit of a processor may be used much more frequently than a multiplier, so optimize it first. It works on dependability as well. If a database server has 50 disks for every processor, as in the next section, storage dependability will dominate system dependability.

In addition, the frequent case is often simpler and can be done faster than the infrequent case. For example, when adding two numbers in the processor, we can expect overflow to be a rare circumstance and can therefore improve performance by optimizing the more common case of no overflow. This may slow down the case when overflow occurs, but if that is rare, then overall performance will be improved by optimizing for the normal case.

We will see many cases of this principle throughout this text. In applying this simple principle, we have to decide what the frequent case is and how much performance can be improved by making that case faster. A fundamental law, called *Amdahl's Law,* can be used to quantify this principle.



## Amdahl's Law

The performance gain that can be obtained by improving some portion of a computer can be calculated using Amdahl's Law. Amdahl's Law states that the performance improvement to be gained from using some faster mode of execution is limited by the fraction of the time the faster mode can be used.

Amdahl's Law defines the *speedup* that can be gained by using a particular feature. What is speedup? Suppose that we can make an enhancement to a computer that will improve performance when it is used. Speedup is the ratio

$$\text{Speedup} = \frac{\text{Performance for entire task using the enhancement when possible}}{\text{Performance for entire task without using the enhancement}}$$

Alternatively,

$$\text{Speedup} = \frac{\text{Execution time for entire task without using the enhancement}}{\text{Execution time for entire task using the enhancement when possible}}$$

Speedup tells us how much faster a task will run using the computer with the enhancement as opposed to the original computer.

Amdahl's Law gives us a quick way to find the speedup from some enhancement, which depends on two factors:

1. *The fraction of the computation time in the original computer that can be converted to take advantage of the enhancement*—For example, if 20 seconds of the execution time of a program that takes 60 seconds in total can use an enhancement, the fraction is 20/60. This value, which we will call $\text{Fraction}_{\text{enhanced}}$, is always less than or equal to 1.
2. *The improvement gained by the enhanced execution mode; that is, how much faster the task would run if the enhanced mode were used for the entire program*—This value is the time of the original mode over the time of the enhanced mode. If the enhanced mode takes, say, 2 seconds for a portion of the program, while it is 5 seconds in the original mode, the improvement is 5/2. We will call this value, which is always greater than 1, $\text{Speedup}_{\text{enhanced}}$.

The execution time using the original computer with the enhanced mode will be the time spent using the unenhanced portion of the computer plus the time spent using the enhancement:

$$\text{Execution time}_{\text{new}} = \text{Execution time}_{\text{old}} \times \left( (1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}} \right)$$

The overall speedup is the ratio of the execution times:

$$\text{Speedup}_{\text{overall}} = \frac{\text{Execution time}_{\text{old}}}{\text{Execution time}_{\text{new}}} = \frac{1}{(1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}}$$

**Example** Suppose that we want to enhance the processor used for Web serving. The new processor is 10 times faster on computation in the Web serving application than the original processor. Assuming that the original processor is busy with computation 40% of the time and is waiting for I/O 60% of the time, what is the overall speedup gained by incorporating the enhancement?

***Answer*** $\text{Fraction}_{\text{enhanced}} = 0.4$, $\text{Speedup}_{\text{enhanced}} = 10$, $\text{Speedup}_{\text{overall}} = \dfrac{1}{0.6 + \dfrac{0.4}{10}} = \dfrac{1}{0.64} \approx 1.56$

Amdahl's Law expresses the law of diminishing returns: The incremental improvement in speedup gained by an improvement of just a portion of the computation diminishes as improvements are added. An important corollary of Amdahl's Law is that if an enhancement is only usable for a fraction of a task, we can't speed up the task by more than the reciprocal of 1 minus that fraction.

A common mistake in applying Amdahl's Law is to confuse "fraction of time converted to use an enhancement" and "fraction of time after enhancement is in use." If, instead of measuring the time that we *could use* the enhancement in a computation, we measure the time *after* the enhancement is in use, the results will be incorrect!

Amdahl's Law can serve as a guide to how much an enhancement will improve performance and how to distribute resources to improve cost-performance. The goal, clearly, is to spend resources proportional to where time is spent. Amdahl's Law is particularly useful for comparing the overall system performance of two alternatives, but it can also be applied to compare two processor design alternatives, as the following example shows.

