**UNIT II PARALLEL PROGRAM CHALLENGES**

Performance – Scalability – Synchronization and data sharing – Data races – Synchronization

primitives (mutexes, locks, semaphores, barriers) – deadlocks and livelocks – communication

between threads (condition variables, signals, message queues and pipes).

**Performance**

**2.1 Defining Performance**

There are two common metrics for performance:

**Items per unit time.**

1. This might be transactions per second, jobs per hour, or some other combination of completed tasks and units of time. Essentially, this is a measure of bandwidth.
2. It places the emphasis on the ability of the system to complete tasks rather than on the duration of each individual task.

**Time per item.**

1. This is a measure of the time to complete a single task. It is basically a measure of latency or response time.

Many systems have a quality of service (QoS) metric that they must meet. The QoS metric will specify the expectations of the users of the system as well as penalties if the system fails to meet these expectations. These are two examples of alternative metrics:

1. Number of transactions of latency greater than some threshold.
2. The amount of time that the system is unavailable, typically called downtime or availability.

**2.1.1 Understanding Algorithmic Complexity**

1. Algorithmic complexity is a measure of how much computation a program will perform when using a particular algorithm.
2. It is a measure of its efficiency and estimate of operation count. It is not a measure of the complexity of the code necessary to implement a particular algorithm.

**Examples of Algorithmic Complexity**

Suppose you want to write a program that sums the first N numbers. You would probably write something like the code shown in Listing 2.1.

**Listing 2.1 Sum of the First N Numbers**

void sum(int N)

{

int total=0;

for (int i=1; i<=N; i++)

{

total += i;

} printf( "Sum of first %i integers is %i\n", N, total );

}

1. For a given input value N, the code will take N trips around the loop and do N additions. The algorithmic complexity focuses on the number of operations, which in this case are the N additions.
2. It assumes that any additional costs are proportional to this number. The time it would take to complete this calculation is some cost per addition, k, multiplied by the number of additions, N. So, the time would be k ∗ N.
3. The algorithmic complexity is a measure of how this time will change as the size of the input changes, so it is quite acceptable to ignore the (constant) scaling factor and say that the calculation will take of the order of N computations. This is typically written **O(N)**. It is a very useful measure of the time it will take to complete the program as the size of the input changes. If N is doubled in value, then it will take twice as long for the program to complete.

**Listing 2.2 Sum of the First N Factorials**

int factorial(int F)

{

int f = 1;

for (int i=1; i<=F; i++)

{

f = f\*i;

}

return f; }

void fsum(int N)

{

int total = 0;

for (int i=1; i<N; i++)

{

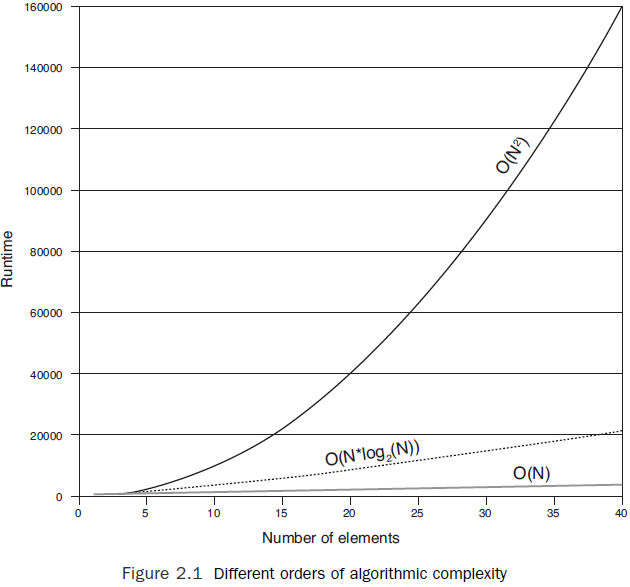
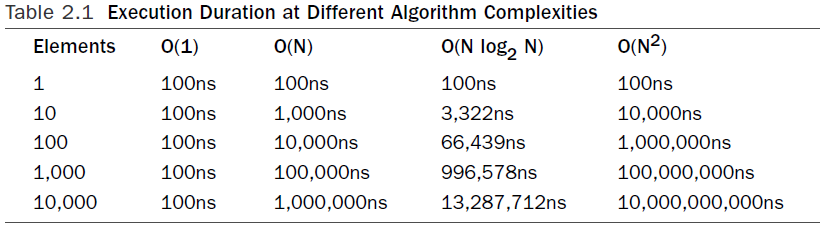
total+=factorial(i); }

