12
Multiprocessor Systems

Up till now, we have discussed the design and implementation of a processor in great detail including several methods to optimize its performance such as pipelining. We observed that by optimizing the processor, and the memory system, it is possible to significantly increase the performance of a program. Now, the question is, "Is this enough?" Or, is it possible to do better?

A short answer to this question is "Maybe Not." For a long answer to this question, the reader needs to read through this entire chapter, and possibly take a look at the references. Let us start out by saying that processor performance has its limits. It is not possible to increase the speed of a processor indefinitely. Even with very complicated superscalar processors (see Chapter 10), and highly optimized memory systems, it is typically not possible to increase the IPC by more than 50%. Secondly, because of power and temperature considerations, it is very difficult to increase processor frequency beyond 3 GHz. The reader should note that processor frequencies have remained more of less the same over the last ten years (2002-2012). Consequently, CPU performance has also been increasing very slowly over the last ten years.

We illustrate these points in Figures 12.1, and 12.2. Figure 12.1 shows the peak frequency of processors released by multiple vendors such as Intel, AMD, Sun, Qualcomm, and Fujitsu from 2001 till 2010. We observe that the frequency has stayed more or less constant (mostly between 1 GHz to 2.5 GHz). The trends do not indicate a gradual increase in frequency. We expect that in the near future also, the frequency of processors will be limited to 3 GHz.

Figure 12.2 shows the average Spec Int 2006 score for the same set of processors from 2001 till 2010. We observe that CPU performance is slowly saturating over time, and it is getting increasingly difficult to increase performance.

Even though the performance of a single processor is not expected to significantly increase in the future, the future of computer architecture is not bleak. This is because processor manufacturing technology is steadily getting better, and this is leading to smaller and faster transistors. Till the late nineties processor designers were utilizing the gains in transistor technology to increase the complexity of a processor by implementing more features. However,

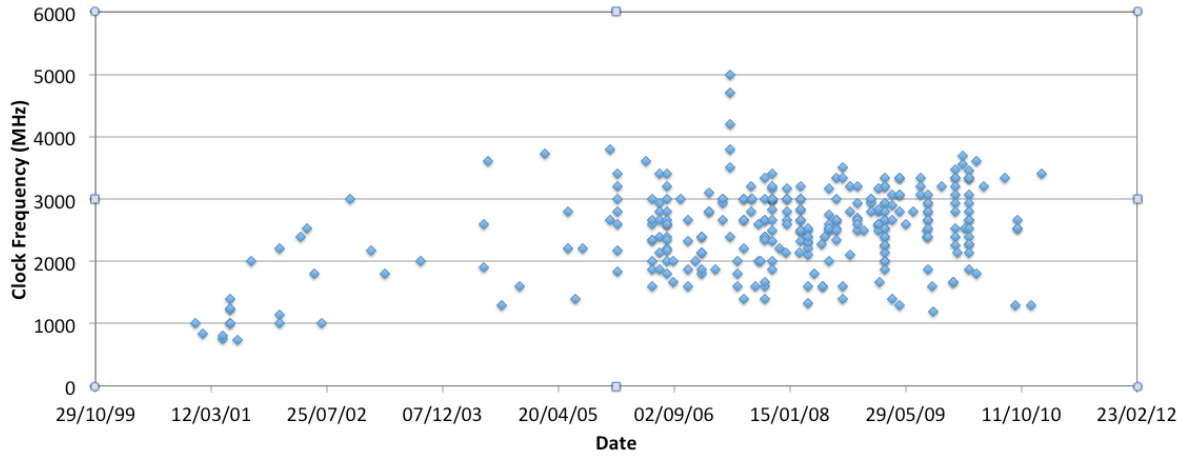


Figure 12.1: CPU frequencies (source [Danowitz et al., 2012])

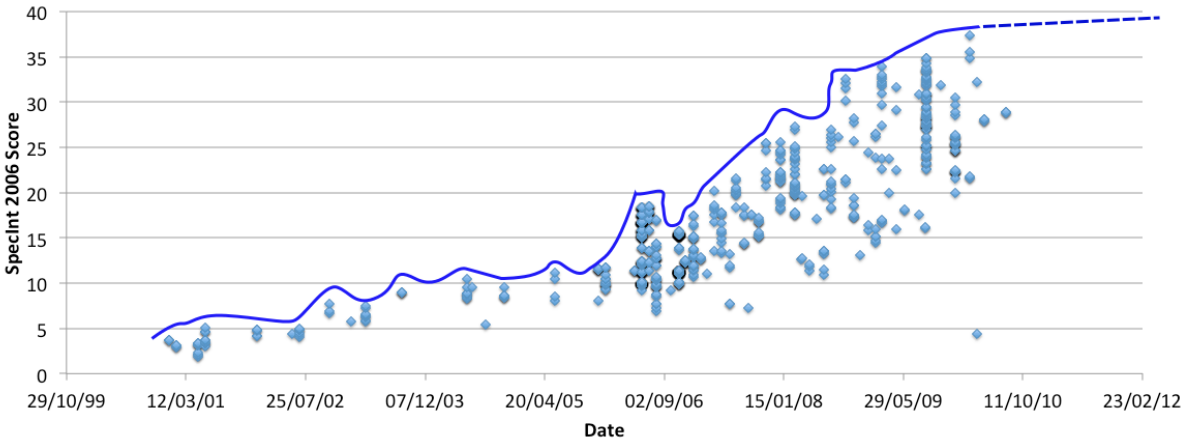


Figure 12.2: CPU performance (source [Danowitz et al., 2012])

due to limitations in complexity, and power, designers resorted to using simpler processors after 2005. Instead of implementing more features in processors, vendors instead decided to put more than one processor on a single chip. This helps us run more than one program at the same time. Alternatively, sometimes it is possible to split a single program into multiple parts and run all the parts in parallel.

This paradigm of using multiple computing units running in parallel is known as *multiprocessing*. The term "multiprocessing" is a rather generic term. It can either refer to multiple processors in the same chip working in parallel, or it can refer to multiple processors across chips working in parallel. A multiprocessor is a piece of hardware that supports multiprocessing. When we have multiple processors within a chip, each processor is known as a *core*, and the chip is called a *multicore* processor.

Definition 116
The term multiprocessing *refers to multiple processors working in parallel. This is a generic definition, and it can refer to multiple processors in the same chip, or processors across different chips. A* multicore *processor is a specific type of multiprocessor that contains all of its constituent processors in the same chip. Each such processor is known as a* core.

We are now entering the era of multiprocessors, especially multicore systems. The number of cores per chip is increasing by roughly a factor of two every two years. New applications are being written to leverage this extra hardware. Most experts opine that the future of computing lies in multiprocessor systems.

Before proceeding to the design of different types of multiprocessors, let us quickly take a look at the background and history of multiprocessors.

12.1 Background

In the 60s and 70s large computers were primarily used by banks and financial institutions. They had a growing base of consumers, and consequently they needed computers that could perform more and more transactions per second. Typically, just one processor proved to be insufficient in providing the computing throughput that was required. Hence, early computer designers decided to put multiple processors in a computer. The processors could share the computing load, and thus increase the computing throughput of the entire system.

One of the earliest multiprocessors was the Burroughs 5000, which had two processors – A and B. A was the main processor, and B was an auxiliary processor. When the load was high, processor A gave processor B some work to do. Almost all the other major vendors at that time also had multiprocessor offerings such as the IBM 370, PDP 11/74, VAX-11/782, and Univac 1108-II. These computers supported a second CPU chip. This was connected to the main processor. In all of these early machines, the second CPU was on a second chip that was physically connected to the first with wires or cables. They came in two flavors: symmetric and asymmetric. A *symmetric multiprocessor* consists of multiple processors, where each processor is of the same type, and has access to the services offered by the operating system, and peripherals; whereas, an *asymmetric multiprocessor* assigns different roles to different processors. There is typically a distinguished processor that controls the operating system, and the peripherals. The rest of the processors are slaves. They take work from the main processor, and return the results.

Definition 117

Symmetric Multiprocessing *This paradigm treats all the constituent processors in a multiprocessor system as the same. Each processor has equal access to the operating system, and the I/O peripherals. These are also known as SMP systems.*

Asymmetric Multiprocessing *This paradigm does not treat all the constituent processors in a multiprocessor system as the same. There is typically one* master *processor that has exclusive control of the operating system and I/O devices. It assigns work to the rest of the processors.*

In the early days, the second processor was connected to the main processor using a set of cables. It was typically housed in a different area of the main computer. Note that in those days, computers used to be the size of a room. With increased miniaturization, gradually both the processors started coming closer. In the late eighties and early nineties, companies started putting multiple processors on the same motherboard. A motherboard is a printed circuit board that contains all the chips that a computer uses. The reader can take the lid off her laptop or desktop. The large green board with chips and metallic lines is the motherboard. By the late nineties, it was possible to have four or eight processors on a single motherboard. They were connected to each other using dedicated high speed buses.

Gradually, the era of multicore processors commenced. It was now possible to have multiple processors in the same chip. IBM was the first to announce a dual core (2 cores) multicore processor called the Power 4 in 2001. Intel and AMD followed with similar offerings in 2005. As of 2012, 8 core, and 10 core versions of multicore processors are available.

12.1.1 Moore's Law

Let us now take a deeper look at what happened between 1960 and 2012 in the world of processors. In the sixties, a computer was typically the size of a room, and today a computer fits in the pocket. A processor in a cell phone is around 1.6 million times faster than the IBM 360 machines in the early sixties. It is also several orders of magnitude more power efficient. The main driver for this continued evolution of computer technology is the miniaturization of the transistor. Transistors used to have a channel length of several millimeters in the sixties, and now they are about 20-30 nanometers long. In 1971, a typical chip used to have 2000-3000 transistors. Nowadays, a chip has billions of transistors.

Over the last forty to fifty years, the number of transistors per chip has been roughly doubling every 1-2 years. In fact, the co-founder of Intel, Gordon Moore, had predicted this trend in 1965. The Moore's law (named in the honor of Gordon Moore) predicts that the number of transistors on a chip is expected to double every one to two years. Originally, Moore had predicted the period of doubling to be every year. However, over time, this period has become about 2 years. This was expected to happen because of the steady rate of advances in manufacturing technology, new materials, and fabrication techniques.

Historical Note 3
In 1965, Gordon Moore (co-founder of Intel) conjectured that the number of transistors on a chip will double roughly every one to two years. Initially, the number of transistors was doubling every year. Gradually, the rate slowed down to 18 months, and now it is about two years.

The Moore's law has approximately held true since it was proposed in the mid-sixties. Nowadays, almost every two years, the dimensions of transistors shrink by a factor of $\sqrt{2}$. This ensures that the area of a transistor shrinks by a factor of 2, and thus it is possible to fit twice the number of transistors on a chip. Let us define the *feature size* as the size of the smallest structure that can be fabricated on a chip. Table 12.1 shows the feature sizes of Intel processors over the last 10 years. We observe that the feature size decreases by a factor of roughly $\sqrt{2}$ (1.41) every two years. This results in a doubling of the number of transistors.

Year	Feature Size
2001	130 nm
2003	90 nm
2005	65 nm
2007	45 nm
2009	32 nm
2011	22 nm

Table 12.1: Feature sizes between 2001 and 2012

12.1.2 Implications of the Moore's Law

Note that the Moore's law is an empirical law. Owing to the fact that it has predicted trends correctly for the last forty years, it is widely quoted in technical literature. It directly predicts a miniaturization in the transistor size. A smaller transistor is more power efficient and faster. Designers were traditionally using these benefits to design bigger processors with extra transistors. They were using the additional transistor budget to add complexity to different units, increase cache sizes, increase the issue width, and the number of functional units. Secondly, the number of pipeline stages were also steadily increasing till about 2002, and there was an accompanying increase in the clock frequency also. However, after 2002 there was a radical change in the world of computer architecture. Suddenly, power and temperature became major concerns. The processor power consumption figures started to exceed 100 W, and on chip temperatures started to exceed $100°C$. These constraints effectively put an end to the scaling in complexity, and clock frequencies of processors.

Instead, designers started to pack more cores per chip without changing its basic design. This ensured that the number of transistors per core remained constant, and according to Moore's law the number of cores doubled once every two years. This started the era of multicore processors, and processor vendors started doubling the number of cores on chip. As of 2012, we have processors that have 8-10 cores per chip. The number of cores per chip are expected to reach 32 or 64 in the next 5 to 10 years (by 2020). A large multiprocessor today has multiple cores per chip, and multiple chips per system. For example, your author is at the moment writing this book on a 32 core server. It has 4 chips, and each chip has 8 cores.

Along with regular multicore processors, there has been another important development. Instead of having 4 large cores per chip, there are architectures that have 64-256 very small cores on a chip such as graphics processors. These processors also follow the Moore's law, and are doubling their cores every 2 years. Such processors are increasingly being used in

computer graphics, numerical and scientific computing. It is also possible to split the resources of a processor to make it support two program counters, and run two programs at the same time. These special kind of processors are known as multithreaded processors. It is not possible to cover the entire design space of multiprocessors in this book. This is the topic of a book on advanced architecture, and the reader can consult [Hwang, 2003, Hennessy and Patterson, 2012, Culler et al., 1998] for a detailed description of different kinds of multiprocessors.

In this chapter, we wish to make the reader aware of the broad trends in multiprocessor design. We shall first look at multiprocessing from the point of view of software. Once we establish the software requirements, we shall proceed to design hardware to support multiprocessing. We shall broadly consider multicore, multithreaded, and vector processors in this chapter.

12.2 Software for Multiprocessor Systems

12.2.1 Strong and Loosely Coupled Multiprocessing

Loosely Coupled Multiprocessing

There are two primary ways to harness the power of multiprocessors. The first method is to run multiple unrelated programs in parallel. For example, it is possible to run a text editor and a web browser at the same time. The text editor can run on processor 1, and the web browser can run on processor 2. Both of them can occasionally request for OS services, and connect to I/O devices. Users often use large multiprocessor systems containing more than 64-128 processors to run a set of jobs (processes) that are unrelated. For example, a user might want to conduct a weather simulation with 128 different sets of parameters. Then she can start 128 separate instances of the weather simulation software on 128 different processors on a large multiprocessor system. We thus have a speedup of 128 times as compared to a single processor system, which is significant. This paradigm is known as *loosely coupled multiprocessing*. Here, the dependences between programs is almost negligible. Note that using a multiprocessor in this manner, is not conceptually very different from using a cluster of computers that comprises completely unrelated machines that communicate over a local area network. The only difference is that the latency between machines in a multiprocessor is lower than cluster computers. A loosely coupled multiprocessor such as a cluster of PCs is also known as a *multicomputer*.

Definition 118
A multicomputer *consists of a set of computers typically connected over the network. It is capable of running a set of programs in parallel, where the programs do not share their memory space with each other.*

Strongly Coupled Multiprocessing

However, the real benefit of a multiprocessor is accrued when there is a strong degree of overlap between different programs. This paradigm is known as *strongly coupled multiprocessing*. Here programs can share their memory space, file and network connections. This method of using

multiprocessors harnesses their true power, and helps us speed up a large amount of existing software. The design and programming of strongly coupled multiprocessors is a very rich field, and is expected to grow significantly over the coming decade.

Definition 119

Loosely Coupled Multiprocessing *Running multiple unrelated programs in parallel on a multiprocessor is known as* loosely coupled multiprocessing.

Strongly Coupled Multiprocessing *Running a set of programs in parallel that share their memory space, data, code, file, and network connections is known as* strongly coupled multiprocessing.

In this book, we shall mainly look at strongly coupled multiprocessing, and primarily focus on systems that allow a set of programs to run co-operatively by sharing a large amount of data and code.

12.2.2 Shared Memory vs Message Passing

Let us now explain the methods of programming multiprocessors. For ease of explanation, let us draw an analogy here. Consider a group of workers in a factory. They co-operatively perform a task by communicating with each other orally. A supervisor often issues commands to the group of workers, and then they perform their work. If there is a problem, a worker indicates it by raising an alarm. Immediately, other workers rush to his assistance. In this small and simple setting, all the workers can hear each other, and see each other's actions. This proximity enables them to accomplish complex tasks.

We can alternatively consider another model, where workers cannot necessarily see or hear each other. In this case, they need to communicate with each other through a system of messages. Messages can be passed through letters, phone calls, or emails. In this setting, if a worker discovers a problem, he needs to send a message to his supervisor such that she can come and rectify the problem. Workers need to be typically aware of each other's identities, and explicitly send messages to all or a subset of them. It is not possible anymore to shout loudly, and communicate with everybody at the same time. However, there are some advantages of this system. We can support many more workers because they do not have to be co-located. Secondly, since there are no constraints on the location of workers, they can be located at different parts of the world, and be doing very different things. This system is thus far more flexible, and scalable.

Inspired by these real life scenarios, computer architects have designed a set of protocols for multiprocessors following different paradigms. The first paradigm is known as *shared memory*, where all the individual programs see the same view of the memory system. If program A changes the value of x to 5, then program B immediately sees the change. The second setting is known as *message passing*. Here multiple programs communicate among each other by passing messages. The shared memory paradigm is more suitable for strongly coupled multiprocessors,

and the message passing paradigm is more suitable for loosely coupled multiprocessors. Note that it is possible to implement message passing on a strongly coupled multiprocessor. Likewise, it is also possible to implement an abstraction of a shared memory on an otherwise loosely coupled multiprocessor. This is known as *distributed shared memory* [Keleher et al., 1994]. However, this is typically not the norm.

Shared Memory

Let us try to add n numbers in parallel using a multiprocessor. The code for it is shown in Example 149. We have written the code in C++ using the OpenMP language extension.

Example 149
Write a shared memory program to add a set of numbers in parallel.

Answer: *Let us assume that all the numbers are already stored in an array called numbers. The array numbers has $SIZE$ entries. Assume that the number of parallel sub-programs that can be launched is equal to N.*

```
/* variable declaration */
int partialSums[N];
int numbers[SIZE];
int result = 0;

/* initialize arrays */
...

/* parallel section */
#pragma omp parallel {
    /* get my processor id */
    int myId = omp_get_thread_num();

    /* add my portion of numbers */
    int startIdx = myId * SIZE/N;
    int endIdx = startIdx + SIZE/N;
    for(int jdx = startIdx; jdx < endIdx; jdx++)
        partialSums[myId] += numbers[jdx];
}

/* sequential section */
for(int idx=0; idx < N; idx++)
    result += partialSums[idx];
```

It is easy to mistake the code for a regular sequential program, except for the directive

#pragma omp parallel. This is the only extra semantic difference that we have added in our parallel program. It launches each iteration of this loop as a separate sub-program. Each such sub-program is known as a *thread*. A thread is defined as a sub-program that shares its address space with other threads. It communicates with them by modifying the values of memory locations in the shared memory space. Each thread has its own set of local variables that are not accessible to other threads.

The number of iterations, or the number of parallel threads that get launched is a system parameter that is set in advance. It is typically equal to the number of processors. In this case, it is equal to N. Thus, N copies of the parallel part of the code are launched in parallel. Each copy runs on a separate processor. Note that each of these copies of the program can access all the variables that have been declared before the invocation of the parallel section. For example, they can access *partialSums*, and the *numbers* arrays. Each processor invokes the function *omp_get_thread_num*, which returns the id of the thread. Each thread uses the thread id to find the range of the array that it needs to add. It adds all the entries in the relevant portion of the array, and saves the result in its corresponding entry in the *partialSums* array. Once all the threads have completed their job, the sequential section begins. This piece of sequential code can run on any processor. This decision is made dynamically at runtime by the operating system, or the parallel programming framework. To obtain the final result it is necessary to add all the partial sums in the sequential section.

Definition 120
A thread is a sub-program that shares its address space with other threads. It has a dedicated program counter, and a local stack that it can use to define its local variables. We refer to a thread as a software thread *to distinguish it from a hardware thread that we shall define later.*

A graphical representation of the computation is shown in Figure 12.3. A parent thread spawns a set of child threads. They do their own work, and finally *join* when they are done. The parent thread takes over, and aggregates the partial results.

There are several salient points to note here. The first is that each thread has its separate stack. A thread can use its stack to declare its local variables. Once it finishes, all the local variables in its stack are destroyed. To communicate data between the parent thread and the child threads, it is necessary to use variables that are accessible to both the threads. These variables need to be globally accessible by all the threads. The child threads can freely modify these variables, and even use them to communicate among each other also. They are additionally free to invoke the operating system, and write to external files and network devices. Once, all the threads have finished executing, they perform a *join* operation, and free their state. The parent thread takes over, and finishes the role of aggregating the results. Here, *join* is an example of a *synchronization operation* between threads. There can be many other types of synchronization operations between threads. The reader is referred to [Culler et al., 1998] for a detailed discussion on thread synchronization. All that the reader needs to understand is that there are a set of complicated constructs that threads can use to perform very complex tasks co-operatively. Adding a set of numbers is a very simple example. Multithreaded programs can

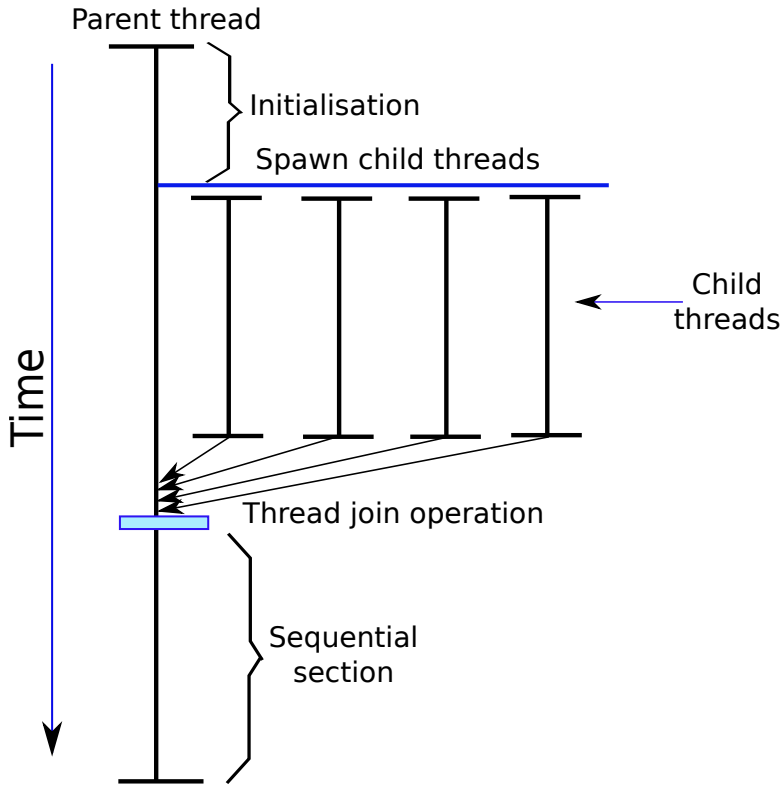


Figure 12.3: Graphical representation of the program to add numbers in parallel

be used to perform other complicated tasks such as matrix algebra, and even solve differential equations in parallel.

