MULTI-CORE PROCESSORS AND THE FUTURE OF PARALLELISM IN SOFTWARE

A Thesis

Presented to the

Faculty of

California State University,

San Bernardino

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

in

Computer Science

---

by

Ryan Christopher Youngman

June 2007

MULTI-CORE PROCESSORS AND THE FUTURE OF PARALLELISM IN SOFTWARE

A Thesis

Presented to the

Faculty of

California State University,

San Bernardino

---

by

Ryan Christopher Youngman

June 2007

Approved by:

Dr. Ernesto Gomez, Advisor, Computer Science	Date
Dr. Keith Schubert	
Dr. Yasha Karant	

ABSTRACT

Recent changes in processor architectures are showing a movement towards utilizing multi-core technology in their designs. The physical limitations of current processors, advances in fabrication technology, and increasing performance demands have provided the impetus for multi-core technology to become a reality. The apparent processing potential, more efficient resource usage and market interest hint that this technology will become more of a standard in modern processor architecture designs. It seems that from now on, computers will no longer be Von Neumann machines, but rather, they will be parallel machines working together.

Multi-core architecture provides benefits such as less power consumption, scalability, and improved application performance enabled by thread-level parallelism. However, with the introduction of this technology, there are implications that we need to understand and consider. What exactly is a "core" and how does a multi-core CPU differ from the traditional single-core CPU? More importantly, how do we take advantage of this hardware when developing software applications? The purpose of this thesis is to examine multi-core technology and answer some of these questions.

We will start by discussing the physical and engineering problems and why multi-core design is the natural evolution of microprocessors. Next, we examine multi-core processors themselves and understand the benefits they provide. A thorough discussion of multi-threading concepts follows, leading to a multi-threaded example application. In the last chapter, we will understand how multi-core processors will change the software world forever and what we need to do to prepare for this significant change.

ACKNOWLEDGEMENTS

I would like to express my appreciation to Dr. Ernesto Gomez for the many though-provoking conversations with him concerning parallel concepts and theory. Due to these, I eventually arrived at the topic of my thesis and I am appreciative of his support. Dr. Josephine Mendoza, acting as my graduate coordinator, assisted me while following the necessary steps to complete my thesis and graduate from Cal State.

A number of individuals at Intel Corporation were very generous to extend their support of my thesis. Greg Anderson supplied me with software licenses and few books from Intel Press. Charles Congdon provided me numerous references and offered his insight on the subject matter. Thanks to Walter Shands and Gary Carleton for writing the article which triggered my initial interest in multi-core processors. Many thanks to Chelsea Goff who was instrumental in helping me acquire remote access to a multi-core lab machine. AMD should also be acknowledged for their support through their Developer Center program, which offered me exclusive access to multi-core hardware.

Lastly, I would like to thank my family for their patience and emotional support. Many long nights were spent away from them while performing the research, writing, and programming tasks for this paper. I am very appreciative and grateful for those who supported and encouraged me throughout the process.

DEDICATION

To my wife Ruthie, and son Lucas.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

In recent years, there has been a considerable shift among microprocessor designs all of which hint at a major evolution in computing hardware. While fabrication technology continues to improve, manufactures are starting to reach the physical limitations of semi-conductor based microelectronics.

1.1 Physical and Engineering Challenges

In accordance with Moore's Law, the transistor density of processors has been increasing exponentially every eighteen to twenty-four months, and surprisingly, this law has held for the past forty years. However, since each transistor is itself a working electrical device that consumes power and produces heat, problems such as transistor leakage and excess heat consumption are starting to become a hindrance to the continuing pace of Moore's Law. If the transistor density rate continues at its present course, processors would eventually generate more heat per square centimeter than the surface of the sun [16].

Along with the heat considerations, we must also notice that with higher transistor densities, this requires more interconnects between the components in the CPU. As the total interconnects increase in length, path delays can surface and effectively nullify the speed increases of the transistors themselves.

Also, serial architectures such as memory have not been increasing as fast as logic processing speeds. A typical Intel i486 CPU in the late 1980s to the early 1990s would require 6 to 8 clock cycles to access memory [15]. Today's processors use more than 200 clock cycles and these wasted cycles undermine the frequency increases typically used to increase processor performance.

It seems as though the traditional methods of increasing the clock rate or packing more transistors onto the chip are starting to reach their practical limits. Even now, due to the heat constraints, semiconductor manufactures are actually decreasing clock rates in order to throttle back heat and obtain cooler running chips [19]. In the effort to produce faster, more efficient processors, manufactures are taking advantage of the improvements in fabrication technology and moving towards a chip-level multiprocessing architecture. Before we look at the current state of processors, let us examine how we arrived at where we are today.

2. EVOLUTION OF THE MICROPROCESSOR

2.1 In Pursuit of Parallel Computation

Over the years, microprocessor manufactures have strived to add more and more parallelism into the design of their chips. The main goal of any microprocessor architecture is to obtain efficient resource utilization in order to maximize performance. One way to do this is to perform as many operations as possible in a given clock cycle. For years, computer architects have used instruction-level parallelism (ILP) to achieve out-of-order execution. ILP allows the CPU to re-order the instructions in such as way that it eliminates pipeline stalls and allows for an increased number of instructions to execute in a single clock cycle.

A basic processor around the year 1989 featured pipelined execution where a number of instructions were active in their respective stages. By duplicating the functional units, superscalar processors arrived on the scene, represented by the Pentium processors of 1993. By having multiple adders, for example, a Pentium processor could execute multiple additions granted they were independent of each other. Later, with the addition of single instruction, multiple data (SIMD) instructions, processors were able to execute a single instruction on several data elements. This parallelism was made possible by MMX, SSE, and SSE2 instruction set additions.

While ILP is recognized for its tremendous benefits of which are only now reaching

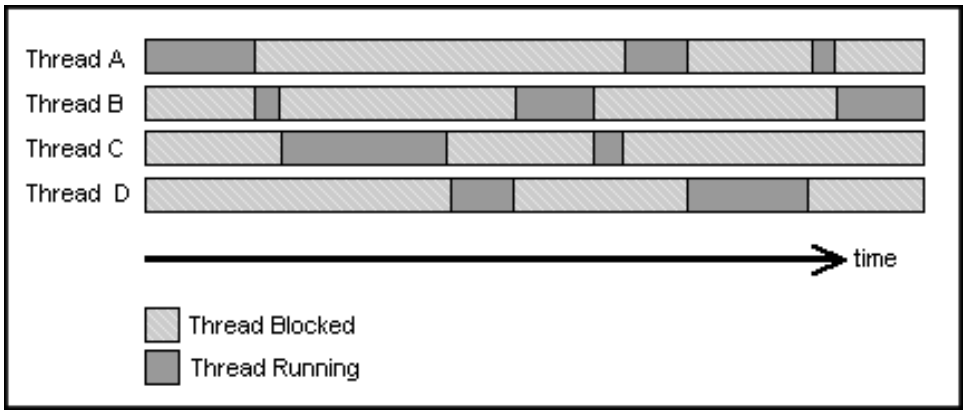

Fig. 2.1: Time-slicing over a Number of Processes

their limits, ILP relies heavily on the dependencies between instructions to be low or non-existent. Higher dependencies between instructions reduce the overall ability for parallel execution to take place. Thread-Level Parallelism (TLP), as we will discuss shortly, builds on top of ILP and is widely recognized as one of the most important areas of how future parallelism will be achieved.

As software became more complex and able to perform multiple tasks simultaneously, the ability for a processor to handle parallel computation was seen as a way of handling this challenge. Another method of achieving concurrency in software as apposed to hardware was to use preemptive, or time-sliced, multitasking. The use of time-slicing allowed developers the ability to hide latencies associated with I/O by interleaving execution of multiple threads running on the system. This gave the appearance of multiple programs running simultaneously, although at any given time, only one thread was executing. Figure 2.1 illustrates time-slicing over a number of process threads.

However, this method of time-slicing was still not parallel execution because of

one simple fact: only one instruction stream could run on a processor at a single point in time [1]. One way to solve this problem was to increase the number of physical processors in the system. A multiprocessor environment truly allows parallel execution because multiple threads or processes can be executed across the processors at the same time. Although this alternative solution worked, it came at a high price as multi-socket motherboards and the numerous processors that went on them it were very costly.

Trying to achieve efficient thread-level parallelism (TLP) on a single processor, computer architects found that the resources of the processor we at some times underutilized. Hence, they developed a technique called simultaneous multi-threading (SMT) which allowed the resources to be used more efficiently. In order to understand SMT, which heavily relies on threads, we first need to define what a thread is.

2.1.1 Definition of a Thread

A thread of execution, or more commonly known as a thread, can be though of as a basic unit of CPU utilization. It has a program counter which points to the current instruction. It has CPU state information for the current thread, and it also contains other resources such as a stack [1]. A processor is made up of a number of different resources: general registers, interrupt logic, caches, buses, execution units, and so forth. These different resources comprise the architecture state of the processor. In order to define a thread on a processor, only the architecture state is required. This is how processors handle time-slicing; by swapping in and out different architectural states over time. This context switching comes at a high cost though, and we will

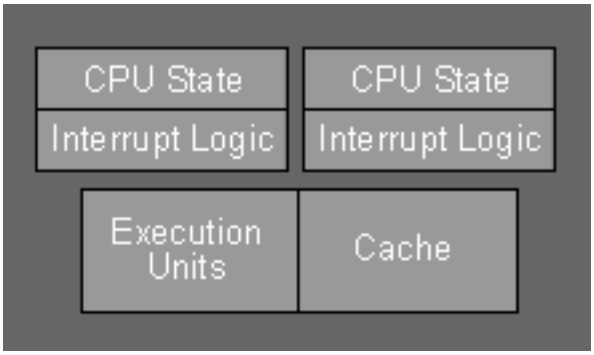

Fig. 2.2: Simultaneous Multi-Threading Architecture

discuss later how too much context switching can hurt application performance.

So, by duplicating the architectural space, designers could implement logical processors on one chip. Execution resources then become shared among the different logical processors. Figure 2.2 shows an example of a SMT processor system.

2.2 Hyper-Threading

Intel released its implementation of SMT in the early 2000 decade which was named Hyper-Threading Technology [15]. A processor using Hyper-Threading Technology appears to the operating system software as multiple logical processors even though the entire unit consists of just one physical processor.