}

1. This program contains a doubly nested loop. The outer loop will do N iterations, and the inner loop will do an average (over the run) of N/2 iterations.
2. Consequently, there will be about N ∗ N/2 multiply operations and N additions. The algorithmic complexity is concerned only with the dominant factor, which is N ∗ N. So, the entire calculation is O(N2). If N doubles in size, the time that this algorithm takes will go up by a factor of 4.

**Why Algorithmic Complexity Is Important**

1. Algorithmic complexity represents the expected performance of a section of code as the number of elements being processed increases. In the limit, the code with the greatest algorithmic complexity will dominate the runtime of the application.



**2.1.2 How Structure Impacts Performance**

1. Three attributes of the construction of an application can be considered as “structure.”
2. The first of these is the build structure, such as how the source code is distributed between the source files. The second structure is how the source files are combined into applications and supporting libraries. Finally, and probably the most obvious, is that way data is organized in the application

**Performance and convenience trade-offs in source code and build structures**

1. The structure of the source code for an application can cause differences to its performance. Source code is often distributed across source files for the convenience of the developers.

**Listing 2.3 Accessor Functions**

#include <stdio.h>

int a;

void setvalue( int v ) { a = v; }

int getvalue() { return a; }

void main() {

setvalue( 3 );

printf( "The value of a is %i\n", getvalue() );

}

The code in Listing 2.3 can be replaced with the equivalent but faster code shown in Listing 2.4. This is an example of inlining within a source file. The calls to the routines getvalue() and setvalue() are replaced by the actual code from the functions.

**Listing 2.4 Pseudosource Code After Inlining Optimization**

#include <stdio.h>

int a;

void main()

{ a = 3;

printf( "The value of a is %i\n", a );

}

At some optimization level, most compilers support inlining within the same source file. Hence, the transformation in Listing 2.4 is relatively straightforward for the compiler to perform. The problem is when the functions are distributed across multiple source files.

Some build methodologies reduce the ability of the compiler to perform optimizations across source files. One common approach to building is to use either static or archive libraries as part of the build process. These libraries combine a number of object files into a single library, and at link time, the linker extracts the relevant code from the library. Listing 2.7 shows the steps in this process. In this case, two source files are combined into a single archive, and that archive is used to produce the application.

**Listing 2.5 Creating an Archive Library**

$ cc -c a.c

$ cc -c b.c

$ ar -r lib.a a.o b.o

ar: creating lib.a

$ cc main.c lib.a

**2.1.3 The Impact of Data Structures on Performance**

1. Data structure is probably what most people think of first when they hear the word structure within the context of applications. Data structure is arguably the most critical structure in the program since each data structure will potentially be accessed millions of times during the run of an application.
2. When an application needs an item of data, it fetches it from memory and installs it in cache. The idea with caches is that data that is frequently accessed will become resident in the cache. The cost of fetching data from the cache is substantially lower than the cost of fetching it from memory. Hence, the application will spend less time waiting for frequently accessed data to be retrieved from memory.
3. The amount of data loaded into each level of cache by a load instruction depends on the size of the cache line. As discussed in “Using Caches to Hold Recently Used Data” in Chapter 1, 64 bytes is a typical length for a cache line;

***Out-of-order execution*** is where the processor will search the instruction stream for future instructions that it can execute. If the processor detects a future load instruction, it can fetch the data for this instruction at the same time as fetching data for a previous load instruction. Both loads will be fetched simultaneously, and in the best case, the total cost of the loads can be potentially halved.

***Hardware prefetching*** of data streams is where part of the processor is dedicated to detecting streams of data being read from memory. When a stream of data is identified, the hardware starts fetching the data before it is requested by the processor. If the hardware prefetch is successful, the data might have become resident in the cache before it was actually needed.

***Software prefetching*** is the act of adding instructions to fetch data from memory before it is needed. Software prefetching has an advantage in that it does not need to guess where the data will be requested from in the memory, because the prefetch instruction can fetch from exactly the right address, even when the address is not a linear stride from the previous address.

