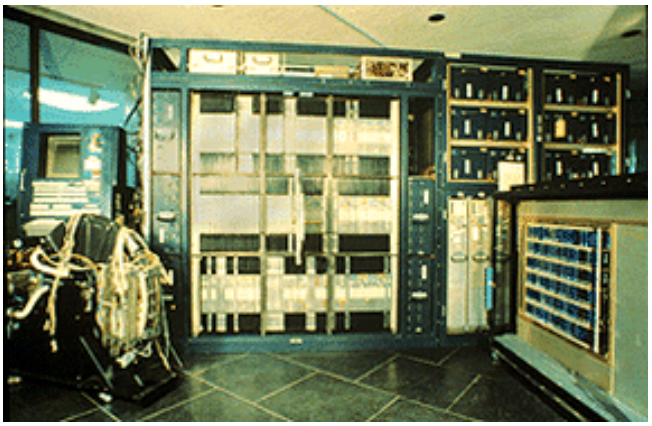
Foundations of Parallel Programming

Calvin Lin
Department of Computer Science
University of Texas, Austin
lin@cs.utexas.edu

Lawrence Snyder
Department of Computer Science and Engineering
University of Washington, Seattle
snyder@cs.washington.edu

*This is a work in progress. It is incomplete, it may be inaccurate, though obviously we don't intend for it to be, and in all likelihood, it has grammar, typing and programming errors. We are circulating it for the purpose of receiving feedback from thoughtful readers. Please send **any** comments to either of us. Thank you for your patience.*



ILLIAC-IV Early Parallel Computer

Draft: 18 September 2006

Table of contents

Chapter 1: Approaching Parallelism

In March of 2005, as techies eagerly awaited the arrival of the first dual core processor chips, Herb Sutter wrote an article in Dr. Dobbs' Journal titled, "The Free Lunch is Over: Fundamental Turn Towards Concurrency in Software." His point was that for 35 years, programmers have ridden the coattails of exponential growth in computing power. During that time, the software community has had the luxury of dealing mainly with incremental conceptual changes. The vast majority of programmers have been able to maintain the same abstract von Neumann model of a computer and the same basic notions of performance-- count instructions, sometimes worrying about memory usage. The community occasionally welcomes a new language, such as Java, and it only rarely changes the programming model, as with the movement towards the object-oriented paradigm. For the most part, however, the community has been spoiled to believe that business will continue as usual except that new generations of microprocessors will arrive every 18 months, providing more computing power and more memory.

What has caused the move to multi-core chips? Over the past 20 years, microprocessors have seen incredible performance gains fueled largely by increased clock rates and deeper pipelines. Unfortunately, these tricks are now showing diminishing returns. As silicon feature sizes have shrunk, wire delay—the number of cycles it takes a signal to propagate across a wire—has increased, discouraging the use of large centralized structures, such as those required for super-pipelined processors. Moreover, as transistor density has increased exponentially, so has power density. Power dissipation has thus become a large issue, and the use of multiple simpler slower cores offers one method of limiting power utilization. All of these trends point towards the use of multiple, simpler cores, so multi-core chips have become a commercial reality. Intel and IBM's latest high end products package 2 CPU's per chip; Sun's Niagara has 8 multi-threaded CPU's; the STI Cell processor has 9 CPU's, and future chips will likely have many more CPU's.

The advent of dual core chips, however, signifies a dramatic change for the software community. The existence of parallel computers is not new. Parallel computers and parallelism have been around for many years, but parallel programming has traditionally been reserved for solving a small class of hard problems, such as computational fluid dynamics and climate modeling, which require large computational resources. Thus, parallel programming was limited to a small group of heroic programmers. What's significant is that parallelism will now become a programming challenge for a much larger segment of programmers, as transparent performance improvements from single-core chips are now a relic of the past. In other words, the Free Lunch is over.

The Characteristics of Parallelism

Why does parallel programming represent such a dramatic change for programmers? Here are a few reasons.

- Explicitly parallel algorithms are fundamentally different from sequential algorithms, because they embody multiple points of execution.
- Programs with concurrent interactions are harder to reason about and harder to debug.
- It's harder to achieve good performance with a parallel program.
    - Small inefficiencies can lead to large performance problems.
    - It's harder to ignore low-level details.
- The performance model is different and more complex.
    - Counting instructions is insufficient.
    - Focusing on communication is insufficient.
    - The performance problem is instead an inseparable problem with multiple dimensions.
- The joint goals of portability and performance are harder to achieve.
- Tools and languages are immature.

In this book, we will explore these topics and more. As a first glimpse, the next two sections address the first two bullets, as we show that parallelism requires us to look at problems differently, and as we show that parallel programming is considerably more challenging than sequential programming.

A Paradigm Shift

Expressing a computation in parallel requires that it be thought about differently. In this section we consider several tiny computations to illustrate some of the issues involved in changing our thinking.

Summation

To begin the illustration, consider the task of adding a sequence of n data values:

$$x_0, x_1, x_2, \ldots, x_{n-1}$$

Perhaps the most intuitive solution is to initialize a variable, call it `sum`, to 0 and then iteratively add the elements of the sequence. Such a computation is typically programmed using a loop with an index value to reference the elements of the sequence, as in

```
sum = 0
for (i=0; i<n; i++) {
    sum += x[i];
}
```

This computation can be abstracted as a graph showing the order in which the numbers are combined; see Figure 1.1. Such solutions are our natural way to think of algorithms.

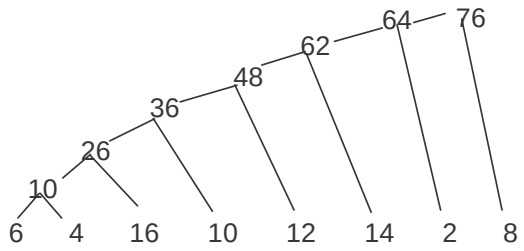


Figure 1.1. Summing in sequence. The order of combining a sequence of numbers (6, 4, 16, 10, 12, 14, 2, 8) when adding them to an accumulation variable.

Of course, addition over the real numbers is an associative and commutative operation, implying that its values need not be summed in the order specified, least index to greatest index. We can add them in another order, perhaps one that admits more parallelism, and get the same answer.

