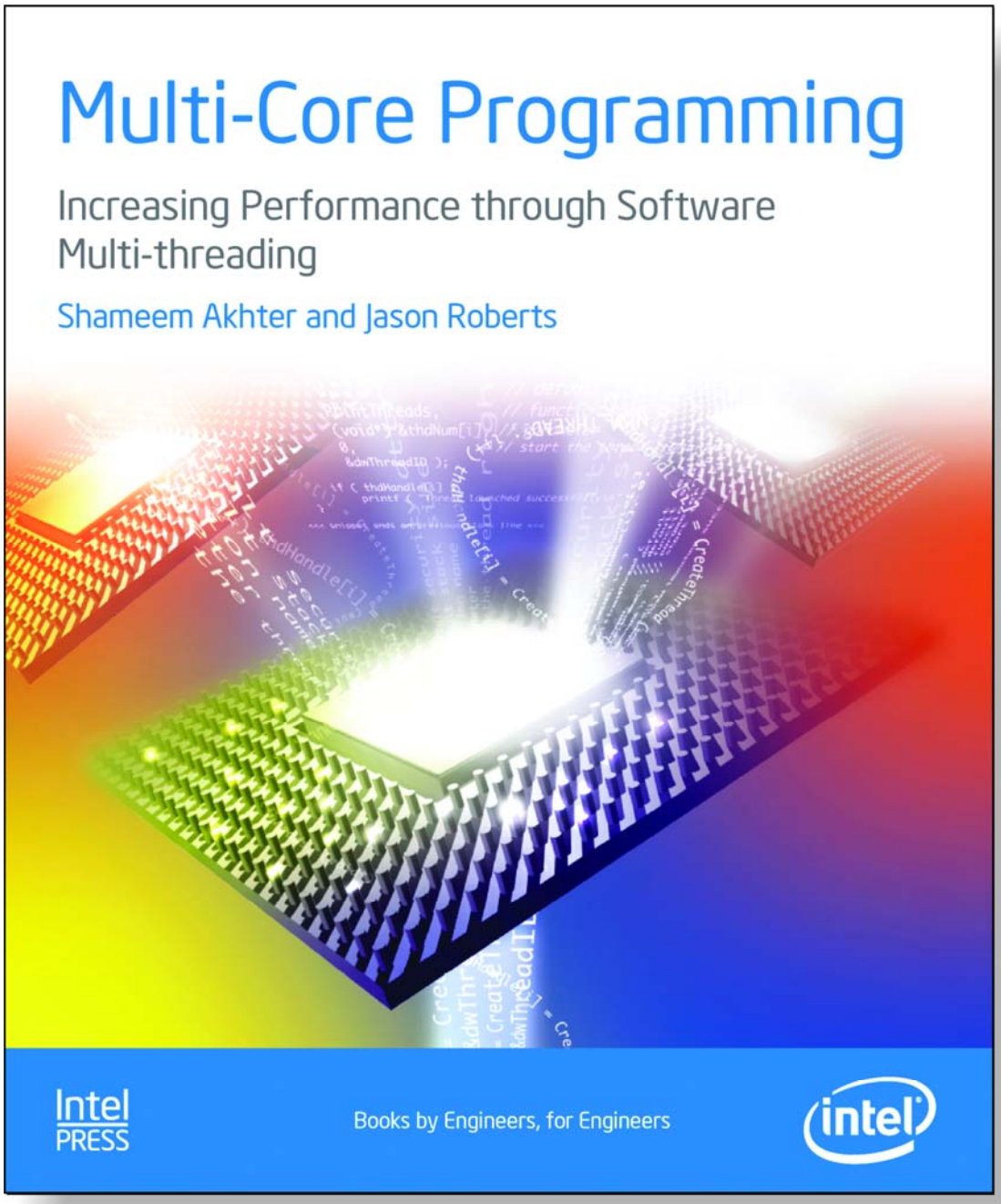
Digital Edition

Digital Editions of selected Intel Press books are in addition to and complement the printed books.



Click the icon to access information on other essential books for Developers and IT Professionals

Visit our website at www.intel.com/intelpress

Chapter 1

Introduction to Multi-Core Architecture

In 1945, mathematician John von Neumann, with the aid of J. Presper Eckert and John Mauchly, wrote a memo proposing the creation of an Electronic Discrete Variable Automatic Computer, more famously known as the EDVAC. In this paper, von Neumann suggested the stored-program model of computing. In the von Neumann architecture, a program is a sequence of instructions stored sequentially in the computer's memory. The program's instructions are executed one after the other in a linear, single-threaded fashion.

As time went on, advancements in mainframe technology expanded upon the ideas presented by von Neumann. The 1960s saw the advent of time-sharing operating systems. Run on large mainframe computers, these operating systems first introduced the concept of concurrent program execution. Multiple users could access a single mainframe computer simultaneously and submit jobs for processing. From the program's perspective, it was the only process in the system. The operating system handled the details of allocating CPU time for each individual program. At this time, concurrency existed at the process level, and the job of task switching was left to the systems programmer.

In the early days of personal computing, personal computers, or PCs, were standalone devices with simple, single-user operating systems. Only one program would run at a time. User interaction occurred via simple text based interfaces. Programs followed the standard model of straight-line instruction execution proposed by the von Neumann architecture. Over time, however, the exponential growth in computing performance

quickly led to more sophisticated computing platforms. Operating system vendors used the advance in CPU and graphics performance to develop more sophisticated user environments. Graphical User Interfaces, or GUIs, became standard and enabled users to start and run multiple programs in the same user environment. Networking on PCs became pervasive.

This rapid growth came at a price: increased user expectations. Users expected to be able to send e-mail while listening to streaming audio that was being delivered via an Internet radio station. Users expected their computing platform to be quick and responsive. Users expected applications to start quickly and handle inconvenient background tasks, such as automatically saving a file with minimal disruption. These challenges are the problems that face software developers today.

Motivation for Concurrency in Software

Most end users have a simplistic view of complex computer systems. Consider the following scenario: A traveling businessman has just come back to his hotel after a long day of presentations. Too exhausted to go out, he decides to order room service and stay in his room to watch his favorite baseball team play. Given that he's on the road, and doesn't have access to the game on his TV, he decides to check out the broadcast via the Internet. His view of the system might be similar to the one shown in Figure 1.1.

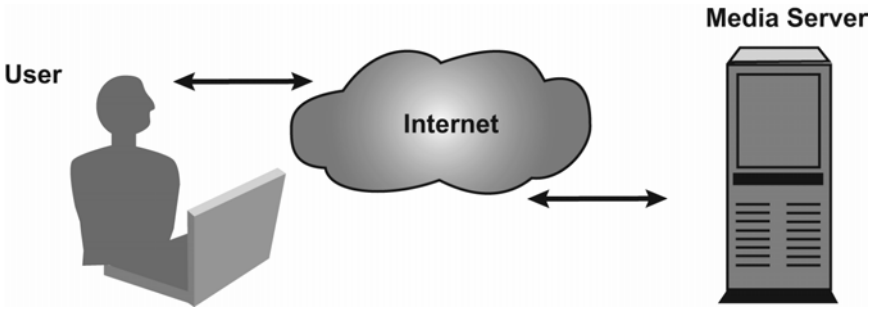


Figure 1.1 End User View of Streaming Multimedia Content via the Internet

The user's expectations are based on conventional broadcast delivery systems which provide continuous, uninterrupted delivery of content. The user does not differentiate between streaming the content over the Internet and delivering the data via a broadcast network. To the user, watching a baseball game on a laptop seems like a simple, straightforward task.

The reality is that the implementation of such a system is far more difficult. From the client side, the PC must be able to download the streaming video data, decompress/decode it, and draw it on the video display. In addition, it must handle any streaming audio that accompanies the video stream and send it to the soundcard. Meanwhile, given the general purpose nature of the computer, the operating system might be configured to run a virus scan or some other system tasks periodically. On the server side, the provider must be able to receive the original broadcast, encode/compress it in near real-time, and then send it over the network to potentially hundreds of thousands of clients. A system designer who is looking to build a computer system capable of streaming a Web broadcast might look at the system as it's shown in Figure 1.2.