**Improving Performance Through Data Density and Locality**

1. Paying attention to the order in which variables are declared and laid out in memory can improve performance. As discussed earlier, when a load brings a variable in from memory, it also fetches the rest of the cache line in which the variable resides. Placing variables that are commonly accessed together into a structure so that they reside on the same cache line will lead to performance gains.

**Listing 2.6 Data Structure**

struct s {

int var1;

int padding1[15];

int var2;

int padding2[15];

}

**Listing 2.7 Reordered Data Structure So That Important Structure Members Are Likely to Share a Cache Line**

struct s {

int var1;

int var2;

int padding1[15];

int padding2[15]; }

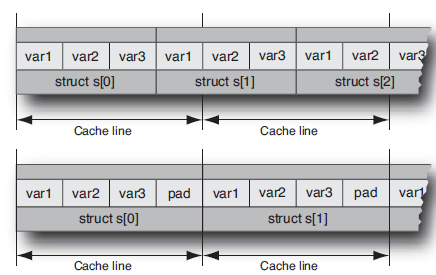


Figure 2.2 Using padding to align structures

**2.2 Synchronization and Data sharing**

1. For a multithreaded application to do useful work, it is usually necessary for some kind of common state to be shared between the threads. The degree of sharing that is necessary depends on the task. At one extreme, the only sharing necessary may be a single number that indicates the task to be performed.
2. Beyond sharing to coordinate work, there is sharing common data. For example, all threads might be updating a database, or all threads might be responsible for updating counters to indicate the amount of work completed.
3. data races, which are situations where multiple threads are updating the same data in an unsafe way. One way to avoid data races is by utilizing proper synchronization between threads.

**2.2.1 Data Races**

1. Data races are the most common programming error found in parallel code.
2. A data race occurs when multiple threads use the same data item and one or more of those threads are updating it.

Suppose you have the code shown in Listing 2.8, where a pointer to an integer variable is passed in and the function increments the value of this variable by 4.

**Listing 2.8 Updating the Value at an Address**

void update(int \* a)

{

\*a = \*a + 4;

}

The SPARC disassembly for this code would look something like the code shown in Listing 2.9.

**Listing 2.9 SPARC Disassembly for Incrementing a Variable Held in Memory**

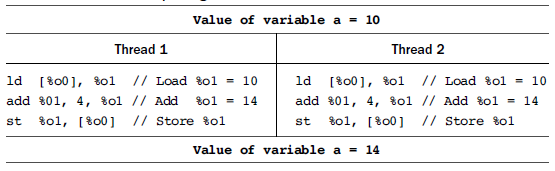
ld [%o0], %o1 // Load \*a

add %o1, 4, %o1 // Add 4

st %o1, [%o0] // Store \*a

Suppose this code occurs in a multithreaded application and two threads try to increment the same variable at the same time. Table2.1 shows the resulting instruction stream.

**Table 2.1 Two Threads Updating the Same Variable**



1. In the example, each thread adds 4 to the variable, but because they do it at exactly the same time, the value 14 ends up being stored into the variable. If the two threads had executed the code at different times, then the variable would have ended up with the value of 18. This is the situation where both threads are running simultaneously. This illustrates a common kind of data race and possibly the easiest one to visualize.
2. Consider the situation where one thread holds the value of a variable in a register and a second thread comes in and modifies this variable in memory while the first thread is running through its code. The value held in the register is now out of sync with the value held in memory.
3. The point is that a data race situation is created whenever a variable is loaded and another thread stores a new value to the same variable: One of the threads is now working with “old” data.
4. The potential for data races is part of what makes parallel programming hard. It is a common error to introduce data races into a code, and it is hard to determine, by inspection, that one exists. Fortunately, there are tools to detect data races.

**Using Tools to Detect Data Races**

The code shown in Listing 4.3 contains a data race. The code uses POSIX threads, which will be introduced in Chapter 5, “Using POSIX Threads.” The code creates two threads, both of which execute the routine func(). The main thread then waits for both the child threads to complete their work.