Hyper-Threading Technology literally works by interleaves the instructions in the execution pipeline. Which instructions get inserted depends on what execution resources are available at execution time [1]. The threads can then be executed in a parallel fashion as the context switching is greatly reduced between the threads.

This latency hiding is the key behind Hyper-Threading Technology's performance gains. By having both thread states in the processor, when one thread blocks due to a

cache miss, branch misprediction, or data dependency, the other thread can execute to maximize processor resource usage. Now, the operating system scheduler can schedule multiple threads to execute just as they would in a multiprocessor system. It then becomes the job of the microprocessor to determine how to execute multiple threads by interleaving them as the necessary resources for each thread become available. In practice, it is recommended for each thread's work to be as different as possible in an effort to reduce contention for processor resources.

Despite the performance increase using SMT, there still exists only one execution resource that is shared among multiple threads. As a result, only one thread of execution is allowed to run at any given time and true thread-level parallelism is not possible.

3. MULTI-CORE ARCHITECTURE

While advents such as ILP, instruction sets additions such as MMX/SSE/SSE2, and Hyper-Threading Technology have improved processor performance, the next logical evolution of the modern microprocessor starts us on a path towards more available parallelism and true chip multiprocessing (CMP).

Modern chip architectures are starting to take advantage of fabrication technology in a whole new way. Current processor technologies are allowing manufactures to take advantage of more real estate on a single die. Manufacturers are using a "divide and conquer" strategy and are now able to implement two or more cores or "execution engines" on a single processor. Each core has its own set of execution and architectural resources such as floating point and integer units, but depending on the design, each core may or may not share a large on-chip cache. A multi-core processor plugs into a single socket, but the operating system perceives each core as a discrete processor with a full set of execution resources. Figures 3.1 and 3.2 demonstrate a simple comparison between single-core and multi-core architecture configurations. Other design configurations may feature differing cache sizes as well as shared or independent front side buses.

Just as improvements in cache utilization has helped with performance in the past, Intel Corporation has developed a technology named Advanced Smart Cache technol-

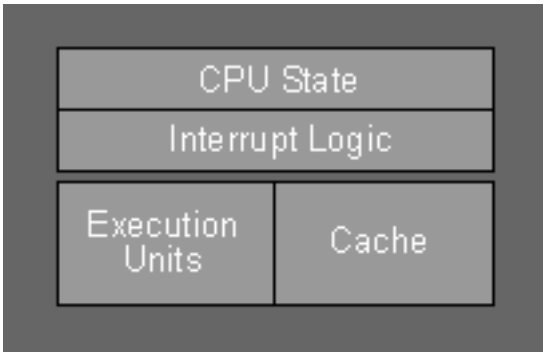

Fig. 3.1: Single-Core Architecture

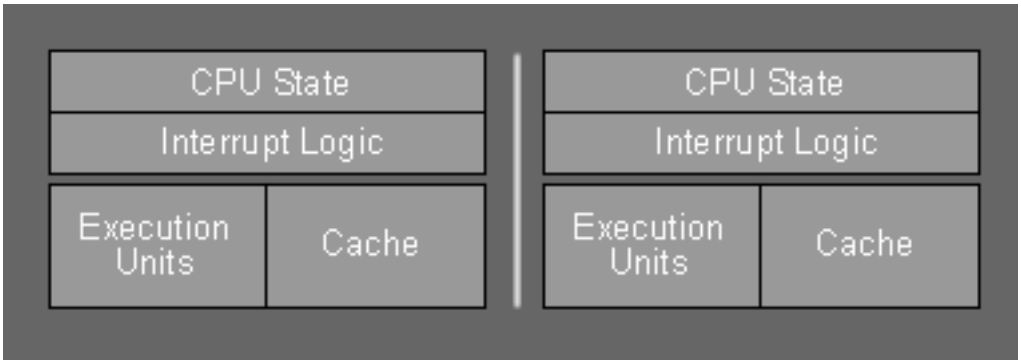

Fig. 3.2: Multi-Core Architecture

ogy (ASC) which should continue to deliver more efficient cache usage. Intel multi-core processors feature an optimized cache that improves efficiency by increasing the probability of each core accessing data from a higher-performance cache-subsystem. To accomplish this, the processor shares its L2 cache between cores. By sharing this cache, ACS also allows each core to dynamically utilize up to 100 percent of the available L2 cache [25]. For example, if one core has minimal cache requirements, other cores can increase their percentage of L2 cache. This results in reduced cache misses and bus traffic, as well as lower latency to data.

3.1 Advantages of Multi-Core Architecture

3.1.1 Increased Performance

Because each core has its own execution pipeline and set of execution resources, multiple threads can truly execute in parallel without blocking on resources needed by other threads. By splitting up computational work among the available cores, a multi-core processor can perform more work in a given cycle. Instead of the previous "scale-up" approach by increasing the clock rate, chip designers are now using a "scale-out" model for more efficient processing of multiple tasks.

Compared to a single-core processor, a multi-core processor with two cores is not twice as fast as a single-core processor of the similar speed. However, it can come close. When comparing an over-clocked single-core processor to a dual-core processor, tests have been shown that a dual-core processor consumes roughly the same amount of power while delivering more than a 70 percent performance improvement [18].

Another reason for increased performance is that multi-core processors greatly reduce the associated cost when doing context switching between threads. This results in much less overhead and greater code processing throughput.

3.1.2 Reduced Power Consumption and Heat

One area in which multi-core architecture promises to deliver is that of reduced heat and power consumption. As most of the processing performance improvements have come from improvements to cache, clock speed, memory access and I/O, each of these has required an additional increase in power consumption. While increasing heat and power resulting in improved performance may lead to the diminishing return issue, it seemed a better solution was needed.

Multi-core architecture is different from single-core processors in that rather then each chip having its own separate architecture components such as memory and I/O, each core shares these resources. The power required to support these shared resources is minimally higher than those required for identical resources on a single-core processor. The only increase in power consumption comes from the addition of the extra execution core(s) to the processor. The result is a processor that provides greater performance than a single-core processor, without doubling the power requirements.

Since multi-core processors do more work in a given clock cycle, they can thus operate at lower frequencies. Semiconductor manufactures are actually starting to scale back clock speeds as they introduce more cores so the chips can run cooler [19]. Since power consumption goes up proportionally with frequency, multi-core architecture helps combat the issue of increasing power and cooling requirements. For a

more in depth look at power and thermal management with respect to Intel microprocessors, please consult [2]. AMD's optimized power management technology named PowerNow! is discussed in [14] and [3]. Please refer to these for more information.

3.1.3 Benefits in the Data Center

Multi-core processors are starting to become a favorable option in the data center due to their intrinsic benefits. IT managers constantly struggle to deliver higher levels of service while reducing costs associated with hardware, power, and physical space requirements. A common practice among server systems today is to deploy one application per server. Also, organizations commonly overprovision IT resources for peak usage, yielding low utilization rates. With multi-core processors, server applications will experience faster throughput rates as multiple simultaneous transactions will be allowed to be processed at the same time. Service applications, databases, and real-time systems will enjoy increased responsiveness. Servers will be able to handle larger application and data loads which will aid in the effort of server consolidation and ultimately result in a reduced total cost of ownership.

With the reduced number of servers also comes the decreased cost of ongoing maintenance and management from IT administrators. As previously mentioned, a processor with multiple cores requires only a single socket, which aids when dealing with scalability issues. Instead of the reliance on multi-processor hardware, which tends to be very expensive, replacing a single multi-core processor with another that contains more cores is all that is needed to upgrade a machine with more processing cores.

3.1.4 Virtualization

Multi-core processors naturally lend themselves to support the virtualization trend which is hot across areas of different industries and tied to the progress towards greater consolidation in the datacenter. Specific operating systems and/or applications can be dedicated to specific cores. Virtual machine strategies are concerned with isolating applications and operating systems from one another while they run on the same hardware. A multi-core processor would allow each core to host an operating system or application in separate physical states, while drawing from a common memory pool.

3.1.5 Economical Changes

Maybe one of the most over-looked aspects of multi-core is the economical change it will bring. By having multi-core processors as commodity items, the economics of computing will change enabling larger and larger systems to be built at a cheaper cost. Grid computers and cluster systems will exhibit even more processing power and will be attainable for many organizations, and not reserved for scientific and academic communities. Multi-core machines will be a more cost-effective solution for high computing needs, and even if the total costs of operation are not dramatically reduced, the increased computational power could be the justification alone.

3.2 Taking Advantage of Multi-Core Architecture

Software applications that will make the most of the thread-level parallelism available in multi-core processors will be those that have included multi-threading in their

design. The software must be written so that it can spread its workload across multiple execution cores allowing each core to execute completely separate threads of code. But what are the challenges associated with multi-threading? Any seasoned software engineer will tell you that multi-threading an application is not a trivial task. There are many factors to consider and one must first evaluate if the need for threading is beneficial at all.

In the next section, we will discuss the aspects of multi-threading programming. We will explore how threading is handled by the application, the operating system, and also the processor. Issues of data synchronization, race conditions, and deadlock will be mentioned as they pertain to multi-threading.

4. MULTI-THREADED PROGRAMMING

To take full advantage of multi-core technology, software applications must be multi-threaded. The work performed must be able to be spread out among the execution units of a multi-core processor so they can execute at the same point in time. A good place to start taking a closer look at multi-threading is to first understand parallel computing and parallel hardware.

In order to achieve parallel execution, we must first have hardware that supports simultaneous execution of threads. In general, a computing system can be described in terms of how instructions and data are processed. Michael J. Flynn proposed a taxonomy that places computing platforms in one of four categories [7]. Flynn's Taxonomy is depicted in Figure 4.1.

Most of the designs for running parallel applications can be classified into two types: distributed memory MIMD or shared memory MIMD architectures. A shared memory system is different from a distributed memory system in that a single address space is shared across all processing elements. As the name suggests, distributed memory systems have distinct memories and the processing elements must interact through an interconnection network. Figure 4.2 and Figure 4.3 illustrate the two different architectures.