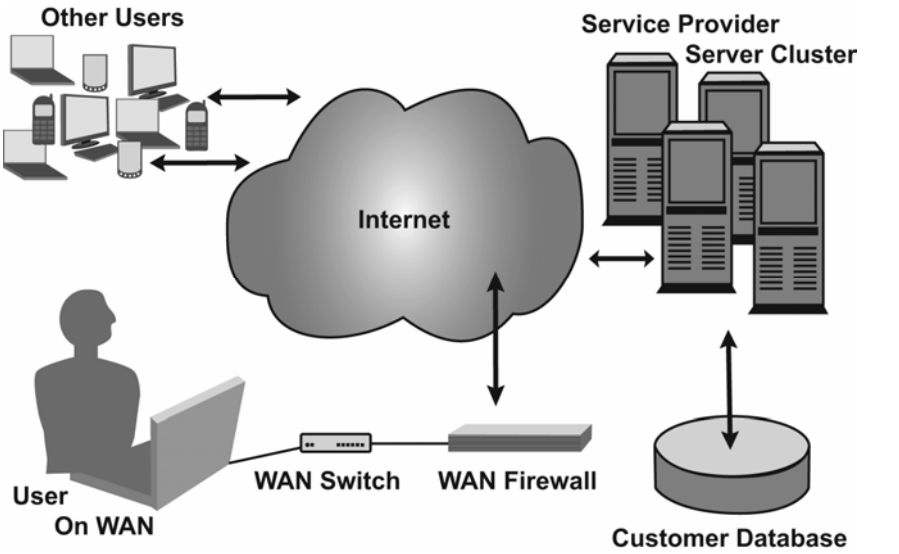


Figure 1.2 End-to-End Architecture View of Streaming Multimedia Content over the Internet

Contrast this view of a streaming multimedia delivery service with the end user's perspective of the system shown in Figure 1.1. In order to provide an acceptable end-user experience, system designers must be able to effectively manage many independent subsystems that operate in parallel.

Careful inspection of Figure 1.2 shows that the problem of streaming media content may be broken into a number of disparate parts; each acting

independently[1] from one another. This decomposition allows us to break down each task into a single isolated problem, making the problem much more manageable.

Concurrency in software is a way to manage the sharing of resources used at the same time. Concurrency in software is important for several reasons:

- Concurrency allows for the most efficient use of system resources. Efficient resource utilization is the key to maximizing performance of computing systems. Unnecessarily creating dependencies on different components in the system drastically lowers overall system performance. In the aforementioned streaming media example, one might naively take this, serial, approach on the client side:
  1. Wait for data to arrive on the network
  2. Uncompress the data
  3. Decode the data
  4. Send the decoded data to the video/audio hardware

  This approach is highly inefficient. The system is completely idle while waiting for data to come in from the network. A better approach would be to stage the work so that while the system is waiting for the next video frame to come in from the network, the previous frame is being decoded by the CPU, thereby improving overall resource utilization.
- Many software problems lend themselves to simple concurrent implementations. Concurrency provides an abstraction for implementing software algorithms or applications that are naturally parallel. Consider the implementation of a simple FTP server. Multiple clients may connect and request different files. A single-threaded solution would require the application to keep track of all the different state information for each connection. A more intuitive implementation would create a separate thread for each connection. The connection state would be managed by this separate entity. This multi-threaded approach provides a solution that is much simpler and easier to maintain.

It's worth noting here that the terms *concurrent* and *parallel* are not interchangeable in the world of parallel programming. When multiple

[1] The term "independently" is used loosely here. Later chapters discuss the managing of interdependencies that is inherent in multi-threaded programming.

software threads of execution are running in parallel, it means that the active threads are running simultaneously on different hardware resources, or processing elements. Multiple threads may make progress simultaneously. When multiple software threads of execution are running concurrently, the execution of the threads is interleaved onto a single hardware resource. The active threads are ready to execute, but only one thread may make progress at a given point in time. In order to have parallelism, you must have concurrency exploiting multiple hardware resources.

Parallel Computing Platforms

In order to achieve parallel execution in software, hardware must provide a platform that supports the simultaneous execution of multiple threads. Generally speaking, computer architectures can be classified by two different dimensions. The first dimension is the number of *instruction streams* that a particular computer architecture may be able to process at a single point in time. The second dimension is the number of *data streams* that can be processed at a single point in time. In this way, any given computing system can be described in terms of how instructions and data are processed. This classification system is known as Flynn's taxonomy (Flynn, 1972), and is graphically depicted in Figure 1.3.

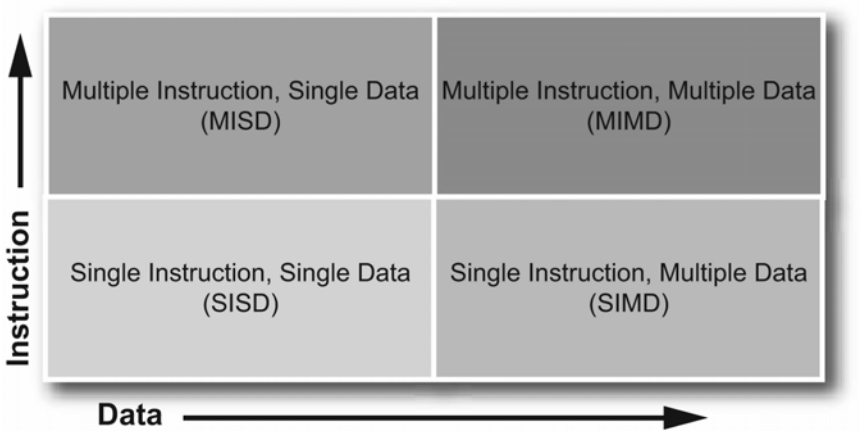


Figure 1.3 Flynn's Taxonomy

Flynn's taxonomy places computing platforms in one of four categories:

- A single instruction, single data (SISD) machine is a traditional sequential computer that provides no parallelism in hardware. Instructions are executed in a serial fashion. Only one data stream is processed by the CPU during a given clock cycle. Examples of these platforms include older computers such as the original IBM PC, older mainframe computers, or many of the 8-bit home computers such as the Commodore 64 that were popular in the early 1980s.
- A multiple instruction, single data (MISD) machine is capable of processing a single data stream using multiple instruction streams simultaneously. In most cases, multiple instruction streams need multiple data streams to be useful, so this class of parallel computer is generally used more as a theoretical model than a practical, mass-produced computing platform.
- A single instruction, multiple data (SIMD) machine is one in which a single instruction stream has the ability to process multiple data streams simultaneously. These machines are useful in applications such as general digital signal processing, image processing, and multimedia applications such as audio and video. Originally, supercomputers known as array processors or vector processors such as the Cray-1 provided SIMD processing capabilities. Almost all computers today implement some form of SIMD instruction set. Intel processors implement the MMX™, Streaming SIMD Extensions (SSE), Streaming SIMD Extensions 2 (SSE2), and Streaming SIMD Extensions 3 (SSE3) instructions that are capable of processing multiple data elements in a single clock. The multiple data elements are stored in the floating point registers. PowerPC† processors have implemented the AltiVec instruction set to provide SIMD support.
- A multiple instruction, multiple data (MIMD) machine is capable of is executing multiple instruction streams, while working on a separate and independent data stream. This is the most common parallel computing platform today. New multi-core platforms such as the Intel® Core™ Duo processor fall into this category.

Given that modern computing machines are either the SIMD or MIMD machines, software developers have the ability to exploit data-level and task level parallelism in software.

Parallel Computing in Microprocessors

In 1965, Gordon Moore observed that the number of transistors available to semiconductor manufacturers would double approximately every 18 to 24 months. Now known as Moore's law, this observation has guided computer designers for the past 40 years. Many people mistakenly think of Moore's law as a predictor of CPU clock frequency, and it's not really hard to understand why. The most commonly used metric in measuring computing performance is CPU clock frequency. Over the past 40 years, CPU clock speed has tended to follow Moore's law. It's an important distinction to make, however, as taking this view of Moore's law imposes unnecessary limits on a silicon designer. While improving straight-line instruction throughput and clock speeds are goals worth striving for, computer architects can take advantage of these extra transistors in less obvious ways.