Message Passing

Let us now briefly look at message passing. Note that message passing based loosely coupled systems are not the main focus area of this book. Hence, we shall just give the reader a flavor of message passing programs. Note that in this case, each program is a separate entity and does not share code, or data with other programs. It is a *process*, where a process is defined as a running instance of a program. Typically, it does not share its address space with any other process.

Definition 121
A process *represents the running instance of a program. Typically, it does not share its address space with any other process.*

Let us now quickly define our message passing semantics. We shall primarily use two

functions, *send* and *receive* as shown in Table 12.2. The $send(pid, val)$ function is used to send an integer (val) to the process whose id is equal to pid. The $receive(pid)$ is used to receive an integer sent by a process whose id is equal to pid. If pid is equal to ANYSOURCE, then the *receive* function can return with the value sent by any process. Our semantics is on the lines of the popular parallel programming framework, MPI (Message Passing Interface) [Snir et al., 1995]. MPI calls have many more arguments, and their syntax is much more complicated than our simplistic framework. Let us now consider the same example of adding n numbers in parallel in Example 150.

Function	Semantics
send (pid, val)	Send the integer, *val*, to the process with an id equal to *pid*
receive (pid)	(1) Receive an integer from process pid (2) The function blocks till it gets the value (3) If the pid is equal to ANYSOURCE, then the *receive* function returns with the value sent by any process

Table 12.2: *send* and *receive* calls

Example 150
Write a message passing based program to add a set of numbers in parallel. Make appropriate assumptions.

Answer: *Let us assume that all the numbers are stored in the array, numbers, and this array is available with all the N processors. Let the number of elements in the numbers array be $SIZE$. For the sake of simplicity, let us assume that $SIZE$ is divisible by N.*

```
/* start all the parallel processes */
SpawnAllParallelProcesses();

/* For each process execute the following code */
int myId = getMyProcessId();

/* compute the partial sums */
int startIdx = myId * SIZE/N;
int endIdx = startIdx + SIZE/N;
int partialSum = 0;
for(int jdx = startIdx; jdx < endIdx; jdx++)
                partialSum += numbers[jdx];

/* All the non-root nodes send their partial sums to the root */
```

```
if(myId != 0) {
        /* send the partial sum to the root */
        send (0, partialSum);
} else {
        /* for the root */
        int sum = partialSum;
        for (int pid = 1; pid < N; pid++) {
                sum += receive(ANYSOURCE);
        }

        /* shut down all the processes */
        shutDownAllProcesses();

        /* return the sum */
        return sum;
}
```

12.2.3 Amdahl's Law

We have now taken a look at examples for adding a set of n numbers in parallel using both the paradigms namely shared memory and message passing. We divided our program into two parts – a sequential part and a parallel part (refer to Figure 12.3). In the parallel part of the execution, each thread completed the work assigned to it, and created a partial result. In the sequential part, the root or master or parent thread initialized all the variables and data structures, and spawned all the child threads. After all the child threads completed (or joined), the parent thread aggregated the results produced by all the child threads. This process of aggregating results is also known as *reduction*. The process of initializing variables, and reduction, are both sequential.

Let us now try to derive the speedup of a parallel program vis-a-vis its sequential counterpart. Let us consider a program that takes T_{seq} units of time to execute. Let f_{seq} be the fraction of time that it spends in its sequential part, and $1 - f_{seq}$ be the fraction of time that it spends in its parallel part. The sequential part is unaffected by parallelism; however, the parallel part gets equally divided among the processors. If we consider a system of P processors, then the parallel part is expected to be sped up by a factor of P. Thus, the time (T_{par}) that the parallel version of the program takes is equal to:

$$T_{par} = T_{seq} \times \left(f_{seq} + \frac{1 - f_{seq}}{P} \right) \tag{12.1}$$

Alternatively, the speedup (S) is given by:

$$S = \frac{T_{seq}}{T_{par}} = \frac{1}{f_{seq} + \frac{1-f_{seq}}{P}} \tag{12.2}$$

Equation 12.2 is known as the Amdahl's Law. It is a theoretical estimate (or rather the upper bound in most cases) of the speedup that we expect with additional parallelism.

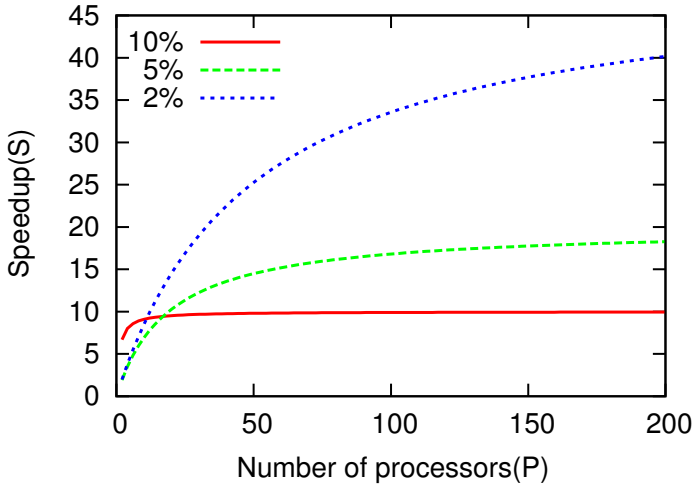


Figure 12.4: Speedup (S) vs number of processors (P)

Figure 12.4 plots the speedups as predicted by Amdahl's Law for three values of f_{seq} (10%, 5%, and 2%). We observe that with an increasing number of processors the speedup gradually saturates and tends to the limiting value, $1/f_{seq}$. We observe diminishing returns as we increase the number of processors beyond a certain point. For example, for $f_{seq} = 5\%$, there is no appreciable difference in speedups between a system with 35 processors, and a system with 200 processors. We approach similar limits for all three values of f_{seq}. The important point to note here is that increasing speedups by adding additional processors has its limits. We cannot expect to keep getting speedups indefinitely by adding more processors, because we are limited by the length of the sequential sections in programs.

To summarize, we can draw two inferences. The first is that to speed up a program it is necessary to have as much parallelism as possible. Hence, we need to have a very efficient parallel programming library, and parallel hardware. However, parallelism has its limits, and it is not possible to increase the speedup appreciably beyond a certain limit. The speedup is limited by the length of the sequential section in the program. To reduce the sequential section, we need to adopt approaches both at the algorithmic level, and at the system level. We need to design our algorithms in such a way that the sequential section is as short as possible. For example, in Examples 149, and 150, we can also perform the initialization in parallel (reduces the length of the sequential section). Secondly, we need a fast processor that can minimize the time it takes to execute the sequential section.

We looked at the latter requirement (designing fast processors) in Chapters 9, 10, and 11. Now, let us look at designing fast and power efficient hardware for the parallel section.

12.3 Design Space of Multiprocessors

Michael J. Flynn proposed the famous Flynn's classification of multiprocessors in 1966. He started out by observing that an ensemble of different processors might either share code, data, or both. There are four possible choices – SISD (single instruction single data), SIMD (single instruction multiple data), MISD (multiple instruction single data), and MIMD (multiple instruction multiple data).

Let us describe each of these types of multiprocessors in some more detail.

SISD This is a standard uniprocessor with a single pipeline as described in Chapter 9 and Chapter 10. A SISD processor can be thought of as a special case of the set of multiprocessors with just a single processor.

SIMD A SIMD processor can process multiple streams of data in a single instruction. For example, a SIMD instruction can add 4 sets of numbers with a single instruction. Modern processors incorporate SIMD instructions in their instruction set, and have special SIMD execution units also. Examples include x86 processors that contain the SSE set of SIMD instruction sets. Graphics processors, and vector processors are special examples of highly successful SIMD processors.

MISD MISD systems are very rare in practice. They are mostly used in systems that have very high reliability requirements. For example, large commercial aircraft typically have multiple processors running different versions of the same program. The final outcome is decided by voting. For example, a plane might have a MIPS processor, an ARM processor, and an x86 processor, each running different versions of the same program such as an autopilot system. Here, we have multiple instruction streams, yet a single source of data. A dedicated voting circuit computes a majority vote of the three outputs. For example, it is possible that because of a bug in the program or the processor, one of the systems can erroneously take a decision to turn left. However, both of the other systems might take the correct decision to turn right. In this case, the voting circuit will decide to turn right. Since MISD systems are hardly ever used in practice other than such specialized examples, we shall not discuss them anymore in this book.

MIMD MIMD systems are by far the most prevalent multiprocessor systems today. They have multiple instruction streams and multiple data streams. Multicore processors, and large servers are all MIMD systems. Examples 149 and 150 pertained to MIMD systems. We need to carefully explain the meaning of multiple instruction streams. This means that instructions come from multiple sources. Each source has its unique location, and associated program counter. Two important branches of the MIMD paradigm have formed over the last few years.

The first is *SPMD* (single program multiple data), and the second is *MPMD* (multiple program multiple data). Most parallel programs are written in the SPMD style (Example 149 and 150). Here, multiple copies of the same program run on different cores, or separate processors. However, each individual processing unit has a separate program counter, and thus perceives a different instruction stream. Sometimes SPMD programs are written in such a way that they perform different actions depending on their thread

ids. We saw a method in Example 149 on how to achieve this using OpenMP functions. The advantage of SPMD is that we do not have to write different programs for different processors. Parts of the same program can run on all the processors, though their behavior might be different.

A contrasting paradigm is MPMD. Here, the programs running on different processors are actually different. They are more useful for specialized processors that have heterogeneous processing units. There is typically a single master program that assigns work to slave programs. The slave programs complete the quanta of work assigned to them, and then return the results to the master program. The nature of work of both the programs is actually very different, and it is often not possible to seamlessly combine them into one program.

From the above description, it is clear that the systems that we need to focus on are SIMD and MIMD. MISD systems are very rarely used, and thus will not be discussed anymore. Let us first discuss MIMD multiprocessing. Note that we shall only describe the SPMD variant of MIMD multiprocessing because it is the most common approach.

12.4 MIMD Multiprocessors

Let us now take a deeper look at strongly-coupled shared memory based MIMD machines. We shall first take a look at them from the point of view of software. After we have worked out a broad specification of these machines from the point of view of software, we can proceed to give a brief overview of the design of the hardware. Note that the design of parallel MIMD machines can take an entire book to describe. For additional information, or for added clarity, the reader can refer to the following references [Culler et al., 1998, Sorin et al., 2011].

Let us call the software interface of a shared memory MIMD machine as the "logical point of view", and refer to the actual physical design of the multiprocessor as the "physical point of view". When we describe the logical point of view, we are primarily interested in how the multiprocessor behaves with respect to software. What guarantees does the hardware make regarding its behavior, and what can software expect? This includes correctness, performance, and even resilience to failures. The physical point of view is concerned with the actual design of the multiprocessor. This includes the physical design of the processors, the memory system, and the interconnection network. Note that the physical point of view has to conform to the logical point of view. The reader will recall that we are taking a similar approach here as we did for uniprocessors. We first explained the software view (architecture) by looking at assembly code. Then we provided an implementation for the assembly code by describing a pipelined processor (organization). We shall follow a similar approach here.

12.4.1 Logical Point of View

Figure 12.5 shows a logical view of a shared memory MIMD multiprocessor. Each processor is connected to the memory system that saves both code and data. The program counter of each processor points to the location of the instruction that it is executing. This is in the code section of memory. This section is typically read only, and thus is not affected by the fact that we have multiprocessors.

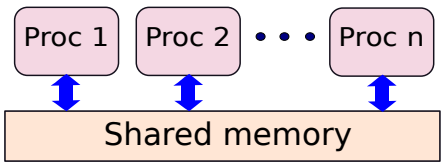


Figure 12.5: Logical view of a multiprocessor system

The main challenge in implementing a shared memory multiprocessor is in correctly handling data accesses. Figure 12.5 shows a scheme in which each computing processor is connected to the memory, and it is treated as a black box. If we are considering a system of processes with different virtual address spaces, then there is no problem. Each processor can work on its private copy of data. Since the memory footprints are effectively disjoint, we can easily run a set of parallel processes in this system. However, the main complexity arises when we are looking at shared memory programs that have multiple threads, and there is data sharing across threads. Note that we can also share memory across processes by mapping different virtual pages to the same physical frame as described in Section 11.4.6. We shall treat this scenario as a special case of parallel multi-threaded software.

A set of parallel threads typically share their virtual and physical address spaces. However, threads do have private data also, which is saved in their stacks. There are two methods to implement disjoint stacks. The first is that all the threads can have identical virtual address spaces, and different stack pointers can start at different points in the virtual address space. We need to further ensure that the size of the stack of a thread is not large enough to overlap with the stack of another thread. Another approach is to map the stack portion of the virtual address space of different threads to different memory frames. Thus, each thread can have different entries in its page table for the stack portion, yet have common entries for the rest of the sections of the virtual address space such as code, read-only data, constants, and heap variables.

In any case, the main problems of complexity of parallel software are not because of code that is read-only, or local variables that are not shared across threads. The main problem is due to data values that are potentially shared across multiple threads. This is what gives the power to parallel programs, and also makes them very complex. In the example that we showed for adding a set of numbers in parallel, we can clearly see the advantage that we obtain by sharing values and results of computation through shared memory.

However, sharing values across threads is not that simple. It is actually a rather profound topic, and advanced texts on computer architecture[Sarangi,] devote several chapters to this topic. We shall briefly look at two important topics in this area namely coherence and memory consistency. *Coherence* is also known as cache coherence, when we refer to it in the context of caches. However, the reader needs to be aware that coherence is just not limited to caches, it is a generic term.

12.4.2 Coherence

The term *coherence* in the memory system refers to the way multiple threads access the same location. We shall see that diverse behaviors are possible, when multiple threads access the same memory location. Some of the behaviors are intuitively wrong, yet possible. Before looking at coherence, we need to note that inside the memory system, we have many entities such as caches, write buffers, and different kinds of temporary buffers. Processors typically write values to temporary buffers, and resume their operation. It is the job of the memory system to transfer the data from these buffers to locations in the cache subsystem. Now, we need to understand that large caches are very slow. Hence, it is common practice to associate a small private cache with each core in a multicore system. For example, each core may have a small, private L1 cache. The ensemble of these L1 caches (across cores) needs to act like a single L1 cache. In other words, an external entity such as a software program should perceive the distributed L1 cache comprising multiple private caches as one entity (insofar as memory accesses are concerned).

It is thus possible that internally, a given memory address might be associated with many physical locations at a given point of time. For example, a variable x can have many replicas that are distributed across the private caches. Any write operation has to update all the replicas. Synchronizing these replicas is the key problem that cache coherence solves. Secondly, the process of transferring data from the processor to the correct location in the memory system (typically a cache block) is not instantaneous. It sometimes takes more than tens of cycles for the memory read or write request to reach its location. Sometimes these memory request messages can wait even longer, if there is a lot of memory traffic. Messages can also get reordered with other messages that were sent after them. This internal complexity of a multiprocessor memory system leads to several interesting behaviors for programs that access the same set of shared variables. Let us consider a set of examples.

In each of these examples, all shared values are initialized to 0. All the local variables start with t such as $t1$, $t2$, and $t3$. They are stored in registers. Let us say that thread 1 writes to a variable x that is shared across threads. Immediately later, thread 2 tries to read its value.

```
Thread 1:          Thread 2:
x = 1              t1 = x
```

Is thread 2 guaranteed to read 1? Or, can it get the previous value 0? What if thread 2, reads the value of x, 2 ns later, or even 10 ns later? What is the time that it takes for a write in one thread to propagate to the other threads? This depends on the implementation of the memory system. If a memory system has fast buses, and fast caches, then a write can propagate very quickly to other threads. However, if the buses and caches are slow, then it can take some time for other threads to see a write to a shared variable.

Now, let us further complicate the example. Let us assume that thread 1 writes to x twice.

Example 151

```
Thread 1:          Thread 2:
x = 1              t1 = x
x = 2              t2 = x
```

Let us now look at the set of possible outcomes. $(t1, t2) = (1, 2)$ is possible. $(t1, t2) = (0, 1)$ is also possible. This is possible when $t1$ was read before thread 1 started, and $t2$ was read after the first statement of thread 1 completed. Likewise, we can systematically enumerate the set of all possible outcomes, which are as follows: (0,0), (0,1), (0,2), (1,1), (1,2), (2,2). The reader is requested to write a simple program using a parallel multithreaded framework such as OpenMP or *pthreads* and look at the set of possible outcomes. The interesting question is whether the outcome (2,1) is possible? This might be possible if somehow the first write to x got delayed in the memory system, and the second write overtook it. The question is whether we should allow such behavior.

The answer is NO. If we were to allow such behavior, then implementing a multiprocessor memory system would undoubtedly become simpler. However, it will become very difficult to write and reason about parallel programs. Hence, most multiprocessor systems disallow such behavior.

Let us now look at the issue of accesses to the same memory location by multiple threads slightly more formally. Let us define the term, *coherence*, as the behavior of memory accesses to the same memory address (such as x in our examples). We ideally want our memory system to be coherent. This basically means that it should observe a set of rules while dealing with different accesses to the same memory address such that it is easier to write programs.

Definition 122
The behavior of memory accesses to the same memory address is known as coherence.

Typically, coherence has two axioms. These are as follows:

1. **Completion** A write must ultimately complete.
2. **Order** All the writes to the same memory address need to be seen by all the threads in the same order.

Both of these axioms are fairly sublime in nature. The completion axiom says that no write is ever lost in the memory system. For example, it is not possible that we write 10 to variable x, and the write request gets dropped by the memory system. It needs to reach the memory location(s) corresponding to x, and then it needs to update its value. It might get overwritten later by another write request. However, the bottom line is that the write request needs to update the memory location at some point of time in the future.

The order axiom says that all the writes to a memory location are perceived to be in the same order by all the threads. This means that it is not possible to read (2,1) in Example 151. Let us now explain the reasons for this. Thread 1 is aware that 2 was written after 1 to the memory location x. By the second axiom of coherence, all other threads need to perceive the same order of writes to x. Their view of x cannot be different from that of thread 1. Hence, they cannot read 2 after 1. If we think about it, the axioms of coherence make intuitive sense. They basically mean that all writes eventually complete, as is true for uniprocessor systems. Second, all the processors see the same view of a single memory location. If its value changes from 0 to 1 to 2, then all the processors see the same order of changes ($0 \rightarrow 1 \rightarrow 2$). No processor sees the updates in a different order. This further means that irrespective of how a memory system is implemented internally, externally each memory location is seen as a globally accessible single location. In terms of theory, this is just the tip of the iceberg. The advanced text on computer architecture by your author [Sarangi,] has more details.

12.4.3 Memory Consistency

Overview

Coherence was all about accesses to the same memory location. What about access to different memory locations? Let us explain with a series of examples.

Example 152

```
Thread 1:          Thread 2:
x = 1              t1 = y
y = 1              t2 = x
```

Let us look at the permissible values of $t1$, and $t2$ from an intuitive standpoint. We can always read (t1,t2)=(0,0). This can happen when thread 2 is scheduled before thread 1. We can also read $(t1, t2)$=(1,1). This will happen when thread 2 is scheduled after thread 1 finishes. Likewise, it is possible to read $(t1, t2)$=(0,1). Figure 12.6 shows how we can get all the three outcomes.

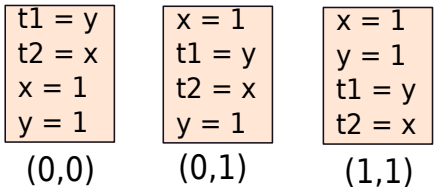


Figure 12.6: Graphical representation of all the possible outcomes

The interesting question is whether $(t1, t2)$=(1,0) is allowed? This will happen when the

write to x is somehow delayed by the memory system, whereas the write to y completes quickly. In this case $t1$ will get the updated value of y, and $t2$ will get the old value of x. The question is whether such kind of behavior should be allowed. Clearly if such kind of behavior is allowed, it will become hard to reason about software, and the correctness of parallel algorithms. It will also become hard to program because the behavior appears non-intuitive. However, if we allow such behavior then our hardware design becomes simpler because we do not have to provide strong guarantees to software.

There is clearly no right or wrong answer? It all depends on how we want to program software, and what hardware designers want to build for software writers. But, still there is something very profound about this example, and the special case of $(t1, t2)$ equal to (1,0). To find out why, let us take a look again at Figure 12.6. In this figure, we have been able to reason about three outcomes by creating an interleaving between the instructions of the two threads. In each of these interleavings, the order of instructions in the same thread is the same as the way it is specified in the program. This is known as *program order*.

Definition 123
An order of instructions (possibly belonging to multiple threads) that is consistent with the control-flow semantics of each constituent thread is said to be in program order. The control-flow semantics of a thread is defined as the set of rules that determine which instructions can execute after a given instruction. For example, the set of instructions executed by a single cycle processor is always in program order.

