

1 A first Step into Parallel Programming

Before we endeavor into cluster programming and scalable parallel architectures we will take a look at the simplest parallel architectures and how they are programmed. The problems that must be addressed with these shared memory machines must also be solved in all other parallel programming models so that one of the reasons why we take the time to learn these ‘non cluster’ architectures and programming techniques, is that we need to crawl before we can walk. Another reason why we look into the non-scalable architectures is that when you understand why we cannot scale these machines into hundreds of processors, you will better understand the choices that are made to make machines scalable and why you as the programmer must pay some of the price of the scalability you want, by increased complexity of the application code that you write.

1.1 Symmetrical Multiprocessor Architectures

The trivial computer contains one CPU and one memory block. If this one CPU runs a multithreaded application, the memory will appear as shared amongst the threads. More realistic versions of shared memory are several CPUs connected to the same memory block or memory blocks using either a shared bus or a crossbar switch. This is sometimes called Symmetrical Multi Processing, SMP, though SMP has come to mean the shared bus approach more specifically. Taken together the two are more often referred to as Uniform Memory Access, UMA, architectures. Uniform because access to a memory address has the same cost independently of the address. This uniform cost may be partly hidden by caching, but this in effect does not change the UMA property.

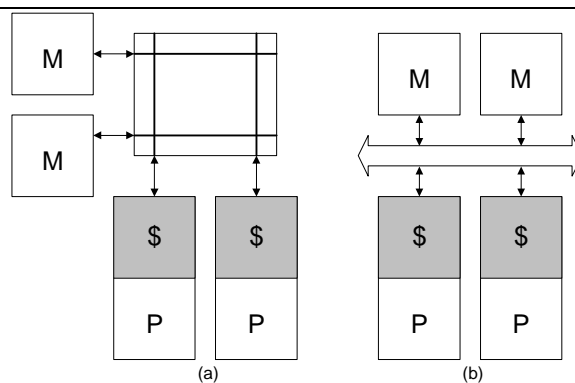


Figure 1 UMA approaches, (a) crossbar switch and (b) Shared bus

The crossbar switch approach can be found in machines such as the HP Exemplar V Class, while the shared bus approach can be found in machines such as the Intel x86 based multiprocessors, SGI Power Challenge and the Sun Enterprise Server. The latter may intuitively not look like an UMA machine, as processors and memory modules are collected on the same system boards, which then interface with the shared bus. Nonetheless this is a standard UMA architecture, since a CPU can access memory in a different system board in the same time it takes to access memory on its own system board.

The core mechanism behind the SMP architecture is the shared memory model, what we intend this model to mean is that if one processor writes a value to an address X , then another processor will receive the same value if it reads from address X . As obvious as this seems, there are many problems associated with achieving this with modern processors, which has come to depend heavily on caches for performance, usually two and some times even three levels of cache. Maintaining the shared memory behavior with multiple CPU's with each their cache is known as cache coherence or cache consistency. Maintaining cache coherence with the crossbar SMP approach is very complex and out of the scope of this text. Achieving coherence on a shared bus is simpler, but by no means trivial either. Since most SMP machines are shared bus machines, we'll go on to describe a coherence technique for these machines very briefly. For a thorough walkthrough of the cache coherence problems and their solutions, consult [Culler 98].

The most common coherence algorithm is the MESI protocol, which is also the one that is used in the Intel x86 architecture. The Modified-Exclusive-Shared-Invalid, MESI, protocol, allows any cache line to be in any of the four states: Modified, Exclusive, Shared and Invalid. A line in Modified state, contains information which is the only correct version of that data, this in turn means that if the cache line is flushed to make room for another entry, then the contents must be written to main memory. Exclusive cache lines are, as the name indicates an exclusive cache copy of the data, e.g. there still exist a coherent version in main memory, but no other CPU has a copy. A

cache line in Shared state contains information, which exists in more than one copy, e.g. two or more CPU's has the cache line in cache memory. An Invalid cache line is a cache line in which the information is no longer valid, e.g. the cached address has been changes by another CPU.

A Modified cache line can turn into a Shared if another CPU reads a copy of the data, or it can become Invalid if another CPU writes to the cache line. An Exclusive entry can enter the Shared or Invalid state in the same way, or it may become Modified if the CPU writes to the cache line. A Shared entry can become Modified if the CPU writes to it, or Invalid if another CPU writes to it. Invalid entries may become Shared or Exclusive on a read to them, Shared if the information is delivered by another cache and Exclusive if the information comes from main memory, if the cache line is written to it shifts state to Modified.

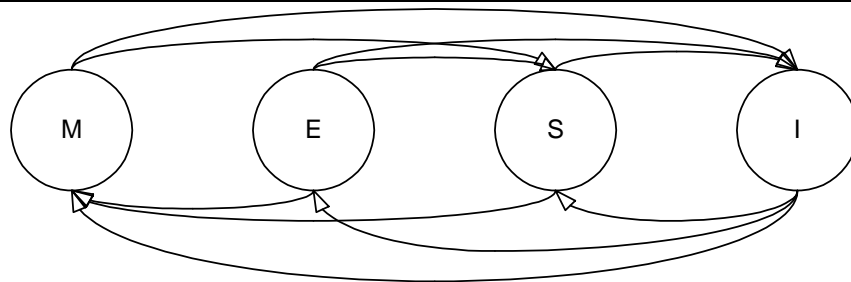


Figure 2 State Machine Transactions for the MESI protocol

Programming shared memory machines are often presented as trivial and as if linear speedup can be achieved without considering architectural details. This perception often results in a very disappointed programmer. There is a multitude of considerations to take, none of which are not part of optimizing sequential programs. If one does not consider these factors, even the simplest parallel applications will show poor performance. The following is a brief description of some of the most important factors to consider.

The operating system architecture is a very important factor with respect to performance. Some operating systems will stop all CPU's in a system when one CPU enters OS modus, this way the application is serialized every time any process accesses the OS, luckily these operating systems are rare today but if one writes an application for such an operating systems, minimizing system calls is imperative for performance. A more common OS flaw is that only one processor may be in kernel modus at a time, this way the other processes won't be stalled simply because one process enters the OS. A process will still be stalled if it tries to go to the kernel and another processor is already in kernel modus. Thus in this case it is important to consider this factor in an application and avoid that different parts of ones application tries to access the kernel at the same time. This is not as rare an event as it might sound, since many applications will naturally have a flow where they synchronize and the all attempt to access the file system at the same time. Other operating systems again will allow any number of processors to be in kernel modus at the same time, granted that they do not access the same data or for some OS that they do not use the same functions. The most important issue with these OS is usually that their Achilles heal is that they still need to interrupt the other processors when the do memory allocation, e.g. in general for any SMP machine it is better to allocate one large chunk of memory than to issue several small allocations.

Interrupts in general presents a problem when programming for performance, and more so when working with multiprocessors. Depending on the architecture and operating system most systems will deliver all interrupts to one processor only, if a parallel application runs a fine grained parallel job interrupting one processor will easily result in a similar sized stall of the other processor before a synchronization point. Imagine a dual processor that need to add two vectors before moving on to another part of the application, adding two vectors is a trivial parallel job, since one processor may work on the first half of the vector, while the other may add the latter part of the vectors. After completion the two processors synchronize before going on. If one processor is delayed x cycles due to interrupts then the other processors will have to wait x cycles before synchronization is possible, thus the actual loss of computation time is $2x$ rather than x , it is evident that scaling to more processors will increase this delay linearly. The cost of interrupts varies much with the CPU architecture, the Intel x86 architecture, which is widely used as in low-cost clusters has a very high interrupt cost, while other CPU architectures, e.g. the HP PA-RISC architecture has a direct interrupt cost of only a few clock-cycles. However there are indirect costs from interrupts, most importantly processing the interrupt will almost always introduce noise in the cache and TLB of the processing CPU. As most high performance applications are highly tuned to efficient cache and TLB utilization, this indirect cost easily outweighs the direct interrupt cost. Programming with interrupt interference in mind is hard, and not event possible at all times, however in some cases, like the vector adding example one might skew the workload slightly to the side of the processor that does not process interrupts. Some operating systems does not

provide the programmer with the means of identifying which processor a given process runs on, and in these cases one can do little to compensate for interrupt cost.

Unfortunately interrupts are by no means the only or even biggest threat to performance in SMP systems. Any resource that may become sparse during execution is a potential bottleneck. The most obvious such resource is the processor-memory, PM, bus. The PM bus of a Pentium III is at the time of writing a 64-bit wide bus running at no more than 133MHz, e.g. a theoretical bandwidth of just more than one giga-byte per second, reading an instruction requires two bus operations, e.g. a theoretical maximum around 500 MB/sec. A 1.5 GHz processor can theoretically issue $4.5 \cdot 10^9$ instructions per second. In other words, the PM bus cannot even keep one CPU busy in the worst-case scenario. Luckily caches eliminate these problems and CPUs are most often stalled due to fetching data and not instructions, the problem is the same here though, thus is a program traverses data structures that either does not exhibit temporal locality or are so large that they does not fit in the cache and thus does not allow the CPU to benefit from temporal locality. So if the PM bus can become a bottleneck for one CPU, then the risk of the PM bus becoming a bottleneck grows with the number of processors that are connected to the bus. The limited bandwidth of the PM bus is also the primary reason why SMP architectures only grow to a very limited number of processors.

The MESI protocol, or any other SMP bus coherence protocol, is also responsible for potential performance degradation. If one processor has modified a variable, e.g. increased a counter, and another processor wish to do the same, then the cache line, which is in Exclusive state, need to be copied to the reading CPU and enter the Shared state at both CPUs, then the second CPU may increase the variable and write the new value, which means that the cache line at the first CPU must be invalidated, e.g. enter the Invalid state. Migrating cache lines back and forth like this may easily reduce the performance of a system to a level where a uni-processor system would perform better. To ensure correct execution in cases as the one of the shared counter being increased by more than one processor, such variables are usually protected by synchronization variables, which increase the problem even further. This scenario is examined in a later example.