For example, in an effort to make the most efficient use of processor resources, computer architects have used instruction-level parallelization techniques to improve processor performance. Instruction-level parallelism (ILP), also known as dynamic, or out-of-order execution, gives the CPU the ability to reorder instructions in an optimal way to eliminate pipeline stalls. The goal of ILP is to increase the number of instructions that are executed by the processor on a single clock cycle[2]. In order for this technique to be effective, multiple, independent instructions must execute. In the case of in-order program execution, dependencies between instructions may limit the number of instructions available for execution, reducing the amount of parallel execution that may take place. An alternative approach that attempts to keep the processor's execution units full is to reorder the instructions so that independent instructions execute simultaneously. In this case, instructions are executed out of program order. This dynamic instruction scheduling is done by the processor itself. You will learn much more about these techniques in a later chapter, but for now what is important to understand is that this parallelism occurs at the hardware level and is transparent to the software developer.

As software has evolved, applications have become increasingly capable of running multiple tasks simultaneously. Server applications today often consist of multiple threads or processes. In order to support this thread-level parallelism, several approaches, both in software and hardware, have been adopted.

---

[2] A processor that is capable of executing multiple instructions in a single clock cycle is known as a super-scalar processor.

One approach to address the increasingly concurrent nature of modern software involves using a preemptive, or time-sliced, multitasking operating system. Time-slice multi-threading allows developers to hide latencies associated with I/O by interleaving the execution of multiple threads. This model does not allow for parallel execution. Only one instruction stream can run on a processor at a single point in time.

Another approach to address thread-level parallelism is to increase the number of physical processors in the computer. Multiprocessor systems allow true parallel execution; multiple threads or processes run simultaneously on multiple processors. The tradeoff made in this case is increasing the overall system cost.

As computer architects looked at ways that processor architectures could adapt to thread-level parallelism, they realized that in many cases, the resources of a modern processor were underutilized. In order to consider this solution, you must first more formally consider what a thread of execution in a program is. A thread can be defined as a basic unit of CPU utilization. It contains a program counter that points to the current instruction in the stream. It contains CPU state information for the current thread. It also contains other resources such as a stack.

A physical processor is made up of a number of different resources, including the architecture state—the general purpose CPU registers and interrupt controller registers, caches, buses, execution units, and branch prediction logic. However, in order to define a thread, only the architecture state is required. A logical processor can thus be created by duplicating this architecture space. The execution resources are then shared among the different logical processors. This technique is known as *simultaneous multi-threading,* or SMT. Intel's implementation of SMT is known as Hyper-Threading Technology, or HT Technology. HT Technology makes a single processor appear, from software's perspective, as multiple logical processors. This allows operating systems and applications to schedule multiple threads to logical processors as they would on multiprocessor systems. From a microarchitecture perspective, instructions from logical processors are persistent and execute simultaneously on shared execution resources. In other words, multiple threads can be scheduled, but since the execution resources are shared, it's up to the microarchitecture to determine how and when to interleave the execution of the two threads. When one thread stalls, another thread is allowed to make progress. These stall events include handling cache misses and branch mispredictions.

The next logical step from simultaneous multi-threading is the multi-core processor. Multi-core processors use chip multiprocessing (CMP). Rather

than just reuse select processor resources in a single-core processor, processor manufacturers take advantage of improvements in manufacturing technology to implement two or more "execution cores" within a single processor. These cores are essentially two individual processors on a single die. Execution cores have their own set of execution and architectural resources. Depending on design, these processors may or may not share a large on-chip cache. In addition, these individual cores may be combined with SMT; effectively increasing the number of logical processors by twice the number of execution cores. The different processor architectures are highlighted in Figure 1.4.

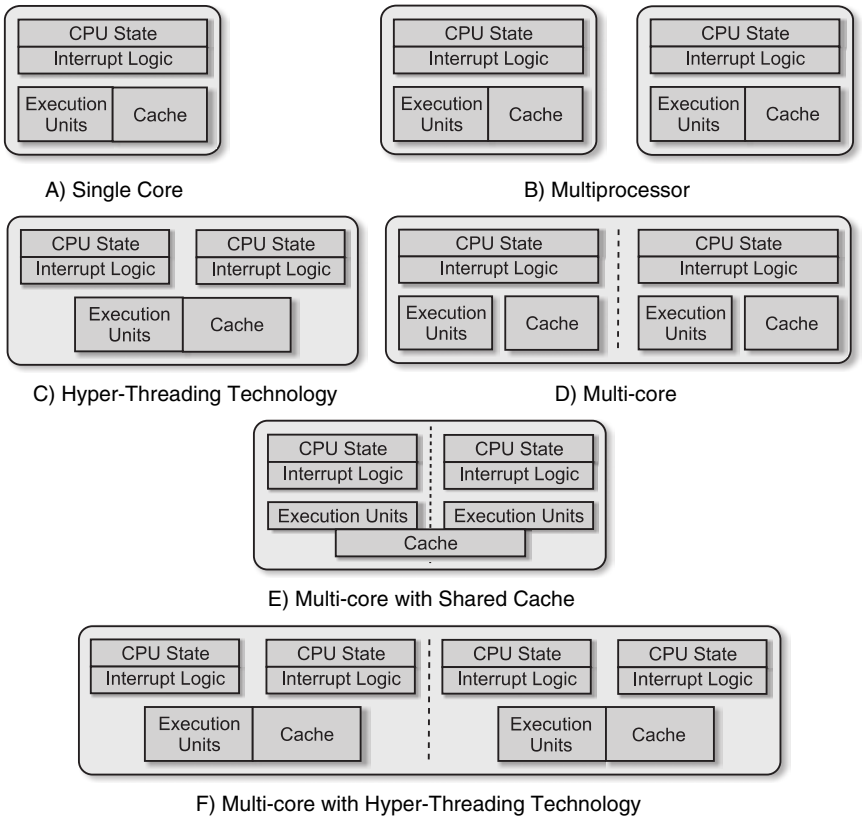


Figure 1.4 Simple Comparison of Single-core, Multi-processor, and Multi-Core Architectures

Differentiating Multi-Core Architectures from Hyper-Threading Technology

With HT Technology, parts of the one processor are shared between threads, while other parts are duplicated between them. One of the most important shared resources is the actual execution engine. This engine works on both threads at the same time by executing instructions for one thread on resources that the other thread is not using. When both threads are running, HT Technology literally interleaves the instructions in the execution pipeline. Which instructions are inserted when depends wholly on what execution resources of the processor are available at execution time. Moreover, if one thread is tied up reading a large data file from disk or waiting for the user to type on the keyboard, the other thread takes over all the processor resources—without the operating system switching tasks—until the first thread is ready to resume processing. In this way, each thread receives the maximum available resources and the processor is kept as busy as possible. An example of a thread running on a HT Technology enabled CPU is shown in Figure 1.5.

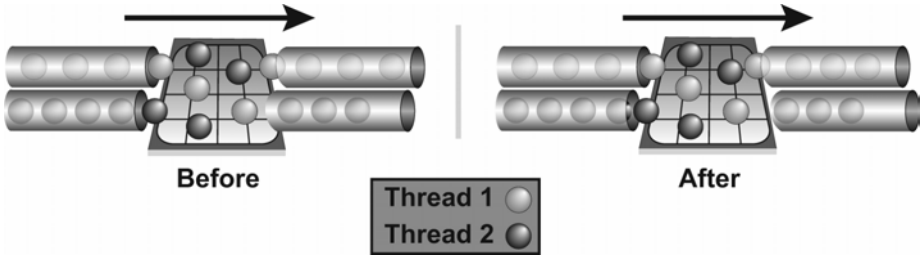


Figure 1.5 Two Threads Executing on a Processor with Hyper-Threading Technology

HT Technology achieves performance gains through latency hiding. Fundamentally, a single execution core is shared among multiple threads. Therefore, thread execution is not parallel. As a result, performance results vary based on application and hardware platform. With HT Technology, in certain applications, it is possible to attain, on average, a 30-percent increase in processor throughput. In other words, in certain cases, the processor can perform 1.3 times the number of executed instructions that it could if it were running only one thread. To see a performance improvement, applications must make good use of threaded programming models and of the capabilities of Hyper-Threading Technology.

The performance benefits of HT Technology depend on how much latency hiding can occur in your application. In some applications, developers may have minimized or effectively eliminated memory latencies through cache optimizations. In this case, optimizing for HT Technology may not yield any performance gains.