Nonassociativity. Strictly speaking, addition is not associative on floating point number's fixed precision representation. For some sequences of values, adding the numbers in different orders will produce different answers, because floating point representations only approximate real numbers. We ignore such issues and reorder computations to improve performance, reasoning that (a) under most circumstances the sequence's order was arbitrary in the first place, and, (b) in those cases where it is not arbitrary *and* numerical precision is a potential issue, error management is required throughout the computation anyway.

Another, more parallel, order of summation is to add even/odd pairs of data values yielding intermediate sums,

$$x_0 + x_1,\ x_2 + x_3,\ x_4 + x_5,\ x_6 + x_7,\ \ldots$$

which are added in pairs,

$$(x_0 + x_1) + (x_2 + x_3),\ (x_4 + x_5) + (x_6 + x_7),\ \ldots$$

yielding more intermediate sums, etc. This solution can be visualized as inducing a tree on the computation, where the original data values are leaves, the intermediate nodes are the sum of the leaves below them, and the root is the overall sum; see Figure 1.2.

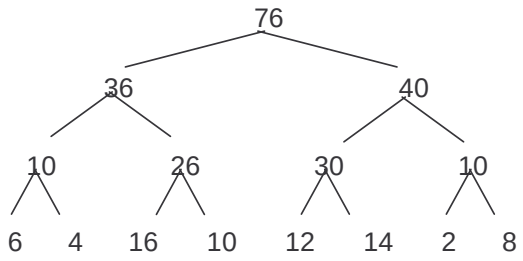


Figure 1.2. Summing in pairs. The order of combining a sequence of numbers (6, 4, 16, 10, 12, 14, 2, 8) by (recursively) combining pairs of values, then pairs of results, etc.

Comparing Figures 1.1 and 1.2, we see that because the two solutions produce the same number of computations and the same number of intermediate sums, there is no time advantage to either solution when using one processor. However, with a parallel computer that has $P=n/2$ processors, all of the additions at the same level of the tree can be computed simultaneously, yielding a solution with time complexity that is proportional to log n. Like the sequential solution this is a very intuitive way to think about the computation.

The crux of the advantage of summing by pairs is that the approach uses separate and independent subcomputations, which can be performed in parallel.

Prefix Summation

A closely related computation is the prefix sum, also known as scan in many programming languages. It begins with the same sequence of n values,

$$x_0, x_1, x_2, \ldots, x_{n-1}$$

but the desired computation is the sequence

$$y_0, y_1, y_2, \ldots, y_{n-1}$$

such that each y_i is the sum of the first i elements of the input, that is,

$$y_i = \sum_{j \le i} x_j$$

Solving the prefix sum in parallel is less obvious than summation, because all of the intermediate values of the sequential solution are needed. It seems as though there is no advantage of, nor much possibility of, finding better solutions. But the prefix sum can be improved.

The observation is that the summing by pairs approach can be modified to compute the prefix values. The idea is that each leaf processor storing x_i could compute the value, y_i, if

it only knew the sum of all elements to its left, i.e. its prefix; in the course of summing by pairs, we know the sum of all substrees, and if we save that information, we can figure out the prefixes, starting at the root, whose prefix—that is, the sum of all elements before the first one in sequence—is 0. This is also the prefix of its left subtree, and the total for its left subtree is the prefix for the right subtree. Applying this idea inductively, we get the following set of rules:

- Compute the grand total by summing pairs, as before.
- On completion, imagine the root receiving a 0 from its (nonexistent) parent.
- All non-leaf nodes receiving a value from their parent, relay that value to their left child, and send their right child the sum of the parent's value and their left child's value; these are the prefixes of their child nodes.
- Leaves add the value—the prefix—received from above.

The values moving down the tree are the prefixes for the child nodes. (See Figure 1.3, where downward moving prefix values are in red.)

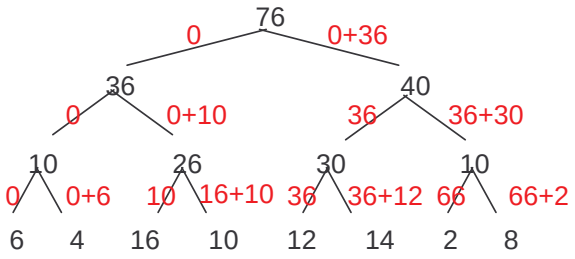


Figure 1.3. Computing the prefix sum. The black values, computed going up the tree, are from the summing by pairs algorithm; the red values, the prefixes, are computed going down the tree by a simple rule: send the value from the parent to the left; add the node's intermediate sum to the value from the parent and send it to the right.

The computation is known as the parallel prefix computation. It requires an up sweep and a down sweep in the tree, but all operations at each level can be performed concurrently. At most two add operations are required at each node, one going up and one coming down, plus the routing logic. Thus, the parallel prefix also has logarithmic time complexity.

Many seemingly sequential operations yield to the parallel prefix approach.

Parallel Programming is Challenging

Though the algorithms are different, they remain intuitive. The programming, even knowing the algorithm can be challenging.

To understand the difficulty of writing correct and efficient parallel programs, consider the problem of counting the number of 3's in an array. This computation can be trivially

expressed in most sequential programming languages, so it is instructive to see what its parallel counterpart looks like.

To simplify matters, let's assume that we will execute our parallel program on a multi-core chip with two processors, see Figure 1.4. This chip has two independent microprocessors that share access to an on-chip L2 cache. Each processor has its own L1 cache. The processors also share an on-chip memory controller so that all access to memory is equidistant" from each processor.

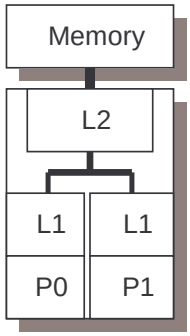


Figure 1.4. Organization of a multi-core chip. Two processors, P0 and P1, have a private L1 cache and share an L2 cache.

We will use a threads programming model in which each thread executes on a dedicated processor, and the threads communicate with one another through shared memory (L2). Thus, each thread has its own process state, but all threads share memory and file state. The serial code to count the number of 3's is shown below:

```
 1 int *array;
 2 int length;
 3 int count;
 4
 5 int count3s ()
 6 {
 7    int i;
 8    count = 0;
 9    for (i=0; i<length; i++)
10    {
11       if (array[i] == 3)
12       {
13          count++;
14       }
15    }
16    return count;
17 }
```

To implement a parallel version of this code, we can partition the array so that each thread is responsible for counting the number of 3's in 1/*t* of the array, where *t* is the number of threads. Figure 1.5 shows graphically how we might divide the work for *t*=4 threads and *length*=16.