**Example** A common transformation required in graphics processors is square root. Implementations of floating-point (FP) square root vary significantly in performance, especially among processors designed for graphics. Suppose FP square root (FPSQR) is responsible for 20% of the execution time of a critical graphics benchmark. One proposal is to enhance the FPSQR hardware and speed up this operation by a factor of 10. The other alternative is just to try to make all FP instructions in the graphics processor run faster by a factor of 1.6; FP instructions are responsible for half of the execution time for the application. The design team believes that they can make all FP instructions run 1.6 times faster with the same effort as required for the fast square root. Compare these two design alternatives.

***Answer*** We can compare these two alternatives by comparing the speedups:

$$\text{Speedup}_{\text{FPSQR}} = \frac{1}{(1-0.2) + \dfrac{0.2}{10}} = \frac{1}{0.82} = 1.22$$

$$\text{Speedup}_{\text{FP}} = \frac{1}{(1-0.5) + \dfrac{0.5}{1.6}} = \frac{1}{0.8125} = 1.23$$



Improving the performance of the FP operations overall is slightly better because of the higher frequency.

Amdahl's Law is applicable beyond performance. Let's redo the reliability example from page 27 after improving the reliability of the power supply via redundancy from 200,000-hour to 830,000,000-hour MTTF, or 4150X better.

**Example** The calculation of the failure rates of the disk subsystem was

$$\text{Failure rate}_{\text{system}} = 10 \times \frac{1}{1{,}000{,}000} + \frac{1}{500{,}000} + \frac{1}{200{,}000} + \frac{1}{200{,}000} + \frac{1}{1{,}000{,}000}$$
$$= \frac{10+2+5+5+1}{1{,}000{,}000 \text{ hours}} = \frac{23}{1{,}000{,}000 \text{ hours}}$$

Therefore, the fraction of the failure rate that could be improved is 5 per million hours out of 23 for the whole system, or 0.22.

***Answer*** The reliability improvement would be

$$\text{Improvement}_{\text{power supply pair}} = \frac{1}{(1-0.22) + \dfrac{0.22}{4150}} = \frac{1}{0.78} = 1.28$$

Despite an impressive 4150X improvement in reliability of one module, from the system's perspective, the change has a measurable but small benefit.

In the examples above we needed the fraction consumed by the new and improved version; often it is difficult to measure these times directly. In the next section, we will see another way of doing such comparisons based on the use of an equation that decomposes the CPU execution time into three separate components. If we know how an alternative affects these three components, we can determine its overall performance. Furthermore, it is often possible to build simulators that measure these components before the hardware is actually designed.

## The Processor Performance Equation

Essentially all computers are constructed using a clock running at a constant rate. These discrete time events are called *ticks, clock ticks, clock periods, clocks, cycles,* or *clock cycles.* Computer designers refer to the time of a clock period by its duration (e.g., 1 ns) or by its rate (e.g., 1 GHz). CPU time for a program can then be expressed two ways:

$$\text{CPU time} = \text{CPU clock cycles for a program} \times \text{Clock cycle time}$$

or

$$\text{CPU time} = \frac{\text{CPU clock cycles for a program}}{\text{Clock rate}}$$

## 4.1 Introduction

As the quotation that opens this chapter shows, the view that advances in uniprocessor architecture were nearing an end has been held by some researchers for many years. Clearly these views were premature; in fact, during the period of 1986–2002, uniprocessor performance growth, driven by the microprocessor, was at its highest rate since the first transistorized computers in the late 1950s and early 1960s.

Nonetheless, the importance of multiprocessors was growing throughout the 1990s as designers sought a way to build servers and supercomputers that achieved higher performance than a single microprocessor, while exploiting the tremendous cost-performance advantages of commodity microprocessors. As we discussed in Chapters 1 and 3, the slowdown in uniprocessor performance arising from diminishing returns in exploiting ILP, combined with growing concern over power, is leading to a new era in computer architecture—an era where multiprocessors play a major role. The second quotation captures this clear inflection point.