Both of these architectures are important and either one is chosen depending on

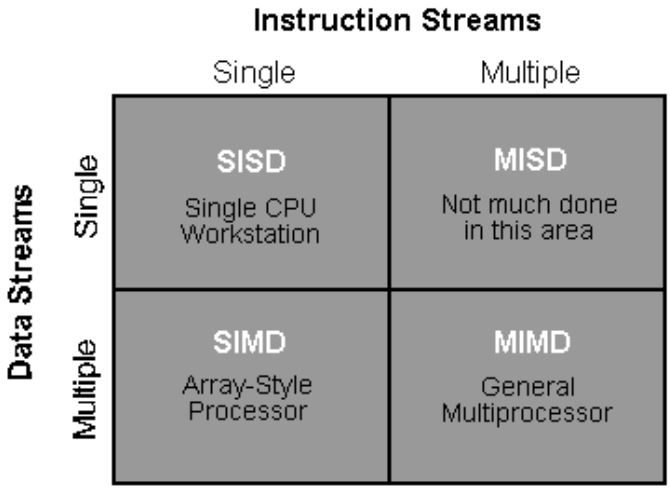

Fig. 4.1: Flynn's Taxonomy

the problem being solved. However, over the next few years, the number of shared memory systems will explode due to the push for multi-core architecture. Soon, shared memory systems will be the norm and programmers will have wider and more available access to parallel hardware on their target platforms than ever before.

Fortunately, high performance computers built to run parallel applications have been around since the early 1980s. Since then, much has been learned about parallel algorithms, parallel programming, and the tools required to make a parallel programmer's life easier and more productive. For years, parallel programming was the domain of high performance computing (HPC) which was widely used by the scientific and academic communities.

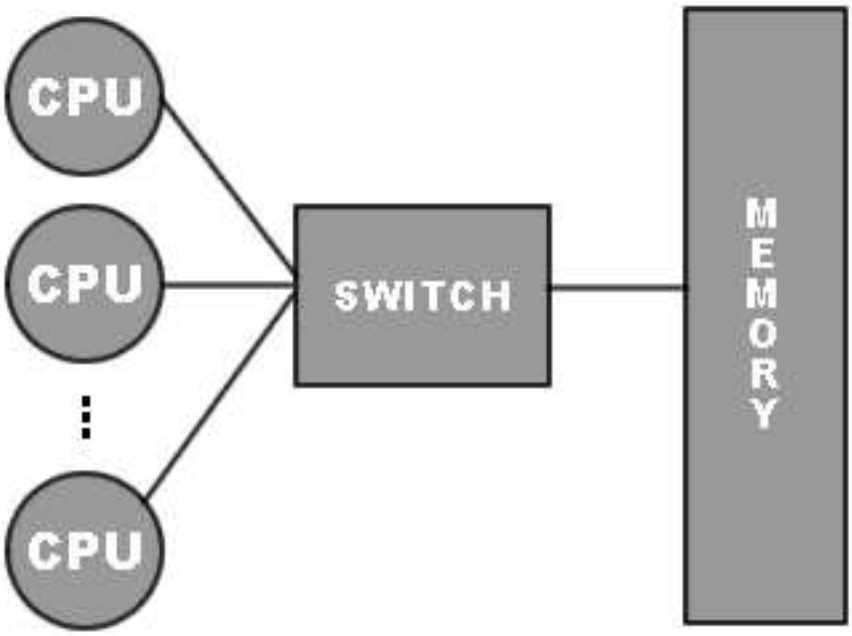

Fig. 4.2: Shared Memory MIMD

4.1 Motivation for Threading

What exactly is the motivation behind the decision to thread an application? First and foremost, the trend toward multi-core for server, desktop, and mobile class processors is expected to continue well into the future. To take full advantage of this new hardware requires that your applications be multi-threaded. By doing this, your applications will experience increased performance by running in a parallel environment. Your users will have improved application responsiveness and productivity as they are able to increase the amount of work that can be done in less time. As hardware manufactures continue to add more cores, applications that are not threaded will only utilize a percentage of the total processing power available to them.

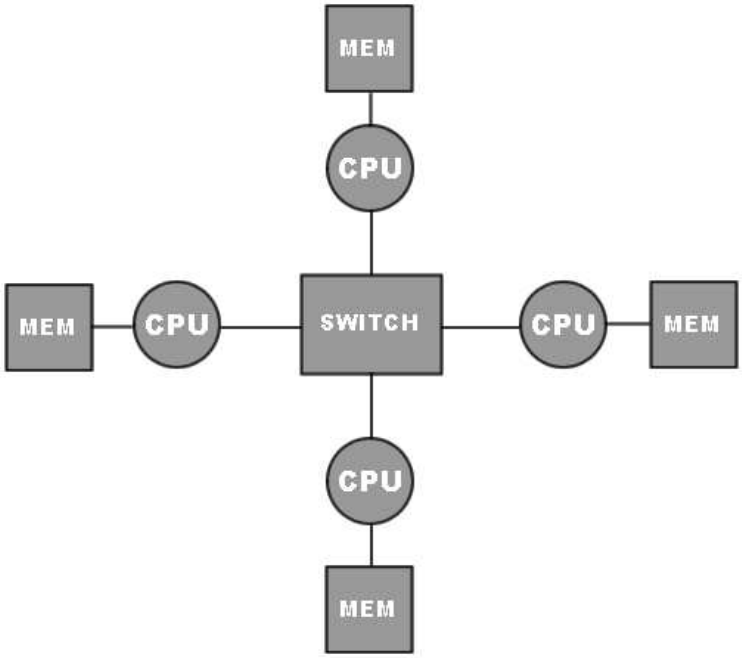

Fig. 4.3: Distributed Memory MIMD

What kinds of problems are good to solve through parallelism? The problems that are ideal are ones that can be partitioned by either of two methods: domain or functional decomposition. The goal of both these methods is to identify independent computations and primitive tasks that have no or limited relationships.

4.2 Domain (Data) Decomposition

Domain decomposition is a form of data parallelism where the same operation is applied to all data. For example, if you were encoding an MP3 file, you could partition the file into multiple chunks of data and send each chunk of data to multiple threads for encoding. Reconstruction of all chunks into the final encoded result would then be necessary. An example of where data decomposition can be found in code is usually

on independent loop iterations. Domain decomposition is a highly scalable approach and allows for better performance as more processors are added.

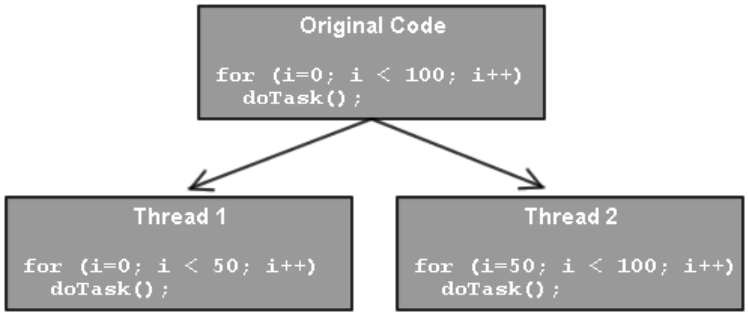

Fig. 4.4: Domain (Data) Decomposition

4.3 Functional (Task) Decomposition

Functional decomposition is a noticeably harder method of parallelism where a problem is segmented into unique jobs or tasks. Each task is then distributed among threads which execute concurrently. Simulation of complex systems falls into this category, for example, simulating the systems and dynamics of an automobile. Each sub-system simulation from the engine model, suspension model, to an aerodynamics model can all be simulated in a parallel fashion if these components are partitioned as independent tasks. Figure 4.5 shows a rather simple example of functional decomposition.

4.4 A Methodology for Multi-Threaded Development

Many experts suggest using a methodology when engaging in a multi-threading development effort. In this section, we will cover a number of stages which are part of

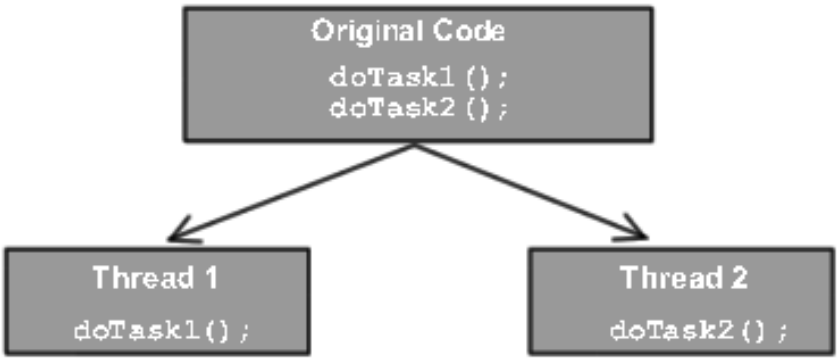

Fig. 4.5: Functional (Task) Decomposition

an iterative process aimed at producing an optimal multi-threaded solution. From identifying candidate areas for parallelism to analyzing the performance, following a process may be of help to you. Figure 4.6 shows a visual representation of this process.

4.4.1 Identifying Parallelism

The first stage of parallel application development begins with identifying opportunities for parallelism in the application. Naturally, the problems in your application must contain areas for parallel work to be done. If the problems are inherently structured as a single task with a fixed order of events, there is just no concurrency to work with. The one goal of this stage is to identify hotspot areas that could be candidates for threading. Hotspot functions with no data dependencies and loops with independent iterations are some common areas where threading may benefit. Software architects who have intimate knowledge of the application and its algorithms can be an excellent source to find and propose areas for multi-threading. You may also use

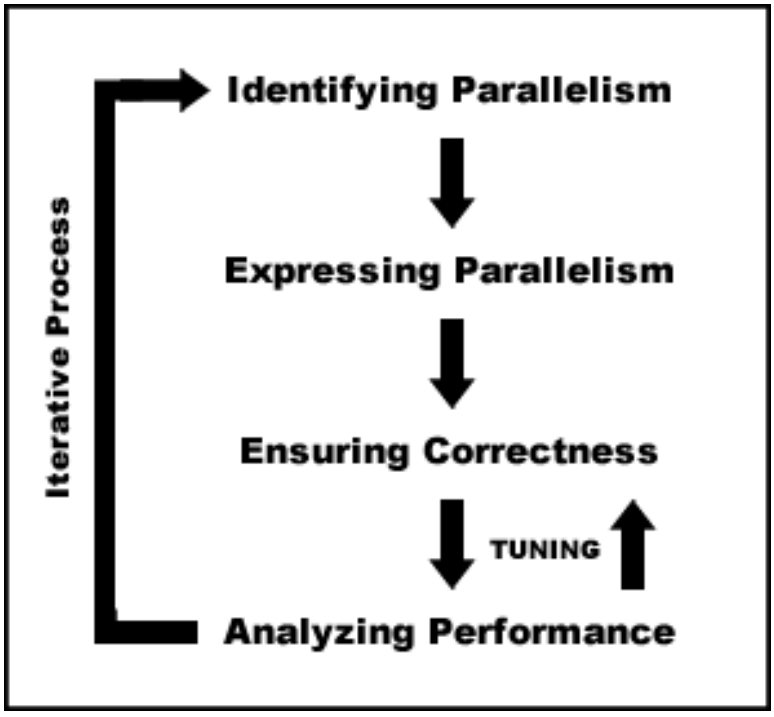

Fig. 4.6: Multi-Threaded Development Methodology

software tools to analyze your application and discover not only hotspots, but which hotspots will give you the greatest performance return on your parallelization efforts.