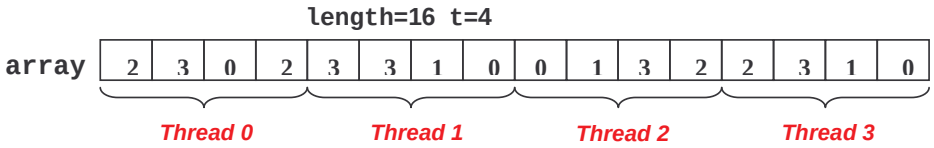


Figure 1.5. Schematic diagram of data allocation to threads. Allocations are consecutive indices.

We can implement this logic with the function `thread_create()`, which takes two arguments—the name of a function to execute and an integer that identifies the thread's ID—and spawns a thread that executes the specified function with the thread ID as a parameter. The resulting program is shown in Figure 1.6.

```
 1 int t;         /* number of threads */
 2 int *array;
 3 int length;
 4 int count;
 5
 6 void count3s ()
 7 {
 8    int i;
 9    count = 0;
10    /* Create t threads */
11    for (i=0; i<t; i++)
12    {
13       thread_create (count3s_thread, i);
14    }
15
16    return count;
17 }
18
19 void count3s_thread (int id)
20 {
21   /* Compute portion of the array that this thread should work on */
22     int length_per_thread = length/t;
23     int start = id * length_per_thread;
24
25     for (i=start; i<start+length_per_thread; i+)
26     {
27        if (array[i] == 3)
28        {
29           count++;
30        }
31     }
32 }
```

Figure 1.6. The first try at a Count 3s solution using threads.

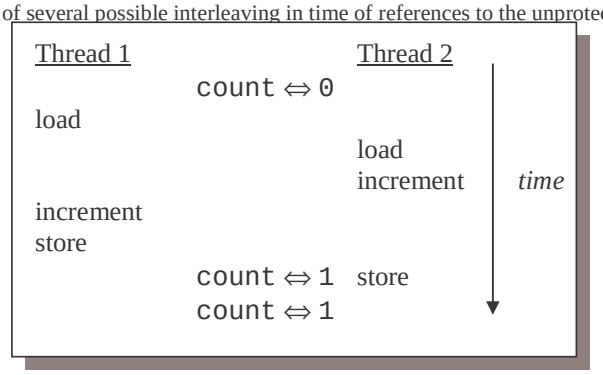
Unfortunately, this seemingly straightforward code will not produce the correct answer because there is a *race condition* in the statement that increments the value of `count` on line 29. A race condition occurs when multiple threads can access the same memory location at the same time. In this case, the problem arises because the statement that

increments `count` is typically implemented on modern machines as a series of primitive machine instructions:

- Load `count` into a register
- Increment `count`
- Store `count` back into memory

Thus, when two threads execute the `Count3s_thread()` code, these instructions might be interleaved as shown in Figure 1.7. The result of the interleaved executions is that `count` ⇔ 1 rather than 2. Of course, many other interleavings can also produce incorrect results, but the fundamental problem is that the increment of `count` is not an *atomic operation*, that is, uninterruptible.

Figure 1.7. One of several possible interleaving in time of references to the unprotected variable `count`



illustrating a race.

We can solve this problem by using a *mutex* to provide *mutual exclusion*. A mutex is an object that has two states—locked and unlocked—and two methods—`lock()` and `unlock()`. The implementation of these methods ensures that when a thread attempts to lock a mutex, it checks to see if it is presently locked our unlocked. If locked, it waits until the mutex is in an unlocked state, before locking it, that is, setting it to the locked state. By using a mutex to protect code that we wish to execute atomically—often referred to as a critical section—we guarantee that only one thread accesses the critical section at any time. For the Count 3s problem, we simply lock a mutex before incrementing `count`, and we unlock the mutex after incrementing `count`, resulting in our second try at a solution, see Figure 1.8.

```
 1 mutex m;
 2
 3 void count3s_thread (int id)
 4 {
 5   /* Compute portion of the array that this thread should work on */
 6    int length_per_thread = length/t;
 7    int start = id * length_per_thread;
 8
 9    for (i=start; i<start+length_per_thread; i+)
10    {
```

```
11       if (array[i] == 3)
12       {
13          mutex_lock(m);
14          count++;
15          mutex_unlock(m);
16       }
17    }
18 }
```

Figure 1.8. The second try at a Count 3s solution showing the `count3s_thread()` with mutex protection for the `count` variable.

With this modification, our second try is a correct parallel program. Unfortunately, as we can see from the graph in Figure 1.9, our parallel program is much slower than our original serial code. With one thread, execution time is five times slower than the original serial code, so the overhead of using the mutexs is harming performance drastically. Worse, when we use two threads, each running on its own processor, our performance is even worse than with just one thread; here lock contention further degrades performance, as each thread spends additional time waiting for the critical section to become unlocked.

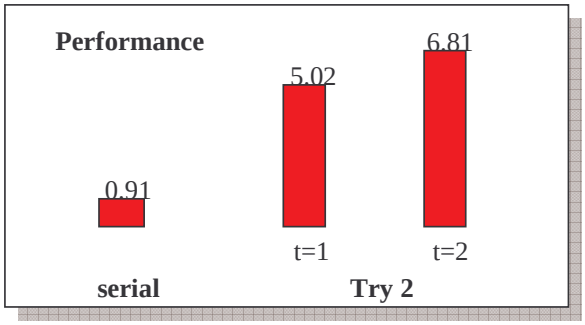


Figure 1.9. Performance of the second Count 3s solution.

Recognizing the problem of lock overhead and lock contention, we can try implementing a third version of our program that operates at a larger granularity of sharing. Instead of accessing a critical section every time `count` must be incremented, we can instead accumulate the local contribution to count in a private variable, `private_count` and only access the critical section of updating `count` once per thread. Our new code for this third solution is shown in Figure 1.10.

```
1 private_count[MaxThreads];
2 mutex m;
3
4 void count3s_thread (int id)
5 {
6   /* Compute portion of the array that this thread should work on */
7    int length_per_thread = length/t;
8    int start = id * length_per_thread;
9
```

```
10    for (i=start; i<start+length_per_thread; i++)
11    {
12       if (array[i] == 3)
13       {
14          private_count[t]++;
15       }
16    }
17    mutex_lock(m);
18    count += private_count[t];
19    mutex_unlock(m);
20 }
```

Figure 1.10. The `count3s_thread()` for the third Count 3s solution using a `private_count` array elements.