This trend toward more reliance on multiprocessing is reinforced by other factors:

- A growing interest in servers and server performance
- A growth in data-intensive applications
- The insight that increasing performance on the desktop is less important (outside of graphics, at least)
- An improved understanding of how to use multiprocessors effectively, especially in server environments where there is significant natural thread-level parallelism
- The advantages of leveraging a design investment by replication rather than unique design—all multiprocessor designs provide such leverage

That said, we are left with two problems. First, multiprocessor architecture is a large and diverse field, and much of the field is in its youth, with ideas coming and going and, until very recently, more architectures failing than succeeding. Full coverage of the multiprocessor design space and its trade-offs would require another volume. (Indeed, Culler, Singh, and Gupta [1999] cover *only* multiprocessors in their 1000-page book!) Second, broad coverage would necessarily entail discussing approaches that may not stand the test of time—something we have largely avoided to this point.

For these reasons, we have chosen to focus on the mainstream of multiprocessor design: multiprocessors with small to medium numbers of processors (4 to 32). Such designs vastly dominate in terms of both units and dollars. We will pay only slight attention to the larger-scale multiprocessor design space (32 or more processors), primarily in Appendix H, which covers more aspects of the design of such processors, as well as the behavior performance for parallel scientific workloads, a primary class of applications for large-scale multiprocessors. In the large-scale multiprocessors, the interconnection networks are a critical part of the design; Appendix E focuses on that topic.



### A Taxonomy of Parallel Architectures

We begin this chapter with a taxonomy so that you can appreciate both the breadth of design alternatives for multiprocessors and the context that has led to the development of the dominant form of multiprocessors. We briefly describe the alternatives and the rationale behind them; a longer description of how these different models were born (and often died) can be found in Appendix K.

The idea of using multiple processors both to increase performance and to improve availability dates back to the earliest electronic computers. About 40 years ago, Flynn [1966] proposed a simple model of categorizing all computers that is still useful today. He looked at the parallelism in the instruction and data streams called for by the instructions at the most constrained component of the multiprocessor, and placed all computers into one of four categories:

1. *Single instruction stream, single data stream* (SISD)—This category is the uniprocessor.
2. *Single instruction stream, multiple data streams* (SIMD)—The same instruction is executed by multiple processors using different data streams. SIMD computers exploit *data-level parallelism* by applying the same operations to multiple items of data in parallel. Each processor has its own data memory (hence multiple data), but there is a single instruction memory and control processor, which fetches and dispatches instructions. For applications that display significant data-level parallelism, the SIMD approach can be very efficient. The multimedia extensions discussed in Appendices B and C are a form of SIMD parallelism. Vector architectures, discussed in Appendix F, are the largest class of SIMD architectures. SIMD approaches have experienced a rebirth in the last few years with the growing importance of graphics performance, especially for the game market. SIMD approaches are the favored method for achieving the high performance needed to create realistic three-dimensional, real-time virtual environments.
3. *Multiple instruction streams, single data stream* (MISD)—No commercial multiprocessor of this type has been built to date.
4. *Multiple instruction streams, multiple data streams* (MIMD)—Each processor fetches its own instructions and operates on its own data. MIMD computers exploit *thread-level parallelism*, since multiple threads operate in parallel. In general, thread-level parallelism is more flexible than data-level parallelism and thus more generally applicable.

This is a coarse model, as some multiprocessors are hybrids of these categories. Nonetheless, it is useful to put a framework on the design space.

Because the MIMD model can exploit thread-level parallelism, it is the architecture of choice for general-purpose multiprocessors and our focus in this chapter. Two other factors have also contributed to the rise of the MIMD multiprocessors:

1. MIMDs offer flexibility. With the correct hardware and software support, MIMDs can function as single-user multiprocessors focusing on high performance for one application, as multiprogrammed multiprocessors running many tasks simultaneously, or as some combination of these functions.
2. MIMDs can build on the cost-performance advantages of off-the-shelf processors. In fact, nearly all multiprocessors built today use the same microprocessors found in workstations and single-processor servers. Furthermore, multicore chips leverage the design investment in a single processor core by replicating it.

One popular class of MIMD computers are *clusters*, which often use standard components and often standard network technology, so as to leverage as much commodity technology as possible. In Appendix H we distinguish two different types of clusters: *commodity clusters*, which rely entirely on third-party processors and interconnection technology, and *custom clusters*, in which a designer customizes either the detailed node design or the interconnection network, or both.