Observation: It is clear that we cannot generate the outcome $(t1, t2)$=(1,0) by interleaving threads in program order.

It would be nice if we can somehow exclude the output (1,0) from the set of possible (allowed) outputs. It will allow us to write parallel software, where we can predict the possible outcomes very easily. A model of the memory system that determines the set of possible outcomes for parallel programs is known as a *memory model*.

Definition 124
The model of a memory system that determines the set of likely outcomes for parallel programs is known as a memory model.

Sequential Consistency

We can have different kinds of memory models corresponding to different kinds of processors. One of the most important memory models is known as *sequential consistency*(SC). Sequential consistency states that only those outcomes are allowed that can be generated by an interleaving of threads in program order. This means that all the outcomes shown in Figure 12.6 are allowed because they are generated by interleaving thread 1 and 2 in all possible ways, without

violating their program order. However, the outcome $(t1, t2)$=(1,0) is not allowed because it violates program order. Hence, it is not allowed in a sequentially consistent memory model. Note that once we interleave multiple threads in program order, it is the same as saying that we have one processor that executes an instruction from one thread in one cycle and possibly another instruction from some other thread in the next cycle. However, the program order (control flow semantics) of each thread is preserved. Hence, a uniprocessor processing multiple threads produces an SC execution. In fact, if we think about the name of the model, the word "sequential" comes from the notion that the execution is equivalent to a uniprocessor *sequentially* executing the instructions of all the threads in some order that is consistent with each thread's program order.

Definition 125
A memory model is sequentially consistent *if the outcome of the execution of a set of parallel threads is equivalent to that of a single processor executing instructions from all the threads in some order. Alternatively, we can define sequential consistency as a memory model whose set of possible outcomes are those that can be generated by interleaving a set of threads in program order.*

Sequential consistency is a very important concept and is widely studied in the fields of computer architecture and distributed systems. It reduces a parallel system to an equivalent uniprocessor system with one processor by equating the execution on a parallel system with the execution on a sequential system. An important point to note is that SC does not mean that the outcome of the execution of a set of parallel programs is the same all the time. This depends on the way that the threads are interleaved and the relative delays f the threads. All that it says is that certain outcomes are not allowed.

Weak Consistency (WC)*

The implementation of SC comes at a cost. It makes software simple, but it makes hardware very slow. To support SC, it is often necessary to wait for a read or write to complete, before the next read or write can be sent to the memory system. A write request W *completes* when all subsequent reads by any processor are guaranteed to get the value written by W, or the value written by a later write to the same location. A read request completes, after it reads the data, and the write request that originally wrote the data completes.

These requirements/restrictions become a bottleneck in high-performance systems. Hence, the computer architecture community has moved to *weak memory models* that violate SC. A weak memory model will allow the outcome $(t1, t2)$=(1,0) in the following multithreaded code snippet.

```
Thread 1:          Thread 2:
x = 1              t1 = y
y = 1              t2 = x
```

Definition 126
A weakly consistent (WC) memory model does not obey SC. It typically allows arbitrary memory orderings.

There are different kinds of weak memory models. Let us look at a generic variant, and call it *weak consistency* (WC). Let us now try to find out why WC allows the (1,0) outcome. Assume that thread 1 is running on core 1, and thread 2 is running on core 2. Moreover, assume that the memory location corresponding to x is near core 2, and the memory location corresponding to y is near core 1. Also assume that it takes tens of cycles to send a request from the vicinity of core 1 to core 2, and the delay is variable. Let us first investigate the behavior of the pipeline of core 1. From the point of view of the pipeline of core 1, once a memory write request is handed over to the memory system, the memory write instruction is deemed to have finished. The instruction moves on to the RW stage. Hence, in this case, the processor will hand over the write to x to the memory system in the n^{th} cycle, and subsequently pass on the write to y in the $(n+1)^{th}$ cycle. The write to y will reach the memory location of y quickly, while the write to x will take a long time.

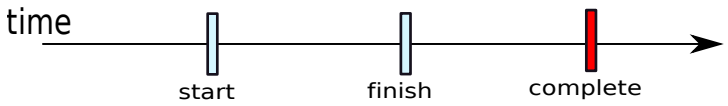
In the meanwhile, core 2 (running thread 2) will try to read the value of y. Assume that the read request arrives at the memory location of y just after the write request (to y) reaches it. Thus, we will get the new value of y, which is equal to 1. Subsequently, core 2 will issue a read to x. It is possible that the read to x reaches the memory location of x just before the write to x reaches it. In this case, it will fetch the old value of x, which is 0. Thus, the outcome (1,0) is possible in a weak memory model.

Now, to avoid this situation, we could have waited for the write to x to complete fully, before issuing the write request to y. It is true that in this case, this would have been the right thing to do. However, in general, when we are writing to shared memory locations, other threads are not reading them at exactly the same point of time. We have no way of distinguishing both the situations at runtime since processors do not share their memory access patterns between each other. Hence, in the interest of performance, it is not worthwhile to delay every memory request till the previous memory requests complete. High-performance implementations thus prefer memory models that allow memory accesses from the same thread to be reordered by the memory system. Note that we shall investigate ways of avoiding the (1,0) outcome in the next subsection.

Let us summarize our discussion that we have had on weak memory models by defining the assumptions that most processors make. Most processors assume that a memory request completes instantaneously at some point of time after it leaves the pipeline. Furthermore, all the threads assume that a memory request completes instantaneously at exactly the same point of time. This property of a memory request is known as *atomicity*. Second, we need to note that the order of completion of memory requests might differ from their program order. When the order of completion is the same as the program order of each thread, the memory model obeys SC. If the completion order is different from the program order, then the memory model is a variant of WC.

Definition 127
A memory request is said to be atomic *or observe* atomicity*, when it is perceived to execute instantaneously by all threads at some point of time after it is issued.*

Important Point 18
To be precise, for every memory request, there are three events of interest namely start*,* finish*, and* completion*. Let us consider a write request. The request* starts *when the instruction sends the request to the L1 cache in the MA stage. The request* finishes *when the instruction moves to the RW stage. In modern processors, there is no guarantee that the write would have reached the target memory location when the memory request* finishes*. The point of time at which the write request reaches the memory location and the write is visible to all the processors, is known as the time of* completion*. In simple processors, the time of completion of a request is in between the start and finish times. However, in high-performance processors, this is seldom the case. This concept is shown in the following illustration.*



What about a read request? Most readers will naively assume that the completion time of a read is between the start and finish times, because it needs to return with the value of the memory location. This is however not strictly true in complex multiprocessors where reads and writes issued by the same core can be reordered if they access different memory locations. Things actually get very complicated when we consider everything that can possibly happen. We also have a bunch of memory models where writes themselves are not atomic – this means that they appear to complete at different points of time to different threads. This further complicates matters.

Trivia 4
Here, is an incident from your author's life. He had 200 US dollars in his bank account. He had gotten a check for 300$ from his friend. He went to his bank's nearest ATM and deposited the check. Three days later, he decided to pay his rent (400$). He wrote a check to his landlord and sent it to his postal address. A day later, he got an angry phone call from his landlord informing him that his check had bounced. How was this possible?

Your author then inquired. It had so happened that because of a snow storm, his bank was not able to send people to collect checks from the ATM. Hence, when his landlord deposited the check, the bank account did not have sufficient money.

This example is related with the problem of memory consistency. Your author leaving his house to drop the check in the ATM is the start time. *He* finished *the job when he dropped the check in the ATM's drop box. However, the* completion time *was 5 days later, when the amount was actually credited to his account. Concurrently, another thread (his landlord) deposited his rent check, and it bounced. This is an example of* weak consistency *in real life.*

There is an important point to note here. In a weak memory model, the ordering between independent memory operations in the same thread is not respected. For example, when we wrote to x, and then to y, thread 2 perceived them to be in the reverse order. However, the ordering of operations of memory instructions to the same address and belonging to the same thread is always respected. For example, if we set the value of a variable x to 1, and later read it in the same thread, we will either get 1 or the value written by a later write to x. All the other threads will perceive the memory requests to be in the same order. Such accesses (to the same variable in the same thread) are not reordered in any memory model that your author is aware of (refer to Figure 12.7). It is possible to prove that doing so breaks the notion of a shared memory altogether (refer to[Sarangi,]).

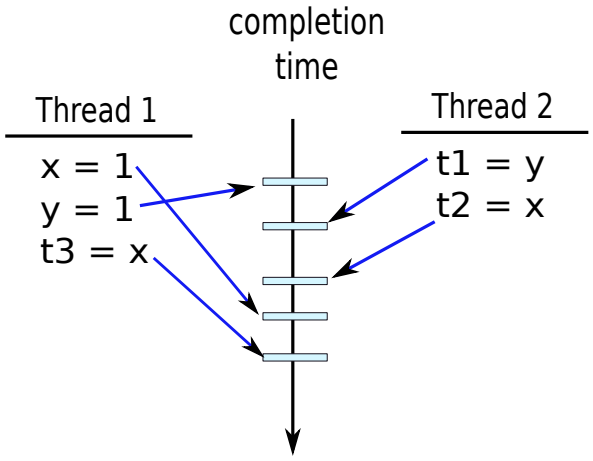


Figure 12.7: Actual completion times of memory requests in a multithreaded program

Examples

Let us now illustrate the difficulty with using a weak memory model that does not obey any ordering rules. Let us write our program to add numbers in parallel assuming a sequentially consistent system. Note that here we do not use OpenMP because OpenMP does a lot behind

the scenes to ensure that programs run correctly on machines with weak memory models. Let us define a *parallel* construct that runs a block of code in parallel, and a $getThreadId()$ function that returns the identifier of the thread. The range of the thread ids is from 0 to $N-1$. The code for the parallel *add* function is shown in Example 153. We assume that before the parallel section begins, all the arrays are initialized to 0. In the parallel section, each thread adds its portion of numbers, and writes the result to its corresponding entry in the array, *partialSums*. Once, it is done, it sets its entry in the $finished$ array to 1.

Let us now consider the thread that needs to aggregate the results. It needs to wait for all the threads to finish the job of computing the partial sums. It does this by waiting till all the entries of the $finished$ array are equal to 1. Once, it establishes that all the entries in the $finished$ array are equal to 1, it proceeds to add all the partial sums to get the final result. The reader can readily verify that if we assume a sequentially consistent system, then this piece of code executes correctly. She needs to note that we compute the result, only when we read all the entries in the array $finished$ to be 1. An entry in the $finished$ array is equal to 1 if the partial sum is computed, and written to the *partialSums* array. Since we add the elements of the *partialSums* array to compute the final result, we can conclude that it is calculated correctly. Note that this is not a formal proof (left as an exercise for the reader).

Example 153

Write a shared memory program to add a set of numbers in parallel on a sequentially consistent machine.

Answer: *Let us assume that all the numbers are already stored in an array called numbers. The array numbers has $SIZE$ entries. The number of parallel threads is given by N.*

```
/* variable declaration */
int partialSums[N];
int finished[N];
int numbers[SIZE];
int result = 0;
int doneInit = 0;

/* initialize all the elements in  partialSums and finished to 0 */
...
doneInit = 1;

/* parallel section */
parallel {
        /* wait till initialization */
        while (!doneInit()){};

        /* compute the partial sum */
        int myId =         getThreadId();
        int startIdx = myId * SIZE/N;
        int endIdx = startIdx + SIZE/N;
```

```
        for(int jdx = startIdx; jdx < endIdx; jdx++)
                partialSums[myId] += numbers[jdx];

        /* set an entry in the finished array */
        finished[myId] = 1;
}

/* wait till all the threads are done */
do {
        flag = 1;
        for (int i=0; i < N; i++){
                if(finished[i] == 0){
                        flag = 0;
                        break;
                }
        }
} while (flag == 0);

/* compute the final result */
for(int idx=0; idx < N; idx++)
        result += partialSums[idx];
```

Now, let us consider a weak memory model. We implicitly assumed in our example with sequential consistency that when the last thread reads $finished[i]$ to be 1, $partialSums[i]$ contains the value of the partial sum. However, this assumption does not hold if we assume a weak memory model because the memory system might reorder the writes to $finished[i]$ and $partialSums[i]$. It is thus possible that the write to the $finished$ array happens before the write to the $partialSums$ array in a system with a weak memory model. In this case, the fact that $finished[i]$ is equal to 1 does not guarantee that $partialSums[i]$ contains the updated value. This distinction is precisely what makes sequential consistency extremely programmer friendly.

Important Point 19
In a weak memory model, the memory accesses issued by the same thread are always perceived to be in program order by that thread. However, the order of memory accesses can be perceived differently by other threads.

Let us come back to the problem of ensuring that our example to add numbers in parallel runs correctly. We observe that the only way out of our quagmire is to have a mechanism to

ensure that the write to $partialSums[i]$ is completed before another thread reads $finished[i]$ to be 1. We can use a generic instruction known as a *fence*. This instruction ensures that all the reads and writes issued before the fence complete before any read or write after the fence begins. Trivially, we can convert a weak memory model to a sequentially consistent one by inserting a fence after every instruction. However, this can induce a large overhead. It is best to introduce a minimal number of fence instructions, as and when required. Let us look at our example for adding a set of numbers in parallel for weak memory models by adding fence instructions.

Example 154

Write a shared memory program to add a set of numbers in parallel on a machine with a weak memory model.

Answer: *Let us assume that all the numbers are already stored in an array called numbers. The array numbers has $SIZE$ entries. The number of parallel threads is given by N.*

```
/* variable declaration */
int partialSums[N];
int finished[N];
int numbers[SIZE];
int result = 0;

/* initialize all the elements in  partialSums and finished to 0 */
...

/* fence */
/* ensures that the parallel section can read the initialized arrays */
fence();

/* All the data is present in all the arrays at this point */
/* parallel section */
parallel {
        /* get the current thread id */
        int myId =         getThreadId();

        /* compute the partial sum */
        int startIdx = myId * SIZE/N;
        int endIdx = startIdx + SIZE/N;
        for(int jdx = startIdx; jdx < endIdx; jdx++)
                partialSums[myId] += numbers[jdx];

        /* fence */
        /* ensures that finished[i] is written after
           partialSums[i] */
        fence();
```

```
        /* set the value of done */
        finished[myId] = 1;
}

/* wait till all the threads are done */
do {
        flag = 1;
        for (int i=0; i < N; i++){
                if(finished[i] == 0){
                        flag = 0;
                        break;
                }
        }
} while (flag == 0) ;

/* sequential section */
for(int idx=0; idx < N; idx++)
        result += partialSums[idx];
```

Example 154 shows the code for a weak memory model. The code is more or less the same as it was for the sequentially consistent memory model. The only difference is that we have added two additional fence instructions. We assume a function called $fence()$ that internally invokes a fence instruction. We first call $fence()$ before creating all the parallel threads. This ensures that all the writes for initializing data structures have completed. After that we start the parallel threads. The parallel threads finish the process of computing and writing the partial sum, and then we invoke the fence operation again. This ensures that before $finished[myId]$ is set to 1, all the partial sums have been computed and written to their respective locations in memory. Secondly, if the last thread reads $finished[i]$ to be 1, then we can say for sure that the value of $partialSums[i]$ is up-to-date and correct. Hence, this program executes correctly, in spite of a weak memory model.

We thus observe that weak memory models do not sacrifice on correctness if the programmer is aware of them, and inserts fences at the right places. Nonetheless, it is necessary for programmers to be aware of weak memory models, and they need to also understand that a lot of subtle bugs in parallel programs occur because programmers do not take the underlying memory model into account. Weak memory models are currently used by most processors because they allow us to build high-performance memory systems. In comparison, sequential consistency is very restrictive, and other than the MIPS R10000 [Yeager, 1996] no other major vendor offers machines with sequential consistency. All our current x86 and ARM based machines use different versions of weak memory models.

12.4.4 Physical View of Memory

Overview

We have looked at two important aspects of the logical view of a memory system for multiprocessors namely coherence and consistency. We need to implement a memory system that respects both of these properties. In this section, we shall study the design space of multiprocessor memory systems, and provide an overview of the design alternatives. We shall observe that there are two ways of designing a cache for a multiprocessor memory system. The first design is called a *shared cache*, where a single cache is shared among multiple processors. The second design uses a set of *private caches*, where each processor or set of processors typically have a private cache. All the private caches co-operate to provide the illusion of a shared cache. This is known as *cache coherence*.

We shall study the design of shared caches in Section 12.4.5, and private caches in Section 12.4.6. Subsequently, we shall briefly look at ensuring memory consistency in Section 12.4.7. We shall conclude that an efficient implementation of a given consistency model such as sequential or weak consistency is difficult, and is thus a subject of study in an advanced computer architecture course. In this book, we propose a simple solution to this problem, and request the reader to look at research papers for more information. The casual reader can skip most of this section without any loss in continuity. Later on we shall summarize the main results, observations and insights.

Design of a Multiprocessor Memory System – Shared and Private Caches

Let us start out by considering the first level cache. We can give every processor its individual instruction cache. Instructions represent read only data, and typically do not change during the execution of the program. Since sharing is not an issue here, each processor can benefit from its small private instruction cache. The main problem is with the data caches. There are two possible ways to design a data cache. We can either have a shared cache, or a private cache. A shared cache is a single cache that is accessible to all the processors. A *private cache* is accessible to either only one processor, or a set of processors. It is possible to have a hierarchy of shared caches, or a hierarchy of private caches as shown in Figure 12.8. We can even have combinations of shared and private caches in the same system.

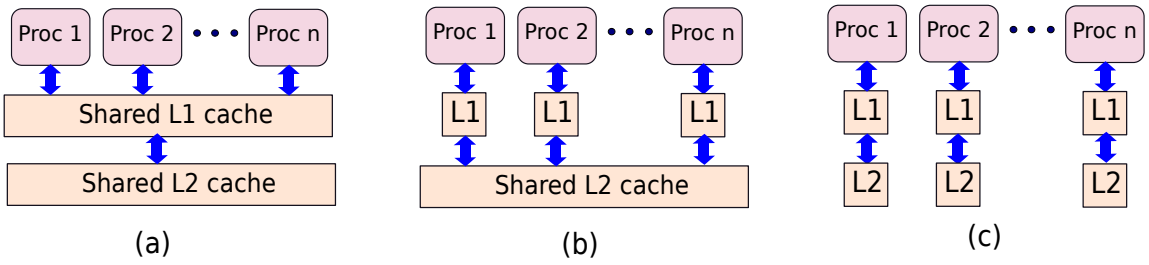


Figure 12.8: Examples of systems with shared and private caches

Let us now evaluate the trade-offs between a shared and private cache. A shared cache is accessible to all the processors, and contains a single entry for a cached memory location. The communication protocol is simple, and is like any regular cache access. The additional complexity arises mainly from the fact that we need to properly schedule the requests coming from different individual processors. However, at the cost of simplicity, a shared cache has its share of problems. To service requests coming from all the processors, a shared cache needs to have a lot of read and write ports for handling requests simultaneously. Unfortunately, the size of a cache increases approximately as a square of the number of ports [Tarjan et al., 2006]. Additionally, the shared cache needs to accommodate the working sets of all the currently running threads. Hence, shared caches tend to become very large and slow. Because of physical constraints, it becomes difficult to place a shared cache close to all the processors. In comparison, private caches are typically much smaller, service requests for fewer cores, and have a lower number of read/write ports. Hence, they can be placed close to their associated processors. A private cache is thus much faster because it can be placed closer to a processor and is also much smaller in size.

To solve the problems with shared caches, designers often use private caches, especially in the higher levels of the memory hierarchy. A private cache can only be accessed by either one processor, or a small set of processors. They are small, fast, and consume a lesser amount of power. The major problem with private caches is that they need to provide the illusion of a shared cache to the programmer. For example, let us consider a system with two processors, and a private data cache associated with each processor. If one processor writes to a memory address, x, the other processor needs to be aware of the write. However, if it only accesses its private cache, then it will never be aware of a write to address x. This means that a write to address x is lost, and thus the system is not coherent. Hence, there is a need to tie the private caches of all the processors such that they look like one unified shared cache, and observe the rules of coherence. Coherence in the context of caches, is popularly known as *cache coherence*. Maintaining cache coherence represents an additional source of complexity for private caches, and limits the scalability of this approach. It works well for small private caches. However, for larger private caches, the overhead of maintaining coherence becomes prohibitive. For large lower level caches, the shared cache is more appropriate. Secondly, there is typically some data replication across multiple private caches. This wastes space.

Definition 128
Coherence in the context of a set of private caches is known as cache coherence.

By implementing a cache coherence protocol, it is possible to convert a set of disjoint private caches to appear as a shared cache to software. Let us now outline the major trade-offs between shared and private caches in Table 12.3.

From the table it is clear that the first level cache should ideally be private because we desire low latency and high throughput. However, the lower levels need to be larger in size. They service significantly fewer requests, and thus they should comprise shared caches. Let us now describe the design of coherent private caches, and large shared caches. To keep matters simple we shall only consider a single level private cache, and not consider hierarchical private

Attribute	Private Cache	Shared Cache
Area	low	high
Speed	fast	slow
Proximity to the processor	near	far
Scalability in size	low	high
Data replication	yes	no
Complexity	high (needs cache coherence)	low

Table 12.3: Comparison of shared and private caches

caches. They introduce additional complexity, and are best covered in an advanced textbook on computer architecture.

Let us discuss the design of shared caches first because they are simpler. Before proceeding further, let us review where we stand.