In exchange for a tiny amount of extra memory, our resulting program now executes considerably faster, as shown by the graph in Figure 1.11.

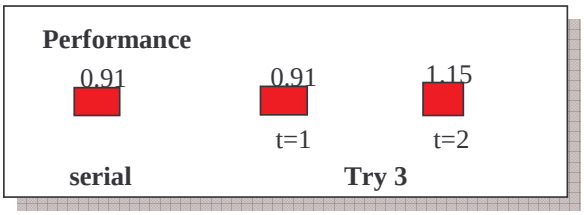


Figure 1.11. Performance results for the third Count 3s solution.

We see that with one thread our execution is the same the serial code, so our latest changes have effectively removed locking overhead. However, with two threads there is still performance degradation. This time, the performance problem is more difficult to identify by simply inspecting the source code. We also need to understand some details of the underlying hardware. In particular, our hardware uses a protocol to maintain the coherence of its caches, that is, to assure that both processors "see" the same memory image: If processor 0 modifies a value at a given memory location, the hardware will invalidate any cached copy of that memory location that resides in processor 1's L1 cache, thereby preventing processor 1 from accessing a stale value of the data. This cache coherence protocol becomes costly if two processors take turns repeatedly modifying the same data, because the data will ping pong between the two caches.

In our code, there does not seem to be any shared modified data. However, the unit of cache coherence is known as a *cache line*, and for our machine the cache line size is 128 bytes. Thus, although each thread has exclusive access to either `private_count[0]` or `private_count[1]`, the underlying machine places them on the same 128 byte cache line, and this cache line ping pongs between the caches as `private_count[0]` and `private_count[1]` are repeatedly updated. (See Figure 1.12.) This phenomenon in which logically distinct data shares a physical cache line is known as *false sharing*. To eliminate false sharing, we can pad our array of private counters so that each resides on a distinct cache line. See Figure 1.13.

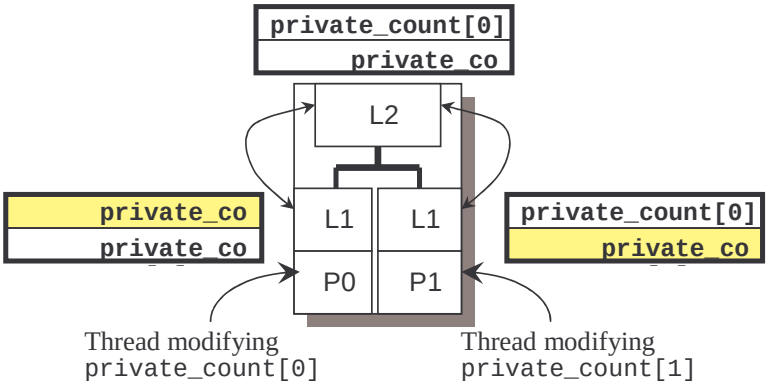


Figure 1.12. False Sharing. A cache line ping-pongs between the L1 caches and the L2 cache, because although the references to private_count don't collide, they use the same cache line.

```
 1 struct padded_int
 2 {
 3    int value;
 4    char padding[32];
 5 } private_count[MaxThreads];
 6
 7 void count3s_thread (int id)
 8 {
 9 /*Compute portion of the array this thread should work on */
10    int length_per_thread = length/t;
11    int start = id * length_per_thread;
12
13    for (i=start; i<start+length_per_thread; i++)
14    {
15       if (array[i] == 3)
16       {
17          private_count[t]++;
18       }
19    }
20    mutex_lock(m);
21    count += private_count[t].value;
22    mutex_unlock(m);
23 }
```

Figure 1.13. The `count3s_thread()` for the fourth solution to the Count 3s computations showing the private count elements padded to force them to be allocated to different cache lines.

With this padding, the fourth solution removes both the overhead and contention of using mutexes, and we have finally achieved success, as shown in Figure 1.14.

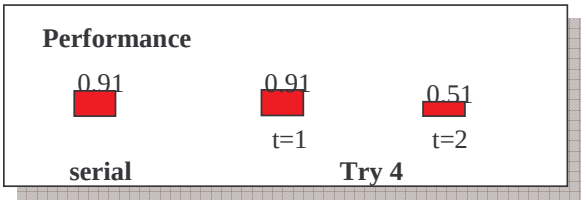


Figure 1.14. Performance for the fourth solution to the Count 3s problem shows that one processor has performance equivalent to the standard sequential solution, and two processors improve the computation time by a factor off 2.

From this example, we can see that obtaining correct and efficient parallel programs can be considerably more difficult than writing correct and efficient serial programs. The use of mutexes illustrates the need to control the interaction among processors carefully. The use of private counters illustrates the need to reason about the granularity of parallelism—that is, the frequency with which processes interact with one another. The use of padding shows the importance of understanding machine details, as sometimes small details can have large performance implications. It is this non-linear aspect of parallel performance that often makes parallel performance tuning difficult. Finally, we have seen two examples where we can trade off a small amount of memory for increased parallelism and increased performance.

The larger lesson from this example is more subtle. Because small details can have large performance implications, there is a tendency to exploit details of the specific underlying hardware. However, because performance tuning can be difficult, it is wise to take a longer term view of the problem. By creating programs that perform well across a wide variety of platforms, we can avoid much of the expense of re-writing parallel programs. For example, solutions that rely on the fact that multi-core chips have only a few cores with low latency communication among cores will need to be re-thought when future hardware provides systems with larger communication latencies.

Looking Ahead

We began this chapter by lamenting the demise of the Free Lunch, which was phrased as a steady sequence of performance improvements provided transparently to programmers by the hardware. In fact much of this performance improvement has come from parallelism. The first ALU's were bit-serial, which quickly gave way to bit-parallel ALUs. Additional parallelism in the form of further increases in data-path width produced additional performance improvements. In the 1990's we saw the introduction of pipelined processors, which used parallelism to increase instruction throughput, followed by superscalar processors that could issue multiple instructions per cycle. Most recently, processors have improved instruction throughput by executing instructions simultaneously and out of order. A key point is that all of these forms of parallelism have been hidden from the programmer. They were available implicitly for no programmer effort.