**Listing 2.10 Code Containing Data Race**

#include <pthread.h>

int counter = 0;

void \* func(void \* params)

{

counter++;

}

void main()

{

pthread\_t thread1, thread2;

pthread\_create( &thread1, 0, func, 0);

pthread\_create( &thread2, 0, func, 0);

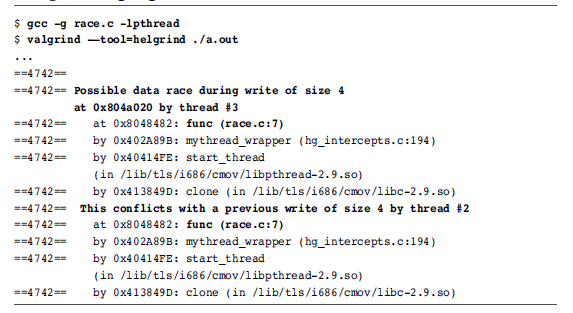
pthread\_join( thread1, 0 );

pthread\_join( thread2, 0 );

}

1. Both threads will attempt to increment the variable counter. We can compile this code with GNU gcc and then use Helgrind, which is part of the Valgrind1 suite, to identify the data race.
2. Valgrind is a tool that enables an application to be instrumented and its runtime behavior examined.
3. The Helgrind tool uses this instrumentation to gather data about data races. Listing 2.11 shows the output from Helgrind.

**Listing 2.11 Using Helgrind to Detect Data Races**



The output from Helgrind shows that there is a potential data race between two threads, both executing line 7 in the file race.c.

Another tool that is able to detect potential data races is the Thread Analyzer in Oracle Solaris Studio. This tool requires an instrumented build of the application, data collection is done by the collect tool, and the graphical interface is launched with the command tha. Listing 2.12 shows the steps to do this.

**Listing 2.12 Detecting Data Races Using the Sun Studio Thread Analyzer**

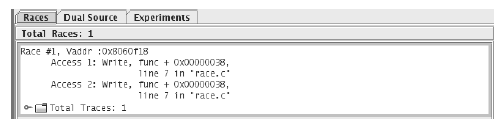
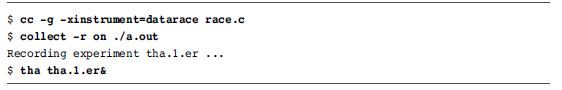
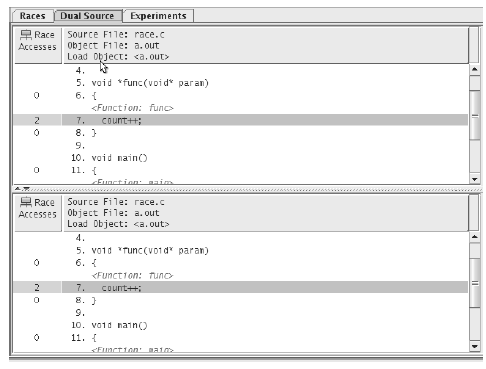


Figure 2.3 List of data races detected by the Solaris Studio Thread Analyzer

The initial screen of the tool displays a list of data races, as shown in Figure 2.3. Once the user has identified the data race they are interested in, they can view the source code for the two locations in the code where the problem occurs. In the example, shown in Figure 2.4, both threads are executing the same source line.

Figure 2.4 Source code with data race shown in Solaris Studio Thread Analyzer



**Avoiding Data Races**

1. Although it can be hard to identify data races, avoiding them can be very simple: Make sure that only one thread can update the variable at a time.
2. The easiest way to do this is to place a synchronization lock around all accesses to that variable and ensure that before referencing the variable, the thread must acquire the lock.
3. Listing 4.6 shows a modified version of the code. This version uses a mutex lock, described in more detail in the next section, to protect accesses to the variable counter.

**Listing 2.13 Code Modified to Avoid Data Races**

void \* func( void \* params )

{

thread\_mutex\_lock( &mutex );

counter++;

pthread\_mutex\_unlock( &mutex );

}

**2.3 Synchronization Primitives**

1. Synchronization is used to coordinate the activity of multiple threads.
2. There are various situations where it is necessary; this might be to ensure that shared resources are not accessed by multiple threads simultaneously or that all work on those resources is complete before new work starts.
3. Most operating systems provide a rich set of synchronization primitives. It is usually most appropriate to use these rather than attempting to write custom methods of synchronization.