Way Point 11

1. *We defined a set of correctness requirements for caches in Section 12.4.1. They were termed as* coherence *and* consistency*.*
2. *In a nutshell, both the concepts place constraints on reordering memory requests in the memory system. The order and semantics of requests to the same memory location is referred to as coherence, and the semantics of requests to different memory locations by the same thread is referred to as consistency.*
3. *For ensuring that a memory system is consistent with a certain model of memory, we need to ensure that the hardware follows a set of rules with regard to reordering memory requests issued by the same program. This can be ensured by having additional circuitry that stalls all the memory requests, till a set of memory requests issued in the past complete. Secondly, programmer support is also required for making guarantees about the correctness of a program.*
4. *There are two approaches for designing caches – shared or private. A shared cache has a single physical location for each memory location. Consequently, maintaining coherence is trivial. However, it is not a scalable solution because of high contention, and high latency.*
5. *Consequently, designers often use private caches at least for the L1 level. In this case, we need to explicitly ensure cache coherence.*

12.4.5 Shared Caches

In the simplest embodiment of a shared cache, we can implement it as a regular cache in a uniprocessor. However, this will prove to be a very bad approach in practice. The reason for this is that in a uniprocessor, only one thread accesses the cache; however, in a multiprocessor multiple threads might access the cache, and thus we need to provide more bandwidth. If all the threads need to access the same data and tag array, then either requests have to stall or we have to increase the number of ports in the arrays. This will have very negative consequences in terms of area and power. Lastly, cache sizes (especially L2 and L3) are roughly doubling as per Moore's law. As of 2012, on-chip caches can be as large as 4-8 MB. If we have a single tag array for the entire cache, then it will be very large and slow. Let us define the term *last level cache* (LLC) as the on chip cache that has the lowest position in the memory hierarchy (with main memory being the lowest). For example, if a multicore processor has an on-chip L3 cache that is connected to main memory, then the LLC is the L3 cache. We shall use the term LLC frequently from now onwards.

To create a multi-megabyte LLC that can simultaneously support multiple threads, we need to split it into multiple subcaches. Let us assume that we have a 4 MB LLC. In a typical design, this will be split into 8-16 smaller subcaches. Thus, each subcache will be 256-512 KB in size, which is an acceptable size. Each such subcache is a cache in its own right, and is known as a *cache bank*. Hence, we have in effect split a large cache into a set of cache banks. A cache bank can either be direct mapped, or can be set associative.

There are two steps in accessing a multibank cache. We first calculate the bank address, and then perform a regular cache access at the bank. Let us explain with an example. Let us consider a 16-bank, 4 MB cache. Each bank thus contains 256 KB of data. Now 4 MB = 2^{22} bytes. We can thus dedicate bits 19-22 for choosing the bank address. Note that bank selection is independent of associativity in this case. After choosing a bank, we can split the remaining 28 bits between the offset within the block, set index, and tag.

There are two advantages of dividing a cache into multiple banks. The first is that we decrease the amount of contention at each bank. If we have 4 threads, and 16 banks, then the probability that 2 threads access the same bank is low. Secondly, since each bank is a smaller cache, it is more power efficient, and faster. We have thus achieved our twin aims of supporting multiple threads, and designing a fast cache. We shall look at the problem of placing processors, and cache banks in a multicore processor in Section 12.6.

12.4.6 Coherent Private Caches

Overview of a Snoopy Protocol

The aim here is to make a set of private caches behave as if it is one large shared cache. From the point of view of software we should not be able to figure out whether a cache is private or shared. A conceptual diagram of the system is shown in Figure 12.9. It shows a set of processors along with their associated caches. The set of caches forms a *cache group*. The entire cache group needs to appear as one cache.

These caches are connected via an interconnection network, which can range from a simple shared bus type topology to more complex topologies. We shall look at the design of different interconnection networks in Section 12.6. In this section, let us assume that all the caches are

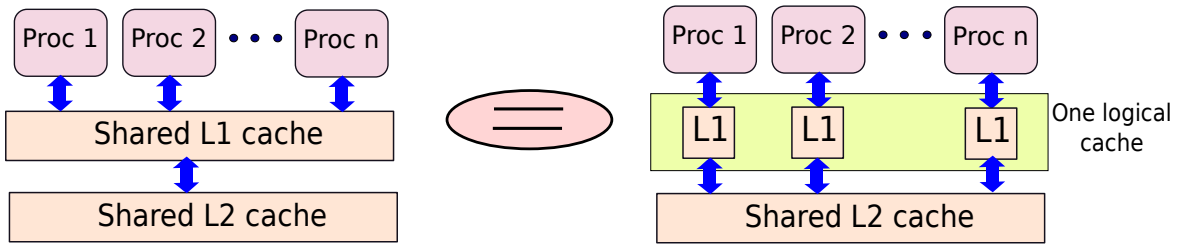


Figure 12.9: A system with many processors and their private caches

connected to a shared bus. A shared bus allows a single writer and multiple readers at any point of time. If one cache writes a message to the bus, then all the other caches can read it. The topology is shown in Figure 12.10. Note that the bus gives exclusive access to only one cache at any point of time for writing a message. Consequently, all the caches perceive the same order of messages. A protocol that implements cache coherence with caches connected on a shared bus is known as a *snoopy protocol*.

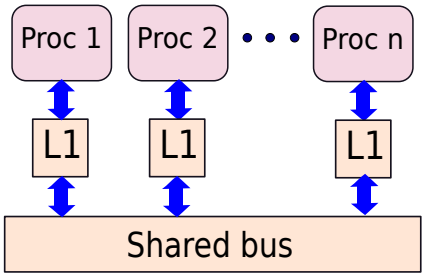


Figure 12.10: Caches connected with a shared bus

Let us now consider the operation of the snoopy protocol from the point of view of the two axioms of coherence – writes always complete (completion axiom), and writes to the same block are seen in the same order by all processors (order axiom). If cache i, wishes to perform a write operation on a block, then this write needs to be ultimately visible to all the other caches. We need to do this to satisfy the completion axiom, because we are not allowed to lose a write request. Secondly, different writes to the same block need to arrive at all the caches that might contain the block in the same order (order axiom). This ensures that for any given block, all the caches perceive the same order of updates. The shared bus automatically satisfies this requirement (the order axiom).

We present the design of two snoopy protocols – write-update and write-invalidate.

Write-Update Protocol

Let us now design a protocol, were a private cache keeps a copy of a write request, and broadcasts the write request to all the caches. This strategy ensures that a write is never lost, and the write messages to the same block are perceived in the same order by all the caches. This strategy requires us to broadcast, whenever we want to write. This is a large additional overhead; however, this strategy will work. Now, we need to incorporate reads into our protocol. A read to a location, x, can first check the private cache to see if a copy of it is already available. If a valid copy is available, then the value can be forwarded to the requesting processor. However, if there is a cache miss, then it is possible that it might be present with another sister cache in the cache group, or it might need to be fetched from the lower level. We need to first check if the value is present with a sister cache. We follow the same process here. The cache broadcasts a read request to all the caches. If any of the caches, has the value, then it replies, and sends the value to the requesting cache. The requesting cache inserts the value, and forwards it to the processor. However, if it does not get any reply from any other cache, then it initiates a read to the lower level.

This protocol is known as the write-update protocol. Each cache block needs to maintain three states, M, S, and I. M refers to the modified state. It means that the cache has modified the block. S(shared) means that the cache has not modified the block, and I(invalid) denotes the fact that the block does not contain valid data.

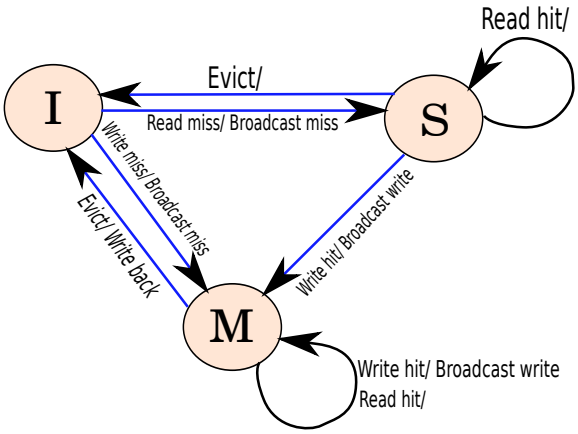


Figure 12.11: State transition diagram in the write-update protocol

Figure 12.11 shows a finite state machine (FSM) for each cache block. This FSM is executed by the cache controller. The format for state transitions is *event / action*. If the cache controller is sent an *event*, then it takes a corresponding *action*, which may include a state transition. Note that in some cases, the action field is blank. This means that in those cases, no action is taken. Note that the state of a cache block is a part of its entry in the tag array. If a block is not present in the cache, then its state is assumed to be invalid (I). Lastly, it is important to mention that Figure 12.11 shows the transitions for events generated by the processor. It does not show the actions for events sent over the bus by other caches in the cache group. Now, let us discuss the protocol in detail.

All blocks, initially are in the I state. If there is a read miss then it moves to the S state. We additionally need to broadcast the read miss to all the caches in the cache group, and either get the value from a sister cache, or from the lower level. Note that we give first preference to a sister cache, because it might have modified the block without writing it back to the lower level. Similarly, if there is a write miss in the I state, then we need to read the block from another sister cache if it is available, and move to the M state. If no other sister cache has the block, then we need to read the block from the lower level of the memory hierarchy.

If there is a read hit in the S state, then we can seamlessly pass the data to the processor. However, if we need to write to the block in the S state, we need to broadcast the write to all the other caches such that they get the updated value. Once, the cache gets a copy of its write request from the bus, it can write the value to the block, and change its state to M. To evict a block in the S state, we need to just evict it from the cache. It is not necessary to write back its value because the block has not been modified.

Now, let us consider the M state. If we need to read a block in the M state, then we can read it from the cache, and send the value to the processor. There is no need to send any message. However, if we wish to write to it, then it is necessary to send a write request on the bus. Once, the cache sees its own write request arrive via the shared bus, it can write its value to the memory location in its private cache. To evict a block in the M state, we need to write it back to the lower level in the memory hierarchy, because it has been modified.

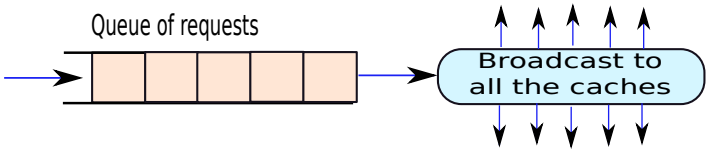


Figure 12.12: The bus arbiter

Every bus has a dedicated structure called an *arbiter* that receives requests to use the bus from different caches. It allots the bus to the caches in FIFO order. A schematic of the bus arbiter is shown in Figure 12.12. It is a very simple structure. It contains a queue of requests to transmit on the bus. Each cycle it picks a request from the queue, and gives permission to the corresponding cache to transmit a message on the bus.

Let us now consider a sister cache. Whenever it gets a *miss* message from the bus, it checks its cache to find if it has the block. If there is a cache hit, then it sends the block on the bus, or directly to the requesting cache. If it receives the notification of a write by another cache, then it updates the contents of the block if it is present in its cache.

Directory Protocol

Note that in the snoopy protocol we always broadcast a write, a read miss, or a write miss. This is strictly not required. We need to send a message to only those caches that contain a copy of the block. The *directory protocol* uses a dedicated structure called a directory to maintain this information. For each block address, the directory maintains a list of *sharers*. A sharer is the id of a cache that might contain the block. The list of sharers is in general a superset of caches

that might contain the given block. We can maintain the list of sharers as a bit vector (1 bit per sharer). If a bit is 1, then a cache contains a copy, otherwise it does not.

The write-update protocol with a directory gets modified as follows. Instead of broadcasting data on the bus, a cache sends all of its messages to the directory. For a read or write miss, the directory fetches the block from a sister cache if it has a copy. It then forwards the block to the requesting cache. Similarly, for a write, the directory sends the write message to only those caches that might have a copy of the block. The list of sharers needs to be updated when a cache inserts or evicts a block. Lastly, to maintain coherence the directory needs to ensure that all the caches get messages in the same order, and no message is ever lost. The directory protocol minimizes the number of messages that need to be sent, and is thus more scalable.

Definition 129

Snoopy Protocol *In a snoopy protocol, all the caches are connected to a shared bus. A cache broadcasts each message to the rest of the caches.*

Directory Protocol *In a directory protocol, we reduce the number of messages by adding a dedicated structure known as a* directory*. The directory maintains the list of caches that might potentially contain a copy of the block. It sends messages for a given block address to only the caches in the list.*

Question 8
Why is it necessary to wait for the broadcast from the bus to perform a write?
Answer: *Let us assume this is not the case, and processor 1, wishes to write 1 to x, and processor 2 wishes to write 2 to x. They will then first write 1 and 2 to their copies of x respectively, and then broadcast the write. Thus, the writes to x will be seen in different orders by both the processors. This violates the order axiom. However, if they wait for a copy of their write request to arrive from the bus, then they write to x in the same order. The bus effectively resolves the conflict between processor 1 and 2, and orders one request after the other.*

Write-Invalidate Protocol

We need to note that broadcasting a write request for every single write is an unnecessary overhead. It is possible that most of the blocks might not be shared in the first place. Hence, there is no need to send an extra message on every write. Let us try to reduce the number of messages in the write update protocol by proposing the *write-invalidate protocol*. Here again, we can either use the snoopy protocol, or the directory protocol. Let us show an example with the snoopy protocol.

Let us maintain three states for each block – M, S, and I. Let us however, change the meaning of our states. The invalid state (I) retains the same meaning. It means that the entry is effectively not present in the cache. The shared state (S) means that a cache can read the block, but it cannot write to it. It is possible to have multiple copies of the same block in different caches in the shared state. Since the shared state assumes that the block is read-only, having multiple copies of the block does not affect cache coherence. The M (modified) state signifies the fact that the cache can write to the block. If a block is in the M state, then all the other caches in the cache group need to have the block in the I state. No other cache is allowed to have a valid copy of the block in the S or M states. This is where the write-invalidate protocol differs from the write-update protocol. It allows either only one writer at a time, or multiple readers at a time. It never allows a reader and a writer to co-exist at the same time. By restricting the number of caches that have write access to a block at any point of time, we can reduce the number of messages.

The basic insight is as follows. The write-update protocol did not have to send any messages on a read hit. It sent extra messages on a write hit, which we want to eliminate. It needed to send extra messages because multiple caches could read or write a block concurrently. For the write-invalidate protocol, we have eliminated this behavior. If a block is in the M state, then no other cache contains a valid copy of the block.

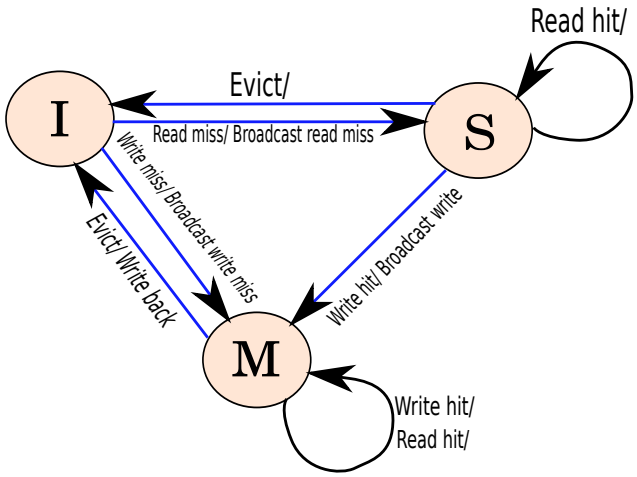


Figure 12.13: State transition diagram of a block due to actions of the processor

Figure 12.13 shows the state transition diagram because of actions of the processor. The state transition diagram is mostly the same as the state transition diagram of the write-update protocol. Let us look at the differences. The first is that we define three types of messages that are put on the bus namely *write*, *write miss*, and *read miss*. When we transition from the I to the S state, we place a read miss on the bus. If a sister cache does not reply with the data, the cache controller reads the block from the lower level. The semantics of the S state remains the same. To write to a block in the S state, we need to transition to the M state, after writing a *write* message on the bus. Now, when a block is in the M state, we are assured of the fact that no other cache contains a valid copy. Hence, we can freely read and write a block in the

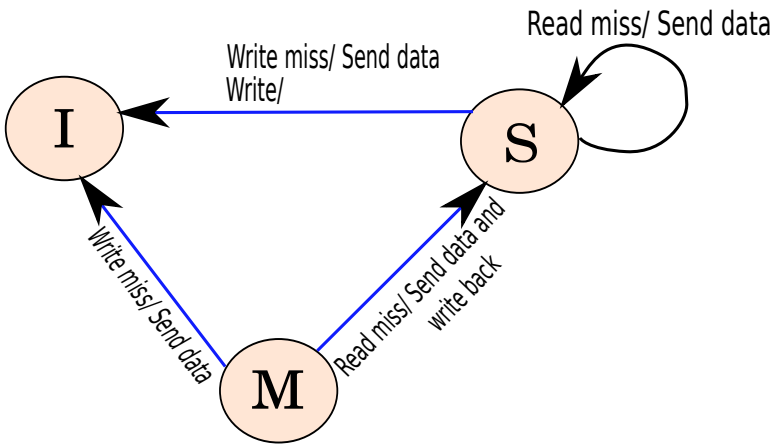


Figure 12.14: State transition diagram of a block due to messages on the bus

M state. It is not necessary to send any messages on the bus. If the processor decides to evict a block in the M state, then it needs to write its data to the lower level.

Figure 12.14 shows the state transitions due to messages received on the bus. In the S state, if we get a *read miss*, then it means that another cache wants read access to the block. Any of the caches that contains the block sends it the contents of the block. This process can be orchestrated as follows. All the caches that have a copy of the block try to get access to the bus. The first cache that gets access to the bus sends a copy of the block to the requesting cache. The rest of the caches immediately get to know that the contents of the block have been transferred. They subsequently stop trying. If we get a *write* or *write miss* message in the S state, then the block transitions to the I state.

Let us now consider the M state. If some other cache sends a *write miss* message then the cache controller of the cache that contains the block, sends the contents of the block to it, and transitions to the I state. However, if it gets a *read miss*, then it needs to perform a sequence of steps. We assume that we can seamlessly evict a block in the S state. Hence, it is necessary to write the data to the lower level before moving to the S state. Subsequently, the cache that originally has the block also sends the contents of the block to the requesting cache, and transitions the state of the block to the S state.

Write-Invalidate Protocol with a Directory

Implementing the write-invalidate protocol with a directory is fairly trivial. The state transition diagrams remain almost the same. Instead of broadcasting a message, we send it to the directory. The directory sends the message to the sharers of the block.

The life cycle of a block is as follows. Whenever, a block is brought in from the lower level, a directory entry is initialized. At this point it has only one sharer, which is the cache that brought it from the lower level. Now, if there are read misses to the block, then the directory keeps adding sharers. However, if there is a write miss, or a processor decides to write to the

block, then it sends a *write* or *write miss* message to the directory. The directory cleans the sharers list, and keeps only one sharer, which is the processor that is performing the write access. When a block is evicted, its cache informs the directory, and the directory deletes a sharer. When the set of sharers becomes empty, the directory entry can be removed.

It is possible to make improvements to the write-invalidate and update protocols by adding an additional state known as the exclusive (E) state. The E state can be the initial state for every cache block fetched from the lower level of the memory hierarchy. This state stores the fact that a block exclusively belongs to a cache. However, the cache has read-only access to it, and does not have write access to it. For an E to M transition, we do not have to send a *write miss* or *write* message on the bus, because the block is owned exclusively by one cache. We can seamlessly evict data from the E state if required. Implementing the MESI protocol is left as an exercise for the reader.

12.4.7 Implementing a Memory Consistency Model*

A typical memory consistency model specifies the types of re-orderings that are allowed between memory operations issued by the same thread. For example, in sequential consistency all read/write accesses are completed in program order, writes are atomic, and all other threads also perceive the memory accesses of any thread in its program order. Let us give a simple solution to the problem of implementing sequential consistency first.

Overview of an Implementation of Sequential Consistency*

Let us build a memory system that is coherent, and provides certain guarantees. Let us assume that all the write operations are associated with a time of *completion*, and appear to execute instantaneously at the time of *completion*. It is not possible for any read operation to get the value of a write before it completes. After a write *completes*, all the read operations to the same address either get the value written by the write operation or a newer write operation. Since we assume a coherent memory, all the write operations to the same memory address are seen in the same order by all the processors. Second, each read operation returns the value written by the latest completed write to that address. Let us now consider the case in which processor 1 issues a write to address x, and at the same time processor 2 issues a read request to the same address, x. In this case, we have a concurrent read and write. The behavior is not defined. The read can either get the value set by the concurrent write operation, or it can get the previous value. However, if the read operation gets the value set by the concurrent write operation, then all subsequent reads issued by any processor, need to get that value or a newer value. In this case, we can say that a read operation *completes* once it has finished reading the value of the memory location, and the write that generated its data also completes.

Now, let us design a multicore processor where each core issues a memory request to coherent memory (defined above) after all the previous memory requests that it had issued have completed. This means that memory requests are issued in program order. This means that after issuing a memory request (read/write), a core waits for it to *complete* before issuing the next memory request. We claim that a multiprocessor with such cores and coherent memory is sequentially consistent. Let us now outline a brief informal proof.

Let us first introduce a theoretical tool called an *access graph* before proving the sequential consistency of this system.

Access Graph*

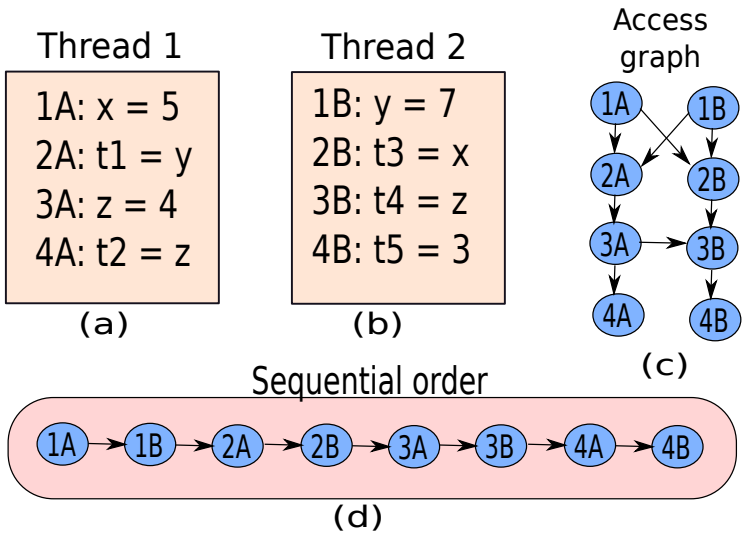


Figure 12.15: Graphical representation of memory accesses

Figure 12.15 shows the execution of two threads and their associated sequence of memory accesses. For each read or write operation, we create a node in the access graph (see Figure 12.15(c)). We add an arrow (or edge) between two nodes if one access follows the other in program order, or if there is a read-write dependence across two accesses from different threads. For example, if we set x to 5 in thread 1, and the read operation in thread 2 reads this value of x, there is a dependence between this read and write. We thus add an arrow in the access graph. The arrow signifies that the destination node (access) must complete after the source node (access).