A related problem, which can be even harder to detect, is called false sharing. The problem is the cache line migration issue again, but this need not be because two different processors access the same variable, but because different processors access two different variables. This may occur because the cache works with a granularity of cache lines, not variables. Imagine two processors the uses each their counter, $x1$ and $x2$ and $x1$ and $x2$ are placed contiguously in memory, then $x1$ and $x2$ are placed on the same cache-line, and thus the same problem of migrating and invalidating as above occurs, even though we explicitly have tried to avoid it by using different variables. There are different ways to address this, but an easy rule of thumb is to use local variable as much as possible, e.g. data placed on the stack rather than the heap.

```
int i,j;
```



Figure 3 A common pitfall in shared memory programming, two apparently independent variables ends up competing for the same cache-line

The use of development libraries is another common source of problems in parallel application development. If the library is not safe for use in parallel applications this is a huge problem which is often discovered late in the debugging process, most modern development platforms use libraries that are safe to use with SMP architectures, these are often referred to as 'thread-safe' libraries, more on threads in section 1.2. Even if the libraries behave correctly on parallel architectures they often represent a significant performance problem, and parallel applications tend to use the standard libraries as little as possible. This is easily seen if we compare the performance on an application that uses random numbers heavily as seen in Figure 4.

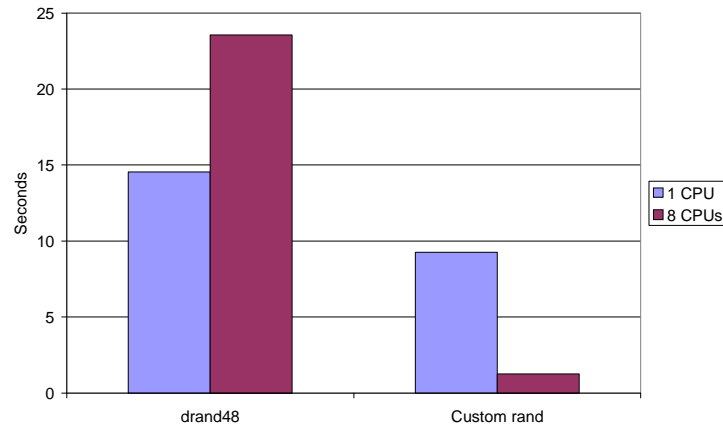


Figure 4 Performance of an application that relies heavily on random numbers, if the standard C-library random function is used the application runs significantly slower with eight CPUs compared to a one

The obvious bottleneck when working with more than one CPU, be it SMP or any other parallel architecture, is IO. Roughly speaking it is of little gain to have much processing power if the CPUs are constantly waiting for data. The IO bottleneck may exist at many levels of a system, the IO bus, e.g. a PCI bus, the disk or tape or a network interface from which the data arrives. IO in parallel systems is a topic in its own right and will not be covered further in this text.

1.1.1 The Intel Multiprocessor architecture

The Intel multiprocessor architecture is one of the most widely available multiprocessors, and most of the control hardware that is needed to implement multi-processing is placed on the CPU itself, with the result that a dual-processor Intel machine is only marginally more expensive than the cost of a uni-processor and the additional CPU. As you can imagine this makes these machines attractive from a cluster point-of-view since a 16 CPU cluster made up of eight dual-processor machines is notably less expensive than a similar cluster made from 16 standard PC. We will look into the factors that one should consider when building a cluster in a later chapter.

Intel's MP architecture is a shared bus architecture, which is why the machines that are based on it are fairly cheap, but this is also the reason why the machines only scale to very few processors in each machine. Each CPU has an execution core, an interrupt controller, a bus controller, which interfaces the processor to the processor-memory bus, and two levels of cache. Both levels of cache are kept coherent in compliance with the MESI protocol, as we've seen earlier in this chapter. A common problem with the MESI protocol is that two or more processors may compete to change the contents of a given address, this typically occurs when a process or thread tries to set a lock variable. If two or more CPUs try to change a given address at the same time they will repeatedly get a shared copy and then try to get exclusive access to this address, this can go on many times before one CPU is able to complete the change the address, and thus obtain the lock. This problem of course is rare and is only a performance problem and it will not result in an incorrect behavior. However to ensure that this is not a problem Intel have introduced the option of locking the bus for the duration of an instruction, and since the Intel processor is a CISC architecture this is enough to easily perform a synchronization lock. Locks will be investigated in further depth in the next section.

```
poll:
MOV AL,0
MOV BL,1
LOCK
CMPXCHG <sem>, BL
JNZ poll
//Now we have the semaphore
```

Example 1. An efficient polling lock in the Intel Architecture

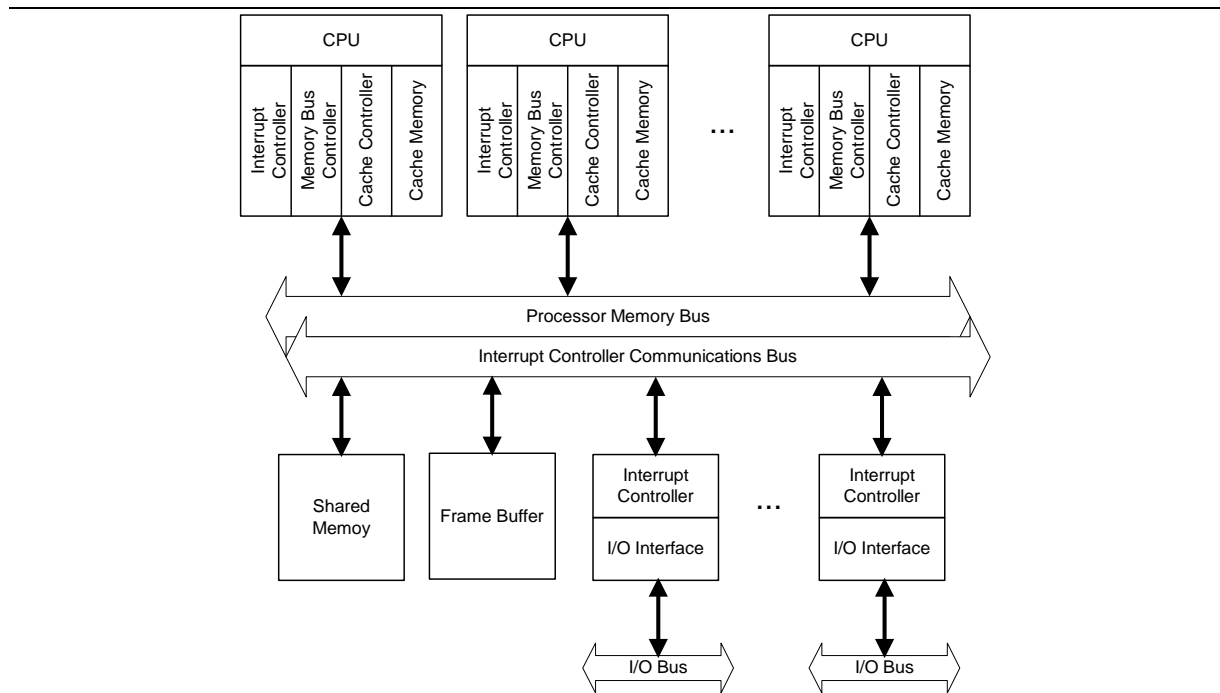


Figure 5 The Intel MP architecture

A more detailed description of the Intel MP architecture may be found in [2].

1.1.2 The SUN Enterprise architecture

The SUN Enterprise architecture represents the best within scalable SMP architectures. The Enterprise architecture may be thought of as a three-tier architecture, which consists of the CPU with a standard cache hierarchy, a gateway to scalable memory called the Ultra Port Architecture, UPA and an intra UPA system called the GigaPlane bus. An important part of this design is that each of these tiers are autonomous protocols, and the actual implementation may be changed without consideration to the other protocols, i.e. SUN also makes smaller SMP machines which eliminates the GigaPlane component and only scales as far as one UPA module allows, and the CPUs are used in small systems without any UPA components as well.

The Ultra Port Architecture is a split transaction memory protocol, which simply means that every memory operation may be split into multiple logical parts and the UPA interface supports many such partial memory operations to exist concurrently. In the Enterprise architecture the UPA modules communicate via the GigaPlane bus, which is a different kind of bus than the Intel MP-bus one we examined in depth previously. The GigaPlane bus is not one physical bus but a set of interconnected busses, the effect of this is that when a UPA module posts an operation to the GigaPlane bus this is not immediately visible to all other UPA modules, but the GigaPlane subsystem will make sure that the operation is propagated to all the GigaPlane sub-busses. In essence this allows the Enterprise to use snooping busses but without requiring the busses to hold the same information at the same time. An in-depth coverage on the transaction protocol and the necessary timing considerations that are required to implement this is beyond the scope of this text. Readers that want a more detailed understanding of the Enterprise architecture are encouraged to read [Culler98][SUN].

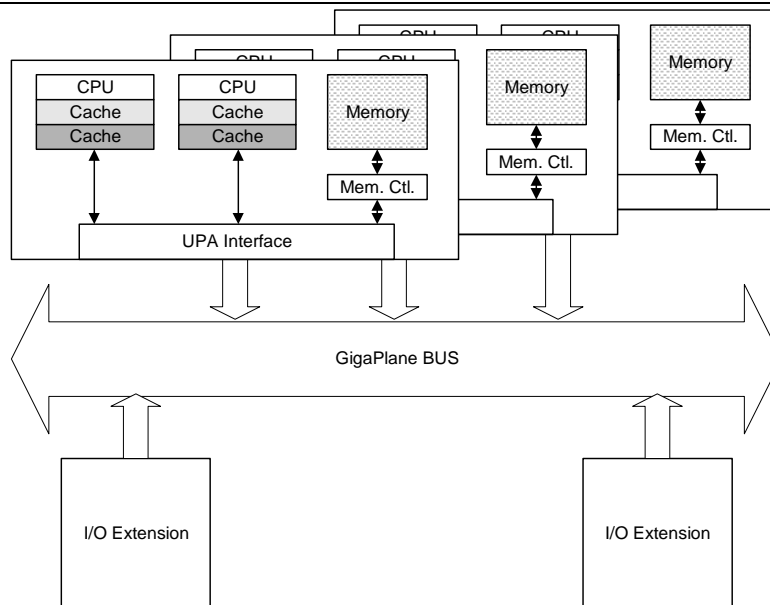


Figure 6 The Sun Enterprise Architecture

1.1.3 The SUN MAJC Architecture

The SUN MAJC (pronounced Magic) processor is an example of a SMP architecture which scales very poorly, however the CPUs that it does include may communicate faster than any of the previous SMP architectures. The idea is to place the CPUs on the same physical chip. This allows the CPUs to share all caches, in effect eliminating the need for cache-coherency protocols. Shared cache and a direct connection between the CPUs allow the MAJC CPUs to share work at a much finer granularity than any other SMP architecture, in fact the MAJC architecture includes components to allow a non-parallel application to use both CPUs by detecting instructions that may execute in parallel at runtime, however if the programmer provides more than one thread this level of parallelism will be used instead.