But what if your application does not have any use for multi-threading? If this is the case, a multi-core processor will simply benefit from improved multi-tasking while running multiple applications. Fortunately, single-threaded applications that run on single-core processors will also run on multi-core processors as well with no code alteration.

Granularity

The issue of granularity is one of many concerns when solving a parallel problem. Often, finding the right size of "chunks" of work and be a challenging task. Having

chunks of work that are too large can lead to an imbalance in the load where one or more threads are overworked, while others are not being used efficiently. On the other hand, too much granularity can lead to synchronization overhead and can effectively reduce the benefits of threading altogether. The levels of granularity are important because of how they affect synchronization and locking of critical sections and shared memory variables. A common solution to this problem is to dynamically adjust the granularity based on the data set and system configuration in order to maintain load balance and reduce synchronization.

Load Balancing

Closely related to granularity, load balancing is also a unique problem. The goal here is to give each thread an equal size amount of work. Doing so will enable the threads to complete as close together in time as possible, which reduces the amount of time on thread waits and synchronization. More often than not, this does not occur and can be very challenging to obtain. For data decomposition problems, equal splitting of the data is recommended. Conversely for task decomposition, creating equal sized tasks will generally result in more efficient load balancing. For this method in particular, it may be data-dependent so adjusting the tasks dynamically is a common practice. For example, one thread might get several tasks instead of a single task. Again, it depends on various factors of the data, but there are profiling tools which can help to assess load and give you insight into how your application executes. In either case of data or task decomposition, you may have to resort to reducing the granularity of the parallel work to obtain better load balancing. Remember though that this

reduction in granularity increases the probability of increased synchronization. This is a tradeoff that must be analyzed and evaluated in order to make the best decision concerning which is the more favorable option for the application.

Overhead and Synchronization

When thinking about analyzing and implementing a parallel application, it is helpful for one to recognize the overhead concerns associated with performing parallel work. First and foremost, the creation of threads in the system is a very expensive operation and should be done infrequently. A recommended practice when working with threads is to utilize a thread pool in which threads can be reused as necessary. This method is an efficient way to remove the costs associated with destroying and subsequently creating new threads.

As a rule, synchronization should occur in the smallest region of code as possible. If your granularity is such that is it quite large, the execution of your application becomes more serialized as other threads must wait on the section being locked. This problem can also occur when multiple threads try to acquire the same lock at the same time. Causing a thread to enter its sleep state, then wake it up is an expensive task and should happen as little as possible. Optimal threads are those that are in their active state as long as possible. Also, the frequency of synchronization should as minimal as possible. As mentioned previously, too much granularity leads to more synchronization and this overhead can eventually dominate the application.

One method of reducing synchronization overhead is to minimize the sharing of data across threads. An excess of data sharing can lead to false-sharing overhead.

False-sharing occurs when two threads are altering data that lie on the same cache line. When one thread changes data on a line of cache, it causes the cache to become invalidated. The second thread must then wait while the cache is reloaded from memory. This does not necessarily mean that an error exists in the program, but if this cache pinging happens frequently, for instance inside of a loop, is it likely to severely affect performance. One way to detect false-sharing behavior is to observe the L2 cache miss rates using software analysis tools.

4.4.2 Expressing Parallelism

Currently, there are a number of options for expressing parallelism into your code. Which method is chosen depends on your application, the skill set of your developers, and on the problem being solved. However, they are also not mutually exclusive; you may mix and match them to meet the requirements of your project. For multi-threading programming applications, the most common methods are to utilize explicit threading, OpemMP, programming language APIs, or internally threaded libraries.

Explicit Threading

By using explicit threading libraries, such as Win32 threads and POSIX threads (for UNIX/Linux operating systems), you can achieve a fine-grained control of all the low level details of managing the threads. This allows for the programmer to handle a wider range of algorithms and do more to tune the application to meet their needs. Explicit threading libraries can support a large range of compilers and languages due to the fact that they only need an interface to the multi-threading library of the

system. A considerable amount of code must be written to create threads and the code that will run within the thread. Therefore, explicit threading libraries are more error prone and harder to use then other methods simply due to the fact that you must control the low level details of thread management.

OpenMP

OpenMP is a portable, industry-wide standard collection of directives and runtime library routines for C, C++, and FORTRAN. It greatly simplifies parallel application development by hiding many of the details of thread management and communication. It consists of a small number of compiler directives, such as pragmas, which specify sections of code which tell the compiler to execute in parallel.

OpenMP works on the concept of parallel regions. After each parallel directive, every thread is executing the same code as the master thread. The parallel directive specifies a number of items for the compiler. These can include the number of threads to use in the parallel region, a list of private and shared variables available to each thread, and even reduction operations on specified variables. At the end of the parallel region, the slave threads disappear leaving only the master thread to continue execution.

The most common use for OpenMP is on loop-level parallelism. In Table 4.1, we see a code sample of how to compute the value of Pi by summing the area under a curve.

One may wonder when looking at the code example in Table 4.1 as to the number of threads created when executing this example. OpenMP provides several environ-

```
#include <omp.h>
static long num_steps = 100000;
double step;


void main() {
  int i;
   double x, pi, sum = 0.0;
   step = 1.0/(double) num_steps;


   #pragma omp parallel for reduction(+:sum) private(x)
   for (i=1; i<=num_steps; i++) {
      x = (i-0.5)*step;
      sum += 4.0/(1.0+x*x);
   }


   pi = step * sum;
}
```

Tab. 4.1: OpenMP example of calculating Pi by integration

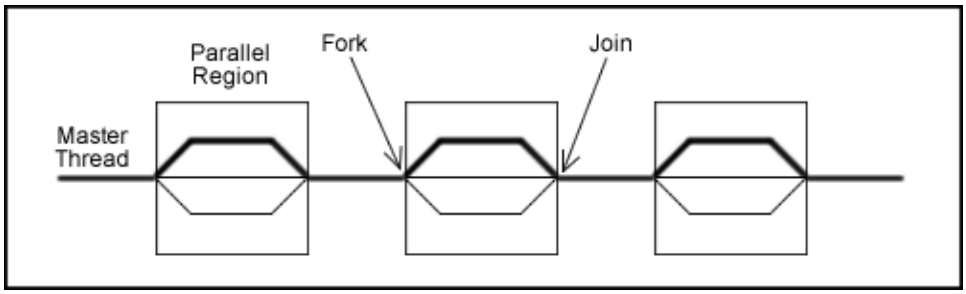

Fig. 4.7: OpenMP Parallel Region/Fork-Join Model

ment variables that can be used to control the behavior of an OpenMP program. An especially important environment variable is `OMP_NUM_THREADS`,which specifies the number of threads (including the master thread) to be used in its parallel regions. `OMP_DYNAMIC` is another environmental variable which dynamically adjusts the number of threads at runtime depending on the underlying implementation. The general rule of thumb is to make the number of threads no larger than the total number of cores in the system.

To use OpenMP, you must have an OpenMP-enabled compiler. However, if your compiler does not support OpenMP, the directives simply compile out. A technique often used to assess speedup is in the use of a compiler switch to enable/disable OpenMP, making the code execute in parallel/non-parallel fashion.

OpenMP is a favorable option when implementing parallel programming because of its simplicity. By having a small set of directives, the introduction of OpenMP usually does not change the program semantics. This allows a developer to prototype possible threading implementations without investing a large amount of time and effort. After an implementation with OpenMP is complete, the programmer can even rewrite the code using native threading APIs for more threading control. Regardless of what

implementation is finally used, OpenMP provides a quick way to add parallelism to your existing code.

Programming Language APIs

Programming Language APIs, such as those for C# and Java, are another method of adding threading to your application. The use of language threading APIs can save you time by hiding the complexity of thread management and also provide powerful threading support for your application. Many of these APIs have support for thread pools and synchronization objects such as monitors, mutexes, and semaphores. Optimizing compilers are another way to add parallelism to your code, specifically ones that offer automatic parallelization features. This feature analyzes loops and creates threaded code for loops which it can determine to be safe and beneficial for parallelization. In the case where a compiler cannot automatically parallelize a loop, it can provide a report of why it could not, which a developer could then analyze and identify regions for manual threading.

Internally Threaded Libraries

Finally, another method for implementing parallelism in your code is to use internally-threaded runtime libraries for common tasks. Intel offers two products, the Integrated Performance Primitives and the Math Kernel Library which aids in solving complex problems such as linear algebra, fast Fourier transform (FFT), and solving large equations. AMD also offers similar libraries like their AMD Performance Library (APL) which is a collection of low level routines ranging from simple arithmetic to

signal processing. AMD also offers an optimized math library, namely the AMD Core Math Library. These libraries transparently hide threading details and are written to immediately take advantage of multi-core hardware.

4.4.3 Ensuring Correctness

Once an application has been implemented using a threading methodology, assuring correctness of the application becomes important. One must verify that the addition of the non-deterministic execution through threading does not alter the expected behavior of the program. Let us explore a number of issues that may occur as a result of adding multi-threading to an application.

Race Conditions

A race condition occurs when two or more threads attempt to access the same resource at once. Because of the non-deterministic execution of the threads, it is impossible to determine which thread will access the resource first. This can lead to inconsistent program results.

The following example illustrates a read/write race condition. Suppose you have two threads, each with access to a shared variable x, which has the initial value of 1. Variables a and b hold the values of 1 and 2 respectively after running their large tasks. Table 4.2 shows the code for this example.