**2.3.1 Mutexes and Critical Regions**

1. The simplest form of synchronization is a mutually exclusive (mutex) lock. Only one thread at a time can acquire a mutex lock, so they can be placed around a data structure to ensure that the data structure is modified by only one thread at a time. Listing 2.14 shows how a mutex lock could be used to protect access to a variable.

**Listing 2.14 Placing Mutex Locks Around Accesses to Variables**

int counter;

mutex\_lock mutex;

void Increment() {

acquire( &mutex );

counter++;

release( &mutex ); }

void Decrement()

{

acquire( &mutex );

counter--;

release( &mutex );

}

1. In the example, the two routines Increment() and Decrement() will either increment or decrement the variable counter. To modify the variable, a thread has to first acquire the mutex lock. Only one thread at a time can do this; all the other threads that want to acquire the lock need to wait until the thread holding the lock releases it. Both routines use the same mutex; consequently, only one thread at a time can either increment or decrement the variable counter.
2. If multiple threads are attempting to acquire the same mutex at the same time, then only one thread will succeed, and the other threads will have to wait. This situation is known as a ***contended mutex***.
3. The region of code between the acquisition and release of a mutex lock is called a ***critical section***, or ***critical region***. Code in this region will be executed by only one thread at a time.

**2.3.2 Spin Locks**

1. Spin locks are essentially mutex locks. The difference between a mutex lock and a spin lock is that a thread waiting to acquire a spin lock will keep trying to acquire the lock without sleeping.
2. In comparison, a mutex lock may sleep if it is unable to acquire the lock. The advantage of using spin locks is that they will acquire the lock as soon as it is released, whereas a mutex lock will need to be woken by the operating system before it can get the lock.
3. The disadvantage is that a spin lock will spin on a virtual CPU monopolizing that resource. In comparison, a mutex lock will sleep and free the virtual CPU for another thread to use.

**2.3.3 Semaphores**

1. Semaphores are counters that can be either incremented or decremented. They can be used in situations where there is a finite limit to a resource and a mechanism is needed to impose that limit.
2. An example might be a buffer that has a fixed size. Every time an element is added to a buffer, the number of available positions is decreased. Every time an element is removed, the number available is increased.
3. Semaphores can also be used to mimic mutexes; if there is only one element in the semaphore, then it can be either acquired or available, exactly as a mutex can be either locked or unlocked. Semaphores will also signal or wake up threads that are waiting on them to use available resources; hence, they can be used for signaling between threads.
4. Depending on the implementation, the method that acquires a semaphore might be called wait,down, or acquire, and the method to release a semaphore might be called post, up,signal, or release. When the semaphore no longer has resources available, the threads requesting resources will block until resources are available.

**2.3.4 Readers-Writer Locks**

1. Data races are a concern only when shared data is modified. Multiple threads reading the shared data do not present a problem. Read-only data does not, therefore, need protection with some kind of lock. However, sometimes data that is typically read-only needs to be updated.
2. A readers-writer lock (or multiple-reader lock) allows many threads to read the shared data but can then lock the readers threads out to allow one thread to acquire a writer lock to modify the data.
3. A writer cannot acquire the write lock until all the readers have released their reader locks. For this reason, the locks tend to be biased toward writers; as soon as one is queued, the lock stops allowing further readers to enter. This action causes the number of readers holding the lock to diminish and will eventually allow the writer to get exclusive access to the lock.

**Listing 2.15 Using a Readers-Writer Lock**

int readData( int cell1, int cell2 )

{

acquireReaderLock( &lock );

int result = data[cell] + data[cell2];

releaseReaderLock( &lock );

return result;

}

void writeData( int cell1, int cell2, int value )

{

acquireWriterLock( &lock );

data[cell1] += value;

data[cell2] -= value;

releaseWriterLock( &lock );

}

**2.3.5 Barriers**

1. There are situations where a number of threads have to all complete their work before any of the threads can start on the next task. In these situations, it is useful to have a barrier where the threads will wait until all are present.
2. One common example of using a barrier arises when there is a dependence between different sections of code. For example, suppose a number of threads compute the values stored in a matrix. The variable total needs to be calculated using the values stored in the matrix.
3. A barrier can be used to ensure that all the threads complete their computation of the matrix before the variable total is calculated. Listing 2.16 shows a situation using a barrier to separate the calculation of a variable from its use.