On the other hand, multi-core processors embed two or more independent execution cores into a single processor package. By providing multiple execution cores, each sequence of instructions, or thread, has a hardware execution environment entirely to itself. This enables each thread run in a truly parallel manner. An example of two threads running on a dual-core processor is shown in Figure 1.6. Compare this with the HT Technology example provided in Figure 1.5, and note that a dual-core processor provides true parallel execution of each thread.

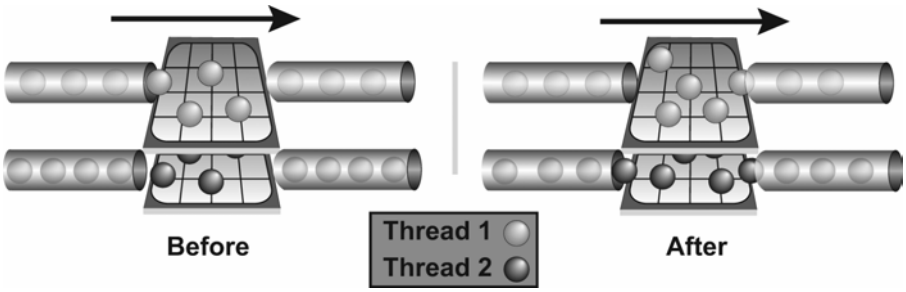


Figure 1.6 Two Threads on a Dual-Core Processor with each Thread Running Independently

It should be noted that HT Technology does not attempt to deliver multi-core performance, which can theoretically be close to a 100-percent, or 2x improvement in performance for a dual-core system. HT Technology is more of a facility in which the programmer may be able to use idle CPU resources in order to accomplish more work. When combined with multi-core technology, HT Technology can provide powerful optimization opportunities, increasing system throughput substantially.

Multi-threading on Single-Core versus Multi-Core Platforms

At this point, many readers may be asking themselves what all the commotion is about. The concept of multiple threads in a single process space has been around for decades. Most modern applications use threads in one fashion or another today. As a result, many developers are

already familiar with the concept of threading, and have probably worked on applications that have multiple threads. There are however, certain important considerations developers should be aware of when writing applications targeting multi-core processors:

- *Optimal application performance on multi-core architectures will be achieved by effectively using threads to partition software workloads.* Many applications today use threads as a tool to improve user responsiveness on single-core platforms. Rather than blocking the user interface (UI) on a time consuming database query or disk access, an application will spawn a thread to process the user's request. This allows the scheduler to individually schedule the main control loop task that receives UI events as well as the data processing task that is running the database query. In this model, developers rely on straight-line instruction throughput improvements to improve application performance.

  This is the significant limitation of multi-threading on single-core processors. Since single-core processors are really only able to interleave instruction streams, but not execute them simultaneously, the overall performance gains of a multi-threaded application on single-core architectures are limited. On these platforms, threads are generally seen as a useful programming abstraction for hiding latency.

  This performance restriction is removed on multi-core architectures. On multi-core platforms, threads do not have to wait for any one resource. Instead, threads run independently on separate cores. As an example, consider two threads that both wanted to execute a shift operation. If a core only had one "shifter unit" they could not run in parallel. On two cores, there would be two "shifter units," and each thread could run without contending for the same resource.

  Multi-core platforms allow developers to optimize applications by intelligently partitioning different workloads on different processor cores. Application code can be optimized to use multiple processor resources, resulting in faster application performance.

- *Multi-threaded applications running on multi-core platforms have different design considerations than do multi-threaded applications running on single-core platforms.* On single-core platforms, assumptions may be made by the developer to simplify writing and debugging a multi-threaded application. These assumptions may not be valid on multi-core platforms. Two areas that highlight these differences are *memory caching* and *thread priority*.

In the case of memory caching, each processor core may have its own cache.[3] At any point in time, the cache on one processor core may be out of sync with the cache on the other processor core. To help illustrate the types of problems that may occur, consider the following example. Assume two threads are running on a dual-core processor. Thread 1 runs on core 1 and thread 2 runs on core 2. The threads are reading and writing to neighboring memory locations. Since cache memory works on the principle of locality, the data values, while independent, may be stored in the same cache line. As a result, the memory system may mark the cache line as invalid, even though the data that the thread is interested in hasn't changed. This problem is known as *false sharing*. On a single-core platform, there is only one cache shared between threads; therefore, cache synchronization is not an issue.

Thread priorities can also result in different behavior on single-core versus multi-core platforms. For example, consider an application that has two threads of differing priorities. In an attempt to improve performance, the developer assumes that the higher priority thread will always run without interference from the lower priority thread. On a single-core platform, this may be valid, as the operating system's scheduler will not yield the CPU to the lower priority thread. However, on multi-core platforms, the scheduler may schedule both threads on separate cores. Therefore, both threads may run simultaneously. If the developer had optimized the code to assume that the higher priority thread would always run without interference from the lower priority thread, the code would be unstable on multi-core and multi-processor systems.

One goal of this book is to help developers correctly utilize the number of processor cores they have available.

Understanding Performance

At this point one may wonder—how do I measure the performance benefit of parallel programming? Intuition tells us that if we can subdivide disparate tasks and process them simultaneously, we're likely

[3] Multi-core CPU architectures can be designed in a variety of ways: some multi-core CPUs will share the on-chip cache between execution units; some will provide a dedicated cache for each execution core; and others will take a hybrid approach, where the cache is subdivided into layers that are dedicated to a particular execution core and other layers that are shared by all execution cores. For the purposes of this section, we assume a multi-core architecture with a dedicated cache for each core.

to see significant performance improvements. In the case where the tasks are completely independent, the performance benefit is obvious, but most cases are not so simple. How does one quantitatively determine the performance benefit of parallel programming? One metric is to compare the elapsed run time of the best sequential algorithm versus the elapsed run time of the parallel program. This ratio is known as the speedup and characterizes how much faster a program runs when parallelized.

$$\text{Speedup}(n_t) = \frac{Time_{best\_sequential\_algorithm}}{Time_{parallel\_implementation}(n_t)}$$

Speedup is defined in terms of the number of physical threads (n_t) used in the parallel implementation.

Amdahl's Law

Given the previous definition of speedup, is there a way to determine the theoretical limit on the performance benefit of increasing the number of processor cores, and hence physical threads, in an application? When examining this question, one generally starts with the work done by Gene Amdahl in 1967. His rule, known as Amdahl's Law, examines the maximum theoretical performance benefit of a parallel solution relative to the best case performance of a serial solution.

Amdahl started with the intuitively clear statement that program speedup is a function of the fraction of a program that is accelerated and by how much that fraction is accelerated.

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

So, if you could speed up half the program by 15 percent, you'd get:

$$\text{Speedup} = 1/((1 - .50) + (.50/1.15)) = 1/(.50 + .43) = 1.08$$

This result is a speed increase of 8 percent, which is what you'd expect. If half of the program is improved 15 percent, then the whole program is improved by half that amount.

Amdahl then went on to explain how this equation works out if you make substitutions for fractions that are parallelized and those that are run serially, as shown in Equation 1.1.

Equation 1.1 Amdahl's Law

$$\text{Speedup} = \frac{1}{S + (1 - S)/n}$$

In this equation, S is the time spent executing the serial portion of the parallelized version and n is the number of processor cores. Note that the numerator in the equation assumes that the program takes 1 unit of time to execute the best sequential algorithm.

If you substitute 1 for the number of processor cores, you see that no speedup is realized. If you have a dual-core platform doing half the work, the result is:

1 / (0.5S + 0.5S/2) = 1/0.75S = 1.33

or a 33-percent speed-up, because the run time, as given by the denominator, is 75 percent of the original run time. For an 8-core processor, the speedup is:

1 / (0.5S + 0.5S/8) = 1/0.75S = 1.78

Setting n = ∞ in Equation 1.1, and assuming that the best sequential algorithm takes 1 unit of time yields Equation 1.2.