An important feature in the MAJC architecture is SUN's concept of 'vertical multithreading'. Vertical multithreading means that each CPU may execute several threads concurrently and the scheduling of these threads are handled in hardware. The concept of threads is covered in the next section, but for now it suffices to think of threads as processes that happen to share the same address space. The hardware handling of threads in the MAJC processor allows the CPU to schedule a new thread if the active thread is delayed. This means that when a thread issues an operation to a memory cell which is not placed in the cache, e.g. a cache-miss, another thread can be scheduled so that the CPU does not stand idle while waiting for main memory. In many ways this is similar to out-of-order execution, but rather than relying on the CPU to detect potential instructions for out-of-order execution, the application programmer can create another thread to run on the CPU.

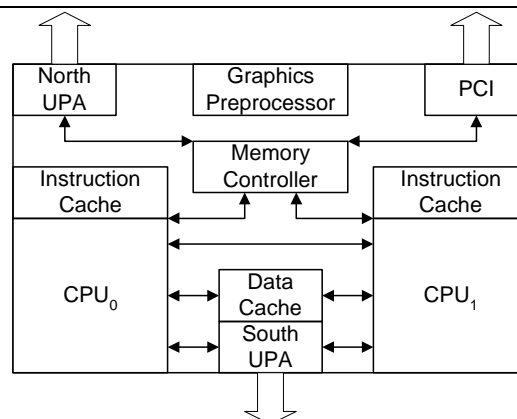


Figure 7 The SUN MAJC 5200 SMP on a chip Architecture

In this context the ‘vertical multithreading’ is quite interesting as it is nothing more than a chip level integration of a well-established technique called latency hiding. The idea is, simply put, to keep several components busy at the same time, e.g. the processor and the memory sub-system or once we start looking at actual clusters, the cluster nodes and the cluster interconnect. Vertical multithreading is also available in other architectures such as the Intel Xeon class CPUs, in this architecture the CPU appears like a dual-CPU on a chip, as in Figure 7, however there are not two CPU cores on the chip but only two register-files and a little extra control logic.

The MAJC architecture is an example of integration of instruction set, memory architecture and runtime environment to improve performance, and extensive coverage of the architecture will not fit here. We are likely to see similar multiprocessor-on-a-chip CPUs in the near future where existing architectures are merged into on-chip multiprocessors. Once these CPUs become readily available they are likely to form the basis of many desktop systems, and we will definitively find them in cluster-nodes.

1.2 Programming the SMP

Now that the SMP architecture has been demystified it is time to start considering how to program with shared memory. While the trivial solution is to run distinct programs or processes on each processor, we mainly focus on speeding up one program/process by using more than one CPU.

1.2.1 The thread model

The usual way of programming shared memory architectures is by using threads. Threads are sometimes referred to as lightweight processes which refer to the fact that a thread is basically everything a process is, except for the address space, e.g. multiple threads exist within the same address space. This is the very feature that is well suited for shared memory architectures, if one thread writes to a variable x then another thread that reads x will read that value.

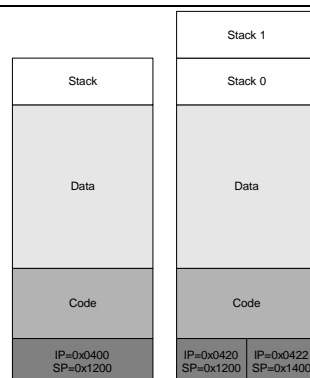


Figure 8 Two processes, a single threaded and one with two threads

Thread packages exist in a multitude, each with their own API and a few special characteristics. The most common thread package is POSIX threads, Solaris threads and Java threads for UNIX like operating systems and WIN32 threads and Java threads for the Windows platform. POSIX and Solaris threads are fairly similar in both API and functionality, while Java threads are an integrated part of Java and as such has an entirely object oriented API, Java threads also represent a functional minimum with fewer options than other thread packages.

For the purpose of high performance programming, which after all is our agenda, it is important to notice that threads are a programming concept, and thus is not linked to any implementation. This is important to us since there are three basic approaches to implementing threads, user-level threads, kernel-level threads and mixed threads.

User level threads, as the name says, are implemented entirely on user level. They provide the programmer with a useful way of structuring her work, but since the operating system only know of the one main thread in the process, user-level threads are not able to utilize more than one processor per process, which in essence makes user-level threads unsuitable for our purpose. So if you write a multi-threaded application and don't receive any performance improvement the first thing to check is whether your application does activate more than one processor.

Kernel level threads too are exactly what the name indicates, i.e. threads which are known by and controlled by the operating system kernel. This way the operating system knows of the threads and can schedule different threads on different processors so that we can achieve a real speedup. On the downside kernel-level threads are controlled

only be the kernel, which means that if one thread is suspended and need to be woken by another thread this can only happen through a costly call to the operating system.

Equally intuitive is the nature of mixed threads, which as you'd imagine seek to provide a best of both worlds to the thread modeling. The main philosophy behind the mixed-level threads is that each process has exactly one kernel level thread per CPU in the system, and as many user-level threads as the programmer creates. The user-level threads are then scheduled on the pool of kernel threads. This way we get true multiprocessing while minimizing the number of costly operating system calls.

1.2.2 Java Threads

In Java threads are an integrated part of the environment and using threads in Java is straightforward. Java contains a generic Thread class and to write code that should run as a thread the programmer simply need to place the code in a class that extends the Thread class. New thread classes must contain a public method called run, which can neither receive nor return data.

```
class dummyThread extends Thread {
    int id;
    public dummyThread(int id){this.id=id;}
    public void run(){System.out.println("Hello World from thread "+id);}
}
```

Example 2. A minimal thread class

The above code implements a class called dummyThread, which can be used to create a new thread – once running the thread will simply write the classic ‘Hello World’ message tagged with an id of the thread. This id is given by the thread that instantiates the dummyThread class, and thus is not an integrated part of the thread sub-system, as we will see below other thread systems do provide a system assigned thread id.

It is important to understand that instantiating a new Java thread class does not start a new thread, starting the thread must be done specifically using the start method which has been inherited from the system Thread class. So if we wish to create a dummyThread we need to first create a new instantiation of the class and then start up the thread.

```
dummyThread dt = new dummyThread(42);
dt.start();
```

Example 3. Instantiating and starting a new thread

If threads are used to divide work between multiple processors it is usually necessary to know when the thread that handle a specific task has finished, this could be done by the programmer but the thread sub-system provides a nice way of doing this by providing a built in method join. Calling the join method in another thread object will suspend the calling thread until the thread in the called object have terminated.

```
dt.join();
```

Example 4. Waiting for a thread to terminate

This technique is often used to ease programming, imagine that we have a machine with eight processors and thus we wish to divide a given job into eight sub-tasks. We can then do one of two things; either the main thread creates eight identical threads to handle each portion of the job and waits for all eight to finish, or the main thread creates only seven threads and perform the last task itself, after which it will return and wait for the termination of the remaining threads. Which of these two techniques is preferable is left as an exercise.

What really makes shared memory programming so easy compared to the techniques we will look at later on, is the fact that we need only concern ourselves with controlling the code, if two or more threads use the same data they will simply read and write them at the same memory address. Reading and writing an address is straightforward but unfortunately here lies the first and perhaps most dangerous pitfall in shared memory programming; the race-condition. Imagine an application where multiple threads all need to increase the same variable is part of the global result, nothing seems simpler than increasing a variable:

```
i++;
```

Example 5. Increasing a variable

However as you probably know this is not how it is done in the computer, what actually goes on is that the variable is read from memory into a register, which is then increased and written back:


```

read a,i
inc a
write i,a

```

Example 6. Assembly pseudo-code to increase a variable

Now consider what happens when two threads wish to increase a shared variable at approximately the same time. Two copies of the instructions in Example 6 may be combined in 22 ways of which only two will return the correct result. So if the simplest instruction with only two threads only has one chance in eleven to execute correctly, you'd expect that race-conditions are easily caught, especially if the operation(s) on shared variable(s) are performed by more than two threads, however to capture the error more than one thread executes the code that changes the shared variables at the same time. Exercise 10) examines this question in further depth.

The solution is to ensure that only one thread is allowed to access the shared variables at a time, such operations are called a critical section. Controlling that only one thread at a time is in a critical section is called mutual exclusion, or often simply 'mutex'. Java provides a very powerful yet simple mechanism to ensure mutual exclusion called monitors, those that are familiar with Hoare-monitors should be forewarned that Java's monitors are **not** Hoare monitors but use a far simpler scheduling model. A monitor is a set of data-variables and methods to access these data, very similar to an object, however in monitors only one thread may be inside the object at a given time. Java monitors are standard objects but the programmer can convert the object to a monitor by stating that one or more methods should ensure mutual exclusive access to the object, the keyword to specify such methods is synchronized.