**Listing 2.16 Using a Barrier to Order Computation**

Compute\_values\_held\_in\_matrix();

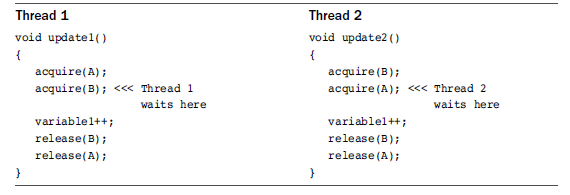
Barrier();

total = Calculate\_value\_from\_matrix();

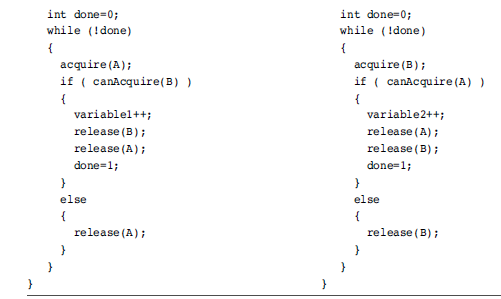
**2.4 Deadlocks and Livelocks**

1. Deadlock is the condition where two or more threads cannot make progress because the resources that they need are held by the other threads.
2. It is easiest to explain this with an example. Suppose two threads need to acquire mutex locks A and B to complete some task. If thread 1 has already acquired lock A and thread 2 has already acquired lock B, then A cannot make forward progress because it is waiting for lock B, and thread 2 cannot make progress because it is waiting for lock A. The two threads are deadlocked. Listing 2.17 shows this situation.

**Listing 2.17 Two Threads in a Deadlock**



1. The best way to avoid deadlocks is to ensure that threads always acquire the locks in the same order. So if thread 2 acquired the locks in the order A and then B, it would stall while waiting for lock A without having first acquired lock B. This would enable thread 1 to acquire B and then eventually release both locks, allowing thread 2 to make progress.
2. A livelock traps threads in an unending loop releasing and acquiring locks. Livelocks can be caused by code to back out of deadlocks.
3. In Listing 2.18, the programmer has tried to implement a mechanism that avoids deadlocks. If the thread cannot obtain the second lock it requires, it releases the lock that it already holds.
4. The two routines update1() and update2() each have an outer loop. Routine update1() acquires lock A and then attempts to acquire lock B, whereas update2() does this in the opposite order. This is a classic deadlock opportunity, and to avoid it, the developer has written some code that causes the held lock to be released if it is not possible to acquire the second lock.
5. The routine canAquire(), in this example, returns immediately either having acquired the lock or having failed to acquire the lock.



Listing 2.18 Two Threads in a Livelock

**2.5 Communication Between Threads and Processes**

1. All parallel applications require some element of communication between either the threads or the processes. There is usually an implicit or explicit action of one thread sending data to another thread.
2. For example, one thread might be signaling to another that work is ready for them. We have already seen an example of this where a semaphore might indicate to waiting threads that initialization has completed.
3. The thread signaling the semaphore does not know whether there are other threads waiting for that signal. Alternatively, a thread might be placing a message on a queue, and the message would be received by the thread tasked with handling that queue. These mechanisms usually require operating system support to mediate the sending of messages between threads or processes.

The following sections outline various mechanisms to enable processes or threads to pass messages or share data.

**2.5.1 Condition Variables**

1. Condition variables communicate readiness between threads by enabling a thread to be woken up when a condition becomes true. Without condition variables, the waiting thread would have to use some form of polling to check whether the condition had become true.
2. Condition variables work in conjunction with a mutex. The mutex is there to ensure that only one thread at a time can access the variable. For example, the producer consumer model can be implemented using condition variables.
3. Suppose an application has one producer thread and one consumer thread. The producer adds data onto a queue, and the consumer removes data from the queue. If there is no data on the queue, then the consumer needs to sleep until it is signaled that an item of data has been placed on the queue. Listing 2.19 shows the pseudocode for a producer thread adding an item onto the queue.