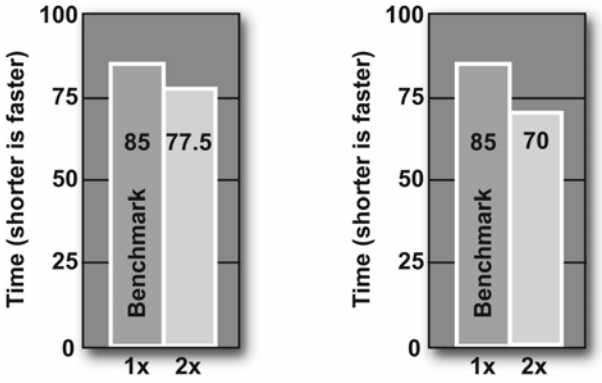
Equation 1.2 Upper Bound of an Application with S Time Spent in Sequential Code

$$\text{Speedup} = \frac{1}{S}$$

As stated in this manner, Amdahl assumes that the addition of processor cores is perfectly scalable. As such, this statement of the law shows the maximum benefit a program can expect from parallelizing some portion of the code is limited by the serial portion of the code. For example, according Amdahl's law, if 10 percent of your application is spent in serial code, the maximum speedup that can be obtained is 10x, regardless of the number of processor cores.

It is important to note that endlessly increasing the processor cores only affects the parallel portion of the denominator. So, if a program is only 10-percent parallelized, the maximum theoretical benefit is that the program can run in 90 percent of the sequential time.

Given this outcome, you can see the first corollary of Amdahl's law: decreasing the serialized portion by increasing the parallelized portion is of greater importance than adding more processor cores. For example, if you have a program that is 30-percent parallelized running on a dual-core system, doubling the number of processor cores reduces run time from 85 percent of the serial time to 77.5 percent, whereas doubling the amount of parallelized code reduces run time from 85 percent to 70 percent. This is illustrated in Figure 1.7. Only when a program is mostly parallelized does adding more processors help more than parallelizing the remaining code. And, as you saw previously, you have hard limits on how much code can be serialized and on how many additional processor cores actually make a difference in performance.



Performance benefit of doubling the number of processor cores

Performance benefit of doubling the amount of parallelism in code

Note: The advantage gained by writing parallel code

Figure 1.7 Theoretical Performance Comparison between Increasing Number of CPU Cores versus Increasing Concurrency in Implementation

To make Amdahl's Law reflect the reality of multi-core systems, rather than the theoretical maximum, system overhead from adding threads should be included:

$$\text{Speedup} = \frac{1}{S + (1 - S)/n + H(n)}$$

where H(n) = overhead, and again, we assume that the best serial algorithm runs in one time unit. Note that this overhead is not linear on a good parallel machine.

This overhead consists of two portions: the actual operating system overhead and inter-thread activities, such as synchronization and other forms of communication between threads. Notice that if the overhead is big enough, it offsets the benefits of the parallelized portion. In fact, if the overhead is large enough, the speedup ration can ultimately have a value of less than 1, implying that threading has actually slowed performance when compared to the single-threaded solution. This is very common in poorly architected multi-threaded applications. The important implication is that the overhead introduced by threading must be kept to a minimum. For this reason, most of this book is dedicated to keeping the cost of threading as low as possible.

Amdahl's Law Applied to Hyper-Threading Technology

The previous section demonstrated Amdahl's law as it applies to multi-processor and multi-core systems. Hyper-Threading Technology imposes an additional factor on how you apply Amdahl's Law to your code. On processors enabled with HT Technology, the fact that certain processor resources are shared between the different threads of execution has a direct effect on the maximum performance benefit of threading an application.

Given the interleaved execution environment provided by HT Technology, it's important to develop a form of Amdahl's law that works for HT Technology. Assume that your application experiences a performance gain of around 30 percent when run on a processor with HT Technology enabled. That is, performance improves by 30 percent over the time required for a single processor to run both threads. If you were using a quad-core platform, with each processor completely dedicated to the thread it was running, the number could, in theory, be up to 4x. That is, the second, third, and fourth processor core could give a 300-percent boost to program throughput. In practice it's not quite 300 percent, due to overhead and code that cannot be parallelized, and the performance benefits will vary based on the application.

Inside the processor enabled with HT Technology, each thread is running more slowly than it would if it had the whole processor to itself. HT Technology is not a replacement for multi-core processing since many processing resources, such as the execution units, are shared. The slowdown varies from application to application. As example, assume each thread runs approximately one-third slower than it would if it

owned the entire processor. Amending Amdahl's Law to fit HT Technology, then, you get:

$$\text{Speedup}_{\text{HTT}} = \frac{1}{S + 0.67((1-S)/n) + H(n)}$$

where n = number of logical processors.

This equation represents the typical speed-up for programs running on processor cores with HT Technology performance. The value of H(n) is determined empirically and varies from application to application.

Growing Returns: Gustafson's Law

Based on Amdahl's work, the viability of massive parallelism was questioned for a number of years. Then, in the late 1980s, at the Sandia National Lab, impressive linear speedups in three practical applications were observed on a 1,024-processor hypercube. The results (Gustafson 1988) demonstrated that near linear speedup was possible in many practical cases, even when Amdahl's Law predicted otherwise.

Built into Amdahl's Law are several assumptions that may not hold true in real-world implementations. First, Amdahl's Law assumes that the best performing serial algorithm is strictly limited by the availability of CPU cycles. This may not be the case. A multi-core processor may implement a separate cache on each core. Thus, more of the problem's data set may be stored in cache, reducing memory latency. The second flaw is that Amdahl's Law assumes that the serial algorithm is the best possible solution for a given problem. However, some problems lend themselves to a more efficient parallel solution. The number of computational steps may be significantly less in the parallel implementation.

Perhaps the biggest weakness, however, is the assumption that Amdahl's Law makes about the problem size. Amdahl's Law assumes that as the number of processor cores increases, the problem size stays the same. In most cases, this is not valid. Generally speaking, when given more computing resources, the problem generally grows to meet the resources available. In fact, it is more often the case that the run time of the application is constant.

Based on the work at Sandia, an alternative formulation for speedup, referred to as scaled speedup was developed by E. Barsis.

$$\text{Scaled speedup} = N + (1-N) * s$$

where N = is the number of processor cores and s is the ratio of the time spent in the serial port of the program versus the total execution time.

Scaled speedup is commonly referred to as Gustafson's Law. From this equation, one can see that the speedup in this case is linear.

Gustafson's Law has been shown to be equivalent to Amdahl's Law (Shi 1996). However, Gustafson's Law offers a much more realistic look at the potential of parallel computing on multi-core processors.

Key Points

This chapter demonstrated the inherent concurrent nature of many software applications and introduced the basic need for parallelism in hardware. An overview of the different techniques for achieving parallel execution was discussed. Finally, the chapter examined techniques for estimating the performance benefits of using proper multi-threading techniques. The key points to keep in mind are:

- Concurrency refers to the notion of multiple threads in progress at the same time. This is often achieved on sequential processors through interleaving.
- Parallelism refers to the concept of multiple threads executing simultaneously.
- Modern software applications often consist of multiple processes or threads that can be executed in parallel.
- Most modern computing platforms are multiple instruction, multiple data (MIMD) machines. These machines allow programmers to process multiple instruction and data streams simultaneously.
- In practice, Amdahl's Law does not accurately reflect the benefit of increasing the number of processor cores on a given platform. Linear speedup is achievable by expanding the problem size with the number of processor cores.

Chapter 3

Fundamental Concepts of Parallel Programming

As discussed in previous chapters, parallel programming uses threads to enable multiple operations to proceed simultaneously. The entire concept of parallel programming centers on the design, development, and deployment of threads within an application and the coordination between threads and their respective operations. This chapter examines how to break up programming tasks into chunks that are suitable for threading. It then applies these techniques to the apparently serial problem of error diffusion.

Designing for Threads

Developers who are unacquainted with parallel programming generally feel comfortable with traditional programming models, such as object-oriented programming (OOP). In this case, a program begins at a defined point, such as the `main()` function, and works through a series of tasks in succession. If the program relies on user interaction, the main processing instrument is a loop in which user events are handled. From each allowed event—a button click, for example, the program performs an established sequence of actions that ultimately ends with a wait for the next user action.

When designing such programs, developers enjoy a relatively simple programming world because only one thing is happening at any given moment. If program tasks must be scheduled in a specific way, it's because the developer imposes a certain order on the activities. At any

point in the process, one step generally flows into the next, leading up to a predictable conclusion, based on predetermined parameters.

To move from this linear model to a parallel programming model, designers must rethink the idea of process flow. Rather than being constrained by a sequential execution sequence, programmers should identify those activities that can be executed in parallel. To do so, they must see their programs as a set of tasks with dependencies between them. Breaking programs down into these individual tasks and identifying dependencies is known as *decomposition*. A problem may be decomposed in several ways: by task, by data, or by data flow. Table 3.1 summarizes these forms of decomposition. As you shall see shortly, these different forms of decomposition mirror different types of programming activities.