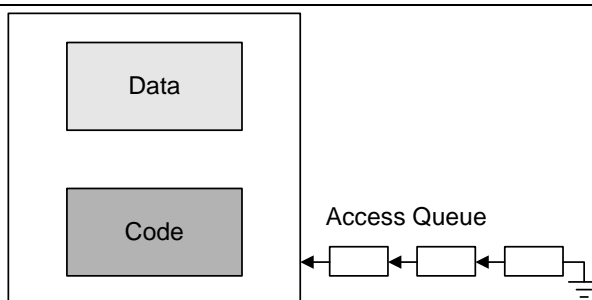


Figure 9 Logical picture of a Hoare Monitor

```

class counter() {
    int i;
    synchronized void inc(){i++;};
}

```

Example 7. A monitor in Java – the increment of I will be atomic even if two threads call it at the same time

If you know a little about compilers and optimizations you'd probably argue that the assembly code in 0 is not realistic since most compilers would try to keep the variable *i* in the register *a* for later use, this objection would be quite right, compilers do try to include the CPU registers into the memory hierarchy and keeps a variable in a register until such a time that it finds it more efficient to use the register to hold another variable. This of course poses a problem to our multithreaded applications, what is the purpose of ensuring mutual exclusion on access to shared variables if we cannot ensure that the result is written to memory so that it can be seen by other threads? The solution to this problem is to tell the compiler not to keep a given variable in a register after use, this is done by telling the compiler that this variable is volatile, i.e. sensitive to changes that the compiler cannot detect.

```

volatile int i;

```

Example 8. Defining a variable as volatile

The great thing about multi-threaded programming is that results that are written by one thread, may immediately be read by another. We use this quite often in iterative algorithms, where the source data for iteration *n* is in fact the result of iteration *n-1*, but the thread that uses the result is not necessarily the same thread that produces the data, thus we need to make sure that result has been written before it is read – otherwise what we read is either outdated or outright garbage. The technique we use for this is called a barrier and the idea is quite simple, when a set of threads need to agree that they have all finished one step of a computation they all go to the barrier, which in turn will only allow them to leave once all have arrived. Barriers may be implemented in two basic ways, either by letting each thread actively spin until all threads have arrived, or by having threads sleep until all other threads have arrived.

```

class counter {
    int cnt;
    public void counter(){cnt=0;}
    synchronized public int inc(){return ++cnt;}
    synchronized public void reset(){cnt=0;}
}

class Barrier {
    counter c;
    int members;
    volatile boolean spin;
    public Barrier(int num){members=num; spin=true; c=new counter();}
    public void meet(){
        boolean state=spin;
        if(c.inc()==members){
            c.reset();
            spin=!spin;
        }
        while(spin==state);
    }
}

```

Example 9. A Barrier which is implemented by spinning the threads until all have arrived

A spinning barrier as the one demonstrated in Example 9 is characterized by two facts; latency is low, as soon as the last thread joins the barrier all threads will return and continue the calculation within a CPU cycles. On the other hand the threads keeps the CPUs active by spinning on the counter variable, which prohibits the CPU from doing valuable work in the mean time. In java this could be the garbage collection thread, which might as well spent the time cleaning up dead objects while the thread is waiting. To allow this one need a barrier that allows a thread to be descheduled at the arrival to the barrier and rescheduled once all threads arrives these are also know as a signal-wait barrier.

```

class Barrier {
    int cnt, members;
    public Barrier(int members){this.members=members;}
    public synchronized void meet() {
        if(++cnt==members){
            cnt=0;
            notifyAll();
        }
        else try {wait();} catch(Exception e){};
    }
}

```

Example 10. A Barrier which is implemented by sleep and wakeup

The tradeoffs in the signal wait barrier are the exact reverse of the spinning implementation, any tread that is not the last to arrive, will go to sleep until the last thread signals the others that the barrier is completed and they may go on. This way the CPUs which were running these threads are free to do valuable work, on the other hand descheduling a thread, scheduling and descheduling another and finally rescheduling the original thread requires a number of CPU cycles so that the latency of the barrier increases significantly.

The remaining question is; which type of barrier should one use? This question does not have a valid answer since each model has its right depending on the application. If an application requires the threads to meet at barriers very often, it is most likely that the code is well balanced, which means that all threads will arrive at the barrier at approximately the same time, in this case the overhead of a signal-wait barrier is too costly and the spinning version is likely to be the fastest solution. On the other hand if barriers are rarely called then the work is also more likely to be unbalanced and utilizing the waiting period on a CPU for the barrier to complete is more likely to be worthwhile. A generic barrier will often combine the two approached, so that threads start out by spinning for completion, once a thread has spun for as long time as a sleep-wakeup takes all spinning threads goes to sleep and the barrier changes to a signal-wait barrier. It is easily seen that this solution is two-worst.

Below is a simulation of a multithreaded application that synchronizes via a barrier. Work is simulated by one thread as adding a new random number to a variable given number of timers, the more loops that simulate work the more unbalanced the application is. The simulated work creates a new Random object each time, which ensures that we will need garbage collection.

```

import java.util.Random;

class Worker extends Thread {
    private int max_work;
    Barrier bar;
    public Worker(int work, Barrier barrier){max_work=work; bar=barrier;}
    public void run(){
        int x=0;
        for(int i=0;i<1000;i++) {
            for(int j=0; j<max_work; j++) x+=new Random().nextInt();
            bar.meet();
        }
    }
}

public class barrier_simulation {
    public static void main(String argv[]){
        int worktime=new Integer(argv[0]).intValue();
        Worker w1,w2;
        Barrier barrier=new Barrier(2);
        w1=new Worker(0, barrier);
        w2=new Worker(worktime, barrier);
        double start=new java.util.Date().getTime();
        w1.start(); w2.start();
        try {w1.join(); w2.join();} catch(Exception e){};
        double stop=new java.util.Date().getTime();
        System.out.println(((stop-start)/1000) + " seconds");
    }
}

```

Example 11. Barrier simulation

Figure 10 show the result of blocking and spinning barriers with the simulation shown in Example 11, as we vary the simulated work from one through 10,000 iterations. If the unbalance between the two threads workload is only one iteration then the spinning barrier performs significantly better than the blocking version, the spinning version is 0.04 seconds faster than the version that initiates rescheduling of the threads. Because the simulation creates a new Random object for each iteration the execution actually do force the garbage-collector thread in the Java virtual machine, to be activated, which means that at some time the blocking-barrier becomes more efficient than the spinning version, in this case it happens somewhere between a simulated workload of 100 and 500 iterations, at 100 iterations the spinning version is 0.01 seconds faster than the blocking version, at 500 the relationship has changed so that the blocking version is 0.01 seconds faster than spinning. When the simulated workload reaches 10,000 iterations the advantage of blocking over spinning has grown to 0.45 seconds, unfortunately we cannot see this from the graph and in percentages it is not as significant at the advantage of spinning over blocking with a small unbalance, but the difference is there nonetheless and one should carefully consider the nature of the application when choosing a barrier technique.

Barriers are used with most parallel programming APIs and once we get to distributed-memory architectures more advanced, and more scalable, barrier techniques will be outlined.

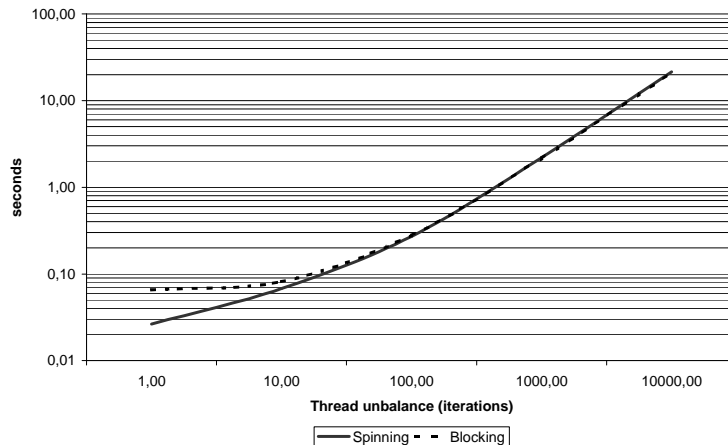


Figure 10 Comparing Blocking and Spinning barriers on a dual processor machine

The results that are shown in Figure 10 clearly show that the Java threads in this case utilize more than one processor, thus the Java virtual machine and the operating system supports kernel level threads. Java in fact allows you to choose whether you want to use user-level or kernel-level threads, though not all Java implementations implement both versions. How you tell Java which thread model to use depends on your particular platform, but user-level threads are generally referred to as ‘green’ threads and kernel level threads as ‘native’ threads.

```
setenv THREADS_FLAG green
setenv THREADS_FLAG native
```

Example 12. Setting Java to use user-level respectively kernel-level threads in Linux

1.2.3 POSIX Threads

Outside the Java world the most common thread package is POSIX threads, these are often simply referred to as pthreads. POSIX threads are supported by most UNIX-like operating systems, including the popular open-source versions Linux and BSD. Since POSIX threads is a language independent library the actual interface to the thread package is a little more complex than Java threads, but the only major difference is the lack of monitors and the fact that threads are active as soon as they are created, as opposed to the Java version which requires threads to be activated after creation. POSIX threads are based on functions and a new thread of execution is given a start address in the form of the address of a function where the thread should start its life. A function that should be used as a stand-alone thread should take a void pointer as its only parameter and return a void pointer.

```
#include <pthread.h>
void * my_thread(void *my_id){
    printf("Hello world from thread #%d thread id #%d\n",\
        (int)my_id, pthread_self());
    return NULL;
}

main(){
    pthread_t thread;
    pthread_create(&thread, NULL, my_thread,(void *)42);
    pthread_join(thread,NULL);
}
```

Example 13. Hello world with POSIX threads

Since POSIX threads are language independent critical regions cannot be implemented as easily as the synchronized methods in Java, instead POSIX threads are based on the classic technique of token holding. Each critical region is associated with a token, and in order to enter the critical region a thread has to pick up the token, leaving the critical region means releasing the token. A pthread token is called a mutex, **mutual exclusion** variable and a critical region is entered and left by locking or unlocking a mutex.

```
static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_lock(&mutex);
pthread_mutex_unlock(&mutex);
```

Example 14. Initializing, locking and unlocking a POSIX mutual exclusion variable

Pthread based applications too, need to allow threads to go to sleep and be woken by another thread. Once again the language independence makes the procedure a little more cumbersome, but it is still straightforward. Pthreads are suspended and woken on condition variables, i.e. a thread issues a wait with a condition variable and is put to sleep until another thread either signals to the same condition variable or broadcasts to it, a signal to a broadcast variable means that one thread may be woken and resume execution, a broadcast means that all threads that are waiting for a condition variable may be woken. There is however an obvious potential race-condition with this kind of signal and wait, if one thread issues a wait and another at the same time issues a signal on the same condition variable. To avoid this the POSIX thread interface requires a thread to pass another variable to a wait call, this second parameter is a mutex variable, which must be locked at the time of calling wait. A wait statement then translates into an operation sequence that goes; unlock mutex, wait for condition, lock mutex, where the mutex unlock and the wait is performed atomically. Any thread that signals a condition variable should then remember to lock the same mutex variable before sending the signal or broadcast. This way race-conditions are avoided.

```
static pthread_cond_t done = PTHREAD_COND_INITIALIZER;
pthread_cond_wait(&done, &mutex);
pthread_cond_signal(&done);
pthread_cond_broadcast(&done);
```

Example 15. Creating, waiting, waking one and waking all threads with POSIX threads

POSIX threads has no immediate semantics for defining shared data variables¹ but most languages, including C which we use here, support the notation of volatile variables, and the defining keyword is usually volatile. If we combine all of the above synchronization techniques for POSIX threads we can define a blocking barrier mechanism similar to the Java version found in Example 10. The C/POSIX threads version is found in Example 16 and it is clear that the language independence which POSIX threads provides do increase the complexity of using it, once you get used to it however it is fairly easy.

```
barrier() {
    static volatile int barcnt=0;
    static pthread_cond_t done = PTHREAD_COND_INITIALIZER;
    static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

    pthread_mutex_lock(&mutex);
    barcnt++;
    if (barcnt < WORKERS) {
        pthread_cond_wait(&done, &mutex);
    }
    else {
        barcnt=0;
        pthread_cond_broadcast(&done);
    }
    pthread_mutex_unlock(&mutex);
}
```

Example 16. A blocking barrier implemented in POSIX threads

1.2.4 Solaris Threads

The first thing to notice with Solaris threads is that the logical elements that we called threads in the previous two sections are called light weight processes, LWPs, in Solaris. Except from the name and the order on some parameters Solaris LWP are very similar to POSIX threads. The Solaris thread model however provides yet another level of sub-tasking which are then named threads. A Solaris thread may be fixed to run within one LWP in which case it is denoted a bounded thread or it may float amongst the different LWPses in the system and is then called unbounded.