Depending on how these instructions are executed in a multi-threaded program, the value for x will vary. If the large tasks for Thread 1 and Thread 2 widely differ in their execution time, a race condition is less likely to occur, however, you cannot

Thread 1 Code	Thread 2 Code
a = LargeTask()	b = LargeTask()
$x = x + a$	$x = x + b$

Tab. 4.2: Example of a race condition

make any assumptions here and doing so could lead to incorrect results. Table 4.3 shows the differing values for x when the operations of both threads are executed in different orders.

The same out-of-order execution concept applies to data races for write/read and write/write conditions as well. To eliminate race conditions, the program must yield the correct result regardless of the interleaving of instructions between threads. It is the job of the programmer to identify all shared objects and protect them with the proper synchronization mechanisms to ensure the correct order of execution.

The problem with race conditions is that they are at some times difficult to detect. Running the program one-thousand times may yield the same result, yet after only one more execution, the program produces a different result. Luckily, there are software tools which can analyze code and detect race conditions in your threaded code. Even more impressive, these race conditions errors do not even have to occur for these tools to detect them [6]. Consequently, these tools can ease the burden of debugging a multi-threaded program significantly.

Values for x	Condition
4	Thread 1 completes then Thread 2 completes, or vice versa
2	Thread 1 reads x, then Thread 2 reads x and writes (1 + 2) to x. Thread 1 should receive the latest value of x written by Thread 2 (which is 3), but instead it has an invalid value (1) and does its sum anyways.
3	Thread 2 reads x, then Thread 1 reads x and writes (1 + 1) to x. Thread 2 should receive the latest value of x written by Thread 1 (which is 2), but instead it has an invalid value (1) and does its sum anyways.

Tab. 4.3: Race condition cases for possible values for x

Synchronization Mechanisms

So how can we prevent race conditions? One method is to use synchronization mechanisms. A critical section is a construct used to denote a block of code where only one thread is allowed to execute that section of code at any time. This ensures that threads access resources inside the critical section in a more organized fashion. Synchronization is a useful technique, but as mentioned previously, you should try to reduce the amount of synchronization as it will add more serialization and degrade performance of your application. While only one thread is allowed inside the critical section, other threads needing to access the shared resources are forced to wait.

Deadlock and Thread Stalls

Although extremely rare, there are certain instances where a multi-threaded program will hang for no apparent reason. If no programming errors are present, this behavior could be attributed to a deadlock. Deadlock occurs when one or more threads are waiting for exclusive access to a resource which will never be released. In order for deadlock to occur, the following criteria must exist in the system [22].

- Exclusive Access: Processes request exclusive access to resources.
- Wait While Hold: Processes hold previously acquired resources while waiting for additional resources.
- No Preemption: A resource cannot be preempted from a process without aborting the process.
- Circular Wait: There exists a set of blocked processes involved in a circular wait.

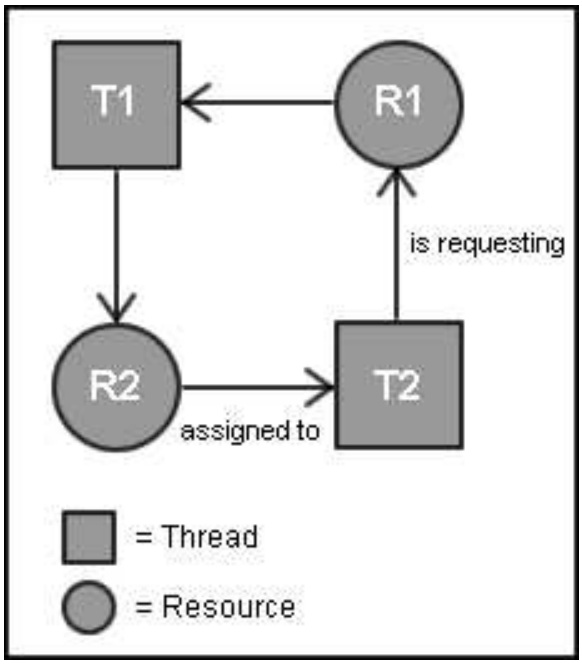

Fig. 4.8: A Simple Deadlock Example

The simplest case of deadlock is illustrated in Figure 4.8. In this instance, Thread 1 (T1) currently has exclusive access to Resource 1 (R1), and is requesting access to Resource 2 (R2) currently held by Thread 2 (T2). Conversely, Thread 2 has exclusive access to Resource B and is requesting access to Resource A currently held by Thread 1. Since neither process relinquishes the hold on its resource, both threads wait forever, and deadlock has occurred. In order to prevent deadlock, one has to be mindful of how the application threads gain and release access to shared resources.

Deadlock has a very low potential for occurrence and will only happen under the right conditions, but it can happen. Another possible explanation of why a multi-threaded application hangs during execution could be due to a thread stall. A thread stall occurs when a thread is waiting on resource owned by a thread where that thread

has already been destroyed. Since the desired resource has a "dangling lock", it will never be released and any threads waiting to acquire it will stall or wait indefinitely. The solution to this is to understand that the resource locks held by a thread are not automatically released when the thread is destroyed. Resource locks must be released within the same thread that obtained them.

4.4.4 Analyzing Performance

Amdahl's Law

We can estimate the expected performance of a parallel program in terms of speedup. Amdahl's Law quantifies the potential speedup from converting serial code to parallel code. Let s be the fraction of code that is inherently serial and cannot be parallelized, while p will represent the fraction of code that can be converted to parallel. Hence, $s + p = 1$. If we use N as the number of processors, we arrive at Amdahl's Law as depicted in Equation 4.1.

$$\frac{1}{s + (p/N)} \tag{4.1}$$

We can observe that with an infinite number of processors, the term p/N will eventually drop out. Thus, the maximum speedup one can achieve by parallelizing a program is the inverse of the fraction of the code that must run in serial. For example, if 20 percent of your code is serial, then you could expect, at most, a speedup of 5. If you are concerned with increasing performance, you should then pay close attention to minimizing the fraction of code that runs in serial.

One interesting note on Amdahl's Law to mention is that as the problem size

grows, p may rise and s may fall. Consider a Monte Carlo simulation which has a large number of iterations. The serial portion of setting up the data and I/O overhead, s, remains the same as p grows and s declines. Furthermore, Amdahl's Law is the best case. Adding processors will eventually lead to diminishing returns, and the equation does not take into account factors such as thread management and coordination. Whatever speedup you achieve will most likely be less than the value provided by Amdahl's Law.

Contrary to what one might expect, it is quite possible for a program to perform more slowly after having been multi-threaded. There are a number of factors which can influence this outcome, some harder to pinpoint then others. Excessive use of shared data can lead to synchronization issues where thread contention is high resulting in long thread wait times. A large amount of locking granularity lends itself to parallel overhead dominating the application and too little granularity may not parallelize enough work to make threading worthwhile. As mentioned previously, good load balancing can sometimes be difficult to achieve as improper distribution of parallel work can affect performance.

Alternatively, instead of performing worse that its single-threaded version, an application may exhibit only a minimal performance boost from threading. This may be due to an issue concerning poor scaling. A common issue resulting in poor scaling is the existence of large sections of serial code that dominate execution as more processors are added. Sometimes, portions of serialized code are not identified as candidate areas for parallelization, and thus the entire application suffers from potential parallel work lost in a multi-core environment. An often overlooked cause of poor scaling is

an application that has exceeded the memory bandwidth of the system or is suffering from memory-related issues, such as false sharing described earlier.

A recommended practice to diagnose scaling problems is to schedule periodic scaling studies using different processor configurations [5]. Performing tests using twice the number of available processors on the common consumer system will help you prepare and stay ahead of your customers.

5. MULTI-THREADED APPLICATION DEVELOPMENT

Now that we are familiar with multi-core processors and understand how multi-threading helps to take full advantage of this new hardware, we will be better prepared to use it effectively when developing applications. Let us examine a simple application using single and multiple threads.

5.1 Card Shuffling Example

In this example program, we will simulate the creation and shuffling of multiple card decks. See Appendix A for the full C# source code listing. This program has the ability to be executed in either a single-threaded or multi-threaded mode, which is denoted by the first command-line parameter passed to the program. The second command-line parameter is the number of times to shuffle each deck. For this example, we will shuffle our decks an outlandish amount of times so that we can observe activity on the cores. The last command-line parameter tells the program how many decks to create. Therefore, the syntax for executing this program is:

CardShuffle.exe [0 or 1] ShufAmt NumDecks

When the program is executed in single-threaded mode it uses its main thread to create the number of decks specified, then it shuffles each deck the number of times specified by the command-line parameter. Figure 5.1 shows the CPU utilization when

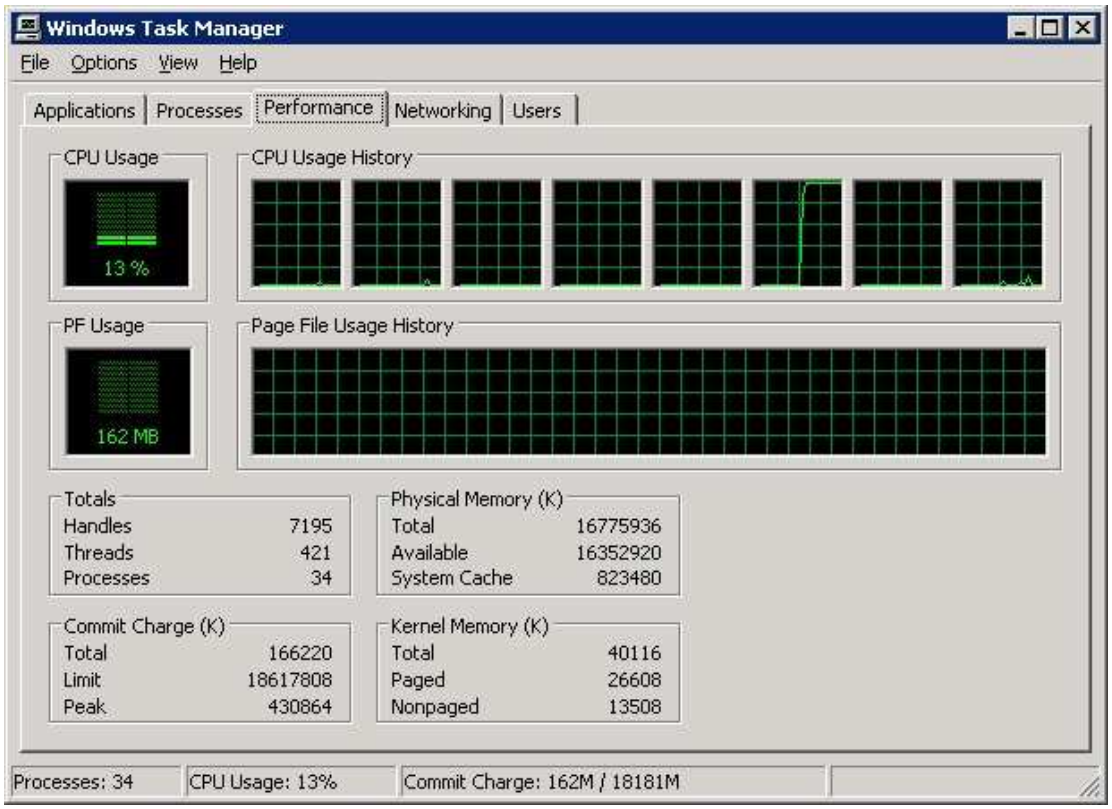

Fig. 5.1: CPU Utilization of Card Shuffle Program in Single-Threaded Mode

the program is running in single-threaded mode. We can clearly see that the program is utilizing only one of the available eight cores of the system, namely core six.