Since there are obvious benefits to hiding the complexity of parallelism, an obvious question is whether we can implement parallelism at some level above the hardware, thereby extending the Free Lunch to higher levels of software? For example, we could imagine a parallelizing compiler that transforms existing sequential programs and map them to new parallel hardware; we could imagine hiding parallelism inside of carefully parallelized library routines; or we could imagine hiding parallelism by using a functional language, which admits copious amounts of parallelism because of its language semantics. All of these techniques have been tried, but none has solved the problem to date.

The costs and benefits of hiding parallelism depend on the setting, the type of problem to be solved, etc. In some settings, a parallelizing compiler is sufficient, in others libraries may be sufficient, and in others, parallelism will simply have to be exposed to the programmer at the highest level. One goal of this book is to help readers understand parallelism so that they can answer such questions and others based on their specific needs.

Summary

This book provides a foundation for those who wish to understand parallel computing. Part 1 (Foundations) focuses on fundamental concepts. Part 2 (State of the Art) then provides a few approaches to parallel programming that represent the current state of the art. The goal is not to espouse these approaches or to describe these languages in exhaustive detail, but to provide a grounding in two low-level approaches and one high level approach so that practitioners can use them. Another goal of Part 2 is to allow researchers to appreciate the limitations of these approaches so that they can help invent the solutions that will replace them in the future. Part 3 (Hot Themes) discusses in more detail various trends in parallel computing, and Part 4 (Capstone Project) puts everything together to help instructors create a capstone project.

Exercises

1. Revise the original Summation computation along the lines of Count 3s to make it parallel.
2. Using the binary encoding of the process ID, use the concepts illustrated in the Count 3s program to Sum Pairs algorithm.

Chapter 2: Parallel Computers

If we're going to write good parallel programs, it's important to understand what parallel computers are. Unfortunately, there is considerable diversity among parallel machines, from multi-core chips with a few processors to cluster computers with many thousands of processors. How much do we need to know about the hardware to write good parallel programs? At one extreme, intimate knowledge of a machine's details can yield significant performance improvements. For example, the Goto BLAS, basic linear algebra subroutines (BLAS) are machine specific programs for core computations hand-optimized by Kazushige Goto that demonstrate enormous performance improvements. However, because hardware typically has a fairly short lifetime, it is important that our programs not become too wedded to any particular machine, for then they will simply have to be re-written when the next machine comes along. This goal of portability thus tempts us to ignore certain machine details.

To resolve this dilemma of needing to know the properties of parallel machines without embedding specifics into our programs, we will take an intermediate approach. We will first discuss essential features that we expect all parallel computers to possess, with the view that these features are precisely those that portable parallel algorithms should exploit. We then take a look at various features that are characteristic of various classes of parallel computers. We close this chapter by exploring in more detail five very different parallel computers.

First There is the RAM Model

To design parallel algorithms, we need to understand our target parallel machine. If we are to have any hope of writing portable parallel programs—specifically, *performance portable programs* that run *well* across a wide *variety* of parallel machines—then we need a single, accurate model of a parallel computer. To reason by analogy, notice that sequential computing has long benefited from such a model: The random access machine (RAM) model is an abstract machine that stores both program and data in its memory and allows one instruction to be fetched and executed at every cycle. We will use an analogous idea for the parallel case, but first, let’s review how we apply the RAM model in sequential programming.

The simplicity of the RAM model is essential, because it allows programmers to estimate overall performance based on instruction counts on the RAM model. For example, if we want to find an item (`searchee`) that might be in an array `A` of sorted items, we could use a sequential search or a binary search; see Figure 1.1. Knowing the RAM model, we know that the sequential search will take an average of $n/2$ iterations of the `for`-loop to find the desired item, and that each iteration will typically require executing fewer than a dozen machine instructions. The binary search is a slightly more complex algorithm to write, but its expected performance is approximately $log_2 n$ iterations of the `while`-loop, which will take fewer than two dozen machine instructions. For $n < 10$ or so, sequential search is likely to be fastest; binary search will be best for larger values of n.

```
1 location = -1;
2 for (j = 0; j < n; j++)
3 {
4      if (A[j] == searchee)
5      {
6        location = j;
7        break;
8      }
9 }
```

```
 1 location = -1;
 2 hi = n-1;
 3 lo = 0;
 4 while (lo != hi)
 5 {
 6   mid=lo+floor((hi–lo+1)/2);
 7  if (A[mid] == searchee)
 8      break;
 9  if (A[mid] < searchee)
10      hi = mid;
11  else
12     lo = mid+1;
13 }
```

Figure 2.1. Two searching computations; (a) linear search, (b) binary search.

The applicability of the RAM model to actual hardware is also essential, because if we had to constantly invent new models, we would have to constantly re-evaluate our algorithms. Instead, this single long-lasting model has allowed algorithm design to proceed for many years without worrying about the myriad details of each particular computer. This feat is impressive considering that hardware has enjoyed 35 years of exponential performance improvement and 35 years of increased hardware complexity.

We note, of course, that the RAM model is unrealistic. For example, the single cycle cost of fetching data is clearly a myth for current processors, as is the illusion of infinite memory, yet the RAM model works because for most purposes, these abstract costs capture those properties that are really important to sequential computers. We also note that significant performance improvements can be obtained by customizing implementations of algorithms to machine details.

And of course, the model does not apply to all hardware. In particular vector processors which can fetch long vectors of data in a single cycle do not fit the RAM model, so conventional programs written with the RAM do not fare well on vector machines. It was not until programmers learned to develop a new vector model of programming that vector processors realized their full potential.

A Parallel Computer Model

To translate the success of sequential algorithms to parallel computers, we need an idealized parallel computer that corresponds to the RAM model. Like the RAM, this model should be minimal and as universal as possible. The model that we will present is known for historical reasons as the Candidate Type Architecture, or simply the CTA.

