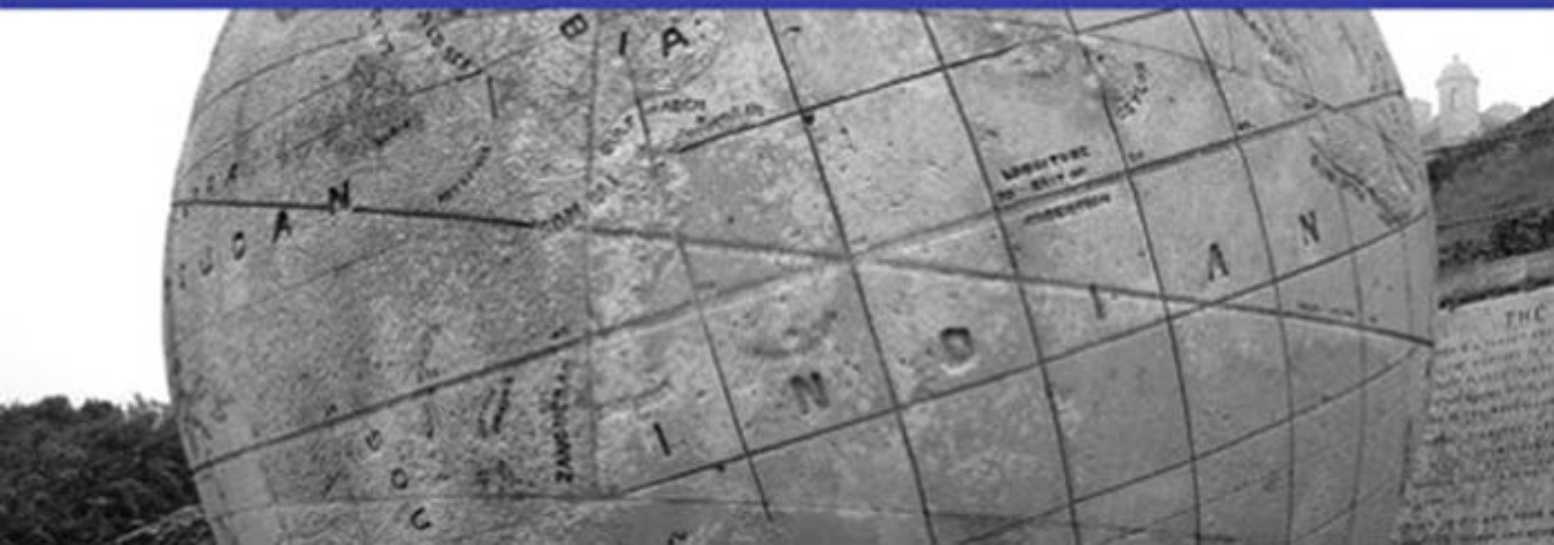
Darryl Gove

Multicore Application Programming

For Windows, Linux, and Oracle® Solaris

Developer's Library



Multicore Application Programming

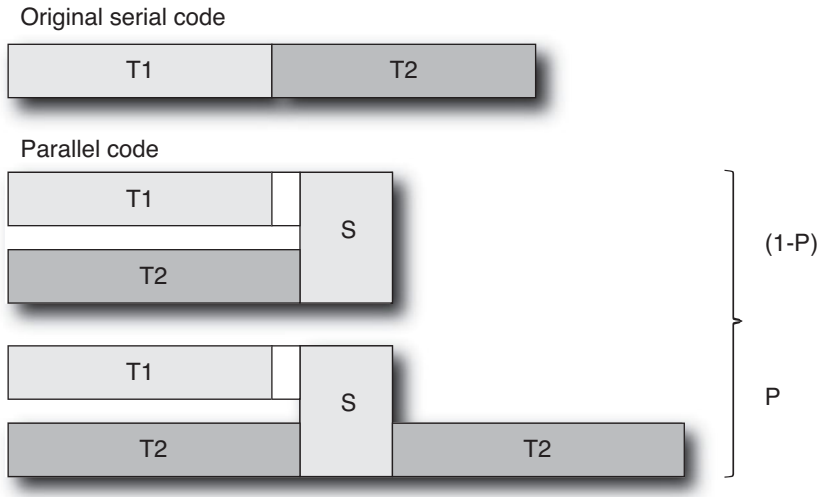


Figure 3.23 Parallelization using value speculation

costs of the two routines. In this instance, P represents the probability that the speculation is incorrect. S represents the synchronization overheads. Figure 3.23 shows the costs of value speculation.

The original code takes T1+T2 seconds to complete. The parallel code takes max(T1,T2)+S+P*T2. For the parallelization to be profitable, one of the following conditions needs to be true:

- If T1 > T2, then for the speculation to be profitable, T1 + S + P*T2 < T1 +T2. So, S < (1-P) * T2. If the speculation is mostly correct, the synchronization costs just need to be less than the costs of performing T2. If the synchronization is often wrong, then the synchronization costs need to be much smaller than T2 since T2 will be frequently executed to correct the misspeculation.
- If T2 > T1, then for the speculation to be profitable, T2 + S + P*T2 < T1 +T2. So, S <T1 – P*T2. The synchronization costs need to be less than the cost of T1 after the overhead of recomputing T2 is included.

As can be seen from the preceding discussion, speculative computation can lead to a performance gain but can also lead to a slowdown; hence, care needs to be taken in using it only where it is appropriate and likely to provide a performance gain.