If we instruct the program to run in its multi-threaded mode, it creates an equal number of decks and threads as there are cores in the system. It assigns one deck per thread and then executes the Shuffle() method on each deck. For example, if we run the program using the command *"CardShuffle.exe 1 10000000 8"*, it will create eight decks, eight threads, then start each thread to shuffle its own deck ten million times.

If we inspect the CPU utilization now, we observe that the program is fully utilizing all eight cores and is thus doing its work in parallel.

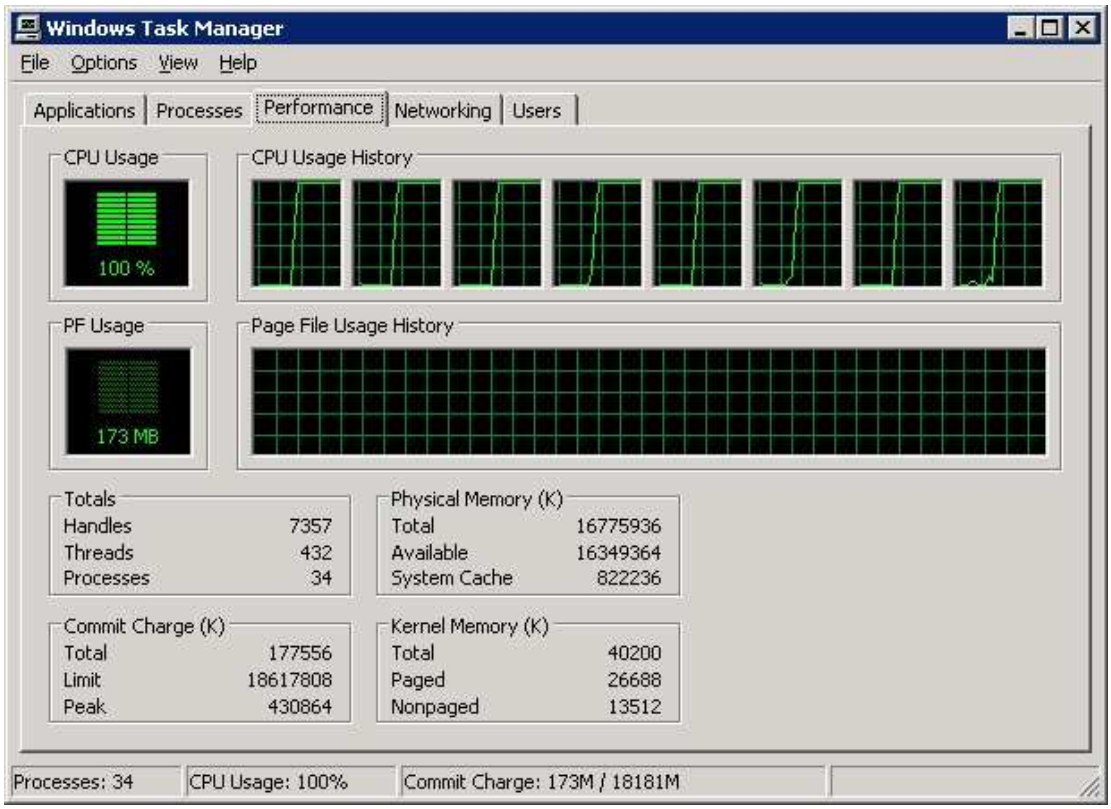

Fig. 5.2: CPU Utilization of Card Shuffle Program in Multi-Threaded Mode

To even further observe parallel behavior by the program, we can inspect the output from each thread to the console. Table 5.1 shows the typical output from the program running in single-threaded mode. Notice how each deck is printed in sequential order; this tells us that only a single thread is doing the work.

However, a multi-threaded run gives us much more interesting output. Because of the non-deterministic behavior of threading we see that the output to the console is dynamic because each thread writes to the console at different points in time. The solution to the interleaving of output statements could be handled simply by synchronizing access to the console using standard conventions. Table 5.2 shows an example console output from a multi-threaded run of the program.

I was fortunate to be given access to a variety of multi-core machines with differing configurations. See Appendix B for a list of system configurations used in this study. The following tables show data from executing the card shuffling program on those different systems.

By reviewing the data, one can determine the benefit of having more processing cores. System A, a dual-core machine, had an average of 50 percent improvement gain while running in multi-threaded mode. System B, an eight-core machine using AMD's Opteron processor, saw an average of 87 percent performance gain. System C, an eight-core machine using Intel's Xeon processor, saw a similar average of 86 percent performance gain. It is worthwhile to note that even while quadrupling the number of processing cores from System A to Systems B and C, we did not observe a linear performance gain. This is likely the existence of diminishing returns at work.

```
Using single-threaded shuffle (3 decks)...
Player 0's hand...
Player 0 0: 7 of Spades
Player 0 1: Ace of Clubs
Player 0 2: 5 of Spades
Player 0 3: 10 of Clubs
Player 0 4: Jack of Clubs

Player 1's hand...
Player 1 0: 10 of Hearts
Player 1 1: 4 of Hearts
Player 1 2: Queen of Clubs
Player 1 3: King of Spades
Player 1 4: Ace of Clubs

Player 2's hand...
Player 2 0: 9 of Clubs
Player 2 1: 2 of Spades
Player 2 2: 3 of Diamonds
Player 2 3: 5 of Spades
Player 2 4: 10 of Hearts

Process completed in 00:00:00.3042325 seconds.
```

Tab. 5.1: Single-threaded card shuffle console output

```
Using multi-threaded shuffle (3 decks)...
Player 1's hand...
Player 0's hand...
Player 0 0: 4 of Clubs
Player 0 1: King of Hearts
Player 0 2: King of Clubs
Player 0 3: 2 of Hearts
Player 0 4: 4 of Diamonds

Player 1 0: 4 of Clubs
Player 1 1: King of Hearts
Player 1 2: King of Clubs
Player 2's hand...
Player 2 0: 4 of Clubs
Player 2 1: King of Hearts
Player 2 2: King of Clubs
Player 2 3: 2 of Hearts
Player 2 4: 4 of Diamonds

Player 1 3: 2 of Hearts
Player 1 4: 4 of Diamonds

Process completed in 00:00:00.1125793 seconds.
```

Tab. 5.2: Multi-threaded card shuffle console output

Run	Multi-Threaded	Decks	Shuffle Amount	Avg. Exec Time(s)	MT Improvement
1	No	2	10^7	2.3112845	-
2	Yes	2	10^7	1.5468929	33.1%
3	No	2	10^8	23.0747055	-
4	Yes	2	10^8	11.6918898	49.3%
5	No	2	10^9	231.0912208	-
6	Yes	2	10^9	117.43444253	49.2%

Tab. 5.3: Execution data from card shuffle program running on System A

Run	Multi-Threaded	Decks	Shuffle Amt.	Avg. Exec Time(s)	MT Improvement
1	No	8	10^7	8.5817276	-
2	Yes	8	10^7	1.2418062	85.6%
3	No	8	10^8	84.9485922	-
4	Yes	8	10^8	10.8091182	87.3%
5	No	8	10^9	848.9395972	-
6	Yes	8	10^9	108.6855996	87.2%

Tab. 5.4: Execution data from card shuffle program running on System B

Run	Multi-Threaded	Decks	Shuffle Amt.	Avg. Exec Time(s)	MT Improvement
1	No	8	10^7	7.6190219	-
2	Yes	8	10^7	1.0773939	85.6%
3	No	8	10^8	77.7749434	-
4	Yes	8	10^8	11.1424368	85.7%
5	No	8	10^9	765.6089332	-
6	Yes	8	10^9	97.8403878	87.2%

Tab. 5.5: Execution data from card shuffle program running on System C

6. THE FUTURE OF COMPUTING WITH MULTI-CORE PROCESSORS

6.1 The Multi-Core Roadmap

It seems evident that the single processor days are over and multi-core architecture is here to stay. Intel forecasts that more than 85 percent of its server processors and more than 70 percent of its mobile and desktop processors will be dual-core by the end of 2006 [21]. At the Spring 2005 Intel Developer Forum in San Francisco, Intel senior fellow and CTO director Justin Rattner spoke of the company's goal to deliver chips with one-hundred or more processing cores by the year 2015 [17].

Intel is also planning to take multi-core to the next level by implementing specialized cores for classes of computation such as graphics, artificial intelligence, speech and handwriting recognition, image processing, and even communication protocol processing [13]. In an interesting question and answer interview with Jerry Bautista, who leads the Intel Microprocessor Lab, Bautista speculates that with enough cores, the graphics processing commonly done by expensive GPUs will eventually be pulled back onto the CPU [24]. Perhaps this is the same thinking from microprocessor competitor Advanced Micro Devices (AMD) as evident in their October 2006 acquisition of graphics chipset manufacturer, ATI Technologies. AMD has also announced its commitment to multi-core architecture progression in its line of server, desktop, and mobile processors. Peter Buhr, a professor of computer science at the University of

Waterloo, states, "You won't be able to buy a computer in five years that doesn't have a dual-core processor" [4]. If this is indeed the case, parallel computing will truly become more mainstream then ever before. Instead of needing expensive hardware or access to a large number of networked machines, every consumer-level processor will be able to perform parallel computation.