Let us now define a *happens-before* relationship between nodes a and b, if there is a path from a to b in the access graph.

Definition 130
Let us define a happens-before *relationship between nodes a and b, if there is a path from a to b in the access graph. A happens-before relationship signifies that b must complete its execution after a completes.*

The access graph is a general tool and is used to reason about concurrent systems. It consists of a set of nodes, where each node is a dynamic instance of an instruction, specifically a memory instruction. There are edges between nodes. An edge of the form $A \rightarrow B$ means that B needs to complete its execution after A. In our simple example in Figure 12.15, we have added two kinds of edges namely *program order* edges, and *causality* edges. Program order edges indicate the order of completion of memory requests in the same thread. In our system,

where we wait for an instruction to complete, before executing the next instruction, there are edges between consecutive instructions issued by the same thread.

Causality edges are between load and store instructions across threads. For example, if a given instruction writes a value, and another instruction reads it in another thread, we add an edge from the store to the load.

To prove sequential consistency, we need to add additional edges to the access graph as follows (see [Arvind and Maessen, 2006]). Let us first assume that we have an *oracle*(a hypothetical entity that knows everything) with us. Now, since we assume coherent memory, all the stores to the same memory location are sequentially ordered. Furthermore, there is an order between loads and stores to the same memory location. For example, if we set x to 1, then set x to 3, then read $t1 = x$, and then set x to 5, there is a store-store-load-store order for the location, x. The oracle knows about such orderings between loads and stores for each memory location. Let us assume that the oracle adds the corresponding happens-before edges to our access graph. We add an edge for each consecutive pair of accesses to the same address (regardless of the threads involved). In this case, the edge between the store and the load is a causality edge, and the additional store-store and load-store edges are examples of *coherence edges*.

To summarize, we add four kinds of edges in our access graph.

Program-order Edges Consecutive memory instructions (in program order) issued by the same thread have such edges between them in the access graph.

Store→Load Edges These edges are added between memory accesses issued by different threads to the same address. They indicate a store→load dependence that represents causality.

Load→Store Edges These are edges introduced between nodes that represent load and store accesses (across threads) to the same address, respectively.

Store→Store Edges These are edges introduced between nodes that represent consecutive writes to the same address. The accesses may be issued by different threads.

The last two types of edges are known as *coherence edges*.

Next, let us describe how to use an access graph for proving properties of multiprocessor systems. An access graph can be used to determine if a given program's execution adheres to a certain consistency model like sequential or weak consistency or not. First, we need to construct an access graph of a program's execution. We add the four types of edges that we mentioned if we want to test if the execution is sequentially consistent. However, if we want to test if the execution adheres to another consistency model, $\mathcal{M}$, then we to add a set of edges that is a subset of the edges that we add to test if the execution is in SC. The set of edges that we add is a field of study in itself for realistic memory models. In general, for weak consistency, we only add coherence edges. Some memory models define additional types of nodes such as synchronization nodes. They need to define additional edges as well to connect these nodes.

The following result follows from graph theory, specifically topological sorting. **If the access graph does not contain cycles, then we can arrange the nodes in a sequential order.** Let us prove this fact. In the access graph, if there is a path from a to b, then let a be known as b's ancestor. We can generate a sequential order by following an iterative process. We first find

a node, which does not have an ancestor. There has to be such a node because some operation must have been the first to complete (otherwise there is a cycle). We remove it from our access graph, and proceed to inductively find another node that does not have any ancestors left. We add each such node in our sequential order as shown in Figure 12.15(d). In each step, the number of nodes in the access graph decreases by 1 till we are finally left with just one node, which becomes the last node in our sequential order. Now, let us consider the case, where this process does not terminate. This is only possible if there is a cycle in the access graph.

If the access graph is acyclic, then it means that the execution is consistent with the memory model, $\mathcal{M}$, that was used to create the access graph.

This is true for atomic memory models. The proof is beyond the scope of the book. The key idea is as follows. We create an access graph from an execution of a parallel program, and add all the edges that need to be added to it as per the memory model. Now, if this access graph is acyclic, then it means that we can arrange all the memory accesses in a sequential order (as seen above).

We can assume that the order of *completion times* is the same as the order of all the accesses in this sequential order. This means that if all the accesses complete as per this order, the execution is *feasible*. In other words, an order of completion times exists such that none of the happens-before relationships in the access graph are violated.

This is sufficient to prove that the execution and its access graph are *consistent* with the memory model $\mathcal{M}$. The key point is that a set of such completion times should *exist*. An astute reader may argue that the memory accesses may not actually complete at those times. That does not matter. As long as we can find a hypothetical set of completion times for the nodes in the access graph that respect its happens-before relationships, we are done. Given that completion times are not visible anyway to external observers, their precise values are not important. With regard to completion times there are two points that are important: their existence and their order.

Hence, if an access graph (for memory model $\mathcal{M}$) does not contain cycles, we can conclude that a given execution follows $\mathcal{M}$. If we can prove that all the possible access graphs that can be generated by a system are acyclic, then we can conclude that the entire system is consistent with the memory model $\mathcal{M}$.

Proof of Sequential Consistency*

Let us thus prove that all possible access graphs (assuming SC) that can be generated by our simple system that waits for memory requests to complete before issuing subsequent memory requests are acyclic. Let us consider one such access graph G. We need to prove that G is acyclic.

Let us assume that G has a cycle, and it contains a set of nodes $\mathcal{S}$ belonging to the same thread, t_1. Let a be the earliest node in program order in $\mathcal{S}$, and b be the latest node in program order in $\mathcal{S}$. Clearly, *a happens before b* because we execute memory instructions in program order, and we wait for a request to complete before starting the next request in the same thread. For a cycle to form due to a causality edge, b needs to write to a value that is read by another memory read request (node), c, belonging to another thread. Alternatively, there can be a coherence edge between b and a node c belonging to another thread. For a cycle to exist, c needs to *happen before* a. Let us assume that there is a chain of nodes between c

and a and the last node in the chain of nodes is d. By definition, $d \notin t_1$. This means that either d writes to a memory location, and node a reads from it, or there is a coherence edge from d to a. Because there is a path from node b to node a (through c and d), it must be the case that the request associated with node b happens before the request of node a. This is not possible since we cannot execute the memory request associated with node b till node a's request completes. Thus, we have a contradiction, and a cycle is not possible in the access graph. Hence, the execution is in SC (proof by contradiction).

Now, let us clarify the notion of an *oracle*. The reader needs to understand that the problem here is not to generate a sequential order, it is to rather prove that a sequential order exists. Since we are solving the latter problem, we can always presume that a hypothetical entity adds additional edges to our access graph. The resulting sequential order respects program orders for each thread, causality and coherence-based happens-before relationships. It is thus a *valid* ordering.

Consequently, we can conclude that it is always possible to find a sequential order for the memory accesses made in our system. Therefore, our multiprocessor is in SC. Now that we have proved that our system is sequentially consistent, let us describe a method to "practically" implement a multiprocessor with the assumptions that we have made. We can implement a system as shown in Figure 12.16.

Design of a Simple (yet impractical) Sequentially Consistent Machine*

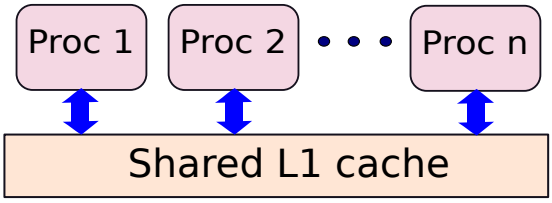


Figure 12.16: A simple sequentially consistent system

Figure 12.16 shows a design that has a large shared L1 cache across all the processors of a multiprocessor. There is only one copy of each memory location that can support only one read or write access at any single time. This ensures coherence. Secondly, a write completes, when it changes the value of its corresponding memory location in the L1 cache. Likewise, a read completes when it reads the value of the memory address in the L1 cache. We need to modify the simple in-order RISC pipeline described in Chapter 10 such that an instruction leaves the memory access (MA) stage only after it **completes** its read/write access. If there is a cache miss, then the instruction waits till the block comes to the L1 cache, and then the access completes. This simple system ensures that memory requests from the same thread complete in program order, and is thus sequentially consistent.

Note that the system described in Figure 12.16 makes some unrealistic assumptions and is thus impractical. If we have 16 processors, and if the frequency of memory instructions is 1 in 3, then every cycle, 5-6 instructions will need to access the L1 cache. Hence, the L1 cache requires at least 6 read/write ports, which will make the structure too large and too slow. Additionally,

the L1 cache needs to be large enough to contain the working sets of all the threads, which further makes the case for a very large and slow L1 cache. Consequently, a multiprocessor system with such a cache will be very slow in practice. Hence, modern processors opt for more high-performance implementations with more complicated memory systems that have a lot of smaller caches. These caches co-operate among each other to provide the illusion of a larger cache (see Section 12.4.6).

It is fairly difficult to prove that a complex system follows sequential consistency(SC). Hence, designers opt to design systems with weak memory models. In the latter case, we need to prove that a *fence* instruction works correctly. If we take all the subtle corner cases that are possible with complicated designs, this also turns out to be a fairly challenging problem. Interested readers can take a look at pointers mentioned at the end of this chapter for research work in this area.

Implementing a Weak Consistency Model*

Let us consider the access graph for a weakly consistent system. We do not have edges to represent program order for nodes in the same thread. Instead, for nodes in the same thread, we have edges between regular read/write nodes and $fence$ operations. We need to add causality and coherence edges to the access graph as we did in the case of SC.

An implementation of a weakly consistent machine needs to ensure that this access graph does not have cycles. We can prove that the following implementation does not introduce cycles to the access graph.

Let us ensure that a $fence$ instruction starts after all the previous instructions in program order complete for a given thread. The $fence$ instruction is a dummy instruction that simply needs to reach the end of the pipeline. It is used for ordering purposes only. We stall the $fence$ instruction in the MA stage till all the previous instructions complete. This strategy also ensures that no subsequent instruction reaches the MA stage. Once, all the previous instructions complete, the $fence$ instruction proceeds to the RW stage, and subsequent instructions can issue requests to memory.

Summary of the Discussion on Implementing a Memory Consistency Model

Let us summarize this section on implementing memory consistency models for readers who decided to skip it. Implementing a memory consistency model such as sequential or weak consistency is possible by modifying the pipeline of a processor, and ensuring that the memory system sends an acknowledgement to the processor once it is done processing a memory request. Many subtle corner cases are possible in high=performance implementations and ensuring that they implement a given consistency model is fairly complicated.

12.4.8 Multithreaded Processors

Let us now look at a different method for designing multiprocessors. Up till now we have maintained that we need to have physically separate pipelines for creating multiprocessors. We have looked at designs that assign a separate program counter to each pipeline. However, let us look at a different approach that runs a set of threads on the same pipeline. This approach is known as multithreading. Instead of running separate threads on separate pipelines, we run

them on the same pipeline. Let us illustrate this concept by discussing the simplest variant of multi-threading known as *coarse-grained multithreading*.

Definition 131
Multithreading is a design paradigm that proposes to run multiple threads on the same pipeline. A processor that implements multithreading is known as a multithreaded *processor.*

Coarse-Grained Multithreading

Let us assume that we wish to run four threads on a single pipeline. Recall that multiple threads belonging to the same process have their separate program counters, stacks, registers; yet, they have a common view of memory. All these four threads have their separate instruction streams, and it is necessary to provide an illusion that these four threads are running separately. Software should be oblivious of the fact that threads are running on a multithreaded processor. It should perceive that each thread has its dedicated CPU. Along with the traditional guarantees of coherence and consistency, we now need to provide an additional guarantee, which is that software should be oblivious to multithreading.

Let us consider a simple scheme, as shown in Figure 12.17.

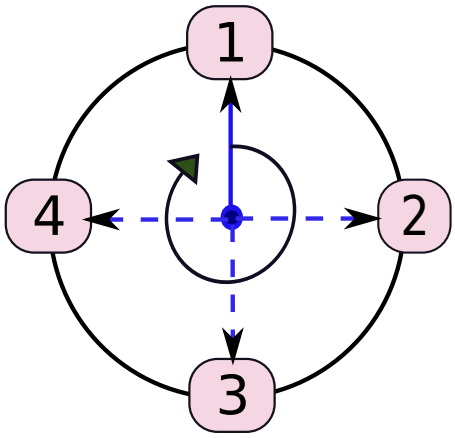


Figure 12.17: Conceptual view of coarse grain multithreading

Here, we run thread 1 for n cycles, then we switch to thread 2 and run it for n cycles, then we switch to thread 3, and so on. After executing thread 4 for n cycles, we start executing thread 1 again. To execute a thread we need to load its state or context. Recall that we had a similar discussion with respect to loading and unloading the state of a program in Section 10.8. We had observed that the context of the program comprises the $flags$ register, the program counter, and the set of registers. We had observed that it is not necessary to keep track of main memory because the memory regions of different processes do not overlap, and in the case of multiple threads, we explicitly want all the threads to share the same memory space.

Instead of explicitly loading and unloading the context of a thread, we can adopt a simpler approach. We can save the context of a thread in the pipeline. For example, if we wish to support coarse-grained multithreading then we can have four separate $flags$ registers, four program counters, and four separate register files (one per each thread). Additionally, we can have a dedicated register that contains the id of the currently running thread. For example, if we are running thread 2, then we use the context of thread 2, and if we are running thread 3, we use the context of thread 3. In this manner it is not possible for multiple threads to overwrite each other's state.

Let us now look at some subtle issues. It is possible that we can have instructions belonging to multiple threads at the same point of time in the pipeline. This can happen when we are switching from one thread to the next. Let us add a thread id field to the instruction packet, and further ensure that the forwarding and interlock logic takes the id of the thread into account. We never forward values across threads. In this manner it is possible to execute four separate threads on a pipeline with a negligible overhead of switching between threads. We do not need to engage the exception handler to save and restore the context of threads, or invoke the operating system to schedule the execution of threads.

Let us now look at coarse-grained multithreading in entirety. We execute n threads in quick succession, and in round robin order. Furthermore, we have a mechanism to quickly switch between threads, and threads do not corrupt each other's state. However, we still do not execute four threads simultaneously. Then, what is the advantage of this scheme?

Let us consider the case of memory intensive threads that have a lot of irregular accesses to memory. They will thus frequently have misses in the L2 cache, and their pipelines need to be stalled for 100-300 cycles till the values come back from memory. Out-of-order pipelines can hide some of this latency by executing some other instructions that are not dependent on the memory value. Nonetheless, it will also stall for a long time. However, at this point, if we can switch to another thread, then it might have some useful work to do. If that thread suffers from misses in the L2 cache also, then we can switch to another thread and finish some of its work. In this way, we can maximize the throughput of the entire system as a whole. We can envision two possible schemes. We can either switch periodically every n cycles, or switch to another thread upon an event such as an L2 cache miss. Secondly, we need not switch to a thread if it is waiting on a high latency event such as an L2 cache miss. We need to switch to a thread that has a pool of ready-to-execute instructions. It is possible to design numerous heuristics for optimizing the performance of a coarse-grained multithreaded machine.

Important Point 20

Let us differentiate between software threads *and* hardware threads. *A software thread is a subprogram that shares a part of its address space with other software threads. The threads can communicate with each other to co-operatively to achieve a common objective. In comparison, a hardware thread is defined as the instance of a software thread or a single threaded program running on a pipeline along with its execution state. A multithreaded processor supports multiple hardware threads on the same processor by splitting its resources across the threads. A software thread might physically be mapped to a separate processor, or to a hardware thread. It is agnostic to the entity that is used to execute it. The important point to be noted here is that a software thread is a programming language concept, whereas*

a hardware thread is physically associated with resources in a pipeline. We shall use the word "thread" for both software and hardware threads. The correct usage needs to be inferred from the context.

Fine-Grained Multithreading

Fine-grained multithreading is a special case of coarse-grained multithreading where the switching interval, n, is a very small value. It is typically 1 or 2 cycles. This means that we quickly switch between threads. We can leverage grained multithreading to execute threads that are memory intensive. However, fine-grained multithreading is also useful for executing a set of threads that for example have long arithmetic operations such as division. In a typical processor, division operations and other specialized operations such as trigonometric or transcendental operations are slow (3-10 cycles). During this period when the original thread is waiting for the operation to finish, we can switch to another thread and execute some of its instructions in the pipeline stages that are otherwise unused. We can thus leverage the ability to switch between threads very quickly for reducing the idle time in scientific programs that have a lot of mathematical operations.

We can thus visualize fine-grained multithreading to be a more flexible form of coarse-grained multithreading where we can quickly switch between threads and utilize idle stages to perform useful work. Note that this concept is not as simple as it sounds. The devil is in the details. We need elaborate support for multithreading in all the structures in a regular in-order or out-of-order pipeline. We need to manage the context of each thread very carefully, and ensure that we do not omit instructions, and errors are not introduced. A thorough discussion on the implementation of multithreading is beyond the scope of this book.

The reader needs to appreciate that the logic for switching between threads is non-trivial. Most of the time the logic to switch between threads is a combination of time based criteria (number of cycles), and event based criteria (high latency event such as L2 cache miss or page fault). The heuristics have to be finely adjusted to ensure that the multithreaded processor performs well for a host of benchmarks.

Simultaneous Multithreading

For a single issue pipeline, if we can ensure that every stage is kept busy by using sophisticated logic for switching between threads, then we can achieve high efficiency. Recall that any stage in a single issue pipeline can process only one instruction per cycle. In comparison, a multiple issue pipeline can process multiple instructions per cycle. We had looked at multiple issue pipelines (both in-order and out-of-order) in Section 10.11. Moreover, we had defined the number of issue slots to be equal to the number of instructions that can be processed by the pipeline every cycle. For example, a 3 issue processor, can at the most fetch, decode, and finally execute 3 instructions per cycle.

For implementing multithreading in multiple issue pipelines, we need to consider the nature of dependences between instructions in a thread also. It is possible that fine and coarse-grained schemes do not perform well because a thread cannot issue instructions to the functional units

for all the issue slots. Such threads are said to have low instruction level parallelism. If we use a 4 issue pipeline, and the maximum IPC for each of our threads is 1 because of dependences in the program, then 3 of our issue slots will remain idle in each cycle. Thus, the overall IPC of our system of 4 threads will be 1, and the benefits of multithreading will be limited.

Hence, it is necessary to utilize additional issue slots such that we can increase the IPC of the system as a whole. A naive approach is to dedicate one issue slot to each thread. Secondly, to avoid structural hazards, we can have four ALUs and allot one ALU to each thread. However, this is a suboptimal utilization of the pipeline because a thread might not have an instruction to issue every cycle. It is best to have a more flexible scheme, where we dynamically partition the issue slots among the threads. This scheme is known as simultaneous multithreading (popularly known as SMT). For example, in a given cycle we might find 2 instructions from thread 2, and 1 instruction each from threads 3, and 4. This situation might reverse in the next cycle. Let us graphically illustrate this concept in Figure 12.18, and simultaneously also compare the SMT approach with fine and coarse-grained multithreading.

The columns in Figure 12.18 represent the issue slots for a multiple issue machine, and the rows represent the cycles. Instructions belonging to different threads have different colors. Figure 12.18(a) shows the execution of instructions in a coarse-grained machine, where each thread executes for two consecutive cycles. We observe that a lot of issue slots are empty because we do not find a sufficient number of instructions that can execute. Fine-grained multithreading (shown in Figure 12.18(b)) also has the same problem. However, in an SMT processor, we are typically able to keep most of the issue slots busy, because we always find instructions from the set of available threads that are ready to execute. If one thread is stalled for some reason, other threads compensate by executing more instructions. In practice, all the threads do not have low ILP[1] phases simultaneously. Hence, the SMT approach has proven to be a very versatile and effective method for leveraging the power of multiple issue processors. Since the Pentium 4 (released in the late nineties), most of the Intel processors support different variants of simultaneous multithreading. In Intel's terminology SMT is known as hyperthreading . The latest (as of 2012) IBM Power 7 processor has 8 cores, where each core is a 4-way SMT (can run 4 threads per each core).

Note that the problem of selecting the right set of instructions to issue is very crucial to the performance of an SMT processor. Secondly, the memory bandwidth requirement of an n-way SMT processor is higher than that of an equivalent uniprocessor. The fetch logic is also far more complicated, because now we need to fetch from four separate program counters in the same cycle. Lastly, the issues of maintaining coherence, and consistency further complicate the picture. The reader can refer to the research papers mentioned in the "Further Reading" section at the end of this chapter.