The CTA Model

A schematic of the CTA parallel computer model is shown in Figure 2.2. It is composed of *P* standard sequential computers, called *processors* or *processor elements*, connected together by an *interconnection network*, also called a *communication network*. The processors, described by the RAM model, are composed of an execution engine and a random access memory, which stores both programs and data. The *P*+1st processor

(denoted by dashed lines) is the *controller*. Its purpose is to assist with various operations such as initialization, synchronization, eurekas, etc. Many parallel computers do not have an explicit controller, and in such cases processor p_0 serves that purpose.

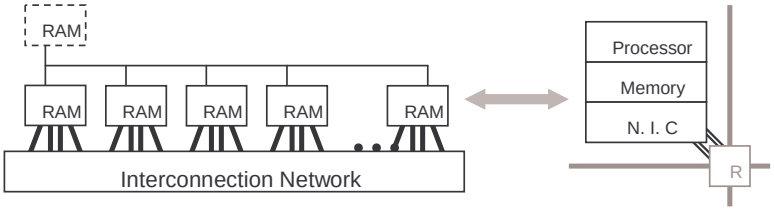
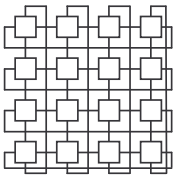


Figure 2.2. The CTA parallel computer model. (a) The schematic shows the CTA as composed of *P* sequential computers connected by a interconnection network; the distinguished (by dashed lines) computer is the controller, and serves such clerical functions as initiating the processing. (b) Detail of a RAM processor element. See the text for further details.

The processors are connected to each other by the interconnection network. These networks are built from wires and routers in a regular topology. Figure 2.3 shows several common topologies used for interconnection networks. The best topology for a parallel computer is a design-decision made by architects based on a variety of technological considerations. The topology is of no interest to programmers.



2D Torus <and others>

Figure 2.3. Common topologies used for interconnection networks; the interconnection network's topology is of little concern to programmers.

A *network interface chip* (NIC) mediates the processor/network connection. The Figure 2.2 schematic shows processors connected to the network by four wires, known as the *node degree*, but the actual number of connections is a property of the topology and the network interface design; it could be as few as one (bidirectional) connection, but typically no more than a half dozen. Data going to or coming from the network is stored in the memory and usually read or written by the direct memory access (DMA) mechanism.

Though the processors are capable of synchronizing and collectively stopping for barriers, they generally execute autonomously, running their own local programs. If the programs are the same in every processor, the computation is often referred to as *single program, multiple data*, or *SPMD* computation. The designation is of limited use, because even though the code is the same in all processors, the fact that they can each

execute different parts of it (they each have a copy of the code and their own program counter) allows them complete autonomy.[1]

Data references can be made to a processor's own *local memory*, which is supported by caches and performs analogously to standard sequential computers. Additionally, processors can reference non-local memory, that is, the memory of some other processor element. (The model has no global memory.) There are three widely used mechanisms to make non-local memory references: shared-memory, one-sided communication, which we abbreviate 1-sided, and message passing. The three mechanisms, described in a moment, place different burdens on programmers and hardware, but from the CTA machine model perspective, they are interchangeable.

A key aspect of parallel computers is that referencing the local and the non-local memory requires different amounts of time to complete. The delay required to make a memory reference is called *memory latency*. Memory latency cannot be specified in seconds, because the model generalizes over many different architectures built of different design elements from different technologies. So, latency is specified relative to the processor's local memory latency, which is taken to be unit time. This implies local memory latency roughly tracks processor rate, and we (optimistically) assume that local (data) memory can be referenced at the rate of one word per instruction. Of course, local memory reference is influenced by cache behavior and many aspects of processor and algorithm design, making it quite variable. An exact value is not needed, however.

The non-local memory latency is designated in the CTA model by the Greek letter λ. Non-local memory references are much more expensive, having λ values 2-5 orders of magnitude larger than local memory reference times. As with local memory reference, non-local references are influenced by many factors including technology, communication protocols, topology, node degree, network congestion, distance between communicating processors, caching, algorithms, etc. But the numbers are so huge that knowing them exactly is unnecessary.

Properties of the CTA

To summarize the characteristics of our abstract machine, we have:

- There are *P* processors, which are standard sequential computers executing local instructions
- Local memory access time is the usual memory access time for the sequential processor

[1] Two classifications commonly referred to in the literature, but not particularly relevant to the CTA model or our study are SIMD and MIMD. In single instruction stream, multiple data stream (SIMD) computers, there is a single program and *all* processors must execute *the same* instruction or no instruction at all. In multiple instruction stream, multiple data stream (MIMD) computers, each processor potentially has a different program to execute. Thus, MIMD and SPMD are logically equivalent: The separate MIMD programs can be conceptually unioned together into one (MIMDà SPMD); conversely, optimize the SPMD code so each processor's copy eliminates any code it never executes (SPMDà MIMD).

- Non-local memory access time, $\lambda >> 1$, can be between 2-5 orders of magnitude larger than local memory access time.
- The low node degree implies a processor cannot have more than a few (typically one or two) transfers in flight at once.
- A global controller (often only logical) assists with basic operations like initiation, synchronization, etc.

Further observations will result from a more complete look at the interconnection network below.

The consequences of these properties for programming parallel computers can be encapsulated into a simple rule:

> ***Locality Rule.*** *Fast programs tend to maximize the number of local memory references and minimize the number of non-local memory references.*

This guideline must remain foremost in every parallel programmer's thinking while designing algorithms.

> **Applying The Locality Rule**. Exploiting locality is the basis of many examples showing how parallel programming differs from sequential programming. Scalar computation is one: Imagine a computation in which the processors need a new random number *r* for each iteration of an algorithm. One approach is for one processor to store the seed and generate *r* on each cycle; then, all other processors reference it. A better approach is for each processor to store the seed locally, and to generate *r* itself on each cycle, that is, redundantly. Though the second solution requires many more instructions to be executed, they are executed in parallel and so do not take any more elapsed time than one processor generating *r* alone. More importantly, the second solution avoids non-local references, and since computing a random number is much faster than a single non-local memory reference, the overall computation is faster.

Though interprocessor communication is extremely expensive, it is helpful if programmers are aware of the effects of certain patterns of communication:

1. All processors can transmit at once; that is, communication is a parallel activity. Referring to the topologies of Figure 2.3 notice that there can in principle be a transmission along each edge simultaneously.
2. The processor graph is not complete, that is, not fully connected. Thus, some communication operations will be indirect, progressing through a series of routers.
3. Processors are only *sparsely connected*, which is a graph theoretic term implying (among other things) that the topology doesn't have the capacity to perform certain communication operations without serious congestion—all-to-all communication or transposes, for example.