In a commodity cluster, the nodes of a cluster are often blades or rack-mounted servers (including small-scale multiprocessor servers). Applications that focus on throughput and require almost no communication among threads, such as Web serving, multiprogramming, and some transaction-processing applications, can be accommodated inexpensively on a cluster. Commodity clusters are often assembled by users or computer center directors, rather than by vendors.

Custom clusters are typically focused on parallel applications that can exploit large amounts of parallelism on a single problem. Such applications require a significant amount of communication during the computation, and customizing the node and interconnect design makes such communication more efficient than in a commodity cluster. Currently, the largest and fastest multiprocessors in existence are custom clusters, such as the IBM Blue Gene, which we discuss in Appendix H.

Starting in the 1990s, the increasing capacity of a single chip allowed designers to place multiple processors on a single die. This approach, initially called *on-chip multiprocessing* or *single-chip multiprocessing*, has come to be called *multicore*, a name arising from the use of multiple processor cores on a single die. In such a design, the multiple cores typically share some resources, such as a second- or third-level cache or memory and I/O buses. Recent processors, including the IBM Power5, the Sun T1, and the Intel Pentium D and Xeon-MP, are multicore and multithreaded. Just as using multiple copies of a microprocessor in a multiprocessor leverages a design investment through replication, a multicore achieves the same advantage relying more on replication than the alternative of building a wider superscalar.

With an MIMD, each processor is executing its own instruction stream. In many cases, each processor executes a different process. A *process* is a segment of code that may be run independently; the state of the process contains all the information necessary to execute that program on a processor. In a multiprogrammed environment, where the processors may be running independent tasks, each process is typically independent of other processes.

It is also useful to be able to have multiple processors executing a single program and sharing the code and most of their address space. When multiple processes share code and data in this way, they are often called *threads*. Today, the term *thread* is often used in a casual way to refer to multiple loci of execution that may run on different processors, even when they do not share an address space. For example, a multithreaded architecture actually allows the simultaneous execution of multiple processes, with potentially separate address spaces, as well as multiple threads that share the same address space.

To take advantage of an MIMD multiprocessor with $n$ processors, we must usually have at least $n$ threads or processes to execute. The independent threads within a single process are typically identified by the programmer or created by the compiler. The threads may come from large-scale, independent processes scheduled and manipulated by the operating system. At the other extreme, a thread may consist of a few tens of iterations of a loop, generated by a parallel compiler exploiting data parallelism in the loop. Although the amount of computation assigned to a thread, called the *grain size*, is important in considering how to exploit thread-level parallelism efficiently, the important qualitative distinction from instruction-level parallelism is that thread-level parallelism is identified at a high level by the software system and that the threads consist of hundreds to millions of instructions that may be executed in parallel.

Threads can also be used to exploit data-level parallelism, although the overhead is likely to be higher than would be seen in an SIMD computer. This overhead means that grain size must be sufficiently large to exploit the parallelism efficiently. For example, although a vector processor (see Appendix F) may be able to efficiently parallelize operations on short vectors, the resulting grain size when the parallelism is split among many threads may be so small that the overhead makes the exploitation of the parallelism prohibitively expensive.

Existing MIMD multiprocessors fall into two classes, depending on the number of processors involved, which in turn dictates a memory organization and interconnect strategy. We refer to the multiprocessors by their memory organization because what constitutes a small or large number of processors is likely to change over time.

The first group, which we call *centralized shared-memory architectures*, has at most a few dozen processor chips (and less than 100 cores) in 2006. For multiprocessors with small processor counts, it is possible for the processors to share a single centralized memory. With large caches, a single memory, possibly with multiple banks, can satisfy the memory demands of a small number of processors. By using multiple point-to-point connections, or a switch, and adding additional memory banks, a centralized shared-memory design can be scaled to a few dozen processors. Although scaling beyond that is technically conceivable,

sharing a centralized memory becomes less attractive as the number of processors sharing it increases.