Table 3.1 Summary of the Major Forms of Decomposition

Decomposition	Design	Comments
Task	Different activities assigned to different threads	Common in GUI apps
Data	Multiple threads performing the same operation but on different blocks of data	Common in audio processing, imaging, and in scientific programming
Data Flow	One thread's output is the input to a second thread	Special care is needed to eliminate startup and shutdown latencies

Task Decomposition

Decomposing a program by the functions that it performs is called *task decomposition*. It is one of the simplest ways to achieve parallel execution. Using this approach, individual tasks are catalogued. If two of them can run concurrently, they are scheduled to do so by the developer. Running tasks in parallel this way usually requires slight modifications to the individual functions to avoid conflicts and to indicate that these tasks are no longer sequential.

If we were discussing gardening, task decomposition would suggest that gardeners be assigned tasks based on the nature of the activity: if two gardeners arrived at a client's home, one might mow the lawn while the other weeded. Mowing and weeding are separate functions broken out as such. To accomplish them, the gardeners would make sure to have some coordination between them, so that the weeder is not sitting in the middle of a lawn that needs to be mowed.

In programming terms, a good example of task decomposition is word processing software, such as Microsoft Word†. When the user opens a very long document, he or she can begin entering text right away. While the user enters text, document pagination occurs in the background, as one can readily see by the quickly increasing page count that appears in the status bar. Text entry and pagination are two separate tasks that its programmers broke out by function to run in parallel. Had programmers not designed it this way, the user would be obliged to wait for the entire document to be paginated before being able to enter any text. Many of you probably recall that this wait was common on early PC word processors.

Data Decomposition

Data decomposition, also known as *data-level parallelism*, breaks down tasks by the data they work on rather than by the nature of the task. Programs that are broken down via data decomposition generally have many threads performing the same work, just on different data items. For example, consider recalculating the values in a large spreadsheet. Rather than have one thread perform all the calculations, data decomposition would suggest having two threads, each performing half the calculations, or n threads performing $1/n^{th}$ the work.

If the gardeners used the principle of data decomposition to divide their work, they would both mow half the property and then both weed half the flower beds. As in computing, determining which form of decomposition is more effective depends a lot on the constraints of the system. For example, if the area to mow is so small that it does not need two mowers, that task would be better done by just one gardener—that is, task decomposition is the best choice—and data decomposition could be applied to other task sequences, such as when the mowing is done and both gardeners begin weeding in parallel.

As the number of processor cores increases, data decomposition allows the problem size to be increased. This allows for more work to be done in the same amount of time. To illustrate, consider the gardening example. Two more gardeners are added to the work crew. Rather than assigning all four gardeners to one yard, we can we can assign the two new gardeners to another yard, effectively increasing our total problem size. Assuming that the two new gardeners can perform the same amount of work as the original two, and that the two yard sizes are the same, we've doubled the amount of work done in the same amount of time.

Data Flow Decomposition

Many times, when decomposing a problem, the critical issue isn't what tasks should do the work, but how the data flows between the different tasks. In these cases, data flow decomposition breaks up a problem by how data flows between tasks.

The *producer/consumer* problem is a well known example of how data flow impacts a programs ability to execute in parallel. Here, the output of one task, the producer, becomes the input to another, the consumer. The two tasks are performed by different threads, and the second one, the consumer, cannot start until the producer finishes some portion of its work.

Using the gardening example, one gardener prepares the tools—that is, he puts gas in the mower, cleans the shears, and other similar tasks—for both gardeners to use. No gardening can occur until this step is mostly finished, at which point the true gardening work can begin. The delay caused by the first task creates a pause for the second task, after which both tasks can continue in parallel. In computer terms, this particular model occurs frequently.

In common programming tasks, the producer/consumer problem occurs in several typical scenarios. For example, programs that must rely on the reading of a file fit this scenario: the results of the file I/O become the input to the next step, which might be threaded. However, that step cannot begin until the reading is either complete or has progressed sufficiently for other processing to kick off. Another common programming example is parsing: an input file must be parsed, or analyzed semantically, before the back-end activities, such as code generation in a compiler, can begin.

The producer/consumer problem has several interesting dimensions:

- The dependence created between consumer and producer can cause significant delays if this model is not implemented correctly. A performance-sensitive design seeks to understand the exact nature of the dependence and diminish the delay it imposes. It also aims to avoid situations in which consumer threads are idling while waiting for producer threads.
- In the ideal scenario, the hand-off between producer and consumer is completely clean, as in the example of the file parser. The output is context-independent and the consumer has no need to know anything about the producer. Many times, however, the producer and consumer components do not enjoy

such a clean division of labor, and scheduling their interaction requires careful planning.

- If the consumer is finishing up while the producer is completely done, one thread remains idle while other threads are busy working away. This issue violates an important objective of parallel processing, which is to balance loads so that all available threads are kept busy. Because of the logical relationship between these threads, it can be very difficult to keep threads equally occupied.

In the next section, we'll take a look at the pipeline pattern that allows developers to solve the producer/consumer problem in a scalable fashion.

Implications of Different Decompositions

Different decompositions provide different benefits. If the goal, for example, is ease of programming and tasks can be neatly partitioned by functionality, then task decomposition is more often than not the winner. Data decomposition adds some additional code-level complexity to tasks, so it is reserved for cases where the data is easily divided and performance is important.

The most common reason for threading an application is performance. And in this case, the choice of decompositions is more difficult. In many instances, the choice is dictated by the problem domain: some tasks are much better suited to one type of decomposition. But some tasks have no clear bias. Consider for example, processing images in a video stream. In formats with no dependency between frames, you'll have a choice of decompositions. Should they choose task decomposition, in which one thread does decoding, another color balancing, and so on, or data decomposition, in which each thread does all the work on one frame and then moves on to the next? To return to the analogy of the gardeners, the decision would take this form: If two gardeners need to mow two lawns and weed two flower beds, how should they proceed? Should one gardener only mow—that is, they choose task based decomposition—or should both gardeners mow together then weed together?

In some cases, the answer emerges quickly—for instance when a resource constraint exists, such as only one mower. In others where each gardener has a mower, the answer comes only through careful analysis of the constituent activities. In the case of the gardeners, task

decomposition looks better because the start-up time for mowing is saved if only one mower is in use. Ultimately, you determine the right answer for your application's use of parallel programming by careful planning and testing. The empirical timing and evaluation plays a more significant role in the design choices you make in parallel programming than it does in standard single-threaded programming.

Challenges You'll Face

The use of threads enables you to improve performance significantly by allowing two or more activities to occur simultaneously. However, developers cannot fail to recognize that threads add a measure of complexity that requires thoughtful consideration to navigate correctly. This complexity arises from the inherent fact that more than one activity is occurring in the program. Managing simultaneous activities and their possible interaction leads you to confronting four types of problems:

- *Synchronization* is the process by which two or more threads coordinate their activities. For example, one thread waits for another to finish a task before continuing.
- *Communication* refers to the bandwidth and latency issues associated with exchanging data between threads.
- *Load balancing* refers to the distribution of work across multiple threads so that they all perform roughly the same amount of work.
- *Scalability* is the challenge of making efficient use of a larger number of threads when software is run on more-capable systems. For example, if a program is written to make good use of four processor cores, will it scale properly when run on a system with eight processor cores?

Each of these issues must be handled carefully to maximize application performance. Subsequent chapters describe many aspects of these problems and how best to address them on multi-core systems.

Parallel Programming Patterns

For years object-oriented programmers have been using design patterns to logically design their applications. Parallel programming is no different than object-oriented programming—parallel programming problems

generally fall into one of several well known patterns. A few of the more common parallel programming patterns and their relationship to the aforementioned decompositions are shown in Table 3.2.

Table 3.2 Common Parallel Programming Patterns

Pattern	Decomposition
Task-level parallelism	Task
Divide and Conquer	Task/Data
Geometric Decomposition	Data
Pipeline	Data Flow
Wavefront	Data Flow

In this section, we'll provide a brief overview of each pattern and the types of problems that each pattern may be applied to.