Distilling the observations, (1) means a lot can be transmitted in one "communication time," (2) means that times will be sensitive to the pattern of communication, and (3)

means some patterns are much worse than others. (Some parallel computers do not have all of these properties, but we will adjust the model for them below.)

The CTA architecture mentions *P* processors, implying that the machine is intended to scale. Programmers will write code that is independent of the exact number of processors, and the actual value will be supplied at runtime. It is a fact that λ will increase as *P* increases, though probably not as fast; doubling the number of processors will usually not double λ in a well engineered computer.

In summary, the CTA is a general purpose parallel computer model that abstracts the key features of all scalable (MIMD) parallel computers built in the last few decades. Though there are variations on the theme (discussed below), the properties that the CTA exhibits should be expected of any parallel computer.

Memory Reference Mechanisms

The CTA model does not specify whether the memory referencing mechanism is by shared memory, 1-sided or message passing communication. All three are commonly used and are described in the next sections.

Shared memory

The shared memory mechanism is a natural extension of the flat memory of sequential computers. It is widely thought to be easier for programmers to use than the other mechanisms, but it has also been frequently criticized as being harder to write a fast program. Shared memory, which presents a single coherent memory image to the multiple threads, generally requires some degree of hardware support to make it perform well.

In shared memory all data items, except those variables explicitly designated as *private* to a thread, can be referenced by all threads. This means that if a processor is executing a thread with the statement

```
x = 2*y;
```

the compiler has generated code so that the processor and shared memory hardware can automatically reference `x` and `y`. Generally, every variable will have its own *home location*, the address where the compiler originally allocated it in some processor's memory. In certain implementations all references will fetch from and store to this location. In other implementations a value can float around the processor's caches until it is changed. So, if the processor had previously referenced `y`, then the value might still be cached locally, allowing a local reference to replace a non-local reference. When the value is changed, all of the copies floating around the caches must be *invalidated*, indicating that they are *stale* values, and the contents of the home location must be updated. There are variations on these schemes, but they share the property of trying to use cache hardware to avoid so many non-local references.

Notice that although it is easy for any thread to reference a memory location, the risk is that two or more threads will attempt to change the same location at the same or nearly the same time. Such "races" have a great potential for introducing difficult-to-find bugs and motivate programmers to scrupulously protect all shared memory references with some type of synchronization mechanism. See Chapter 6 for more information.

1-sided

One-sided communication, also known on Cray machines by the name *shmem*, is a relaxation of the shared memory concept as follows: It supports a single shared *address space*, that is, all threads can *reference* all memory locations, but it doesn't attempt to keep the memory coherent. This change places greater burdens on the programmer, though it simplifies the hardware because if a processor caches a value and another processor changes its home location, the cached value is not updated or invalidated. Different threads can see different values for the same variable.

In 1-sided all addresses except those explicitly designated as *private* can be referenced by all processes. References to local memory use the standard load/store mechanism, but references to non-local memory use either a `get()` or `put()`. The `get()` operation takes a memory location, and fetches the value from the non-local processor's memory. The `put()` operation takes both a memory location and a value, and deposits the value in the non-local memory location. Both operations are performed without notifying the processor whose memory is referenced. Accordingly, like shared memory, 1-sided requires that programmers protect key program variables with some synchronization protocol to assure that no processes mistakenly use stale data.

The term "one-sided" derives from the property that a communication can be initiated by only one side of the transfer.

Message Passing

The message passing mechanism is the most primitive and requires the least hardware support. Being a "two-sided" mechanism, both ends of a communication must participate, which requires greater attention from the programmer. However, because message passing does not involve shared addresses, there is no chance for races or unannounced modifications to variables, and therefore less chance of accidentally trashing the memory image. There *are* other problems, discussed momentarily.

Because there are no shared addresses, processes refer to other processes by number. (For convenience, assume one process per processor.) Processes use the standard load/store mechanism for all data references, since the only kind of reference they are allowed are local. To reference non-local data, two basic operations are available, `send()` and receive, usually abbreviated `recv()`. The `send()` operation takes as arguments a process number and the address in local memory of a *message*, a sequence of data values, and transmits the message to the (non-local) process. The `recv()` operation takes as arguments a process number and an address in local memory, and stores the message from that process into the memory. If the message from the process has not arrived prior to executing the `recv()`, the receiver process stalls until the

message arrives. There are several variations on the details of the interaction. Both sides of the communication must participate.

Notice that message passing is an operation initiated by the owner of the data values, implying that a protocol is required for most processing paradigms. For example, when a process *pr* completes an operation on a data structure and is available to perform another, it cannot simply take one from the work queue if the queue is stored on another processor. It must request one from the work queue manager, *mgr*. But that manager, to receive the request, must anticipate the situation and have an (asynchronous) `recv()` waiting for the request from *pr*. Though such protocols are cumbersome, they quickly become second nature to message passing programmers.

Programming approaches that build literally upon message passing machines are often difficult to use because they provide two distinct mechanisms for moving data: memory references are used with a local memory, and message passing is used across processes. Chapter 8 explains how higher level programming languages can be built on top of message passing machines.

Alternative Models. On encountering the CTA for the first time, it might seem complicated; isn't there a simpler idealization of a parallel computer? There is. It is called the PRAM, parallel random access machine. It is simply a large number of processor cores connected to a common, coherent memory; that is, all processors operate on the global memory and all observe the (single) sequence of state changes. Like the RAM, memory access is "unit time." One complication of the PRAM model is handling the case of two (or more) processors accessing the same memory location at the same time. For reading, simultaneous access is often permitted. For writing, there is a host of protocols, ranging from "only one processor accesses at a time" to "any number can access and some processor wins." There is a huge literature on all of these variations. The problem with the PRAM for programmers intending to write practical parallel programs is that by specifying unit time for all memory accesses, the model leads programmers to develop the "wrong" algorithms. That is, programmers exploit the unimplementable unit cost memory reference and produce inefficient programs. For that reason the CTA explicitly separates the inexpensive (local) from the expensive (non-local) memory references. Modeling parallel algorithms is a complex topic, but the CTA will serve our needs well.