**Listing 2.19 Producer Thread Adding an Item to the Queue**

Acquire Mutex();

Add Item to Queue();

If ( Only One Item on Queue )

{

Signal Conditions Met();

}

Release Mutex();

1. The producer thread needs to signal a waiting consumer thread only if the queue was empty and it has just added a new item into that queue. If there were multiple items already on the queue, then the consumer thread must be busy processing those items and cannot be sleeping.
2. If there were no items in the queue, then it is possible that the consumer thread is sleeping and needs to be woken up. Listing 2.20 shows the pseudocode for the consumer thread.

**Listing 2.20 Code for Consumer Thread Removing Items from Queue**

Acquire Mutex();

Repeat

Item = 0;

If ( No Items on Queue() )

{

Wait on Condition Variable();

}

If (Item on Queue())

{

Item = remove from Queue();

}

Until ( Item != 0 );

Release Mutex();

**2.5.2Signals and Events**

1. Signals are a UNIX mechanism where one process can send a signal to another process and have a handler in the receiving process perform some task upon the receipt of the message. Many features of UNIX are implemented using signals. Stopping a running application by pressing ^C causes a SIGKILL signal to be sent to the process.
2. Windows has a similar mechanism for events. The handling of keyboard presses and mouse moves are performed through the event mechanism. Pressing one of the buttons on the mouse will cause a click event to be sent to the target window.
3. Signals and events are really optimized for sending limited or no data along with the signal, and as such they are probably not the best mechanism for communication when compared to other options.

**Listing 2.21 Installing and Using a Signal Handler**

void signalHandler(void \*signal)

{ ... }

int main()

{

installHandler( SIGNAL, signalHandler );

sendSignal( SIGNAL );

}

**2.5.3 Message Queues**

1. A message queue is a structure that can be shared between multiple processes. Messages can be placed into the queue and will be removed in the same order in which they were added. Constructing a message queue looks rather like constructing a shared memory segment.
2. The first thing needed is a descriptor, typically the location of a file in the file system. This descriptor can either be used to create the message queue or be used to attach to an existing message queue. Once the queue is configured, processes can place messages into it or remove messages from it. Once the queue is finished, it needs to be deleted.

Listing 2.22 shows code for creating and placing messages into a queue. This code is also responsible for removing the queue after use.

**Listing 2.22 Creating and Placing Messages into a Queue**

ID = Open Message Queue Queue( Descriptor );

Put Message in Queue( ID, Message );

**...**

Close Message Queue( ID );

Delete Message Queue( Description );

**2.5.4 Named Pipes**

1. UNIX uses pipes to pass data from one process to another. For example, the output from the command ls, which lists all the files in a directory, could be piped into the wc command, which counts the number of lines, words, and characters in the input. The combination of the two commands would be a count of the number of files in the directory.
2. Named pipes provide a similar mechanism that can be controlled programmatically. Named pipes are file-like objects that are given a specific name that can be shared between processes. Any process can write into the pipe or read from the pipe.
3. There is no concept of a “message”; the data is treated as a stream of bytes. The method for using a named pipe is much like the method for using a file: The pipe is opened, data is written into it or read from it, and then the pipe is closed.

**Listing 2.23** shows the steps necessary to set up and write data into a pipe, before closing and deleting the pipe. One process needs to actually make the pipe, and once it has been created, it can be opened and used for either reading or writing. Once the process has completed, the pipe can be closed, and one of the processes using it should also be responsible for deleting it.

**Listing 2.23 Setting Up and Writing into a Pipe**

Make Pipe( Descriptor );

ID = Open Pipe( Descriptor );

Write Pipe( ID, Message, sizeof(Message) );

**...**

Close Pipe( ID ); Delete Pipe( Descriptor );

**2.6 Scalability**

**Scaling with multi-core processors**

1. The key advantage of a multicore processor is that it is able to allocate more cores to solving a compute problem. Hence, to get the best out of a multicore processor, each thread of an application needs to be efficient, and the application needs to be able to effectively utilize multiple threads.
2. An ideal application will double in performance when run with two hardware threads and will quadruple in performance with four.