6.2 Industry Adoption

Already, the software industry is preparing for the multi-core era by adapting multi-threading into their applications and reaping the benefits. In the future, users can expect more performance and responsive applications which take advantage of multi-core processors. Companies are noticing the potential for parallel processing and are eager to use multi-core to gain a competitive edge for their products. From digital content creation applications to computer games, multi-core is applicable to virtually all industries.

6.2.1 Digital Content Creation

Pixar Animation Studios has already been enjoying the benefits of multi-core processing as they recently multi-threaded their award-winning RenderMan rendering software. Since its inception in 1984, Pixar has always been looking for ways to improve frame rendering which can speed up the film production cycle and results in greater detailed scenes and realistic imagery. Adding more scene objects and detail requires more computing power, but Pixar was reaching the limits of its data center power and cooling capacity. A solution was needed to obtain more processing power

from the same amount of real estate. The solution was to choose multi-core and add multi-threading to RenderMan.

RenderMan was originally designed to manage many independent servers operating in parallel using its integrated network rendering dispatcher. However, it did not incorporate multi-threading, which was needed to take full advantage of their multi-core processors. To guide them in this herculean effort, the RenderMan team was educated on threading concepts and trained to use software development tools from Intel. One developer noted that Intel's Thread Checker tool helped him to find race conditions in the threaded code which could have led to weeks of analysis to uncover [10].

Ultimately, the RenderMan team achieved their goal of a 75 percent improvement when scaling up to four physical processors. RenderMan could now render up to five times faster in threaded mode on a system with four dual-core processors than when rendered in a non-threaded mode on a single processor system. Also, the team found that two cores working from the same memory rendered at nearly the same speed as two physical processors with distinct memories. According to Dana Batali, director of RenderMan development, by adding threading to RenderMan, Pixar was able to pack more processing power capacity into their render farm while lowering their RAM costs as well [10]. Batali noted that the threading expertise and software tools from Intel were "extremely helpful" and essential for creating a stable, reliable multi-threaded application [10].

Some applications were already posed to give immediate benefits with multi-core processors to its users without the need for code changes. For over a decade, Adobe Systems has been developing its line of video processing software, such as its Adobe Premiere Pro line, for use with multi-processor systems [8]. This product uses threads to process and render video frames for faster, even real-time, data manipulation. By already utilizing threading in their design, the application currently supports multi-core processors in the same way it already supports multiple processor systems. What this means for the user is increased performance without the need for expensive hardware systems or a software upgrade.

6.2.2 Computer Game Development

Multi-core processors even have benefits for the digital entertainment industry, bringing larger and richer experiences to gamers worldwide. One area in which multi-core processing is already being used is for asynchronous background loading [11]. In modern computer games, a "zone" is a section of the in-game universe which can be explored by the player. As the player travels to different zones the player must wait for different zones to be loaded into memory and rendered. Asynchronous background loading allows the machine to pre-load nearby zones so as the player approaches them, the transition is seamless. What this could mean is the end of the dreaded "Loading" screen for computer gaming.

Mark Rein, vice president at Epic Games is expecting big changes in the gaming industry. "When the Intel folks first told us that they were taking a multi-core approach, we cheered and clapped" [11]. Epic Games is already thinking ahead to

the time when multi-core is more pervasive and how they can leverage that extra processing power in the form of user-to-user communication. Rein suggests multi-core will eventually allow for integration of real-time high-quality video in the game itself.

Multi-core processors will allow game developers to effectively separate tasks between the available cores. For instance, a photorealistic rendering algorithm could use multiple cores for on-the-fly graphics. Expensive in-game physics calculations could be dedicated to one core while artificial intelligence could be processed on another core creating a more rich and immersive experience. As the gaming industry is known for pushing the boundaries of hardware, it will be interesting to see the innovations made possible by multi-core processors and how it will affect this ever-changing field of computing.

6.3 Academic Acknowledgement

By recognizing the effect that multi-core will have on parallel computing potential, education institutions are already adapting their courses to provide students with the knowledge and skills necessary to succeed as computer scientist, system architects, and programmers. The College of Computing (CoC) at the Georgia Institute of Technology is starting to re-emphasize concurrency and parallelism to their students in an effort to teach these important concepts. Over the next two years, Georgia Tech expects to upgrade their core curriculum to convey the principles of multi-core processing and the techniques needed to take advantage of the architecture [9]. Professor Karsten Schwan at the CoC says, "we have to start educating and thinking

in terms of parallel" [9].

Through their efforts, Georgia Tech will be helping to educate and train the next generation of engineers and architects who are ready for the parallel age of computing. In light of this, I anticipate other academic institutions to follow Georgia Tech's lead and adapt their undergraduate, postgraduate, and doctoral programs to have an increased emphasis on multi-core and parallel computing concepts.

6.4 The Next Revolution in Software: Concurrency

Every so often a technology comes around and shakes the software world to its foundation. Multi-core architecture and parallel computing enabled by this technology is such a change. Analysts and experts are predicting the move to multi-core will be as profound as the object-oriented paradigm shift of the 80s and 90s. Who knows, they may be correct. If they are, software developers will likely be motivated to re-think application design and implementation to support concurrency.

6.4.1 The "Free Lunch" Is Over

For years, applications gained a performance "free lunch" as better and faster processors were released to the market. Improvements in clock speed, cache, and execution optimizations such as pipelining, branch prediction and ILP, made applications run faster without any code changes. It is interesting to note that a surprising eighty percent of performance gained was simply due to the increased processor clock rates [13]. Unfortunately, this trend will not continue. Due to the inherent problems with heat consumption and current leakage that accompany clock speed increases, only

advances in the area of cache will continue to deliver performance benefits for the time being. It seems as though the "speed wars" are over and most microprocessor vendors are pursuing the multi-core direction.

6.4.2 Paradigm Shift for Software Architects and Developers

So how then can we as software developers best utilize the performance improvements available in multi-core processors? The answer, as you may have guessed by now, lies in software multi-threading. In the future, it will be advantageous for software engineers and designers to embrace and introduce multi-threaded designs into their applications to truly benefit from TLP offered by multi-core processors. We'll still use the traditional "divide and conquer" methodology, but we'll need to adopt the "work smarter, not harder" mindset as well.

Unfortunately, as I have found from my research, only a small fraction of developers have been doing concurrent programming regularly. Multi-threading is considered by most programmers to be an advanced topic only useful if the application specifications require it. The vast majority of applications today are single-threaded, most likely due to the fact that the majority of target platforms did not have parallel hardware.

This fact is going to change; and change rapidly. Here is an example of the rate of change we can expect. When I started my Masters Thesis research in July of 2006, most major microprocessor manufactures had dual-core processors on the market. At the time of this writing, only six months later, quad-core processors are already available for purchase. An interesting fact comes from IDC, a market researcher specializing in the information technology and telecommunications industries. IDC

makes an interesting claim for the server market and says that the future growth of this market can be mapped directly to the similar growth we have seen in the storage industry for the past five years [20]. As the number of cores per processor increases, IDC predicts the power of the server market will increase at an exponential rate.

Like any major technology change, it may take some time for the majority of software developers to understand parallel programming concepts and have their applications take full advantage of multi-core processors. For example, object-orientation of code did not become mainstream until 30 years after its initial use with the SIMULA language back in the mid 1960s [23]. Object-oriented concepts and principles ultimately became popular years after their debut simply due to application requirements needing to consist of larger and larger systems. The powerful abstraction offered by object-orientation became a natural fit and thus it began to pick up adoption by the industry. A lack of motivation prevented object-orientation of code to hit the mainstream, until there was a need.

Although parallel programming and object-orientation concepts may have a similar complexity and learning curve, I estimate parallel programming will not take as long to permeate into the software industry. Why do I believe this? Because the need for more performance will always be present, and as user expectation rises through the years as it has in the past, parallel programming will be the answer in utilizing the new hardware given to us through multi-core architecture.

6.4.3 Stronger Reliance on Software Analysis Tools

Due to the inherent difficulty of parallel programming, I anticipate, as it becomes a more prevalent part of software development, we will see a stronger reliance for software analysis tools than ever before. These tools will provide helpful insight into the use of parallel constructs and will aid developers to find and fix bugs before they are introduced into production software. Analysis tools will help to remove the guess work involved in locating candidate code for parallelism, striving for appropriate load balancing, and minimizing issues revolved around parallel programming such as synchronization and efficient cache usage.

If we as software developers understand the new architectural changes brought about by multi-core processors, we can begin to take advantage of parallelism and create the next generation of applications. These future applications will need to use threading to fully exploit the architectural innovations from multi-core processors. A single threaded application on a 100-core processor could be utilizing only 1/100th of the total processing throughput of the machine. The inherent benefits are too large to ignore.

7. CONCLUSION

It seems that the future of microprocessor architecture has multi-core written all over it. The major vendors are already shipping their dual and quad-core processors with promise of many more cores in the coming years. Although it seems that clock rates are stabilizing, transistor counts should continue to explode and it seems that CPUs will exhibit Moore's Law-like throughput gains for some years to come.

In this paper, we explored the current state of microprocessors today and showed how multi-core is the natural progression of this technology. We defined exactly what multi-core means, and the benefits we can obtain by utilizing it. We then explored the topic of multi-threading and learned that threading is the single greatest method for maximizing use of a multi-core processor. Concepts surrounding threaded development were discussed which are important to keep in mind while implementing a threaded solution. We then learned how the move to multi-core design in processor architecture will affect software developers.

7.1 Preparing for the Future

Many are expecting multi-core to bring a huge impact to the software industry. Herb Sutter of Microsoft says, "[Multi-core processing is creating]... the biggest sea change in software development since the object-oriented programming revolution..." [12].

James Reinders, director of marketing and business of Intel's Software Development Products Division has been quoted as saying, "It strikes me that in terms of future development, the magnitude of the change that software developers are going to experience will be substantial" [12].

Whether they like it or not, parallelism has gone mainstream and software developers need to keep their threading skills current and be prepared. We should utilize this time now to get up to speed on our threading skills. No doubt, engineers who possess the skills necessary to analyze, design, and debug multi-threaded applications will be in high demand in the foreseeable future.