Because there is a single main memory that has a symmetric relationship to all processors and a uniform access time from any processor, these multiprocessors are most often called *symmetric (shared-memory) multiprocessors* (SMPs), and this style of architecture is sometimes called *uniform memory access* (UMA), arising from the fact that all processors have a uniform latency from memory, even if the memory is organized into multiple banks. Figure 4.1 shows what these multiprocessors look like. This type of symmetric shared-memory architecture is currently by far the most popular organization. The architecture of such multiprocessors is the topic of Section 4.2.

The second group consists of multiprocessors with physically distributed memory. Figure 4.2 shows what these multiprocessors look like. To support larger processor counts, memory must be distributed among the processors rather than centralized; otherwise the memory system would not be able to support the bandwidth demands of a larger number of processors without incurring excessively long access latency. With the rapid increase in processor performance and the associated increase in a processor's memory bandwidth requirements, the size of a multiprocessor for which distributed memory is preferred continues to shrink. The larger number of processors also raises the need for a high-bandwidth interconnect, of which we will see examples in Appendix E.

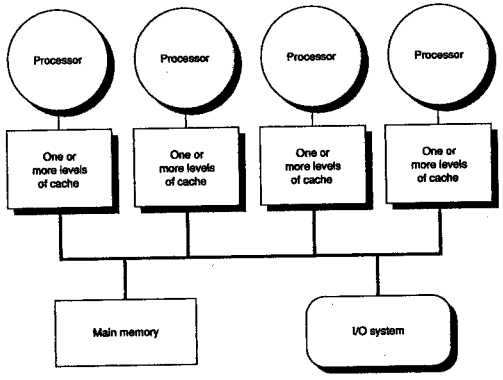


**Figure 4.1 Basic structure of a centralized shared-memory multiprocessor.** Multiple processor-cache subsystems share the same physical memory, typically connected by one or more buses or a switch. The key architectural property is the uniform access time to all of memory from all the processors.

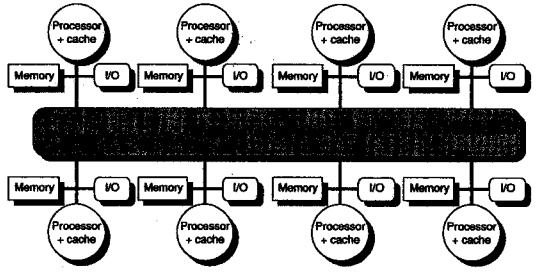


**Figure 4.2 The basic architecture of a distributed-memory multiprocessor consists of individual nodes containing a processor, some memory, typically some I/O, and an interface to an interconnection network that connects all the nodes.** Individual nodes may contain a small number of processors, which may be interconnected by a small bus or a different interconnection technology, which is less scalable than the global interconnection network.

Both direction networks (i.e., switches) and indirect networks (typically multidimensional meshes) are used.

Distributing the memory among the nodes has two major benefits. First, it is a cost-effective way to scale the memory bandwidth if most of the accesses are to the local memory in the node. Second, it reduces the latency for accesses to the local memory. These two advantages make distributed memory attractive at smaller processor counts as processors get ever faster and require more memory bandwidth and lower memory latency. The key disadvantages for a distributed-memory architecture are that communicating data between processors becomes somewhat more complex, and that it requires more effort in the software to take advantage of the increased memory bandwidth afforded by distributed memories. As we will see shortly, the use of distributed memory also leads to two different paradigms for interprocessor communication.

## Models for Communication and Memory Architecture

As discussed earlier, any large-scale multiprocessor must use multiple memories that are physically distributed with the processors. There are two alternative architectural approaches that differ in the method used for communicating data among processors.

In the first method, communication occurs through a shared address space, as it does in a symmetric shared-memory architecture. The physically separate memories can be addressed as one logically shared address space, meaning that a

memory reference can be made by any processor to any memory location, assuming it has the correct access rights. These multiprocessors are called *distributed shared-memory* (DSM) architectures. The term *shared memory* refers to the fact that the *address space* is shared; that is, the same physical address on two processors refers to the same location in memory. Shared memory does *not* mean that there is a single, centralized memory. In contrast to the symmetric shared-memory multiprocessors, also known as UMAs (uniform memory access), the DSM multiprocessors are also called NUMAs (nonuniform memory access), since the access time depends on the location of a data word in memory.