¹ It does however provide the programmer with an interface to provide each threads with a private data-segment if you should need this.

```
thr_create(NULL, 0, my_thread, (void *)42, NULL, &thread);
thr_join(thread, NULL, NULL);
```

Example 17. Creating and joining with a Solaris thread is very similar to the same operations with a POSIX thread, apart from the order of the parameter order

Newer versions of the Solaris operating system also support POSIX threads directly and SUN discourages the use of Solaris threads.

1.2.5 WIN32 Threads

WIN32 threads are designed to interface naturally with the WIN32 programming API, WIN32 threads also extends threads with another level called fibers.

```
threads[0] = CreateThread(NULL, 0, my_thread, &args, 0, &thread_ID);
WaitForMultipleObjects(1, threads, TRUE, INFINITE);
```

Example 18. Creating and joining with a new thread in the WIN32 systems

1.2.6 OpenMP

OpenMP is a fairly new approach to easing parallel programming while adding as little new semantics as possible. Currently OpenMP exist in a C/C++ and a Fortran version. OpenMP is based on an well established parallel programming technique called Fork-Join. The idea is that an application runs sequentially up until a point where a section may be parallelized. At this point a set of threads are created, each of which execute its own part of the code section, after finishing the parallel block the threads terminates and the main thread continues after all parallel threads have finished.

OpenMP is a crude macro-oriented approach to parallel programming, which is its main attraction. If a sequential version of an application exists this can be expanded with OpenMP macros, without modifying the original code, e.g. the sequential code for a vector addition:

```
for(i=0; i<length; i++)
    c[i]=a[i]+b[i];
```

Example 19. Sequential Vector addition

Can be turned into a parallel version simply by adding an OpenMP macro:

```
#pragma omp parallel for
for(i=0; i<length; i++)
    c[i]=a[i]+b[i];
```

Example 20. Parallel Vector addition using OpenMP

After which all processors in a machine may be used to perform the addition in parallel.

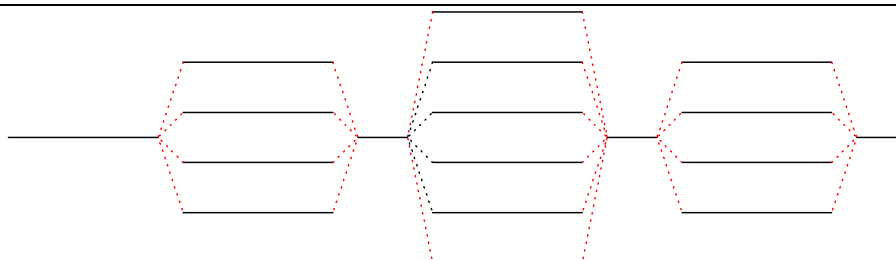


Figure 11 The Fork-Join parallel execution model in OpenMP

It is important to realize that OpenMP is a simple macro-oriented language extension, which means that if a programmer specifies a loop for parallel execution but this is not immediately possible, e.g. because of the result of one iteration is dependant on the result of a previous iteration, the OpenMP will still execute the loop in parallel, but the result will be incorrect.

While OpenMP seems like an attractive way of writing parallel applications, it is important to recognise that one can only expect limited processor utilization with OpenMP. First of all only sections that are explicitly made parallel will execute in parallel, which in turn means that speedup is limited by the part of the execution time that is

spent in the sections that are parallelizable. Another cost that limits the efficiency of OpenMP is the Fork-Join execution model, creating and joining threads easily becomes costly if it happens often. However for a set of applications the OpenMP is a simple and efficient model, it is simply no a silve bullet for genereel parallel programming. As clusters frequently use SMP machines as nodes the use of OpenMP is also becoming increasingly popular in the cluster world.

OpenMP consist of a large number of macros, most of which are more complex than the parallel for example that was described here. In addition to simplifying parallelaization of existing sequential applications OpenMP is also gaining popularity as an intermediate language for self-parallelizing compilers.

1.3 Orchestrating for Shared Memory

The term orchestration means dividing the problem into sub-tasks and distributing these tasks among the processors. Orchestration is a central part of creating a parallel application and errors that are introduced in this step may introduce significant correctness or performance problems. Orchestration of applications requires considerations towards both architecture and programming language, since the orchestration will be directly visible in the end code.

The importance of correct orchestration of parallel applications cannot be over-rated, and in many ways the field of parallel application design and implementation boils down to this one issue, how to do orchestration. Orchestration principles and techniques may be coarsely divided into two groups, general techniques and architecture specific techniques. Thus as we revisit orchestration in the later chapters some of the techniques that are introduced will be applicable for other architectures also, however to the extend that it is possible orchestration techniques are presented with the architectures that they are most often used with.

Orchestration for SMP architectures may be fairly simply on an abstract level or it may be detailed to the level where cache-line-width, main memory layout and other issues are considered. For now we will consider only the orchestration techniques on the most abstract levels, finer points will be made later in the chapter and some are left as exercises.

1.3.1 Static Orchestration of Parallel Threads

The easiest approach to orchestration of parallel threads is to divide the data into n chunks and have n workers work on each their portion of the data. Not only is this the simplest way to divide the work it is also likely to be the most efficient in the cases where it is doable. The catch is naturally that a static division of the data amongst processors is rarely immediately possible, most often we need to redefine a function or in some other way modify the application to allow data to be distributed amongst the processors. Usually the computed value of a data-element will in some way depend on the result of one or more of its neighbors and we need to add synchronization and define update-order to ensure correct execution. We will investigate an example of such a scenario later in this chapter.

A different, but less obvious, problem with static orchestration is that it requires the work that should be done on the different data-blocks must be fairly well balanced. If the work is not well balanced one processor might finish quickly while another processor will work for a long time, the result is that the speedup that we get is far smaller than what we would expect. This problem often comes as a surprise to the application programmer and results in a reorchestration of the application. There are two primary approaches to resolving an unbalanced application, either change the static layout of the data or shift to a dynamic orchestration of the work. Dynamic orchestration is described in the next section and the approach of well-considered work distribution is the main point of the first programming project ‘The Desert Map’.

1.3.2 Dynamic Orchestration of Parallel Threads

Orchestrating the work in a dynamic fashion is the natural opposite to the static approach and is best suited if the time it takes to perform a sub-task is varying and unpredictable. Dynamic distribution of the data is best suited if the data is not dependent on sub-results from the same iteration over the data. Dynamic orchestration is usually implemented by dividing the work into many small tasks, where the number of tasks should be much larger than the number of processors. The tasks are then placed on a queue, and the workers simply work by repeatedly fetching a job from the queue and executing it. Once the work-queue is empty the workers terminate. It is important that the tasks on the queue are small, otherwise the balance of the execution becomes poor with a low speedup as the result.

The dynamic task orchestration is a special case of a common technique in parallel processing known as the ‘bag-of-tasks’ technique, where workers both consume and generate tasks for execution. This technique will be further investigated in later chapters.

1.3.3 Specialized Workers

The third and last of the basic orchestration techniques is known as the specialized workers approach. This approach is quite different in that it is not focused on the data but rather on the instructions, i.e. if an application consists of two or more logical tasks then a thread may handle each such task. This is exactly what happens within a pipelined CPU, the execution of a single instruction is divided into several logical steps and to implement each such step there is a pipeline stage to execute that portion of the instruction. Because of this specialized worker implementations are also often referred to as pipelined applications.

There are several reasons why the specialized workers approach is not common in parallel processing, first of all it is less intuitive to consider the instructions for parallelization than the data, parallelizing the code also requires a much more detailed understanding of the code than previous orchestration strategies. A more important reason why we rarely see the specialized workers paradigm in use is the fact that it often limits the degree of parallelism that may be achieved, since we need to identify a unique task for each CPU we wish to activate. There are however applications where specialized workers are superior to the data-parallel versions, one of these are ray-tracing and we will examine this and others later on in the programming project ‘The Clone Machine’

1.4 Monte Carlo Simulation

The following example represents a somewhat stupid way of finding π , based on a Monte Carlo simulation. The model is fairly simple; if we take a square with two units side length and inscribe that by a circle with unit radius then we can throw a dart at the drawing multiple times, each time the dart fall within the circle we will increase a counter. We know that the area of the square is four unit-squared and the area of the circle is π , thus the ratio of darts that fall within the circle is the ratio of four unit-squared that π represents. For simplicity we reduce the problem to the upper right quadrant of the circle, e.g. one fourth of the original problem.

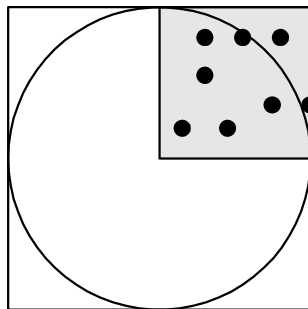


Figure 12 The Monte Carlo Pi model

Obviously this is an extremely inefficient way of determining the value of π , however Monte Carlo methods and the closely related Las Vegas methods are quite valuable tools in various fields of science and the simple model that we present here may be replaced by a set of ‘real’ problems within physics, chemistry and biology. See the exercises section for more examples of Monte Carlo methods.