**Constraints to Application Scaling**

1. Most applications, when run in parallel over multiple cores, will get less than linear speedup.
2. Amdahl’s law indicates that a section of serial code will limit the scalability of the application over multiple cores.
3. There are other limitations that stop an application from scaling perfectly. These limitations could be hardware bottlenecks where some part of the system has reached a maximum capacity.
4. Adding more threads divides this total amount of resources between more consumers but does not increase the amount available.
5. Scaling can also be limited by hardware interactions where the presence of multiple threads causes the hardware to become less effective.
6. Software limitations can also constrain scaling where synchronization overheads become a significant part of the runtime.

**Performance Limited by Serial Code**

1. The serial sections of code will limit how fast an application can execute given unlimited numbers of threads. Consider the code shown in Listing . This application has two sections of code; one section is serial code, and the other section is parallel code.

#include <math.h>

#include <stdlib.h>

void func1( double\*array, int n )

{

for( int i=1; i<n; i++ )

{

array[i] += array[i-1];

}

}

void func2( double \*array,int n )

{

#pragma omp parallel for

for( int i=0; i<n; i++ )

{

array[i] = sin(array[i]);

} }

int main()

{

double \* array = calloc( sizeof(double), 1024\*1024 );

for ( int i=0; i<100; i++ )

{

func1( array, 1024\*1024 );

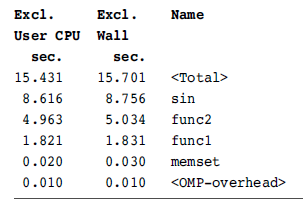
func2( array, 1024\*1024 );

}

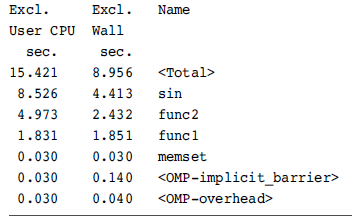
return 0;

}

**Profile of Code Run with a Single Thread**

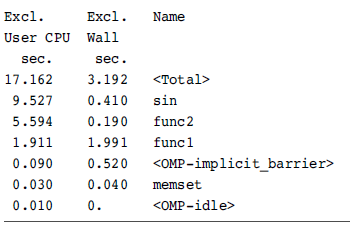


**Profile of Code Run with Two Threads**



If two threads were to run this application, we would expect each thread to take about seven seconds to complete the parallel code and one thread to spend about two seconds completing the serial code. The total wall time for the application should be about nine seconds.

**Profile of Code Run with 32 Threads**

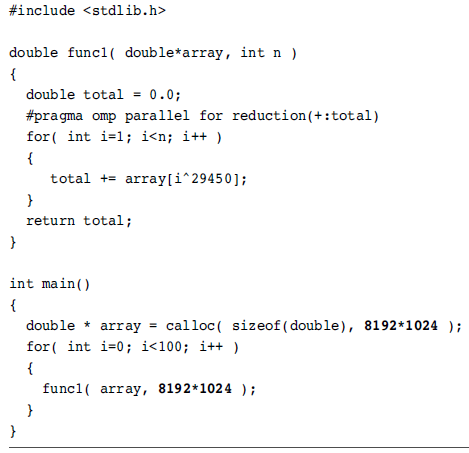


With perfect scaling, we would expect the runtime of the application with 32 threads to be 2 seconds of serial time plus 14 seconds divided by 32 threads, making a total of just under 3 seconds. The actual wall time is not that far from this ideal number. However, notice that the total user time has increased.

**Superlinear Scaling**

1. Imagine that you hurt your hand and were no long able to use both hands to type but you still had a report to finish. For most people, it would take more than twice as long to produce the report using one hand as using two. When your hand recovers, the rate at which you can type will more than double. This is an example of superlinear speedup.
2. In most instances, going from one thread to two will result in, at most, a doubling of performance. However, there will be applications that do see superlinear scaling—the application ends up running more than twice as fast. This is typically because the data that the application uses becomes cache resident at some point. Imagine an application that uses 4MB of data.
3. On a processor with a 2MB cache, only half the data will be resident in the cache. Adding a second processor adds an additional 2MB of cache; then all the data becomes cache resident, and the time spent waiting on memory becomes substantially lower.

**Program with 64MB Memory Footprint**



1. When the program is run on a single processor with 32MB of second-level cache, the program takes about 25 seconds to complete. When run using two threads on the same processor, the code completes in about 12 seconds and takes 25 seconds of user time. This is the anticipated performance gain from using multiple threads.