- *Task-level Parallelism Pattern*. In many cases, the best way to achieve parallel execution is to focus directly on the tasks themselves. In this case, the task-level parallelism pattern makes the most sense. In this pattern, the problem is decomposed into a set of tasks that operate independently. It is often necessary remove dependencies between tasks or separate dependencies using replication. Problems that fit into this pattern include the so-called *embarrassingly parallel* problems, those where there are no dependencies between threads, and *replicated data* problems, those where the dependencies between threads may be removed from the individual threads.
- *Divide and Conquer Pattern*. In the divide and conquer pattern, the problem is divided into a number of parallel sub-problems. Each sub-problem is solved independently. Once each sub-problem is solved, the results are aggregated into the final solution. Since each sub-problem can be independently solved, these sub-problems may be executed in a parallel fashion.
- The divide and conquer approach is widely used on sequential algorithms such as merge sort. These algorithms are very easy to parallelize. This pattern typically does a good job of load balancing and exhibits good locality; which is important for effective cache usage.
- *Geometric Decomposition Pattern*. The geometric decomposition pattern is based on the parallelization of the data structures

used in the problem being solved. In geometric decomposition, each thread is responsible for operating on data 'chunks'. This pattern may be applied to problems such as heat flow and wave propagation.

- *Pipeline Pattern*. The idea behind the pipeline pattern is identical to that of an assembly line. The way to find concurrency here is to break down the computation into a series of stages and have each thread work on a different stage simultaneously.
- *Wavefront Pattern*. The wavefront pattern is useful when processing data elements along a diagonal in a two-dimensional grid. This is shown in Figure 3.1

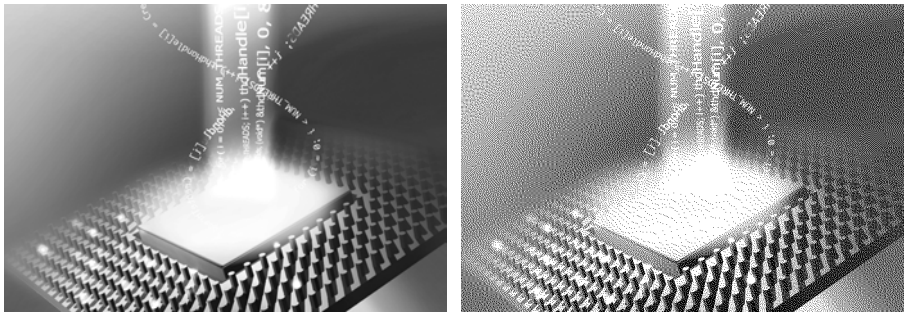
Figure 3.1 Wavefront Data Access Pattern

The numbers in Figure 3.1 illustrate the order in which the data elements are processed. For example, elements in the diagonal that contains the number "3" are dependent on data elements "1" and "2" being processed previously. The shaded data elements in Figure 3.1 indicate data that has already been processed. In this pattern, it is critical to minimize the idle time spent by each thread. Load balancing is the key to success with this pattern.

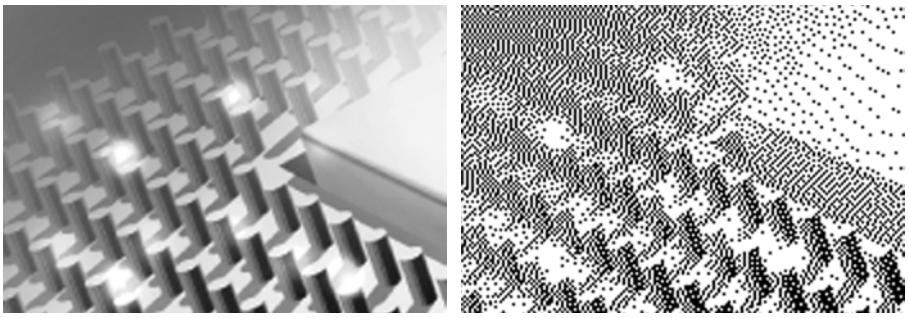
For a more extensive and thorough look at parallel programming design patterns, refer to the book *Patterns for Parallel Programming* (Mattson 2004).

A Motivating Problem: Error Diffusion

To see how you might apply the aforementioned methods to a practical computing problem, consider the error diffusion algorithm that is used in many computer graphics and image processing programs. Originally proposed by Floyd and Steinberg (Floyd 1975), *error diffusion* is a technique for displaying continuous-tone digital images on devices that have limited color (tone) range. Printing an 8-bit grayscale image to a black-and-white printer is problematic. The printer, being a bi-level device, cannot print the 8-bit image natively. It must simulate multiple shades of gray by using an approximation technique. An example of an image before and after the error diffusion process is shown in Figure 3.2. The original image, composed of 8-bit grayscale pixels, is shown on the left, and the result of the image that has been processed using the error diffusion algorithm is shown on the right. The output image is composed of pixels of only two colors: black and white.



Original 8-bit image on the left, resultant 2-bit image on the right. At the resolution of this printing, they look similar.



The same images as above but zoomed to 400 percent and cropped to 25 percent to show pixel detail. Now you can clearly see the 2-bit black-white rendering on the right and 8-bit gray-scale on the left.

Figure 3.2 Error Diffusion Algorithm Output

The basic error diffusion algorithm does its work in a simple three-step process:

1. Determine the output value given the input value of the current pixel. This step often uses quantization, or in the binary case, thresholding. For an 8-bit grayscale image that is displayed on a 1-bit output device, all input values in the range [0, 127] are to be displayed as a 0 and all input values between [128, 255] are to be displayed as a 1 on the output device.
2. Once the output value is determined, the code computes the error between what should be displayed on the output device and what is actually displayed. As an example, assume that the current input pixel value is 168. Given that it is greater than our threshold value (128), we determine that the output value will be a 1. This value is stored in the output array. To compute the error, the program must normalize output first, so it is in the same scale as the input value. That is, for the purposes of computing the display error, the output pixel must be 0 if the output pixel is 0 or 255 if the output pixel is 1. In this case, the display error is the difference between the actual value that should have been displayed (168) and the output value (255), which is –87.
3. Finally, the error value is distributed on a fractional basis to the neighboring pixels in the region, as shown in Figure 3.3.

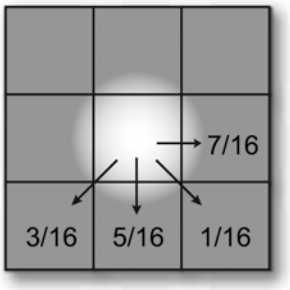


Figure 3.3 Distributing Error Values to Neighboring Pixels

This example uses the Floyd-Steinberg error weights to propagate errors to neighboring pixels. 7/16ths of the error is computed and added

to the pixel to the right of the current pixel that is being processed. 5/16ths of the error is added to the pixel in the next row, directly below the current pixel. The remaining errors propagate in a similar fashion. While you can use other error weighting schemes, all error diffusion algorithms follow this general method.

The three-step process is applied to all pixels in the image. Listing 3.1 shows a simple C implementation of the error diffusion algorithm, using Floyd-Steinberg error weights.

```
/************************************
* Initial implementation of the error diffusion algorithm.
************************************/

void error_diffusion(unsigned int width,
                     unsigned int height,
                     unsigned short **InputImage,
                     unsigned short **OutputImage)
{
   for (unsigned int i = 0; i < height; i++)
   {
      for (unsigned int j = 0; j < width; j++)
      {
         /* 1. Compute the value of the output pixel*/
         if (InputImage[i][j] < 128)
            OutputImage[i][j] = 0;
         else
            OutputImage[i][j] = 1;

         /* 2. Compute the error value */
         int err = InputImage[i][j] - 255*OutputImage[i][j];

         /* 3. Distribute the error */
         InputImage[i][j+1]   += err * 7/16;
         InputImage[i+1][j-1] += err * 3/16;
         InputImage[i+1][j]   += err * 5/16;
         InputImage[i+1][j+1] += err * 1/16;
      }
   }
}
```

Listing 3.1 C-language Implementation of the Error Diffusion Algorithm

Analysis of the Error Diffusion Algorithm

At first glance, one might think that the error diffusion algorithm is an inherently serial process. The conventional approach distributes errors to neighboring pixels as they are computed. As a result, the previous pixel's error must be known in order to compute the value of the next pixel. This interdependency implies that the code can only process one pixel at a time. It's not that difficult, however, to approach this problem in a way that is more suitable to a multithreaded approach.