12.5 SIMD Multiprocessors

Let us now discuss SIMD multiprocessors. SIMD processors are typically used for scientific applications, high intensity gaming, and graphics. They do not have a significant amount of general purpose utility. However, for a limited class of applications, SIMD processors tend to

[1]ILP (instruction level parallelism, defined as the number of instructions that are ready to execute in parallel each cycle)

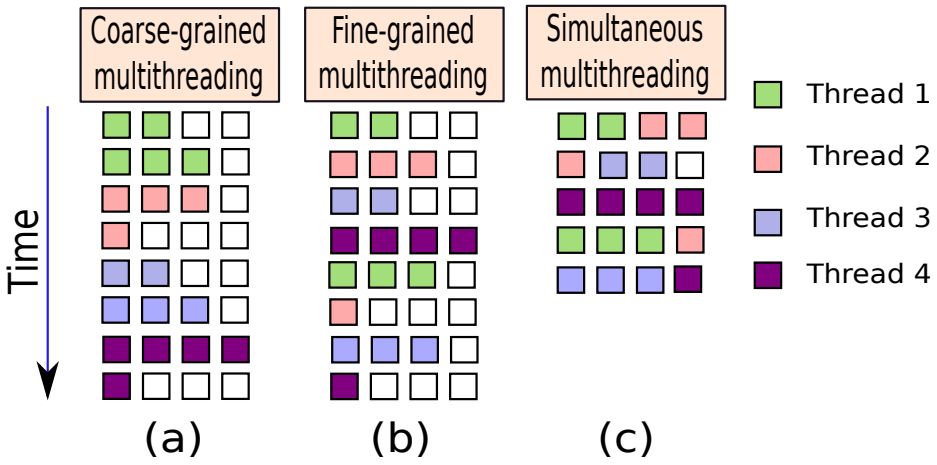


Figure 12.18: Instruction execution in multithreaded processors

outperform their MIMD counterparts.

SIMD processors have a rich history. In the good old days we had processors arranged as arrays. Data typically entered through the first row, and first column of processors. Each processor acted on input messages, generated an output message, and sent the message to its neighbors. Such processors were known as *systolic arrays*. Systolic arrays were used for matrix multiplication, and other linear algebra operations. Subsequently, several vendors notably, Cray, incorporated SIMD instructions in their processors to design faster and more power efficient supercomputers. Nowadays, most of these early efforts have subsided. However, some aspects of classical SIMD computers where a single instruction operates on several streams of data, have crept into the design of modern processors.

We shall discuss an important development in the area of modern processor design, which is the incorporation of SIMD functional units, and instructions, in high-performance processors.

12.5.1 SIMD – Vector Processors

Background

Let us consider the problem of adding two n element arrays. In a single threaded implementation, we need to load the operands from memory, add the operands, and store the result in memory. Consequently, for computing each element of the destination array, we require two load instructions, one add instruction, and one store instruction. Traditional processors try to attain speedups by exploiting the fact that we can compute $(c[i] = a[i] + b[i])$, in parallel with $(c[j] = a[j] + b[j])$ because these two operations do not have any dependences between them. Hence, it is possible to increase IPC by executing many such operations in parallel.

Let us now consider superscalar processors. If they can issue 4 instructions per cycle, then their IPC can at the most be 4 times that of a single cycle processor. In practice, the peak speedup over a single cycle processor that we can achieve with such inherently parallel array processing operations is around 3 to 3.5 times for a 4 issue processor. Secondly, this

method of increasing IPC by having wide issue widths is not scalable. We do not have 8 or 10 issue processors in practice because the logic of the pipeline gets very complicated, and the area/power overheads become prohibitive.

Hence, designers decided to have special support for vector operations that operate on large vectors (arrays) of data. Such processors were known as *vector processors*. The main idea here is to process an entire array of data at once. Normal processors use regular *scalar* data types such as integers and floating point numbers; whereas, vector processors use vector data types, which are essentially arrays of scalar data types.

Definition 132
A vector processor *considers a vector of primitive data types (integer or floating point numbers) as its basic unit of information. It can load, store, and perform arithmetic operations on entire vectors at once. Such instructions that operate on vectors of data, are known as* vector instructions.

One of the most iconic products that predominantly used vector processors was the Cray 1 Supercomputer. Such supercomputers were primarily used for scientific applications that mainly consisted of linear algebra operations. Such operations work on vectors of data and matrices, and are thus well suited to be run on vector processors. Sadly, beyond the realm of high intensity scientific computing, vector processors did not find a general purpose market till the late nineties.

In the late nineties, personal computers started to be used for research, and for running scientific applications. Secondly, instead of designing custom processors for supercomputers, designers started to use regular commodity processors for building supercomputers. Since then, the trend continued till the evolution of graphics processors. Most supercomputers between 1995 and 2010, consisted of thousands of commodity processors. Another important reason to have vector instructions in a regular processor was to support high intensity gaming. Gaming requires a massive amount of graphics processing. For example, modern games render complex scenes with multiple characters, and thousands of visual effects. Most of these visual effects such as illumination, shadows, animation, depth, and color processing, are at its core basic linear algebra operations on matrices containing points or pixels. Due to these factors regular processors started to incorporate a limited amount of vector support. Specifically, the Intel processors provided the MMX, SSE 1-4 vector instruction sets, AMD processors provided the 3DNow! vector extensions, and ARM processors provide the ARM® Neon™vector ISA. There are a lot of commonalities between these ISAs, and hence let us not focus on any specific ISA. Let us instead the discuss the broad principles behind the design and operation of vector processors.

12.5.2 Software Interface

Let us first consider the model of the machine. We need a set of vector registers. For example, the x86 SSE (Streaming SIMD Extensions) instruction set defines sixteen 128-bit registers (XMM0 . . . XMM15). Each such register can contain four integers, or four floating point values.

Alternatively, it can also contain eight 2-byte short integers, or sixteen 1-byte characters. On the same lines, every vector ISA defines additional vector registers that are wider than normal registers. Typically, each register can contain multiple floating point values. Hence, in our *SimpleRisc* ISA, let us define eight 128-bit vector registers: $vr0 \ldots vr7$.

Now, we need instructions to load, store, and operate on vector registers. For loading, vector registers, there are two options. We can either load values from contiguous memory locations, or from non-contiguous memory locations. The former case is more specific, and is typically suitable for array based applications, where all the array elements are anyway stored in contiguous memory locations. Most vector extensions to ISAs support this variant of the load instruction because of its simplicity, and regularity. Let us try to design such a vector load instruction *v.ld* for our *SimpleRisc* ISA. Let us consider the semantics shown in Table 12.4. Here, the *v.ld* instruction reads in the contents of the memory locations ([r1+12], [r1+16], [r1+20], [r1+ 24]) into the vector register $vr1$. In the table below note that $\langle vreg \rangle$ is a vector register.

Example	Semantics	Explanation
v.ld vr1, 12[r1]	v.ld $\langle vreg \rangle$, $\langle mem \rangle$	vr1 ← ([r1+12], [r1+16], [r1+20], [r1+ 24])

Table 12.4: Semantics of the contiguous variant of the vector load instruction

Now, let us consider the case of matrices. Let us consider a 10,000 element matrix, $A[100][100]$, and assume that data is stored in row major order (see Section 3.2.2). Assume that we want to operate on two columns of the matrix. In this case, we have a problem because the elements in a column are not saved in contiguous locations. Hence, a vector load instruction that relies on the assumption that the input operands are saved in contiguous memory locations, will cease to work. We need to have dedicated support to fetch all the data for the locations in a column and save them in a vector register. Such an operation is known as a *scatter-gather* operation. This is because, the input operands are essentially *scattered* in main memory. We need to *gather*, and put them in one place, which is the vector register. Let us consider a scatter-gather variant of the vector load instruction, and call it *v.sg.ld*. Instead of making assumptions about the locations of the array elements, the processor reads another vector register that contains the addresses of the elements (semantics shown in Table 12.5). In this case, a dedicated vector load unit reads the memory addresses stored in $vr2$, fetches the corresponding values from memory, and writes them in sequence to the vector register, $vr1$.

Example	Semantics	Explanation
v.sg.ld vr1, vr2	v.sg.ld $\langle vreg \rangle$, $\langle vreg \rangle$	vr1 ← ([vr2[0]], [vr2[1]], [vr2[2]], [vr2[3]])

Table 12.5: Semantics of the non-contiguous variant of the vector load instruction

Once, we have data loaded in vector registers, we can operate on two such registers directly. For example, if we consider 128-bit vector registers, $vr1$, and $vr2$. Then, the assembly statement $v.add\ vr3, vr1, vr2$, adds each pair of corresponding 4-byte floating point numbers stored in the input vector registers ($vr1$ and $vr2$), and stores the results in the relevant positions in the

output vector register ($vr3$). Note that we use the vector add instruction ($v.add$) here. We show an example of a vector add instruction in Figure 12.19.

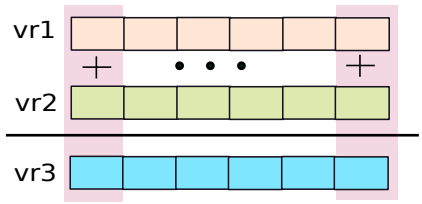


Figure 12.19: Example of a vector addition

Vector ISAs define similar operations for vector multiplication, division, and logical operations. Note that it is not necessary for a vector instruction to always have two input operands, which are vectors. We can multiply, a vector with a scalar, or we can have an instruction that operates on just one vector operand. For example, the SSE instruction set has dedicated instructions for computing trigonometric functions such as *sin*, and *cos*, for a set of floating point numbers packed in a vector register. If a vector instruction can simultaneously perform operations on n operands, then we say that we have n data lanes, and the vector instruction simultaneously performs an operation on all the n data lanes.

Definition 133
If a vector instruction can simultaneously perform operations on n operands, then we say that we have n data lanes, and the vector instruction simultaneously performs an operation on all the n data lanes.

The last step is to store the vector register in memory. Here again, there are two options. We can either store to contiguous memory locations, which is simpler, or save to non-contiguous locations. We can design two variants of the vector store instruction (contiguous and non-contiguous) on the lines of the two variants of vector load instructions ($v.ld$ and $v.sg.ld$). Sometimes it is necessary to introduce instructions that transfer data between scalar and vector registers. We shall not describe such instructions for the sake of brevity. We leave designing such instructions as an exercise for the reader.

12.5.3 A Practical Example using SSE Instructions

Let us now consider a practical example using the x86 based SSE instruction set. We shall not use actual assembly instructions. We shall instead use functions provided by the *gcc* compiler that act as wrappers for the assembly instructions. These functions are called *gcc intrinsics*.

Let us now solve the problem of adding two arrays of floating point numbers. In this case, we wish to compute $c[i] = a[i] + b[i]$, for all values of i.

The SSE instruction set contains 128-bit registers. Each register can be used to store four 32-bit floating point numbers. Hence, if we have an array of N numbers, we need to have $\lceil N/4 \rceil$

iterations, because we can add at the most 4 pairs of numbers in each cycle. In each iteration, we need to load vector registers, add them, and store the result in memory. This process of breaking up a vector computation into a sequence of loop iterations based on the sizes of vector registers is known as *strip mining*.

Definition 134
The process of breaking up a vector computation into a sequence of loop iterations based on the sizes of vector registers is known as strip mining. *For example, if a vector register can hold 16 integers, and we wish to operate on 1024 integer vectors, then we need a loop with 64 iterations.*

Example 155
Write a function in C/C++ to add the elements in the arrays a and b pairwise, and save the results in the array, c, using the SSE extensions to the x86 ISA. Assume that the number of entries in a and b are the same, and are a multiple of 4.
Answer:

vector addition

```
1  void sseAdd (const float a[], const float b[], float c[],  int N)
2  {
3          /* strip mining */
4          int numIters = N / 4;
5
6          /* iteration */
7          for (int i = 0; i < numIters; i++) {
8                  /* load the values */
9                  __m128 val1 = _mm_load_ps (a);
10                 __m128 val2 = _mm_load_ps (b);
11
12                 /* perform the vector addition */
13                 __m128 res = _mm_add_ps(val1, val2);
14
15                 /* store the result */
16                 _mm_store_ps(c, res);
17
18                 /* increment the pointers */
19                 a += 4 ; b += 4; c+= 4;
20         }
21 }
```

Let us consider the C code snippet in Example 155. We first calculate the number of iterations in Line 4. In each iteration, we consider a block of 4 array elements. In Line 9,

we load a set of four floating point numbers into the 128-bit vector variable, $val1$. $val1$ is mapped to a vector register by the compiler. We use the function *_mm_load_ps* to load a set of 4 contiguous floating point values from memory. For example, the function $\_mm\_load\_ps(a)$ loads four floating point values in the locations, a, $a+4$, $a+8$, and $a+12$ into a vector register. Similarly, we load the second vector register, $val2$, with four floating point values starting from the memory address, b. In Line 13, we perform the vector addition, and save the result in a 128-bit vector register associated with the variable res. We use the intrinsic function, *_mm_add_ps*, for this purpose. In Line 16, we store the variable, res, in the memory locations namely c, $c+4$, $c+8$, and $c+12$.

Before proceeding to the next iteration, we need to update the pointers a, b, and c. Since we process 4 contiguous array elements every cycle, we update each of the pointer by 4 (4 array elements) in Line 19.

We can quickly conclude that vector instructions facilitate bulk computations such as bulk loads/stores and adding a set of numbers pairwise, in one go. We compared the performance of this function, with a version of the function that does not use vector instructions on a quad core Intel Core i7 machine. The code with SSE instructions ran 2-3 times faster for million-element arrays. If we had wider SSE registers, then we could have gained more speedups. The latest AVX vector ISA on x86 processors supports 256 and 512-bit vector registers. Interested readers can implement the function shown in Example 155 using the AVX vector ISA, and compare the performance.

12.5.4 Predicated Instructions

We have up till now considered vector load, store, and ALU operations. What about branches? Typically, branches have a different connotation in the context of vector processors. For example, let us consider a processor with vector registers that are wide enough to hold 32 integers, and we have a program which requires us to pair-wise add only 18 integers, and then store them in memory. In this case, we cannot store the entire vector register to memory because we risk overwriting valid data.

Let us consider another example. Assume that we want to apply the function $inc10(x)$ on all elements of an array. In this case, we wish to add 10 to the input operand, x, if it is less than 10. Such patterns are very common in programs that run on vector processors, and thus we need additional support in vector ISAs to support them.

```
function inc10(x):
if (x < 10)
        x = x + 10;
```

Let us add a new variant of a regular instruction, and call it a *predicated instruction* (similar to conditional instructions in ARM). For example, we can create *predicated* variants of regular load, store, and ALU instructions. A predicated instruction executes if a certain condition is true, otherwise it does not execute at all. If the condition is false, a predicated instruction is equivalent to a *nop*.

Definition 135
A predicated instruction *is a variant of a normal load, store, or ALU instruction. It executes normally, if a certain condition is true. However, if the associated condition is false, then it gets converted to a nop. For example, the addeq instruction in the ARM ISA, executes like a normal add instruction if the last comparison has resulted in an equality. However, if this is not the case, then the add instruction does not execute at all.*

Let us now add support for predication in the *SimpleRisc* ISA. Let us first create a vector form of the *cmp* instruction, and call it *v.cmp*. It compares two vectors pair-wise, and saves the results of the comparison in the $v.flags$ register, which is a vector form of the $flags$ register. Each component of the $v.flags$ register contains an E and GT field, similar to the $flags$ register in a regular processor.

```
v.cmp vr1, vr2
```

This example compares $vr1$, and $vr2$, and saves the results in the $v.flags$ register. We can have an alternate form of this instruction that compares a vector with a scalar.

```
v.cmp vr1, 10
```

Now, let us define the predicated form of the vector add instruction. This instruction adds the i^{th} elements of two vectors, and updates the i^{th} element of the destination vector register, if the $v.flags[i]$ (i^{th} element of $v.flags$) register satisfies certain properties. Otherwise, it does not update the i^{th} element of the destination register. Let the generic form of the predicated vector add instruction be: *v.p.add*. Here, p is the predicate condition. Table 12.6 lists the different values that p can take.

Predicate Condition	Meaning
lt	less than
gt	greater than
le	less than or equal
ge	greater than or equal
eq	equal
ne	not equal

Table 12.6: List of conditions for predicated vector instructions in *SimpleRisc*

Now, let us consider the following code snippet.

```
v.lt.add vr3, vr1, vr2
```

Here, the value of the vector register $vr3$ is the sum of the vectors represented by $vr1$ and $vr2$. The predication condition is less than (lt). This means that if both the E and GT flags

are false for element i in the $v.flags$ register, then only we perform the addition for the i^{th} element, and set its value in the $vr3$ register. The elements in the $vr3$ register that are not set by the add instruction maintain their previous value. Thus, the code to implement the function $inc10(x)$ is as follows. We assume that $vr1$ contains the values of the input array.

```
v.cmp vr1, 10
v.lt.add vr1, vr1, 10
```

Likewise, we can define predicated versions of the load/store instructions, and other ALU instructions.

12.5.5 Design of a Vector Processor

Let us now briefly consider the design of vector processors. We need to add a vector pipeline similar to the scalar pipeline. In specific, the OF stage reads the vector register file for vector operands, and the scalar register file for scalar operands. Subsequently, it buffers the values of operands in the pipeline registers. The EX stage sends scalar operands to the scalar ALU, and sends vector operands to the vector ALU. Similarly, we need to augment the MA stage with vector load and store units. For most processors, the size of a cache block is an integral multiple of the size of a vector register. Consequently, vector load and store units that operate on contiguous data do not need to access multiple cache blocks. Hence, a vector load and store access is almost as fast as a scalar load and store access because the atomic unit of storage in a cache is a *block*. In both cases (scalar and vector), we read the value of a block and choose the relevant bytes using the column muxes. We need to change the structure of the L1 cache to read in more data at a time. Lastly, the writeback stage writes back scalar data to the scalar register file and vector data to the vector register file.

In a pipelined implementation of a vector processor, the interlock and forwarding logic is complicated. We need to take into account the conflicts between scalar instructions, between vector instructions, and between a scalar and vector instruction. The forwarding logic needs to forward values between different functional unit types, and thus ensure correct execution. Note that vector instructions need not always be as fast as their scalar counterparts. Especially, scatter-gather based vector load store instructions are slower. Since modern out-of-order pipelines already have dedicated support for processing variable latency instructions, vector instructions can seamlessly plug into this framework.

12.6 Interconnection Networks

12.6.1 Overview

Let us now consider the problem of interconnecting different processing and memory elements. Typically, multicore processors use a checkerboard design. Here, we divide the set of processors into *tiles*. A *tile* typically consists of a set of 2-4 processors. A tile has its private caches (L1 and possibly L2). It also contains a part of the shared last level cache (L2 or L3). The part of the shared last level cache that is a part of a given tile is known as a *slice* . Typically, a slice consists of 2-4 banks (see Section 12.4.5). Additionally, a tile, or a group of tiles might share a memory controller in modern processors. The role of the *memory controller* is to co-ordinate

the transfer of data between the on-chip caches, and the main memory. Figure 12.20 shows a representative layout of a 32 core multiprocessor. The cores have a darker color as compared to the cache banks. We use a tile size of 2 (2 processors and 2 cache banks), and assume that the shared L2 cache has 32 cache banks evenly distributed across the tiles. Moreover, each tile has a dedicated memory controller, and a structure called a *router*.

A router is a specialized unit, and is defined in Definition 136.

Definition 136

1. *A router sends messages originating from processors or caches in its tile to other tiles through the on chip network.*
2. *The routers are interconnected with each other via an on-chip network.*
3. *A message travels from the source router to the destination router (of a remote tile) via a series of routers. Each router on the way forwards the message to another router, which is closer to the destination.*
4. *Finally, the router associated with the destination tile forwards the message to a processor or cache in the remote tile.*
5. *Adjacent routers are connected via a* link*. A link is a set of passive copper wires that are used to transmit messages (more details in Chapter 13).*
6. *A router typically has many incoming links, and many outgoing links. One set of incoming and outgoing links connect it to processors and caches in its tile. Each link has a unique identifier.*
7. *A router has a fairly complicated structure, and typically consists of a 3 to 5-stage pipeline. Most designs typically dedicate pipeline stages to buffering a message, computing the id of the outgoing link, arbitrating for the link, and sending the message over the outgoing link.*
8. *The arrangement of routers and links is referred to as the* on chip network*, or* network on chip*. It is abbreviated as the* NOC*.*
9. *Let us refer to each router connected to the NOC as a* node*. Nodes communicate with each other by sending* messages*.*

During the course of the execution of a program, it sends billions of messages over the NOC. The NOC carries coherence messages, LLC (last level cache) request/response messages, and messages between caches, and memory controllers. The operating system also uses the NOC to send messages to cores for loading and unloading threads. Due to the high volume of messages, large parts of the NOC often experience a sizable amount of congestion. Hence, it is essential to design NOCs that reduce congestion to the maximum extent possible, are easy to design

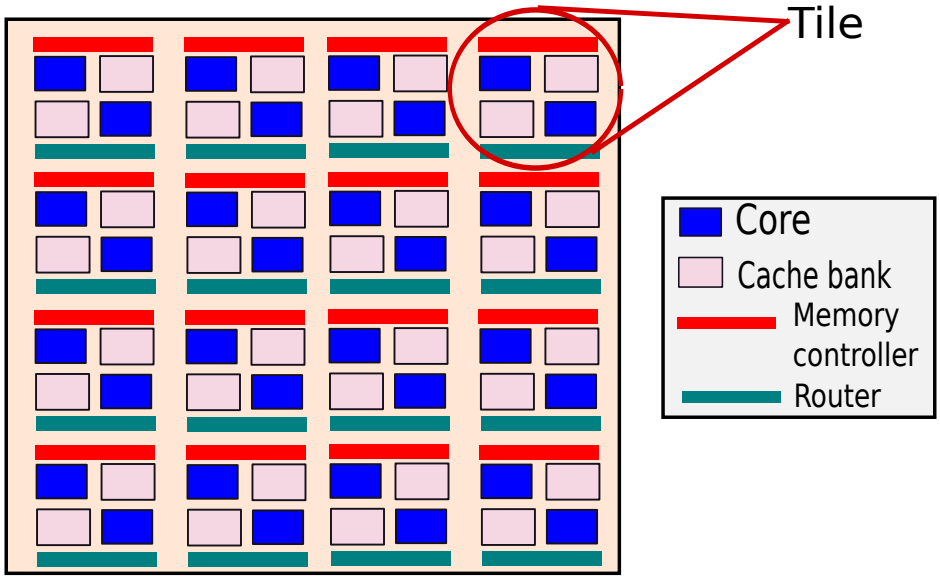


Figure 12.20: The layout of a multicore processor

and manufacture, and ensure that messages quickly reach their destination. Let us define two important properties of an NOC namely bisection bandwidth and diameter.