Alternatively, the address space can consist of multiple private address spaces that are logically disjoint and cannot be addressed by a remote processor. In such multiprocessors, the same physical address on two different processors refers to two different locations in two different memories. Each processor-memory module is essentially a separate computer. Initially, such computers were built with different processing nodes and specialized interconnection networks. Today, most designs of this type are actually clusters, which we discuss in Appendix H.

With each of these organizations for the address space, there is an associated communication mechanism. For a multiprocessor with a shared address space, that address space can be used to communicate data implicitly via load and store operations—hence the name *shared memory* for such multiprocessors. For a multiprocessor with multiple address spaces, communication of data is done by explicitly passing messages among the processors. Therefore, these multiprocessors are often called *message-passing multiprocessors*. Clusters inherently use message passing.

## Challenges of Parallel Processing

The application of multiprocessors ranges from running independent tasks with essentially no communication to running parallel programs where threads must communicate to complete the task. Two important hurdles, both explainable with Amdahl's Law, make parallel processing challenging. The degree to which these hurdles are difficult or easy is determined both by the application and by the architecture.

The first hurdle has to do with the limited parallelism available in programs, and the second arises from the relatively high cost of communications. Limitations in available parallelism make it difficult to achieve good speedups in any parallel processor, as our first example shows.

---

**Example** Suppose you want to achieve a speedup of 80 with 100 processors. What fraction of the original computation can be sequential?

***Answer*** Amdahl's Law is

$$\text{Speedup} = \frac{1}{\frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}} + (1 - \text{Fraction}_{\text{enhanced}})}$$

For simplicity in this example, assume that the program operates in only two modes: parallel with all processors fully used, which is the enhanced mode, or serial with only one processor in use. With this simplification, the speedup in enhanced mode is simply the number of processors, while the fraction of enhanced mode is the time spent in parallel mode. Substituting into the previous equation:

$$80 = \frac{1}{\frac{\text{Fraction}_{\text{parallel}}}{100} + (1 - \text{Fraction}_{\text{parallel}})}$$

Simplifying this equation yields

$$\begin{aligned} 0.8 \times \text{Fraction}_{\text{parallel}} + 80 \times (1 - \text{Fraction}_{\text{parallel}}) &= 1 \\ 80 - 79.2 \times \text{Fraction}_{\text{parallel}} &= 1 \\ \text{Fraction}_{\text{parallel}} &= \frac{80 - 1}{79.2} \\ \text{Fraction}_{\text{parallel}} &= 0.9975 \end{aligned}$$

Thus, to achieve a speedup of 80 with 100 processors, only 0.25% of original computation can be sequential. Of course, to achieve linear speedup (speedup of $n$ with $n$ processors), the entire program must usually be parallel with no serial portions. In practice, programs do not just operate in fully parallel or sequential mode, but often use less than the full complement of the processors when running in parallel mode.

---

The second major challenge in parallel processing involves the large latency of remote access in a parallel processor. In existing shared-memory multiprocessors, communication of data between processors may cost anywhere from 50 clock cycles (for multicores) to over 1000 clock cycles (for large-scale multiprocessors), depending on the communication mechanism, the type of interconnection network, and the scale of the multiprocessor. The effect of long communication delays is clearly substantial. Let's consider a simple example.

---

**Example** Suppose we have an application running on a 32-processor multiprocessor, which has a 200 ns time to handle reference to a remote memory. For this application, assume that all the references except those involving communication hit in the local memory hierarchy, which is slightly optimistic. Processors are stalled on a remote request, and the processor clock rate is 2 GHz. If the base CPI (assuming that all references hit in the cache) is 0.5, how much faster is the multiprocessor if there is no communication versus if 0.2% of the instructions involve a remote communication reference?

***Answer*** It is simpler to first calculate the CPI. The effective CPI for the multiprocessor with 0.2% remote references is

$$\text{CPI} = \text{Base CPI} + \text{Remote request rate} \times \text{Remote request cost}$$
$$= 0.5 + 0.2\% \times \text{Remote request cost}$$

The remote request cost is

$$\frac{\text{Remote access cost}}{\text{Cycle time}} = \frac{200 \text{ ns}}{0.5 \text{ ns}} = 400 \text{ cycles}$$

Hence we can compute the CPI:

$$\text{CPI} = 0.5 + 0.8 = 1.3$$

The multiprocessor with all local references is 1.3/0.5 = 2.6 times faster. In practice, the performance analysis is much more complex, since some fraction of the noncommunication references will miss in the local hierarchy and the remote access time does not have a single constant value. For example, the cost of a remote reference could be quite a bit worse, since contention caused by many references trying to use the global interconnect can lead to increased delays.