Critical Paths

One way of looking at parallelization is by examining the *critical paths* in the application. A critical path is the set of steps that determine the minimum time that the task can

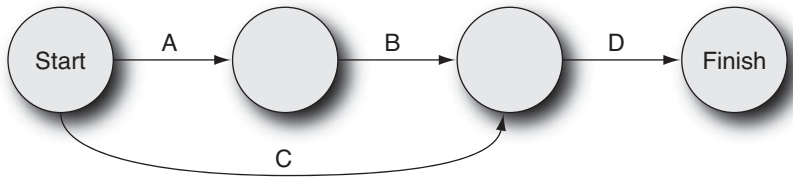


Figure 3.24 Critical paths

complete in. A serial program might complete tasks A, B, C, and D. Not all of the tasks need to have dependencies. B might depend on the results of A, and D might depend on the results of B and C, but C might not depend on any previous results. This kind of data can be displayed on a graph such as the one in Figure 3.24.

It is relatively straightforward to identify the critical path in a process once the dependencies and durations have been identified. From this graph, it is apparent that task C could be performed in parallel with tasks A and B. Given timing data, it would be possible to estimate the expected performance of this parallelization strategy.

Identifying Parallelization Opportunities

The steps necessary to identify parallelization opportunities in codes are as follows:

1. Gather a representative runtime profile of the application, and identify the regions of code where the most time is currently being spent.
2. For these regions, examine the code for dependencies, and determine whether the dependencies can be broken so that the code can be performed either as multiple parallel tasks or as a loop over multiple parallel iterations. At this point, it may also be worth investigating whether a different algorithm or approach would give code that could be more easily made parallel.
3. Estimate the overheads and likely performance gains from this parallelization strategy. If the approach promises close to linear scaling with the number of threads, then it is probably a good approach; if the scaling does not look very efficient, it may be worth broadening the scope of the analysis.
4. Broaden the scope of the analysis by considering the routine that calls the region of interest. Is it possible to make this routine parallel?

The important point to remember is that parallelization incurs synchronization costs, so the more work that each thread performs before it needs synchronization, the better the code will scale. Consequently, it is always worth looking further up the call stack of a region of code to determine whether there is a more effective parallelization point. For example, consider the pseudocode shown in Listing 3.14.

Listing 3.14 **Opportunities for Parallelization at Different Granularities**

```
void handlePacket(packet_t *packet)
{
   doOneTask(packet);
   doSecondTask(packet);
}

void handleStream( stream_t* stream )
{
  for( int i=0; i < stream->number_of_packets; i++)
  {
    handlePacket( stream->packets[i] );
  }
}
```

In this example, there are two long-running tasks; each performs some manipulation of a packet of data. It is quite possible that the two tasks, `doOneTask()` and `doSecondTask()`, could be performed in parallel. However, that would introduce one synchronization point after every packet that is processed. So, the synchronization cost would be O(N) where N is the number of packets.

Looking further up the stack, the calling routine, `handleStream()`, iterates over a stream of packets. So, it would probably be more appropriate to explore whether this loop could be made to run in parallel. If this was successful, then there would be a synchronization point only after an entire stream of packets had been handled, which could represent a significant reduction in the total synchronization costs.

Summary

This chapter has discussed the various strategies that can be used to utilize systems more efficiently. These range from virtualization, which increases the productivity of the system through increasing the number of active applications, to the use of parallelization techniques that enable developers to improve the throughput or speed of applications.

It is important to be aware of how the amount of code that is made to run in parallel impacts the scaling of the application as the number of threads increases. Consideration of this will enable you to estimate the possible performance gains that might be attained from parallelization and determine what constraints need to be met for the parallelization to be profitable.

The chapter introduces various parallelization strategies, and these should provide you with insights into the appropriate strategy for the situations you encounter. Successful parallelization of applications requires identification of the dependencies present in code. This chapter demonstrates ways that the codes can be made parallel even in the presence of dependencies.

This chapter has focused on the strategies that might be employed in producing parallel applications. There is another aspect to this, and that is the handling of data in parallel applications. The individual threads need to coordinate work and share information. The appropriate method of sharing information or synchronizing will depend on the implementation of the parallelization strategy. The next chapter will discuss the various mechanisms that are available to support sharing data between threads and the ways that threads can be synchronized.

4

Synchronization and Data Sharing

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. For example, a thread in a web server might be told only the port number to respond to. At the other extreme, a pool of threads might be passing information constantly among themselves to indicate what tasks are complete and what work is still to be completed. 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.

This chapter discusses the various methods for sharing data between threads and the costs of these approaches. It starts with a discussion of *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. This chapter provides an overview of the common approaches to data sharing supported by most operating systems. This discussion focuses, as much as possible, on the abstract methods of synchronization and coordination. The following chapters will provide implementation-specific details for the POSIX and Windows environments.

Data Races

Data races are the most common programming error found in parallel code. A data race occurs when multiple threads use the same data item and one or more of those threads are updating it. It is best illustrated by an example. Suppose you have the code shown in Listing 4.1, where a pointer to an integer variable is passed in and the function increments the value of this variable by 4.

Listing 4.1 **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 4.2.

Listing 4.2 **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. Table 4.1 shows the resulting instruction stream.

Table 4.1 **Two Threads Updating the Same Variable**

Value of variable a = 10	
Thread 1	**Thread 2**
`ld  [%o0], %o1  // Load %o1 = 10`	`ld  [%o0], %o1  // Load %o1 = 10`
`add %01, 4, %o1 // Add  %o1 = 14`	`add %01, 4, %o1 // Add %o1 = 14`
`st  %o1, [%o0]  // Store %o1`	`st  %o1, [%o0]  // Store %o1`
Value of variable a = 14	

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.

Another situation might be when one thread is running, but the other thread has been context switched off of the processor. Imagine that the first thread has loaded the value of the variable `a` and then gets context switched off the processor. When it eventually runs again, the value of the variable `a` will have changed, and the final store of the restored thread will cause the value of the variable `a` to regress to an old value.

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.