12.6.2 Bisection Bandwidth and Network Diameter

Bisection Bandwidth

Let us consider a network topology where the vertices are the nodes, and the edges between the vertices are the links. Suppose there is a link failure, or for some other reason such as congestion, a link is unavailable, then it should be possible to route messages through alternate paths. For example, let us consider a network arranged as a ring. If one link fails, then we can always send a message through the other side of the ring. If we were sending the message in a clockwise fashion, we can send it in an anti-clockwise fashion. However, if there are two link failures, it is possible that the network can get disconnected into two equal parts. We would thus like to maximize the number of link failures that are required to completely disconnect the network into sizable large parts (possibly equal). Let us refer to the number of such failures as the *bisection bandwidth*. The bisection bandwidth is a measure of the reliability of the network. It is precisely defined as the minimum number of links that need to fail to partition the network into two equal parts.

There can be an alternative interpretation of the bisection bandwidth. Let us assume that nodes in one half of the network are trying to send messages to nodes in the other half of the network. Then the number of messages that can be simultaneously sent is at least equal to the bisection bandwidth. Thus, the bisection bandwidth is also a measure of the bandwidth of a network.

Definition 137
The bisection bandwidth *is defined as the minimum number of link failures that are required to partition a network into two equal parts.*

Network Diameter

We have discussed reliability, and bandwidth. Now, let us focus on latency. Let us consider pairs of nodes in the network. Let us subsequently consider the shortest path between each pair of nodes. Out of all of these shortest paths, let us consider the path that has the maximum length. The length of this path is an upper bound on the proximity of nodes in the network, and is known as the *diameter* of the network. Alternatively, we can interpret the diameter of a network as an estimate of the worst case latency between any pair of nodes.

Definition 138
Let us consider all pairs of nodes, and compute the shortest path between each pair. The length of the longest such path is known as the network diameter*. It is a measure of the worst case latency of the network.*

12.6.3 Network Topologies

Let us review some of the most common network topologies in this section. Some of these topologies are used in multicore processors. However, most of the complex topologies are used in loosely coupled multiprocessors that use regular Ethernet links to connect processors. For each topology, let us assume that it has N nodes. For computing the bisection bandwidth, we can further make the simplistic assumption that N is divisible by 2. Note that measures like the bisection bandwidth, and the diameter are approximate measures, and are merely indicative of broad trends. Hence, we have the leeway to make simplistic assumptions. Let us start out with considering simpler topologies that are suitable for multicores. We need to aim for a high bisection bandwidth, and low network diameter.

Chain and Ring

Figure 12.21 shows a chain of nodes. Its bisection bandwidth is 1, and the network diameter is $N-1$. This is our worst configuration. We can improve both the metrics by considering a ring of nodes (Figure 12.22). The bisection bandwidth is now 2, and the network diameter is $N/2$. Both of these topologies are fairly simple, and have been superseded by other topologies. Let us now consider a topology known as a *fat tree*, which is commonly used in cluster computers. A cluster computer refers to a loosely coupled multiprocessor that consists of multiple processors connected over the local area network.

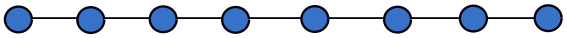


Figure 12.21: Chain

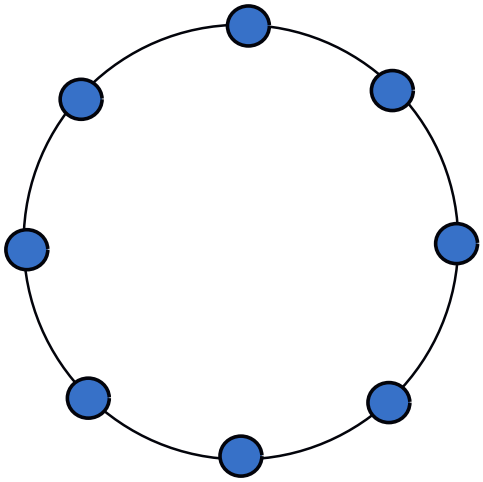


Figure 12.22: Ring

Definition 139
A cluster computer refers to a loosely coupled computer that consists of multiple processors connected over the local area network.

Fat Tree

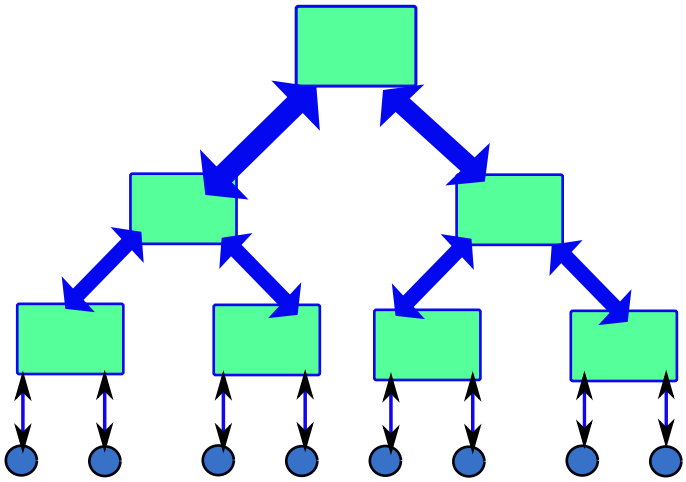


Figure 12.23: Fat Tree

Figure 12.23 shows a fat tree. In a fat tree, all the nodes are at the leaves, and all the internal nodes of the tree are routers dedicated to routing messages. Let us refer to these internal nodes as *switches*. A message from node a to node b first travels to the closest node that is a common ancestor of both a and b. Then it travels downwards towards b. Note that the density of messages is the highest near the root. Hence, to avoid contention, and bottlenecks, we gradually increase the number of links connecting a node and its children, as we move towards the root. This strategy reduces the message congestion at the root node.

In our example, two subtrees are connected to the root node. Each subtree has 4 nodes. At the most the root can receive 4 messages from each subtree. Secondly, at the most, it needs to send 4 messages to each subtree. Assuming a duplex link, the root needs to have 4 links connecting it to each of its children. Likewise, the next level of nodes need 2 links between them and each of their child nodes. The leaves need 1 link each. We can thus visualize the tree growing fatter, as we proceed towards the root, and hence it is referred to as a *fat tree*.

The network diameter is equal to 2log(N). The bisection bandwidth is equal to the minimum number of links that connect the root node to each of its children. If we assume that the tree is designed to ensure that there is absolutely no contention for links at the root, then we need to connect the root with $N/2$ links to each subtree. Thus, the bisection bandwidth in this case is $N/2$. Note that we do not allot $N/2$ links between the root and its children in most practical scenarios. This is because the probability of all the nodes in a subtree transmitting messages at the same time is low. Hence, in practice we reduce the number of links at each level.

Mesh and Torus

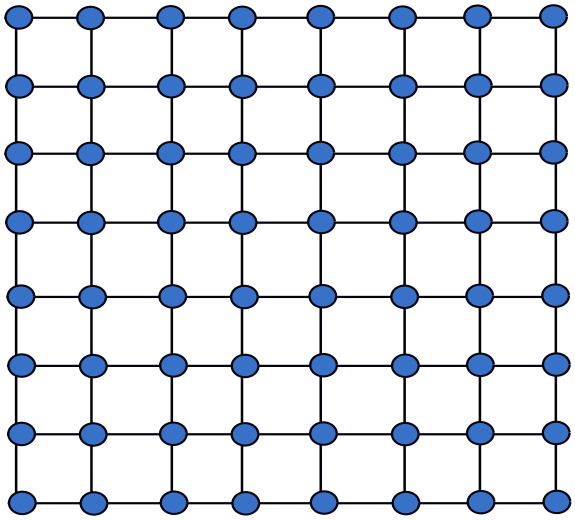


Figure 12.24: Mesh

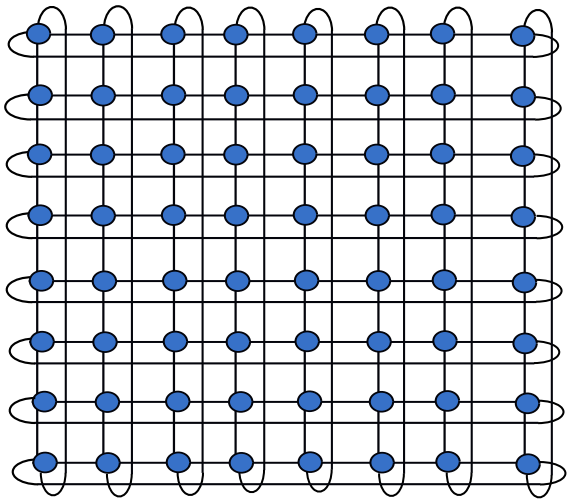


Figure 12.25: Torus

Let us now look at topologies that are more suitable for multicores. One of the most common topologies is a mesh where all the nodes are connected in a matrix like fashion (see Figure 12.24). Nodes at the corner have two neighbors, nodes on the rim have three neighbors,

and the rest of the nodes have four neighbors. Let us now compute the diameter and bisection bandwidth of a mesh. The longest path is between two corner nodes. The diameter is thus equal to ($2\sqrt{N}$ - 2). To divide the network into two equal halves we need to either split the mesh in the middle (horizontally or vertically). Since we have $\sqrt{N}$ nodes in a row, or column, the bisection bandwidth is equal to $\sqrt{N}$. The mesh is better than a chain and a ring in terms of these parameters.

Unfortunately, the mesh topology is asymmetric in nature. Nodes that are at the rim of the mesh are far away from each other. Consequently, we can augment a mesh with cross links between the extremities of each row and column. The resulting structure is known as a *torus*, and is shown in Figure 12.25. Let us now look at the properties of tori (plural of torus). In this case, nodes on opposite sides of the rim of the network, are only one hop apart. The longest path is thus between any of the corner nodes and a node at the center of the torus. The diameter is thus again equal to (ignoring small additive constants) $\sqrt{N}/2 + \sqrt{N}/2 = \sqrt{N}$. Recall that the length of each side of the torus is equal to $\sqrt{N}$.

Now, to divide the network into two equal parts let us split it horizontally. We need to thus snap $\sqrt{N}$ vertical links, and $\sqrt{N}$ cross links (links between the ends of each column). Hence, the bisection bandwidth is equal to $2\sqrt{N}$.

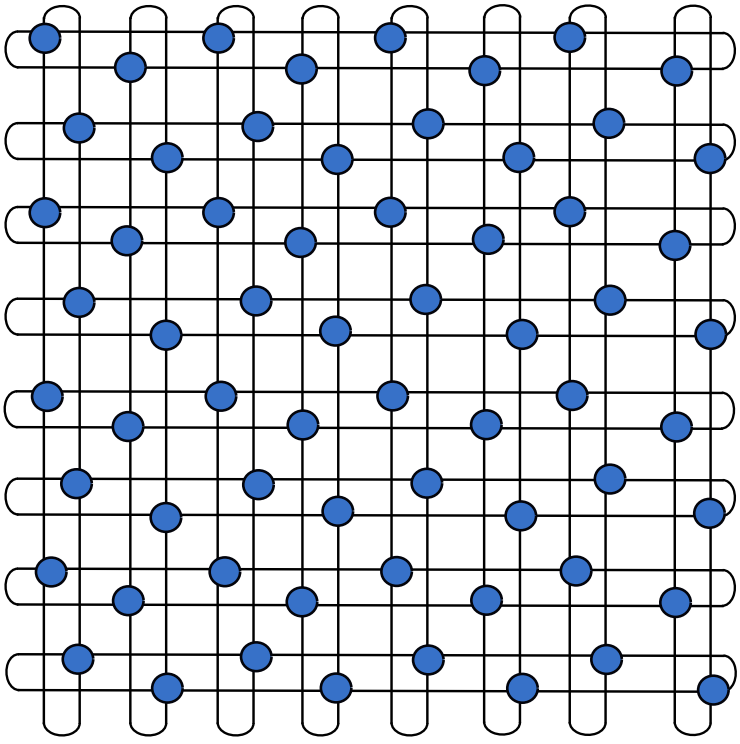


Figure 12.26: Folded Torus

By adding $2\sqrt{N}$ cross links ($\sqrt{N}$ for rows, and $\sqrt{N}$ for columns), we have halved the diameter, and doubled the bisection bandwidth of a torus. However, this scheme still has some problems. Let us elaborate.

While defining the diameter, we made an implicit assumption that the length of every link is almost the same, or alternatively the time a message takes to traverse a link is almost the same for all the links in the network. Hence, we defined the diameter in terms of the number of links that a message traverses. This assumption is not very unrealistic because in general the propagation time through a link is small as compared to the latencies of routers along the way. Nevertheless, there are limits to the latency of a link. If a link is very long, then our definition of the diameter needs to be revised. In the case of tori, we have such a situation. The cross links are physically $\sqrt{N}$ times longer than regular links between adjacent nodes. Hence, as compared to a mesh, we have not significantly reduced the diameter in practice because nodes at the ends of a row are still far apart.

We can fortunately solve this problem by using a slightly modified structure called a *folded torus* as shown in Figure 12.26. Here, the topology of each row and column is like a ring. One half of the ring consists of regular links that were originally a part of the mesh topology, and the other half comprises the cross links that were added to convert a mesh into a torus. We alternately place nodes on the regular links and on the cross links. This strategy ensures that the distance between adjacent nodes in a folded torus is twice the distance between adjacent nodes in a regular torus. However, we avoid the long cross links ($\sqrt{N}$ hops long) between the two ends of a row or column.

The bisection bandwidth and the diameter of the network remain the same as that of the torus. In this case, there are several paths that can qualify as the longest path. However, the path between a corner to the center is not the longest. One of the longest paths is between opposite corners. The folded torus is typically the preferred configuration in multicore processors because it avoids long cross links.

Hypercube

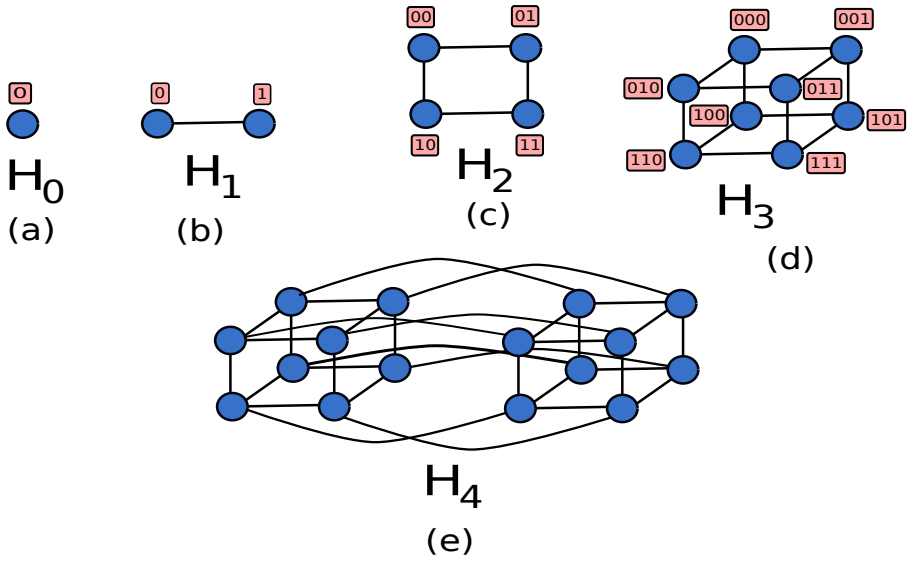


Figure 12.27: Hypercube

Let us now consider a network that has $O(log(N))$ diameter. These networks use a lot of links; hence, they are not suitable for multicores. However, they are used often in larger cluster computers. This network is known as a *hypercube*. A hypercube is actually a family of networks, where each network has an *order*. A hypercube of order k is referred to as H_k.

Figure 12.27(a) shows a hypercube of order 0 (H_0). It is a single point. To create a hypercube of order 1, we take two copies of a hypercube of order 0, and connect the corresponding points with lines. In this case, H_0 has a single point. Therefore, H_1 is a simple line segment (see Figure 12.27(b)). Now, let us follow the same procedure to create a hypercube of order 2. We place two copies of H_1 close to each other and connect corresponding points with lines. Hence, H_2 is a rectangle (see Figure 12.27(c)). Let us follow the same procedure to create a hypercube of order 3 in Figure 12.27(d). This network is equivalent to a normal cube (the "cube" in hypercube). Finally, Figure 12.27(e), shows the topology of H_4, where we connect the corresponding points of two cubes. We can proceed in a similar manner to create hypercubes of order k.

Let us now investigate the properties of a hypercube. The number of nodes in H_k is equal to twice the number of nodes in H_{k-1}. This is because, we form H_k by joining two copies of H_{k-1}. Since H_1 has 1 node, we can conclude that H_k has 2^k nodes. Let us propose a method to label the nodes of a hypercube as shown in Figure 12.27. We label the single node in H_0 as 0. When we join two copies of a hypercube of order, $k-1$, we maintain the same labeling of nodes for $(k-1)$ least significant digits. However, for nodes in one copy, we set the MSB as 1, and for nodes in the other copy, we set the MSB to be 0.

Let us consider, H_2 (Figure 12.27(c)). The nodes are labeled 00, 01, 10, and 11. We have created a similar 3-bit labeling for H_3 (Figure 12.27(d)). In our labeling scheme, a node is connected to all other nodes that have a label differing in only one bit. For example, the neighbors of the node 101, are 001, 111, and 100. A similar labeling for H_k requires $k = log(N))$ bits per node.

This insight will help us compute the diameter of a hypercube. Let us explain through an example. Consider the 8 node hypercube, H_3. Assume that we want a message to travel from node A (000) to node B (110). Let us scan the labels of both the nodes from the MSB to the LSB. The first bit (MSB) does not match. Hence, to make the first bit match, let us route the message to node A_1 (100). Let us now scan the next bit. Here, again there is a mismatch. Hence, let us route the message to node A_2 (110). Now, we take a look at the third bit (LSB), and find it to match. The message has thus reached its destination. We can follow the same approach for an N-bit hypercube. Since each node has a $log(N)$-bit label, and in each step we flip at most one bit in the label of the current node, we require a maximum of $log(N)$ routing steps. Thus, the network diameter is $log(N)$.

Let us now compute the bisection bandwidth. We shall state the result without proof because the computation of the bisection bandwidth of a hypercube requires a thorough theoretical treatment of hypercubes. This is beyond the scope of this book. The bisection bandwidth of an N-node hypercube is equal to $N/2$.

Butterfly

Let us now look at our last network called the *butterfly* that also has $O(log(N))$ diameter, yet, is suitable for multicores. Figure 12.28 shows a butterfly network for 8 nodes. Each node is

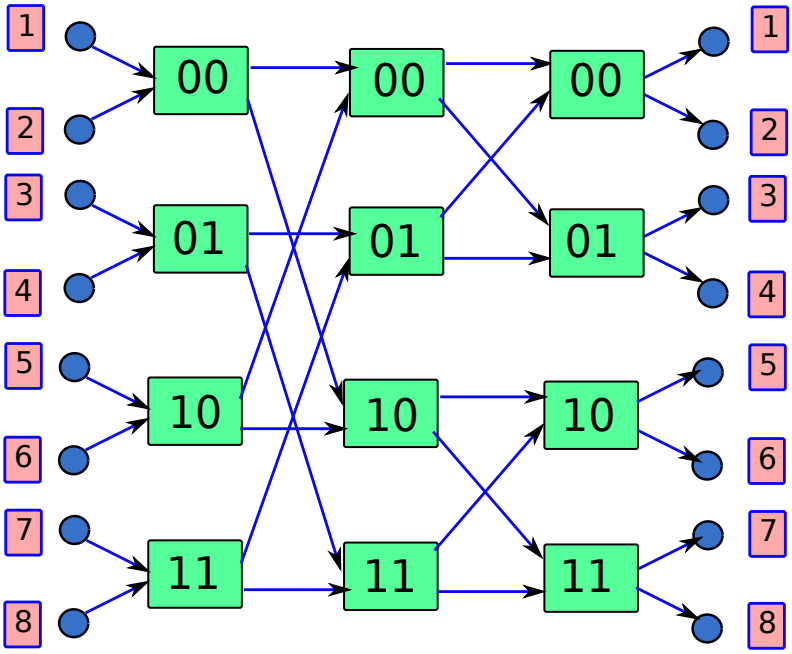


Figure 12.28: Butterfly

represented by a circle. Along with the nodes, we have a set of switches or internal nodes (shown with rectangles) that route messages between nodes. The messages start on the left side, pass through the switches, and reach the right side of the diagram. Note that the nodes on the leftmost and rightmost sides of the figure are actually the same set of nodes. We did not want to add left to right cross links to avoid complicating the diagram; hence, we show the set of nodes in duplicate.

Let us start from the left side of the diagram. For N nodes (assuming N is a power of 2), we have $N/2$ switches in the first column. Two nodes are connected to each switch. In our example in Figure 12.28, we have labeled the nodes $1 \ldots 8$. Each node is connected to a switch. Once a message enters a switch, it gets routed to the correct switch in the rightmost column through the network of switches. For 8 nodes, we have 4 switches in each column. For each column, we have labeled them $00 \ldots 11$ in binary.