An Alternate Approach: Parallel Error Diffusion

To transform the conventional error diffusion algorithm into an approach that is more conducive to a parallel solution, consider the different decomposition that were covered previously in this chapter. Which would be appropriate in this case? As a hint, consider Figure 3.4, which revisits the error distribution illustrated in Figure 3.3, from a slightly different perspective.

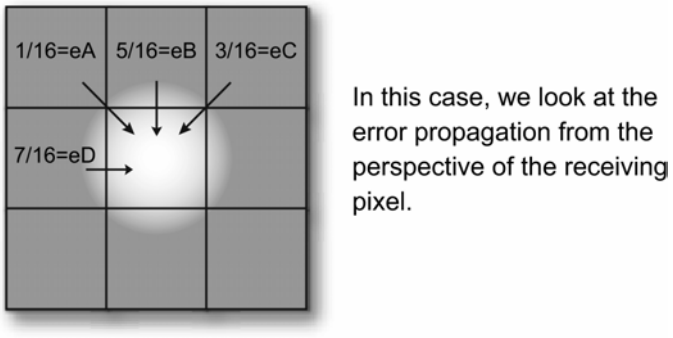


Figure 3.4 Error-Diffusion Error Computation from the Receiving Pixel's Perspective

Given that a pixel may not be processed until its spatial predecessors have been processed, the problem appears to lend itself to an approach where we have a producer—or in this case, multiple producers—producing data (error values) which a consumer (the current pixel) will use to compute the proper output pixel. The flow of error data to the current pixel is critical. Therefore, the problem seems to break down into a data-flow decomposition.

Now that we identified the approach, the next step is to determine the best pattern that can be applied to this particular problem. Each independent thread of execution should process an equal amount of work (load balancing). How should the work be partitioned? One way, based on the algorithm presented in the previous section, would be to have a thread that processed the even pixels in a given row, and another thread that processed the odd pixels in the same row. This approach is ineffective however; each thread will be blocked waiting for the other to complete, and the performance could be worse than in the sequential case.

To effectively subdivide the work among threads, we need a way to reduce (or ideally eliminate) the dependency between pixels. Figure 3.4 illustrates an important point that's not obvious in Figure 3.3—that in order for a pixel to be able to be processed, it must have three error values (labeled eA, eB, and eC[1] in Figure 3.3) from the previous row, and one error value from the pixel immediately to the left on the current row. Thus, once these pixels are processed, the current pixel may complete its processing. This ordering suggests an implementation where each thread processes a row of data. Once a row has completed processing of the first few pixels, the thread responsible for the next row may begin its processing. Figure 3.5 shows this sequence.

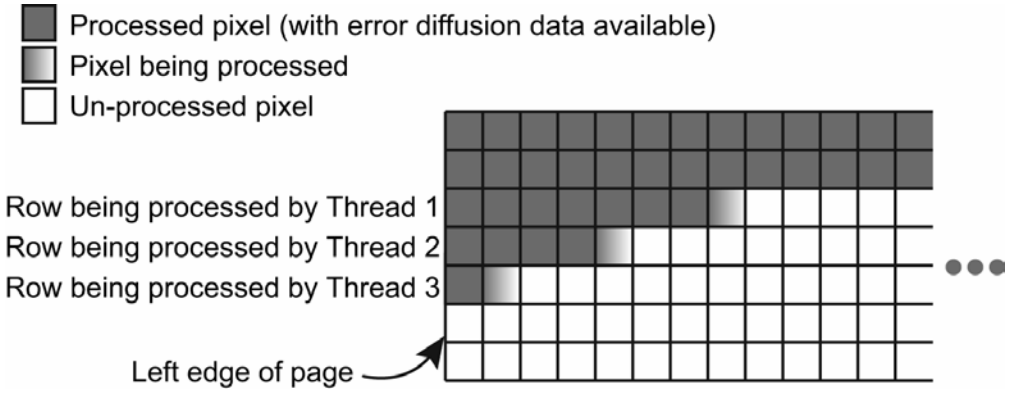


Figure 3.5 Parallel Error Diffusion for Multi-thread, Multi-row Situation

[1] We assume eA = eD = 0 at the left edge of the page (for pixels in column 0); and that eC = 0 at the right edge of the page (for pixels in column W-1, where W = the number of pixels in the image).

Notice that a small latency occurs at the start of each row. This latency is due to the fact that the previous row's error data must be calculated before the current row can be processed. These types of latency are generally unavoidable in producer-consumer implementations; however, you can minimize the impact of the latency as illustrated here. The trick is to derive the proper workload partitioning so that each thread of execution works as efficiently as possible. In this case, you incur a two-pixel latency before processing of the next thread can begin. An 8.5" X 11" page, assuming 1,200 dots per inch (dpi), would have 10,200 pixels per row. The two-pixel latency is insignificant here.

The sequence in Figure 3.5 illustrates the data flow common to the wavefront pattern.

Other Alternatives

In the previous section, we proposed a method of error diffusion where each thread processed a row of data at a time. However, one might consider subdividing the work at a higher level of granularity. Instinctively, when partitioning work between threads, one tends to look for independent tasks. The simplest way of parallelizing this problem would be to process each page separately. Generally speaking, each page would be an independent data set, and thus, it would not have any interdependencies. So why did we propose a row-based solution instead of processing individual pages? The three key reasons are:

- *An image may span multiple pages.* This implementation would impose a restriction of one image per page, which might or might not be suitable for the given application.
- *Increased memory usage.* An 8.5 x 11-inch page at 1,200 dpi consumes 131 megabytes of RAM. Intermediate results must be saved; therefore, this approach would be less memory efficient.
- *An application might, in a common use-case, print only a single page at a time.* Subdividing the problem at the page level would offer no performance improvement from the sequential case.

A hybrid approach would be to subdivide the pages and process regions of a page in a thread, as illustrated in Figure 3.6.

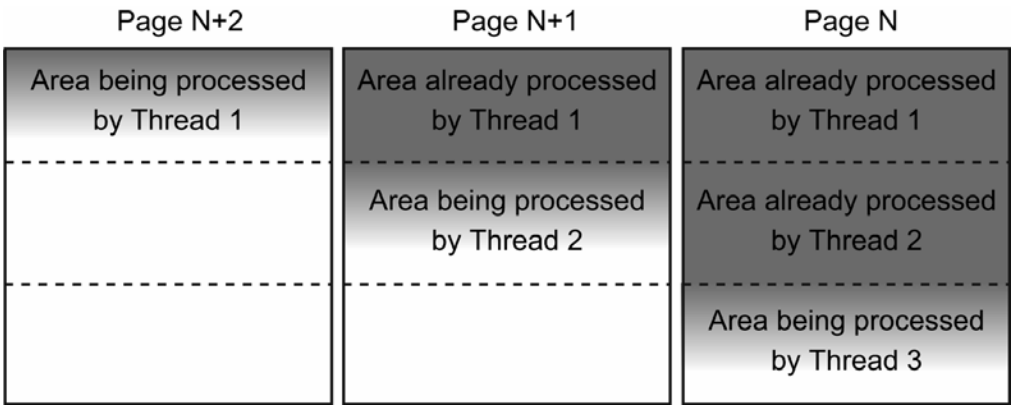


Figure 3.6 Parallel Error Diffusion for Multi-thread, Multi-page Situation

Note that each thread must work on sections from different page. This increases the startup latency involved before the threads can begin work. In Figure 3.6, Thread 2 incurs a 1/3 page startup latency before it can begin to process data, while Thread 3 incurs a 2/3 page startup latency. While somewhat improved, the hybrid approach suffers from similar limitations as the page-based partitioning scheme described above. To avoid these limitations, you should focus on the row-based error diffusion implementation illustrated in Figure 3.5.

Key Points

This chapter explored different types of computer architectures and how they enable parallel software development. The key points to keep in mind when developing solutions for parallel computing architectures are:

- Decompositions fall into one of three categories: task, data, and data flow.
- Task-level parallelism partitions the work between threads based on tasks.
- Data decomposition breaks down tasks based on the data that the threads work on.

- Data flow decomposition breaks down the problem in terms of how data flows between the tasks.
- Most parallel programming problems fall into one of several well known patterns.
- The constraints of synchronization, communication, load balancing, and scalability must be dealt with to get the most benefit out of a parallel program.

Many problems that appear to be serial may, through a simple transformation, be adapted to a parallel implementation.