1.4.1 Sequential version

The sequential code is straightforward:


```

import java.util.Random;
public class PI {
    public static void main(String argv[]){
        final int LIM=10000000; //We throw 10E7 darts
        int i,count=0;
        double x,y;
        Random r=new Random();

        for(i=1;i<=LIM;i++) {
            x=r.nextDouble();
            y=r.nextDouble();
            if((x*x+y*y)<=1.0)count++;
        }
        System.out.println("PI is " + 4.0*count/LIM);
    }
}

```

Example 21. Sequential Monte Carlo simulation to find π

Performance

Running the code with the time command returns:

```

time java PI
PI is 3.141338
0.100u 0.010s 1:51.26 0.0%      0+0k 0+0io 4739pf+0w

```

It takes one minute and 51 seconds to perform the ten million simulated dart throws, and returns π with only three correct decimals. There are of course much better ways of calculating π but the Monte Carlo simulation technique is in fact a useful mechanism for other problems, so speeding up the simulation is of real interest.

1.4.2 Statically Orchestrated Parallel Version

Parallelizing the code is straightforward, instead of one simulated dart and one target, we'll simply make two. The two darts can be thrown in parallel without any problem since the result of one dart does not influence the result of any other dart.

Using threads the code is very similar to the above, but has grown some in order to handle the threads:

```

import java.util.Random;

class data {
public int count;
data(){ count=0; }
}

class Worker extends Thread {
private int id,LIM; data cnt;
public Worker(int id, int LIM, data cnt){this.id=id; this.LIM=LIM;
this.cnt=cnt;}
public void run() {
int i;
double x,y;
Random r=new Random();
for(i=0;i<LIM;i++) {
x=r.nextDouble(); y=r.nextDouble();
if((x*x+y*y)<=1.0)cnt.count++;
}
}
}

public class PI {
public static void main(String argv[]){
Worker w1,w2;
data cnt;
int LIM=10000000;
cnt=new data();
w1=new Worker(0,LIM/2,cnt); w2=new Worker(1,LIM/2+LIM%2,cnt);
w1.start(); w2.start();
try { w1.join(); w2.join();} catch (InterruptedException e) {};
System.out.println("PI is " + 4.0*cnt.count/LIM);
}
}

```

Example 22. Statically orchestrated parallel Monte Carlo simulation to find PI (faulty!)

The actual simulation code is almost identical to the sequential version the only difference is that the counter that is increased when a dart falls within the circle is not a local counter, but a shared counter. The main program creates the shared counter this way the main program can read the resulting number of hits directly once the threads have finished the work.

Running the code with the time command returns:

```

PI is 3.1221796
118.910u 0.090s 0:59.73 199.2% 0+0k 0+0io 7109pf+0w

```

The immediate focus of attention is the achieved speedup, which is $111/60=1.85$, which in turn corresponds to a processor utilization of 0.925, which is not too bad.

However, when we look at the result, the error on π has been increased from three correct decimals to only one. Though we cannot expect to get identical results from two runs, since we use random input, the added error is too large to be based on a unfortunate random number chain, something must have gone wrong in the conversion to a parallel version.

The problem is the increasing of the global counter. `cnt.count++` looks very atomic, but what actually happen is `read(count);increase(count);write(count)` and then it is far more obvious what happens, while one processor is executing the increase part the other may be reading count, which will result in the first increase is lost.

Fixing the synchronization problem

Because of the synchronization problem we need to protect the count variable from concurrent operations. This may be done in a variety of ways, each of which is preferable for a given problem and/or system. The classic way of doing this is via a semaphore, e.g. before an increase the semaphore must be locked and then unlocked after the write. Alternatively the increase can be performed in a monitor, this approach has the advantage that one does not accidentally forget to lock the semaphore before increasing the protected variable. In applications with thousands of lines of code, finding the one spot where a variable is accessed without correct protection may be very hard. We

have already covered that Java provides a very simple mechanism for monitor-like behavior, the keyword `synchronized`. Using the `synchronized` instruction the code becomes:

```
import java.util.Random;

class data {
    public int count;
    public void data(){ count=0;}
    synchronized void inc() { count++;}
}

class Worker extends Thread {
    private int id,LIM;
    data cnt;
    public Worker(int id, int LIM, data cnt)
        {this.id=id; this.LIM=LIM; this.cnt=cnt;}
    public void run() {
        int i;
        double x,y;
        Random r=new Random();
        for(i=0;i<LIM;i++) {
            x=r.nextDouble(); y=r.nextDouble();
            if((x*x+y*y)<=1.0)cnt.inc();
        }
    }
}

public class PI {
    Nothing changed here!
}
```

Example 23. Statically orchestrated parallel Monte Carlo simulation to find PI (correct – but slow!)

Running the code with the time command returns:

```
PI is 3.1424512
139.530u 14.160s 1:21.66 188.2% 0+0k 0+0io 7109pf+0w
```

Now the third decimal is not quite there, but this is due to the randomness of the algorithm, more interestingly our speedup has fallen to $111/82=1.35$ which is a processor utilization of 0.68, much worse than the first version. So the question is must we sacrifice speed to get the right result?

Fixing the speed problem

The only change from the initial parallel version to the parallel version that yields the correct result is that the shared counter is updated via a call to the monitor construct rather than performed directly, thus the 1/3 increase in execution time must derive from this. We can conclude that to get good performance we must minimize the use of the monitor construct. Since all the counter does is to keep track of the darts that fall within the circle, there is no reason why the counter needs to be increased as soon as a dart falls within the circle. Instead each worker may keep track of its own hits and update the counter once, just before the worker terminates. All that is needed for this change is that each worker gets its own, local, counter and that the increase function in the counter monitor, is changed to increase by a parameter, instead of simply by one. The resulting code looks as:

```

import java.util.Random;

class data {
    public int count;
    public void data(){ count=0;}
    synchronized void inc(int by) {count+=by;}
}

class Worker extends Thread {
    private int id,LIM;
    data cnt;
    public Worker(int id, int LIM, data cnt)
        {this.id=id;this.LIM=LIM;this.cnt=cnt;}
    public void run(){
        int i;
        double x,y;
        int lcount=0;
        Random r=new Random();
        for(i=0;i<LIM;i++) {
            x=r.nextDouble(); y=r.nextDouble();
            if((x*x+y*y)<=1.0)lcount++;
        }
        cnt.inc(lcount);
    }
}

public class PI {
    Nothing changed here!
}

```

Example 24. Statically orchestrated parallel Monte Carlo simulation to find PI (correct and fast)

Running the code with the time command returns:

```

PI is 3.1424888
117.180u 0.200s 0:59.04 198.8% 0+0k 0+0io 7109pf+0w

```

Now the result is correct again, and the performance is in fact slightly better than with the faulty parallel version with a processor utilization of 0.942 vs. 0.925, so that in conclusion to the question posed above, the answer is no! We need not sacrifice performance to get the correct result. While this is true for this example this is not always so, often there is a mandatory cost to pay to ensure correct execution. In fact one could argue that this is true in this example, see exercise 5).

1.4.3 Dynamic Orchestrated Version

While a static orchestrated version is usually preferable when we have exclusive access to the hardware and know its configuration, this is sometimes not the case. The application may have to coexist with other applications and/or we don't know the exact configuration of the hardware.

In the π example jobs are fairly easy to model, since rather than splitting the number of darts evenly amongst the processors, the darts are divided into more blocks. Each worker then repeatedly retrieves a job and executes the assigned number of darts, until all darts has been thrown.

The dynamic code looks as:

```

import java.util.Random;

class data {
    public int count;
    private int LIM,task;
    public data(int LIM, int task){this.LIM=LIM; this.task=task; count=0;}
    synchronized int get_job() {
        if(LIM>0){LIM-=task; return task;}
        return 0;
    }
    synchronized void inc(int by){ count+=by; }
}

class Worker extends Thread {
    private int id,LIM;
    data cnt;
    public Worker(int id, data cnt) {this.id=id; this.cnt=cnt;}
    public void run(){
        int i;
        double x,y;
        int lcount=0;
        Random r=new Random();
        LIM=cnt.get_job();
        while(LIM>0){
            for(i=0;i<LIM;i++) {
                x=r.nextDouble(); y=r.nextDouble();
                if((x*x+y*y)<=1.0)lcount++;
            }
            LIM=cnt.get_job();
        }
        cnt.inc(lcount);
    }
}

public class PI {
    public static void main(String argv[]){
        Worker w1,w2;
        data cnt;
        int LIM=10000000;
        cnt=new data(LIM,100);
        w1=new Worker(0,cnt); w2=new Worker(1,cnt);
        w1.start(); w2.start();
        try {w1.join(); w2.join();} catch (InterruptedException e) {};
        System.out.println("PI is " + 4.0*cnt.count/LIM);
    }
}

```

Example 25. Dynamic orchestrated parallel Monte Carlo simulation to find PI

Running the code with the time command returns:

```

PI is 3.14221
121.790u 0.830s 1:01.70 198.7% 0+0k 0+0io 7109pf+0w

```

So the dynamically scheduled version is slightly slower than the static version, if the node had been occupied with other stuff the situation could have been reversed. In addition the job size of 100 might be too small for this type of problem.

1.4.4 Specialized Worker Orchestration

Monte Carlo Pi is not the most obvious application to orchestrate with the specialized workers paradigm, but not the less it is straightforward, at least for two CPUs beyond that it is much harder. The Pi application may be divided into two basic tasks, generating the random numbers and checking whether they hit our unitary circle. Our approach to a specialized worker version is thus to have one thread generate the random number and other to test for circle-hit. This approach is also known as producer-consumer parallelism since one thread produces the random numbers and another consumes them, this type of parallelism will be examined further in the next chapter.