Let us consider an example. Assume that we want to send a message from node 4 to node 7. In this case, the message enters switch 01 in the first column. It needs to reach switch 11 in the third column. We start out by comparing the MSBs of the source switch, and the destination switch. If they are equal, then the message proceeds horizontally rightwards to a switch in the adjacent column and same row. However, if the MSBs are unequal (as is the case in our example), then we need to use the second output link to send it to a switch in a different row in the next column. The label of the new switch differs from the label of the original switch by just 1 bit, which is the MSB bit in this case. Similarly, to proceed from the second to the third column, we compare the second bit position (from the MSB). If they are equal, then the message proceeds horizontally, otherwise it is routed to a switch such that the first two bits

match. In this case, the label of the switch that we choose in the second column is 11. The first two bits match the label of the switch in the third column. Hence, the message proceeds horizontally, and is finally routed to node 7.

We can extend this method for a butterfly consisting of k columns. In the first column, we compare the MSB. Similarly, in the i^{th} column, we compare the i^{th} bit (MSB is the 1^{st} bit). Note that the first $k-i$ bits of the label of the switch that handles the message in the i^{th} column are equal to the first $k-i$ bits of the destination switch. This strategy ensures that the message gets routed to ultimately the correct switch in the k^{th} column.

Now, let us investigate the properties of the network. Let us assume that we have N nodes, where N is a power of 2. We require $log(N)$ columns, where each column contains $N/2$ switches. Thus, we require an additional $Nlog(N)/2$ switches. An astute reader would have already concluded that routing a message in a butterfly network is almost the same as routing in a message in a hypercube. In every step we increase the size of the matching prefix between the labels of the source and destination switches by 1. We thus require $log(N)+1$ steps (1 additional step for sending the message to the destination node from the last switch) for sending a message between a pair of nodes. We can thus approximate the diameter of the network to $log(N)$.

Let us now compute the bisection bandwidth. Let us consider our example with 8 nodes first. Here, we can split the network horizontally. We thus need to snap 4 links, and hence the bisection bandwidth of the network shown in Figure 12.28 is 4. Let us now consider $N > 8$ nodes. In this case, also the best solution is to split the network horizontally. This is because if we draw an imaginary horizontal line between the $(N/4)^{th}$ and $(N/4+1)^{th}$ row of switches then it will only intersect the links between the first and second columns. The outgoing links of the rest of the columns will not intersect with our imaginary line. They will either be below it or above it. Since each switch in the first column has only one outgoing link that intersects the imaginary line, a total of $N/2$ links intersect the imaginary line. All of these links need to be disconnected to divide the network into two equal parts. Hence, the bisection bandwidth is equal to $N/2$.

Comparison of Topologies

Let us now compare the topologies with respect to four parameters – number of internal nodes (or switches), number of links, diameter, and bisection bandwidth in Table 12.7. In all the cases, we assume that the networks have N nodes that can send and receive messages, and N is a power of 2.

12.7 Summary and Further Reading

12.7.1 Summary

Summary 12

1. *Processor frequency and performance is beginning to saturate.*

Topology	# Switches	# Links	Diameter	Bisection Bandwidth
Chain	0	N-1	N-1	1
Ring	0	N	$N/2$	2
Fat Tree	N - 1	N log(N)‡	2 log(N)	N/2†
Mesh	0	$2N - 2\sqrt{N}$	$2\sqrt{N} - 2$	$\sqrt{N}$
Torus	0	2N	$\sqrt{N}$	$2\sqrt{N}$
Folded Torus	0	2N	$\sqrt{N}$	$2\sqrt{N}$
Hypercube	0	$Nlog(N)/2$	$log(N)$	$N/2$
Butterfly	$Nlog(N)/2$	$N + Nlog(N)$	$log(N) + 1$	$N/2$
‡ Assume that the size of each link is equal to the size of the subtree under it. † Assume that the capacity of each link is equal to the number of leaves in its subtree				

Table 12.7: Comparison of topologies

2. *Concomitantly, the number of transistors per chip is roughly doubling very two years as per the original predictions of Gordon Moore. This empirical law is known as the* **Moore's Law**.

3. *The additional transistors are not being utilized to make a processor larger, or more complicated. They are instead being used to add more processors on chip. Each such processor is known as a* core*, and a chip with multiple cores is known as a* multicore *processor.*

4. *We can have multiprocessor systems where the processors are connected over the network. In this case the processors do not share any resources between them, and such multiprocessors are known as* loosely coupled multiprocessors*. In comparison, multicore processors, and most small sized server processors that have multiple processors on the same motherboard, share resources such as the I/O devices, and the main memory. Programs running on multiple processors in these systems might also share a part of their virtual address space. These systems are thus referred to as* strongly coupled multiprocessors*.*

5. *Multiprocessor systems can be used to run multiple sequential programs simultaneously, or can be used to run parallel programs. A parallel program contains many sub-programs that run concurrently. The sub-programs co-operate among themselves to achieve a bigger task. When sub-programs share their virtual memory space, they are known as* threads*.*

6. *Parallel programs running on strongly coupled multiprocessors typically communicate values between themselves by writing to a shared memory space. In comparison, programs running on loosely coupled multiprocessors communicate by passing messages between each other.*

7. *Most parallel programs have a sequential section, and a parallel section. The parallel section can be divided into smaller units and distributed among the processors of a multiprocessor system. If we have N processors, then we ideally expect the parallel section to be sped up by a factor of N. An equation describing this relationship is known as the Amdahl's Law. The speedup, S is given by:*

$$S = \frac{T_{seq}}{T_{par}} = \frac{1}{f_{seq} + \frac{1-f_{seq}}{P}}$$

8. *The Flynn's taxonomy classifies computing systems into four types : SISD (single instruction, single data), SIMD (single instruction, multiple data), MISD (multiple instruction, single data), and MIMD (multiple instruction, multiple data).*

9. *The memory system in modern shared memory MIMD processors is in reality very complex. Coherence and consistency are two important aspects of the behavior of the memory system.*

10. *Coherence refers to the rules that need to be followed for accessing the same memory location. Coherence dictates that a write is never lost, and all writes to the same location are seen in the same order by all the processors.*

11. *Consistency refers to the behavior of the memory system with respect to different memory locations. If memory accesses from the same thread get reordered by the memory system (as is the case with modern processors), many counter-intuitive behaviors as possible. Hence, most of the time we reason in terms of the* sequentially consistent memory model *that prohibits reordering of messages sent to the memory system by the same thread. In practice, multiprocessors follow a weak consistency model that allows arbitrary reorderings between accesses that access different memory addresses. We can still write correct programs because such models define synchronization instructions (example: $fence$) that try to enforce an ordering between memory accesses when required.*

12. *We can either have a large shared cache, or multiple private caches (one for each core or set of cores). Shared caches can be made more performance and power efficient by dividing it into a set of subcaches known as banks. For a set of private caches to logically function as one large shared cache, we need to implement* cache coherence.

    (a) *The snoopy cache coherence protocol connects all the processors to a shared bus.*

    (b) *The MSI write-update protocol works by broadcasting every write to all the cores.*

    (c) *The MSI write-invalidate protocol guarantees coherence by ensuring that only one cache can write to a block at any single point of time.*

13. *To further improve performance, we can implement a multithreaded processor that shares a pipeline across many threads. We can either quickly switch between threads (fine and coarse-grained multithreading), or execute instructions from multiple threads in the same cycle using a multi-issue processor (simultaneous multithreading).*

14. *SIMD processors follow a different approach. They operate on arrays of data at once.* Vector processors *have a SIMD instruction set. Even though, they are obsolete now, most modern processors have vector instructions in their ISA.*
    (a) *Vector arithmetic/logical instructions fetch their operands from the vector register file, and operate on large vectors of data at once.*
    (b) *Vector load-store operations can either assume that data is stored in contiguous memory regions, or assume that data is* scattered *in memory.*
    (c) *Instructions on a branch path are implemented as predicated instructions in vector ISAs.*
15. *Processors and memory elements are connected through an interconnection network. The basic properties of an interconnection network are the diameter (worst case end-to-end delay), and the bisection bandwidth (number of links that need to be snapped to partition the network equally). We discussed several topologies: chain, ring, fat tree, mesh, torus, folded torus, hypercube, and butterfly.*

12.7.2 Further Reading

The reader should start by reading the relevant sections in advanced textbooks on computer architecture. The first recommendation is the textbook written by your author on advanced computer architecture [Sarangi,]. Other textbooks [Culler et al., 1998, Hwang, 2003, Baer, 2010, Jacob, 2009] in this space can also be consulted, especially the book by Jacob [Jacob, 2009] for details of the memory system.

For parallel programming, the reader can start with Michael Quinn's book on parallel programming with OpenMP and MPI [Quinn, 2003]. The formal MPI specifications are available at `http://www.mpi-forum.org`. For an advanced study of cache coherence the reader can start with the survey on coherence protocols by Stenstrom [Stenstrom, 1990], and then look at one of the earliest practical implementations [Borrill, 1987]. The most popular reference for memory consistency models is a tutorial by Adve and Gharachorloo [Adve and Gharachorloo, 1996], and a paper published by the same authors [Gharachorloo et al., 1992]. For a different perspective on memory consistency models in terms of ordering, and atomicity, readers can refer to [Arvind and Maessen, 2006]. [Guiady et al., 1999] looks at memory models from the point of view of performance. [Peterson et al., 1991] and [russell, 1978] describe two fully functional SIMD machines. For interconnection networks the reader can refer to [Jerger and Peh, 2009].

Exercises

Overview of Multiprocessor Systems

Ex. 1 — Differentiate between *strongly coupled*, and *loosely coupled* multiprocessors.

Ex. 2 — Differentiate between *shared memory*, and *message passing* based multiprocessors.

Ex. 3 — Why is the evolution of multicore processors a direct consequence of Moore's Law?

Ex. 4 — The fraction of the potentially parallel section in a program is 0.6. What is the maximum speedup that we can achieve over a single core processor, if we run the program on a quad-core processor?

Ex. 5 — You need to run a program, 60% of which is strictly sequential, while the rest 40% can be fully parallelized over a maximum of 4 cores. You have 2 machines:

(a) A single core machine running at 3.2 GHz

(b) A 4-core machine running at 2.4 GHz

Which machine is better if you have to minimize the total time taken to run the program? Assume that the two machines have the same IPC per thread and only differ in the clock frequency and the number of cores.

* **Ex. 6** — Consider a program, which has a sequential and a parallel portion. The sequential portion is 40% and the parallel portion is 60%. Using Amdahl's law, we can compute the speedup with n processors, as $S(n)$. However, increasing the number of cores increases the cost of the entire system. Hence, we define a utility function, $g(n)$, of the form:

$$g(n) = e^{-n/3}(2n^2 + 7n + 6)$$

The buyer wishes to maximize $S(n) \times g(n)$. What is the optimal number of processors, n?

Ex. 7 — Define the terms: SISD, SIMD, MISD, and MIMD. Give an example of each type of machine.

Ex. 8 — What are the two classes of MIMD machines introduced in this book?

Coherence and Consistency

Ex. 9 — What are the axioms of cache coherence?

Ex. 10 — Define sequential and weak consistency.

Ex. 11 — Is the outcome (t1,t2) = (2,1) allowed in a system with coherent memory?

Thread 1:	Thread 2:
`x = 1;`	`t1 = x;`
`x = 2;`	`t2 = x;`

Ex. 12 — Assume that all the global variables are initialized to 0, and all variables local to a thread start with 't'. What are the possible values of $t1$ for a sequentially consistent system, and a weakly consistent system? (source [Adve and Gharachorloo, 1996])

Thread 1:

```
x = 1;
y = 1;
```

Thread 2:

```
while(y == 0){}
t1 = x;
```

Ex. 13 — Is the outcome (t1,t2) = (1,1) possible in a sequentially consistent system?

Thread 1:

```
x = 1;
if(y == 0)
        t1 = 1;
```

Thread 1:

```
y = 1;
if(x == 0)
        t2 = 1;
```

Ex. 14 — Is the outcome $t1 \neq t2$ possible in a sequentially consistent system? (source [Adve and Gharachorloo, 1996])

Thread 1:

```
z = 1;
x = 1;
```

Thread 2:

```
z = 2;
y = 1;
```

Thread 3:

```
while (x != 1) {}
while (y != 1) {}
t1 = z;
```

Thread 4:

```
while (x != 1) {}
while (y != 1) {}
t2 = z;
```

* **Ex. 15** — Is the outcome (t1 = 0) allowed in a system with coherent memory and atomic writes? Consider both sequential and weak consistency?

Thread 1:

```
x = 1;
```

Thread 2:

```
while(x != 1) {}
y = 1;
```

Thread 3:

```
while (y != 1) {}
t1 = x;
```

* **Ex. 16** — Consider the following code snippet for implementing a *critical section*. A critical section is a region of code that can only be executed by one thread at any single point of time. Assume that we have two threads with ids 0 and 1 respectively. The function *getTid*() returns the id of the current thread.

```
void enterCriticalSection() {
        tid = getTid();
        otherTid = 1 - tid;
        interested[tid] = true;
        flag = tid;

        while ( (flag == tid) && (interested[otherTid] == 1) ) {}
}

void leaveCriticalSection{
        tid = getTid();
        interested[tid] = false;
}
```

Is it possible for two threads to be in the critical section at the same point of time?

Ex. 17 — In the snoopy protocol, why do we write back data to the main memory upon an M to S transition?

Ex. 18 — Assume that two nodes desire to transition from the S state to the M state at exactly the same point of time. How will the snoopy protocol ensure that only one of these nodes enters the M state, and finishes its write operation? What happens to the other node?

Ex. 19 — The snoopy protocol clearly has an issue with scalability. If we have 64 cores with a private cache per core, then it will take a long time to broadcast a message to all the caches. Can you propose solutions to circumvent this problem?

Ex. 20 — Let us assume a cache coherent multiprocessor system. The L1 cache is private and the coherent L2 cache is shared across the processors. Let us assume that the system issues a lot of I/O requests. Most of the I/O requests perform DMA (Direct Memory Access) from main memory. It is possible that the I/O requests might overwrite some data that is already present in the caches. In this case we need to extend the cache coherence protocol that also takes I/O accesses into account. Propose one such protocol.

Ex. 21 — Let us define a new state in the traditional MSI states based snoopy protocol. The new E state refers to the "exclusive" state, in which a processor is sure that no other cache contains the block in a valid state. Secondly, in the E state, the processor hasn't modified the block yet. What is the advantage of having the E state? How are evictions handled in the E state?

Ex. 22 — Show the state transition diagrams for an MSI protocol with a directory. You need to show the following:

1.Structure of the directory

2.State transition diagram for events received from the host processor.

3.State transition diagram for events received from the directory.

4.State transition diagram for an entry in the directory (if required).

Ex. 23 — Assume that we have a system with private L1 and L2 caches. The L1 layer is not coherent. However, the L2 layer maintains cache coherence. How do we modify our MSI snoopy protocol to support cache coherence for the entire system?

Ex. 24 — In the snoopy write-invalidate protocol, when should a processor actually perform the write operation? Should it perform the write as soon as possible, or should it wait for the write-invalidate message to reach all the caches? Explain your answer.

* **Ex. 25** — Assume that a processor wants to perform an atomic exchange operation between two memory locations a and b. a and b cannot be allocated to registers. How will you modify the MSI coherence protocol to support this operation? Before proceeding with the answer think about what are the things that can go wrong. An exchange is essentially equivalent to the following sequence of operations: (1) temp = a; (2) a = b; (3) b = temp. If a read arrives between operations (2) and (3) it might get the wrong value of b. We need to prevent this situation.

** **Ex. 26** — Assume that we want to implement an instruction called $MCAS$. The $MCAS$ instruction takes k (known and bounded) memory locations as arguments, a set of k old values, and a set of k new values. Its pseudo-code is shown below. We assume here that mem is a hypothetical array representing the entire memory space.

```
/* multiple compare and set */
boolean MCAS(int memLocs[], int oldValues[], int newValues[]){
        /* compare */
        for(i=0; i < k; i++) {
                if(mem[memLocs[i]] != oldValues[i]) {
                        return false;
                }
        }

        /* set */
        for(i=0; i < k; i++) {
                mem[memLocs[i]] = newValues[i];
        }

        return true;
}
```

The challenge is to implement this instruction such that it appears to execute instantaneously. Let us look at some subtle cases. Assume that we want to write (4,5,6) to three memory locations if their previous contents are (1,2,3). It is possible that after writing 4, and 5, there is a small delay. During this time another thread reads the three memory locations, and concludes that their values are 4,5, and 3 respectively. This result is incorrect because it violates our assumption that $MCAS$ executes instantaneously. We should either read (1,2,3) or (4,5,6).
Now, let us look at the case of reading the three memory locations. Let us say that their initial values are 1,2, and 0. Our $MCAS$ instruction reads the first two locations and since they are equal to the old values, proceeds to the third location. Before reading it, a store operation from another thread changes the values of the three locations as follows. (1,2,0) $\rightarrow$ (5,2,0) $\rightarrow$ (5,2,3). Subsequently, the $MCAS$ instruction takes a look at the third memory location and finds it to be 3. Note that the three memory locations were never equal to (1,2,3). We thus arrive at a wrong conclusion.
How should we fix these problems? We want to implement an $MCAS$ instruction purely in hardware, which provides an illusion of instantaneous execution. It should be free of deadlocks, and should complete in a finite amount of time. How can we extend our coherence protocols to implement it?

*** **Ex. 27** — Assume a processor that has a sequentially consistent(SC) memory. We implement SC by making each thread wait for a memory request to complete before issuing the next request. Now, assume that we modify the architecture by allowing a processor to read a value that the immediately preceding instruction has written without waiting for it to complete. Prove that the memory system still follows SC.

*** **Ex. 28** — Assume a processor with a weak consistency model. Let us run a "properly

labeled" program on it. A *properly labeled*(PL) program does not allow conflicting accesses (read-write, write-read, or write-write) to a shared variable at the same time. For example, the following code sequence is not properly labeled because it allows x to be modified concurrently.

Thread 1:	Thread 2:
`x = 0`	`x = 1`

In reality, the coherence protocol orders one write access before the other. Nevertheless, both the threads **try** to modify x concurrently at the programmer's level. This is precisely the behavior that we wish to avoid.

In a PL program, two threads do not **try** to modify x at the same time. This is achieved by having two magic instructions known as *lock* and *unlock*. Only one thread can lock a memory location at any point of time. If another thread tries to lock the location before it is unlocked, then it stalls till the lock is free. If multiple threads are waiting on the same lock, only one of them is given the lock after a *unlock* instruction. Secondly, both the *lock* and *unlock* instructions have a built-in $fence$ operation, and all the *lock* and *unlock* instructions execute in program order. The PL version of our program is as follows:

Thread 1:	Thread 2:
`lock(x)`	`lock(x)`
`x = 0`	`x = 1`
`unlock(x)`	`unlock(x)`

We can thus think of a lock-unlock block as a sequential block that can only be executed by one thread at a given time. Moreover, assume that a lock-unlock block can only have one memory instruction inside it.

Now, prove that all PL programs running on a weakly consistent machine have a sequentially consistent execution. In other words we can interleave the memory accesses of all the threads such that they appear to be executed by a single cycle processor that switches among the threads. [HINT: Construct access graphs for your system, and prove that they are acyclic.]

Multithreading

Ex. 29 — What is the difference between a fine grained and coarse grained multithreaded machine?

Ex. 30 — Describe a simultaneous multithreaded (SMT) processor in detail.

Ex. 31 — Describes the steps that we need to take to ensure that an SMT processor executes correctly.

Ex. 32 — Assume a mix of workloads in a 4-way SMT processor. 2 threads are computationally intensive, 1 thread is I/O intensive, and the last thread sleeps for a long time. Design an efficient instruction selection scheme.

Interconnection Networks

Ex. 33 — What is the bisection bandwidth and diameter of a 2D $n \times n$ mesh?

Ex. 34 — What is the bisection bandwidth and diameter of a 3D $n \times n \times n$ mesh?

Ex. 35 — What is the diameter of a ring containing n nodes? Give a precise answer that holds for even and odd n.

Ex. 36 — What is the bisection bandwidth and diameter of a hypercube of order n.

Ex. 37 — What is the bisection bandwidth and diameter of an $n \times n \times n$, 3D torus?

Ex. 38 — What is the bisection bandwidth and diameter of a clique of n nodes (n is even)? In a clique, all pairs of nodes are connected.

** **Ex. 39** — Assume we have an $n \times n$ mesh. There are n^2 routers, and each processor is connected to one router. Note that at any point of time, a router can only store 1 message. It will discard a message only if the message gets stored in another router. In our previous example, router (i, j) will keep the message until it has been delivered and stored at a neighboring router such as $(i + 1, j)$. Now, an interesting deadlock situation can develop. Let us assume the following scenario.

- (1,1) wants to send a message to (1,2).
- (1,2) wants to send a message to (2,2).
- (2,2) wants to send a message to (2,1).
- (2,1) wants to send a message to (1,1).

In this case all the four nodes have 1 message each. They are not able to forward the packet to the next node, because the next node already stores a packet, and is thus busy. Since there is a cyclic wait, we have a deadlock. Design a message routing protocol between a source and destination node that is provably deadlock free.

Vector Processors

Ex. 40 — What is the advantage of vector processors over scalar processors?

Ex. 41 — Why are vector load-store instructions easy to implement in systems that have caches with large block sizes?

Ex. 42 — How can we efficiently implement a scatter-gather based load-store unit?

Ex. 43 — What is a predicated instruction, and how does it help speed up a vector processor?

* **Ex. 44** — Assume that we have a processor with a 32 entry vector register file. We wish to add two arrays that have 17 entries each. How can we implement this operation, with the *SimpleRisc* vector instructions introduced in the chapter? Feel free to introduce new vector instructions if required.

* **Ex. 45** — Design a dedicated SIMD hardware unit to sort n integers in roughly n time steps by using the bubble sort algorithm. You have a linear array of n processors connected end to end. Each processor is capable of storing two integers, and has some logic inside it. Design the logic for each processor and explain the overall working of the system.

Design Problems

Ex. 46 — Write a program to sort a billion integers using OpenMP and MPI.

Ex. 47 — Implement a distributed shared memory system on a cluster of computers connected via an Ethernet LAN.