No matter the change, it is surely an exciting time to be a software developer. For the first time in history, parallelism, for everyone, is going to be ubiquitous. The level of performance we can expect from our applications will continue to soar. Even so, software vendors are already putting multi-core processors to work as we saw. With this age of parallelism, new applications will surface, and new methods of solving current problems will emerge. I believe this is a wonderful time for opportunity.

But it won't be an easy change to cope with. Software developers will need to rethink how they design their applications to take advantage of threading; the real power behind multi-core processors. Training and experience will be crucial to ensure that correctness and efficiency are maintained in the implementation of multi-threaded applications. If we know and understand the hardware, we will be more prepared to effectively utilize all the benefits that multi-core processors can provide.

Parallel computing has finally hit the mainstream. From the hardware engineer to the end-user, multi-core will reach the masses and affect everyone. The time to adapt to all that multi-core brings is now because this "wave of parallelism" isn't on the way, it's already here.

APPENDIX A

CARD SHUFFLING EXAMPLE SOURCE CODE

```
// ******************************************
// File: Main.cs
// Single/Multi-Threaded Card Shuffling Example
//   usage: CardShuffle.exe [0|1] #1 #2")
//   0 = Single Threaded Example (required)
//   1 = Multi-Threaded Example
//   #1 = Shuffle Iterations     (required)
//   #2 = Number of Threads      (optional)
// ******************************************

using System;
using System.Collections.Generic;
using System.Text;
using System.Threading;
using System.Diagnostics;
using System.Collections;

namespace CardShuffle {
  class MainClass {
    public static void Main(string[] args) {
      double ShufAmt;
      ArrayList myDecks, myThreads;
```

```
Deck deck;
Thread myShuffle;
int numDecks, useSingleThreading;

if (args.Length < 2)
  Environment.Exit(0);
else {
  useSingleThreading = Convert.ToInt32(args[0]);
  ShufAmt = Convert.ToDouble(args[1]);

  if (args.Length == 3)
    numDecks = Convert.ToInt32(args[2]);
  else
    numDecks = System.Environment.ProcessorCount;

  if (useSingleThreading == 0) {
    // Single-Threaded Version
    Console.WriteLine("Using single-threaded shuffle
      (" + numDecks + " decks)...");

    Stopwatch watch = new Stopwatch();
    watch.Start();
```

```
  deck = new Deck("Player 1", ShufAmt);

  for (int i = 0; i < numDecks; i++)
    deck.Shuffle();

  watch.Stop();       // Stop the clock
  Console.WriteLine("Process completed in " +
      watch.Elapsed.ToString() + " seconds.");
  }
else {
  // Multi-Threaded Version
  Console.WriteLine("Using multi-threaded shuffle
    (" + numDecks + " decks)...");

  Stopwatch watch = new Stopwatch();
  watch.Start();

  myDecks = new ArrayList();
  myThreads = new ArrayList();

  // Create our decks and threads
  for (int i = 0; i < numDecks; i++) {
    deck = new Deck("Player " + i, ShufAmt);
```

```
            myDecks.Add(deck);

            myShuffle = new Thread(new ThreadStart(
                        (myDecks[i] as Deck).Shuffle));
            myThreads.Add(myShuffle);
          }

          // Start all the threads
          for (int i = 0; i < numDecks; i++)
            (myThreads[i] as Thread).Start();

          // Wait for all threads to complete
          for (int i = 0; i < numDecks; i++)
            (myThreads[i] as Thread).Join();

            watch.Stop();       // Stop the clock
            Console.WriteLine("Process completed in " +
            watch.Elapsed.ToString() + " seconds.");
        }
      }
    }
  }
}
```

```
// File: Deck.cs

using System;
using System.Collections.Generic;
using System.Text;

namespace CardShuffle {
  public class Deck {
    private string[] oCards;
    private string oDeckOwner;
    private double oShufAmt;

    public Deck(string deckOwner, double ShufAmt) {
      oCards = new string[52];
      oShufAmt = ShufAmt;
      oDeckOwner = deckOwner;
      int thisCard;
      int cnt = 0;
      for (int suit=0; suit < 4; suit++) {
        for (int num=0; num < 13; num++) {
          cnt = (suit * 13) + num;

            // Which card are we on?
```

```
switch(num) {
  case 0:
    oCards[cnt] = "Ace";
    break;
  case 10:
    oCards[cnt] = "Jack";
    break;
  case 11:
    oCards[cnt] = "Queen";
    break;
  case 12:
    oCards[cnt] = "King";
    break;
  default:
    thisCard = num + 1;
    oCards[cnt] = thisCard.ToString();
    break;
}

// Append the suit
switch(suit) {
  case 0:
    oCards[cnt] += " of Spades";
```

```
            break;
          case 1:
            oCards[cnt] += " of Hearts";
            break;
          case 2:
            oCards[cnt] += " of Clubs";
            break;
          case 3:
            oCards[cnt] += " of Diamonds";
            break;
        }
      }
    }
}

public void Shuffle() {
  int card1, card2;
  string tmpCard;
  Random RandomClass = new Random();
  for (int i=0; i < oShufAmt; i++) {
    card1 = RandomClass.Next(0,52);
    card2 = RandomClass.Next(0,52);
    tmpCard = oCards[card1];
```

```
      oCards[card1] = oCards[card2];
      oCards[card2] = tmpCard;
    }

    Console.WriteLine(oDeckOwner + "'s hand...");
    Print(5);
    Console.WriteLine();
    }

    public void Print(int numCardsFromStart) {
      for (int i=0; i < numCardsFromStart; i++)
        Console.WriteLine(oDeckOwner + " " +
        i.ToString() + ": " + oCards[i]);
    }
  }
}
```

APPENDIX B

MULTI-CORE SYSTEM CONFIGURATIONS

- SYSTEM A

  PROCESSOR(S): Intel Core 2 Duo

  L2 CACHE: 2MB

  FSB: 1066 MHz

  MEMORY: 1 GB

  OPERATING SYSTEM: Windows XP Professional w/ SP2

- SYSTEM B

  PROCESSOR(S): Dual-Core AMD Opteron 2.6 GHz x 4

  L2 CACHE: 4MB x 2

  FSB: HyperTransport Technology (1GHz true, 2GHz effective)

  MEMORY: 16GB DDR400 ECC Registered

  OPERATING SYSTEM: Windows Server 2003

- SYSTEM C

  PROCESSOR(S): Quad-Core Intel Xeon 2.67 GHz x 2

  L2 CACHE: 4MB x 2

  FSB: 1066 MHz

  MEMORY: 4GB FBDIMM DDR2 533 MHz

  OPERATING SYSTEM: Windows Vista

REFERENCES

[1] Shameem Akhter and Jason Roberts. *Multi-Core Programming*. Intel Press, 2006.

[2] Avi Mendelson Simcha Gochman Rajshree Chabukswar Karthik Krishnan Arun Kumar Alon Naveh, Efraim Rotem. Power and thermal management in the intel core duo processor. *Intel Technology Journal*, 10(2), May 2006.

[3] AMD. Power and cooling in the data center. Technical report, AMD, 2005.

[4] Thomas Burger. Intel multi-core processors: Quick reference guide. webpage, Sep 2006. http://www.intel.com/cd/ids/developer/asmo-na/eng/231914.htm.

[5] Charles Congdon. Multithreaded programming quickstart. webcast, May 2006.

[6] Max Domeika and Lerie Kane. Optimization techniques for intel multi-core processors. webpage, Aug 2006. http://www.intel.com/cd/ids/developer/asmo-na/eng/261221.htm.

[7] John L. Hennessy and David A. Patterson. *Computer Architecture A Quantitative Approach*.

[8] Intel Software Insight. For adobe, foresight brings immediate benefit to users. *Intel Software Insight*, Aug 2006.

[9] Intel Software Insight. Geogia tech takes a parallel path. *Intel Software Insight*, Sep 2006.

[10] Intel Software Insight. The inside story from hollywood: How pixar multi-threaded renderman. *Intel Software Insight*, Aug 2006.

[11] Intel Software Insight. Multi-core changes the game for epic. *Intel Software Insight*, Aug 2006.

[12] Intel Software Insight. Sea change in the software world: Multi-core processing opens innovative business possibilities. *Intel Software Insight*, page 6, Sep 2006.

[13] Intel. Evolution of parallel computing. webpage, Oct 2006. http://www.intel.com/platforms/parallel.htm.

[14] Brent Kerby. Managing data center power and cooling with amd operton processors and amd powernow! technology. *Dell Power Solutions*, page 5, February 2007.

[15] Geoff Koch. Discovering multi-core: Extending the benefits of moore's law, July 2005. http://www.intel.com/technology/magazine/computing/multi-core-0705.pdf.

[16] Geoff Koch. Intel multi-core processor architecture: Faq. webpage, Sep 2006. http://www.intel.com/cd/ids/developer/asmo-na/eng/221188.htm.

[17] Geoff Koch. Intel's road to multi-core chip architecture. webpage, Sep 2006. http://www.intel.com/cd/ids/developer/asmo-na/eng/220997.htm.

[18] Geoff Koch. Transitioning to multi-core architecture. webpage, Sep 2006. http://www.intel.com/cd/ids/developer/asmo-na/eng/recent/221170.htm.

[19] Intel Software Network. Multi-core capability. webpage, Aug 2006. http://www.intel.com/cd/ids/developer/asmo-na/eng/235413.htm.

[20] Kelly Quinn and et al. The next evolution in enterprise computing: The convergence of multicore x86 processing and 64-bit operating systems. webpage, Aug 2006. http://multicore.amd.com/Resources/IDC_Convergence_en.pdf.

[21] R.M. Ramanathan. Intel multi-core processors leading the next digital revolution. webpage, Aug 2006. http://www.intel.com/cd/ids/developer/asmo-na/eng/strategy/trends/234550.htm.

[22] Mukesh Singhal and Nirajan G. Shivaratri. *Advanced Concepts in Operating Systems: Distributed, Database, and Multiprocessor Operating Systems.*

[23] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, March 2005.

[24] Computer Power User. Q&a with jerry bautista. *Computer Power User Magazine*, 6(9), Sep 2006.

[25] Ofri Wechsler. Inside intel core microarchitecture: Setting new standards for energy-efficient performance. Technical report, Intel, 2006.