The problem we need to consider first is known as the handover problem, e.g. the thread which generates the random numbers need to hand them to the thread which consumes them. Correct handover requires attention to two issues, first the producer may never overwrite random numbers before they have been read by the consumer and second that the consumer may never read random numbers before they have been generated. If we follow this approach strictly we will end up with a handover algorithm that uses two barriers.

```
barrier.meet();
if(producer)coordinates.set(random_coors);
barrier.meet();
if(consumer)random_coors=coordinates.get();
```

Example 26. A strict handover implementation

It is obvious that this approach is very costly and we don't need to test it to realize that we won't get a good speedup. The solution is to use a well know data-structure, the bounded buffer. Using a bounded buffer the two threads are able to execute with some skew, e.g. the producer is allowed to ahead of the consumer.

```
public class bounded_buffer {
    random_coor [] buf;
    int size, head, tail;
    bounded_buffer(int len){
        size=len; buf=new random_coor[len];
        head=0; tail=len-1;
    }
    synchronized void put(random_coor c){
        if(tail==head)notify();
        buf[head]=c;
        head=(head+1)%size;
        if(head==tail) try{wait();}catch(Exception e){};
    }
    synchronized random_coor get(){
        if(tail==head)notify();
        tail=(tail+1)%size;
        if(head==tail) try{wait();}catch(Exception e){};
        return buf[tail];
    }
}
```

Example 27. Bounded buffer implementation in Java

A two specialized workers version of the Monte Carlo Pi then consist of a producer of random numbers and a consumer.

```

import java.util.Random;

class Producer extends Thread {
    int LIM;
    random_coor c;
    bounded_buffer b;
    Producer(bounded_buffer b, int LIM) {this.b=b; this.LIM=LIM;}
    public void run(){
        int i;
        Random r=new Random();
        for(i=0;i<LIM;i++) {
            c=new random_coor();
            c.x=r.nextDouble(); c.y=r.nextDouble();
            b.put(c);
        }
    }
}

class Consumer extends Thread {
    int LIM, hit;
    random_coor c;
    bounded_buffer b;
    Consumer(bounded_buffer b, int LIM) {this.b=b; this.LIM=LIM;}
    public void run(){
        int i;
        random_coor c;
        for(i=0;i<LIM;i++){
            c=b.get();
            if(c==null)System.out.println("Strange NULL returned as "+i);
            if((c.x*c.x+c.y*c.y)<=1.0)hit++;
        }
    }
}

public class special_worker_mcpi {
    public static void main(String argv[]){
        int LIM=10000000;
        bounded_buffer b=new bounded_buffer(128);
        Producer p=new Producer(b,LIM);
        Consumer c=new Consumer(b,LIM);
        p.start(); c.start();
        try {p.join(); c.join();} catch (InterruptedException e) {};
        System.out.println("PI is " + 4.0*c.hit/LIM);
    }
}

```

Example 28. Specialized worker version of Monte Carlo Pi

As you might imagine the specialized version is not very efficient, running the code in Example 28 returns:

[TEXT NEEDED: Run output]

1.5 WATOR

The Water TORus world, WATOR, is a classic discrete event simulation, and while it provides valuable information in itself we chose to introduce it here for reasons similar to the Monte Carlo PI example, namely that it is simple and easy to understand, while still being typical for the class of applications that it represents. Discrete event simulations are widely used to model everything from digital systems to financial forecasts, logistics and traffic simulations.

WATOR is the simulation of a very special world, first of all the planet is not a sphere as most planets we know, rather the planet is shaped as a doughnut, or a torus, which greatly simplifies mapping the world into discrete blocks, as shown in Figure 13. As the whole surface of WATOR is covered with water there are only two types of life, which we are interested in; fish and sharks.

Fish are simple organisms that move around at random, and at some point when a fish comes of age it will have two children and die itself. A fish can move into any of its neighboring eight squares, given that the square is empty, if all eight neighboring squares are occupied the fish remains in place. At any discrete time step a fish becomes one time-unit older, and once it has come of age it will split into two fish, one of which will go to a neighbor cell while the other stays in the cell where it was born, both new fish is age zero.

Sharks are similar simple creatures, however sharks also need to eat fish in order to survive. At each time-step a shark moves to a random neighboring cell which holds a fish, if there are no fish which neighbors the shark it moves to a random neighboring cell and increases its hunger index, if the hunger index reaches a starvation limit the shark dies, when the shark eats it resets the hunger index to zero. Similar to fish, sharks will at some point get old enough to breed and once this age is reached the shark is replaced by two new sharks, both with age and hunger index zero. Thus the simulation of one time-step consists of two steps, first all fish are moved, then the sharks.

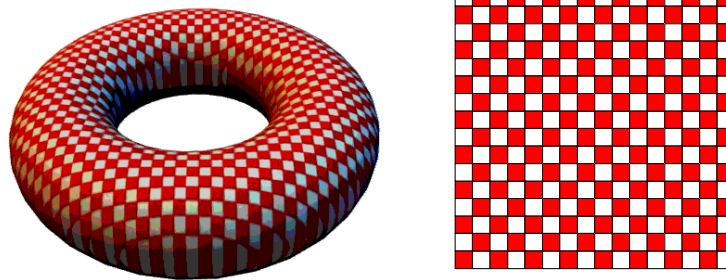


Figure 13 The WATOR world, to the left is the 3-dimensional version and to the right is the unfolded 2-dimensional version

1.5.1 Sequential version

The sequential version is very simple, we have let out the actual simulation code here since it has no influence on the problem we will investigate in the following, which is porting this application to run in parallel in SMP architectures.

```
main() {
    int i;
    initworld();
    for(i=0; i< TIMESTEPS; i++){
        movefish();
        movesharks();
    }
}
```

Example 29. Main loop for the sequential version of WATOR

The complete code can be found on the web-site.

1.5.2 Parallel Version

Turning the WATOR code parallel may be done in various ways, one may either split the fish and sharks among the processors or parts of the torus may be split amongst the processors, either statically where each processor operates on a given portion of the torus throughout the execution or dynamically so that the torus is re-divided at each iteration to attempt to balance the work between the processors. Of the two previous options, the latter is generally preferred. Unfortunately none of the solutions are as straightforward as they may seem if the result of the parallel version should return the same result as the sequential version.

Lets assume that the sequential version moves the fish and sharks as they appear on the unfolded torus, i.e. from the upper-left corner to the lower right. If we let one processor move the fish in the upper half and another move the ones in the lower half, then the first fish that the second processor are likely move into the cells above them, to which the fish on the two rows above them should actually have had precedence. In other words there is no way the processor that moves fish in the lower half can begin moving its fish before the preceding processor is done.

There are three fundamentally different approaches to solving this problem, all of which are practically applicable. One way to handle the dependency problem is to skew the execution of the parts of the torus, as in a pipeline. The idea here is that the processor that handles the lower portion of the torus waits until the upper processor is done moving its fish, before starting to move its own, while the fish are being moved in the lower part the sharks are moved in the upper, and then vice versa. This approach has two major disadvantages, first of all the solution will not scale to hundreds of processors since the latency due to the depth of the pipeline becomes a significant part of the overall execution time. However the really significant problem with this approach is the unbalanced workload that comes from the fact that we must expect the virtual world to hold far fewer sharks than fish, so even though the geometric division of the workspace seem intuitive it does by no means translate into a balanced workload.

A fairly attractive solution in this case is the ostrich approach, e.g. stick our head in the sand and ignore the problem. While this is often an attractive solution, it is rare that one can argue that it is a true alternative, but in this case it does present a usable alternative. Ignoring the problem is acceptable only because we base the movement of both fish and sharks on random numbers and we have no chance of replicating the sequence of random numbers generated by one process on two or more processes in such a way that the same random number is returned to the same fish or shark at the same iteration. One can therefore argue that either we cannot make the application parallel or we can choose to argue that the nature of random numbers allow us to process the movements in a different order than the sequential since movements are randomized anyway. If we choose the latter argument, which is quite valid in this scenario, turning the application parallel is almost trivial. If we divide the work geometrically on the torus we need to ensure that no fish or shark within one iteration is moved from the beginning of one partition to the end of another and then moved once again within the new partition, this is easily done by tagging the object as moved. Alternatively we can split the work by dividing the fish consequently sharks amongst the processors. This way we do not risk that an object is moved twice within one iteration. This solution as a hidden pitfall however, if we uncritically divide the objects amongst the processors then all processors are likely to traverse most or all of the torus surface checking neighboring cells, this will not only result in a poor cache utilization but also raise the chance that performance will suffer from false sharing. Thus if one does choose this approach the objects should be sorted before each iteration and then divided between the processors, in effect creating a combination of the geometric and object splitting.

Since we have already argued that WATOR is typical for many discrete event simulators it is natural to ask what to do if movement is not random but controlled by a set of causal rules, in this case the ostrich approach won't help us. Imagine that instead of WATOR we should simulate traffic on a road, where each car has its own desired speed and we have a set of rules describing the willingness of a driver to take an outer lane and his willingness to leave it again. In such a scenario we cannot simply ignore the problem and we will need a more complex solution to the problem if we wish to build a parallel version. The technique, which is used for such cases, is called time-warping, and is based on the fact that the problem of a collision between objects, which are handled by different processors, is not likely, e.g. it is a rare event. Because such collisions are rare we can accept that recovering from them is relatively costly. What time-warping does is to introduce the concept of 'undo' to the execution, whenever an object is moved the previous position is saved, if the object ends up competing for the new position with another object the object which loses the battle, the rules of which is defined by the rules of the simulation, then reverses to its previous state and recalculates the move, now with the knowledge that the first choice in fact is illegal. If the population of objects is dense such a time-warping operation may cascade to other objects, thus if time-warping should be efficient it does require that collisions are rare. Time-warping is an advanced simulation issue, and we won't pursue this technique any further in this text.

Since the WATOR model is based on randomized operations and because time-warping would be outside the scope of this text, we will base the parallel version of WATOR on a straightforward splitting of the work area into blocks, with one block per processor. While this could result in a very unbalanced execution the reality of the WATOR model is that most runs will distribute the creatures evenly across the torus and we can base our parallel version on a simple geometrical partition of the workload.