These problems—insufficient parallelism and long-latency remote communication—are the two biggest performance challenges in using multiprocessors. The problem of inadequate application parallelism must be attacked primarily in software with new algorithms that can have better parallel performance. Reducing the impact of long remote latency can be attacked both by the architecture and by the programmer. For example, we can reduce the frequency of remote accesses with either hardware mechanisms, such as caching shared data, or software mechanisms, such as restructuring the data to make more accesses local. We can try to tolerate the latency by using multithreading (discussed in Chapter 3 and later in this chapter) or by using prefetching (a topic we cover extensively in Chapter 5).

Much of this chapter focuses on techniques for reducing the impact of long remote communication latency. For example, Sections 4.2 and 4.3 discuss how caching can be used to reduce remote access frequency, while maintaining a coherent view of memory. Section 4.5 discusses synchronization, which, because it inherently involves interprocessor communication and also can limit parallelism, is a major potential bottleneck. Section 4.6 covers latency-hiding techniques and memory consistency models for shared memory. In Appendix I, we focus primarily on large-scale multiprocessors, which are used predominantly for scientific work. In that appendix, we examine the nature of such applications and the challenges of achieving speedup with dozens to hundreds of processors.

Understanding a modern shared-memory multiprocessor requires a good understanding of the basics of caches. Readers who have covered this topic in our introductory book, *Computer Organization and Design: The Hardware/Software Interface*, will be well-prepared. If topics such as write-back caches and multilevel caches are unfamiliar to you, you should take the time to review Appendix C.



## Symmetric Shared-Memory Architectures

The use of large, multilevel caches can substantially reduce the memory bandwidth demands of a processor. If the main memory bandwidth demands of a single processor are reduced, multiple processors may be able to share the same memory. Starting in the 1980s, this observation, combined with the emerging dominance of the microprocessor, motivated many designers to create small-scale multiprocessors where several processors shared a single physical memory connected by a shared bus. Because of the small size of the processors and the significant reduction in the requirements for bus bandwidth achieved by large caches, such symmetric multiprocessors were extremely cost-effective, provided that a sufficient amount of memory bandwidth existed. Early designs of such multiprocessors were able to place the processor and cache subsystem on a board, which plugged into the bus backplane. Subsequent versions of such designs in the 1990s could achieve higher densities with two to four processors per board, and often used multiple buses and interleaved memories to support the faster processors.

IBM introduced the first on-chip multiprocessor for the general-purpose computing market in 2000. AMD and Intel followed with two-processor versions for the server market in 2005, and Sun introduced T1, an eight-processor multicore in 2006. Section 4.8 looks at the design and performance of T1. The earlier Figure 4.1 on page 200 shows a simple diagram of such a multiprocessor. With the more recent, higher-performance processors, the memory demands have outstripped the capability of reasonable buses. As a result, most recent designs use a small-scale switch or a limited point-to-point network.

Symmetric shared-memory machines usually support the caching of both shared and private data. *Private data* are used by a single processor, while *shared data* are used by multiple processors, essentially providing communication among the processors through reads and writes of the shared data. When a private item is cached, its location is migrated to the cache, reducing the average access time as well as the memory bandwidth required. Since no other processor uses the data, the program behavior is identical to that in a uniprocessor. When shared data are cached, the shared value may be replicated in multiple caches. In addition to the reduction in access latency and required memory bandwidth, this replication also provides a reduction in contention that may exist for shared data items that are being read by multiple processors simultaneously. Caching of shared data, however, introduces a new problem: cache coherence.

### What Is Multiprocessor Cache Coherence?

Unfortunately, caching shared data introduces a new problem because the view of memory held by two different processors is through their individual caches, which, without any additional precautions, could end up seeing two different values. Figure 4.3 illustrates the problem and shows how two different processors