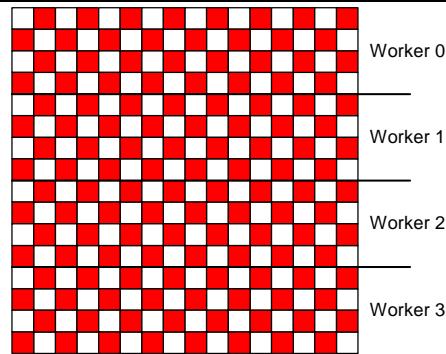


Figure 14 Splitting the WATOR world into four workers

The fastest way of porting the sequential version into a threaded version is to use the fork-join model. We simply modify the `move_fish` and `move_sharks` functions to traverse a portion of the torus, specified by a job-parameter. We then replace the call to `move_fish` by a loop, which spawns a thread per processor, wait until all threads have terminated and continue with the `move_sharks` in the same fashion.

```
typedef struct {int from, to} job;

main(){
    int blocksize=worldsize/nthreads;
    job jobs[WORKERS];
    pthread_t t[WORKERS];
    for (i=0; i<nthreads; i++){
        jobs[i].from=i*blocksize;
        jobs[i].to=(i+1)*blocksize;
    }
    jobs[nthreads-1].to=worldsize;
    initworld();
    for(i=0; i<timesteps; i++){
        for(j=0;j<nthreads;j++){
            pthread_create(&(t[j]), NULL, move_fish,(void *)(&jobs[j]));
        }
        for(j=0;j<nthreads;j++){
            pthread_join(t[j]);
        }
        for(j=0;j<nthreads;j++){
            pthread_create(&(t[j]), NULL, move_sharks,(void *)(&jobs[j]));
        }
        for(j=0;j<nthreads;j++){
            pthread_join(t[j]);
        }
    }
}
```

Example 30. Fork-join implementation of parallel WATOR

If we run this on a dual-processor machine and WATOR world size of 1000 we get a running time of dd.dd seconds, compared to a sequential time of dd.dd seconds. (conclusions on dd vs dd)

```

typedef struct {int from, to} job;

void * worker(job *myjob){
    int i;
    for(i=0; i<myjob->itt; i++){
        move_fish(myjob->from, i<myjob->to);
        barrier();
        move_sharks(myjob->from, i<myjob->to);
        barrier();
        return (void *)1;
    }
}

main() {
    int blocksize=worldsize/nthreads;
    job jobs[WORKERS];
    pthread_t t[WORKERS];

    for (i=0; i<nthreads; i++){
        jobs[i].from=i*blocksize;
        jobs[i].to=(i+1)*blocksize;
    }
    jobs[nthreads-1].to=worldsize;
    initworld();
    for (j=0; j<nthreads; j++){
        pthread_create(&t[j], NULL, worker, (void *)&jobs[j]);
    }
    for (j=0; j<nthreads; j++){
        pthread_join(t[j]);
    }
}

```

Example 31. Parallel WATOR where the threads synchronize between themselves

At this point you might wonder why the flat representation of the torus is divided in one dimension only, after all dividing the model into squares instead of stripes will reduce the number of bytes that need to be passed between the caches as we increase the number of processors. The dilemma is known as blocking vs. striping and it is not quite as simple as it might present itself. If we seek to lower the communication between the processors we wish to minimize the border area between any data-block and its neighbors. If the WATOR world is n -by- n cells then the striped version results in a $2n$ border between any job and its neighbors, while a blocked version would only result

in a border area of $4\frac{n}{\sqrt{p}}$, where p is the number of processors. The immediate problem is of course that we

require p to be a square number if we wish to deploy the straightforward tiling approach, if p is not a square number, and most often it won't be, we can devise a more flexible scheme of sub-division into tasks. Such a non-symmetric division results in more complex code, which makes it easier to introduce errors, but other from that it is simple enough and several applications exist, which use this approach. The real problem is that much of the advantage may disappear as the number of processors becomes harder to split into large integer factors. If you have a prime number of processors available you will end up with the striped version we decided on earlier. A far more interesting problem that may occur is that division of the data into tiles may result in a large degree of false sharing, so to implement a tiled version we would need a radical different memory layout in order to avoid false sharing. This is a major issue in programming shared memory systems, but to truly understand and appreciate the problem we will need a more thorough understanding of the cost of communication so we will postpone this problem until the chapter on scalable shared memory systems.

Porting the WATOR model to a tiled approach is left as an exercise.

1.5.3 OpenMP version

The WATOR application is fairly easily ported to use OpenMP rather than hand-coded parallelism. Since the basic algorithm simply traverses the data row-by-row we can simply tell OpenMP to parallelize the outer loop. The result is that we maintain the main loop as it is in the sequential version and modifies the move operations instead. It's not that simple though, we still need to tag the objects with the iteration flag we used in the POSIX threaded version in order to avoid that the same object is moved twice within the same iteration.

```

move_fish(){
//Local declarations here
#pragma omp parallel for
for(i=0; i<WORLD_SIZE; i++)
    for(j=0; j<WORLD_SIZE; j++)
        //Handle each point (i,j)

```

Example 32. OpenMP version of WATOR, only a skeleton of move_fish but move_sharks is identical for the shown code

If you consider the execution of the OpenMP version the fork-join paradigm is quite obvious, for each iteration the application will call move_fish and then fork into parallel workers which update each their portion of the matrix, the workers then join before calling move_sharks which in turn results in a fork-join step.

1.6 Exercises

- 1) Threads exist in three basic types, kernel level, user level and mixed threads. The kernel handles kernel-level threads and user-level threads are handled at the user level, while the mixed version uses both. What are the pros and cons of each solution from a supercomputing perspective?
- 2) False sharing is a problem that exist at various levels in cluster computing, and as this chapter described, also in the simplest SMP node. To verify this, write a small test program where two threads each sum up a million numbers. Ensure that in one version the target variables are on the same cache line and on another they are on different cache lines. HINTS: You do not have enough control over memory in Java to ensure the placement of variables, so if you cannot provoke the expected results use C. Also any fair compiler will use a register for the destination variable so you must somehow ensure that data is in fact written and read from memory each time. Many compilers will find out if a result is never used and optimise the whole functionality away.
- 3) This chapter has described how the processor-memory bus is likely to become a bottleneck in the system, identify:
 - a) Cases where this may happen
 - b) Approaches to avoiding this problem in software
 - c) Approaches to avoiding this using hardware
 - d) Identify other potential bottlenecks that may exist in a SMP system. Outline how they may be avoided (if possible).
- 4) The final static Monte Carlo PI example still uses a synchronized increase of the shared counter, even though this is only done once. Why is this necessary and why is it even more dangerous to forget the synchronization in this case?
- 5) The examples in this chapter came to the surprising conclusion that the correct parallel version was slightly faster than the straightforward, but faulty, version. Although the difference is small it is never the less consistent every time we run the codes. Since both worker threads need to perform one synchronized monitor call, why is this version faster than the incorrect version? And how could we make the faulty version go as fast as the correct one?
- 6) Why is the dynamic orchestrating approach preferable when the hardware needs to be shared with other applications?
- 7) Experiment with the dynamic version of the PI code, what happens as the job-size increases? And when it decreases?
- 8) Experiment with the fast static version of the PI program, what happens if you run other jobs on the machine?

- 9) The fast static version of the PI program uses one thread per CPU and while these threads are active the main thread is waiting. In the section on Java Threads two techniques were mentioned, the one where the main thread sleeps and another one where the main thread creates one less thread than there are CPUs and then performs the last task itself. Take the fast static version of the PI code and transform it so that the main thread also participates in the work! How much work would you say that this was? Did you see any performance difference?
- 10) This chapter claims that race-conditions are dangerous because they are the kind of error that is not visible in every execution but only occur under special conditions. Write a Monte Carlo simulator that simulates just this. The user should be able to specify or vary the number of threads and the percentage of the total execution time a thread spends in the critical region. How many executions do we need to see the wrong result with two threads and 10% time spent in the critical region; 10 threads and 1%, 32 threads and 0.001%?
- 11) Write a virtual Geiger-counter where a user sets as parameters the amount of radioactive material that should be tested (in number of atoms) and the probability that a given atom will fall. The virtual Geiger-counter should then use a Monte-Carlo simulator to illustrate the development in radioactivity.
- 12) The parallel version of the WATOR example is implemented using striping. Port the code to use tiling instead. Do you get any performance improvement? Why (not)?
- 13) **Small project.** The purpose of this exercise is to implement a super-simplified protein folding simulation. Real protein folds are quite complex and require much insight and considers a large number of parameters, in this super-simplified example we will use proteins of a much simpler nature. Our proteins consist of only two types of acids, Hs and Ps. The Ps are neutral and simply exist in the protein, the Hs are hydrophobic and thus likes to have other Hs as neighbors. We know that a given protein consist of a sequence of Hs and Ps, so to build a protein fold all we have to do is
 - a) Take the acid as a start
 - b) Go to a random position relative to the current; North, South, West, East, Up or Down so that the protein does not cross its own path
 - c) Place the next acid of the remaining acids at the new position
 - d) If there are more acids left to place go to b)
 - e) Calculate the strength of the fold as the number of neighbourhood relations between Hs
 - f) If the resulting fold is stronger than the best known fold then replace the old with the new

A large number of proteins must be tested before we have a good suggestion to an efficient folding of a protein, consider for yourself the number of permutations needed to exhaustively search all folds of a protein consisting of 1000 acids.

This project is an example of the derivation of Monte Carlo simulations called random-walks, but the way it is simulated is different than the PI example since each parallel task now simulates its own protein and simply competes with other threads to get the best result, this kind of parallelism is often called parameter testing, even though in this example we generate the parameters as random numbers. This project will be revisited in a later chapter.

- 14) **Large project.** The PI example is a well known but rather worthless example of Monte Carlo simulations and while there are many actual uses for Monte Carlo methods most of these require some level of understanding of the physics, chemistry, biology or other sciences. If you know of an experiment that may be simulated using Monte Carlo methods choose to implement this, otherwise here is an actual experiment will require some more programming than the previous examples, but should be understandable without intricate knowledge of glaciology. We have all seen beautiful icicles during the winter and know how they seem as individual in size and shape, but how are they formed?

A Monte Carlo Simulation can model this fairly simply. We let the icicle start as one frozen drop of water, and at random let more drops of water drip down the icicle. A given drop of water has an initial speed and an initial temperature. The more time a drop is in contact with the icicle the closer its temperature will get to the temperature of the icicle, if a drop reaches 0°C it will freeze and become part of the icicle. While a drop is still liquid it will go down the icicle, the colder it is the slower it will go, due to higher viscosity. A drop of water will usually take the direct route down the icicle, however it may at (seemingly – but not really) random choose to go another direction, i.e. left and down or right and down, if a drop encounters an obstacle, e.g. a previous drop which have frozen, the drop will have to go ‘out’ rather than down, which will improve the probability that the drop instead changes direction to a more direct path downwards.

Write a multi-threaded icicle maker using these simple rules you can vary the different parameters; icicle temperature, average and spread of drop frequency, temperature and speed, and the probabilities that describe the path taken by a drop.

The resulting icicles can be visualized with graphics, if you know how, or simply as weight and height of the icicle.

1.7 Further reading

Threads Programming:

OpenMP:

HPF:

- [1] High Performance Fortran; history, overview and current developments, Harvey Richardson, Thinking Machines Corporation, 1996.

SMP architectures:

- [2] MultiProcessor Specification Version 1.4, Intel, May 1997

A paper on Alpha, SGI or SUN SMP