Brief Overview of Parallel Computers

Though we will not need to learn the specifics of parallel architectures, we can clarify our abstract model by giving examples of real machines. In this section we consider very briefly the following implemented computers:

- Sun Fire E25K —A symmetric memory processor
- Red Storm – Commodity processors with engineered interconnect
- Cell – High performance, but heterogeneous processors
- Clusters – Building with Myrinet or Infiniband
- Blue Gene – Snazzy name, weenie processors; top dog on the Top 500

Though these machines only begin to show the variety of parallel computer architecture, they suggest the origins of our abstract machine model. <To be completed>

A Closer Look at Communication

The large non-local memory latency, λ, specified by the CTA model represents an extreme cost. To the extent that we can avoid it, our programs will run faster. Reducing its impact will be at the heart of nearly all of our programming efforts. We might wonder, "Can't something be done about reducing communication latency?" It would certainly simplify programming. In this section we consider that question.

For P processors to communicate directly with each other, that is, for processor p_i to make, say, a DMA reference to memory on processor element p_j requires that there be wires connecting p_i and p_j. A quick review of the topologies in Figure 2.3 indicates that not all pairs of processors are directly connected. Technically, no processor is directly connected to any other; every processor is at least one hop from any other because it must "enter" the network. However, if in all cases pairs of processors could communicate in one hop we could count this as a "direct" connection, that is, not requiring navigation through the network. For the topologies of Figure 2.3 information must be switched through the network and is subject to switching delays, collisions, congestion, etc. These phenomena delay the movement of the information.

For sound mathematical reasons, there are essentially two ways to make direct connections between all pairs of P processors: a bus and a crossbar; see Figure 2.4.

- In the *bus* design all processors connect to a common set of wires. When processor p_i communicates with processor p_j, they transmit information on the wires; no other pair of processors can be communicating at that time, because their signals would trash the p_i-p_j communication. Ethernet is a familiar bus design. Though there is a direct connection, a bus can only be used for one communication operation at a time; we say the communication operations are *serialized*.

- The *crossbar* overcomes the problem of one-at-a-time communication by connecting each processor to every other processor, which allows any set of distinct pairs of processors to communicate simultaneously. This is ideal from a computational perspective, but it is too expensive. The number of wires necessary to implement a crossbar grows as n^2, making it unrealistic except for very small computers, say P=16 or fewer.

With just these two basic designs available direct connection is possible only for a small number of processors, either to reduce the likelihood that communication operations contend (bus) or reduce the cost of the device (crossbar).

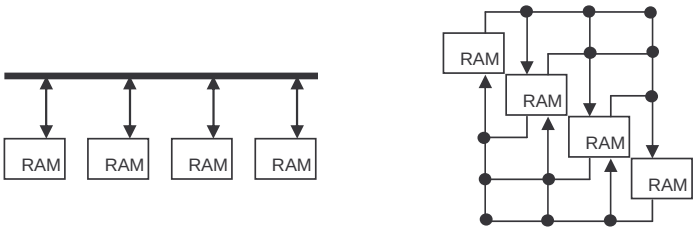


Figure 2.4. Schematics for directly connect parallel computers; (a) bus-based, (b) crossbar-based, where a solid circle can be set to connect one pair of incident wires.

Because of the difficulties of direct connections, architects have invented many communication networks with varying topologies and protocols in order to build computers that can scale. There is a large literature on the subject, and Figure 2.3 indicates only a few representatives. All of these interconnection networks provide less connectivity than the crossbar with fewer resources, and therefore more delays, but at a lower cost. The greater delays force us to adopt the large λ value.

Three Special Cases

Though scalable parallel computers are well modeled by the CTA abstract machine, three cases require us to adjust our thinking slightly:

- Symmetric Multiprocessors (SMPs) and other bus architectures
- Multicore-processor chips
- Cluster computers built with Ethernet

In all cases the issues concern how the processors are connected.

SMP Architectures

Symmetric multiprocessors are bus-based parallel computers that maintain a coherent memory image. Being bus-based implies that they are necessarily small. SMPs achieve high performance in two ways: first, by being small and necessarily clustered near to the bus, they tend to be fast; second, by using sophisticated caching protocols, SMPs tend to use the shared resource of the bus efficiently, reducing the likelihood that multiple communication operations will contend for the bus and possibly be delayed.

The bus-design prevents the SMP from matching the characteristics of the CTA. For example, the serialized use of the bus violates the "parallel communication" property; the bus effectively causes high node degree, etc. However, SMPs are well designed, and their non-local memory reference times[2] are only a small factor more expensive than their

[2] For those familiar with these architectures, non-local reference times here would refer to either a main-memory reference, or a reference that is dirty in another processor's cache.

cache hit times, which is probably the relevant distinction. Accordingly, SMPs perform better than the CTA model predicts, making it reasonable to treat them as CTA machines: The observed performance is unlikely to be worse than that predicted by the CTA model, and it will usually be much better; more importantly, algorithms that are CTA-friendly exploit locality, a property that is very beneficial to SMPs.

Multicore Chips

Being relatively new, multicore processor chips presently show a broad range of designs that make it difficult at the moment to generalize.

The Cell processor, mentioned above, has a single, general purpose core together with eight specialized cores with more SIMD operation. This architecture, originally designed for gaming, has high bandwidth communication among processors making it extremely effective at processing image data. The Cell extends to general parallel computation, too, but at the moment has not been well abstracted.

The AMD and Intel multi-core processors are more similar to each other than to the Cell, though they have significant differences. Each has multiple general purpose processors connected via the L2 cache, as illustrated in Figure 1.4. As a first approximation, both chips can be modeled as SMPs because of their coherency protocols. Because the cores can communicate faster than predicted by the CTA, their performance will tend to better than predicated by the CTA. As before, using CTA-friendly algorithms emphasizes locality, which is good for all parallel computers. Moreover, as the technology advances with more cores and greater on-chip latencies, higher non-local communication times are inevitable, making CTA-friendly algorithms even more desirable.

Cluster Computers

Cluster computers are a popular parallel computer design because they are inexpensively and easily constructed out of commodity parts, and because they scale incrementally. If the cluster is built using networking technologies, for example, Infiniband or Myrinet, to create a true interconnection network, we call it a networked cluster and observe that it is properly modeled by the CTA abstraction. If instead the cluster is built using an Ethernet for communication, then it is not. As mentioned above, Ethernet is a bus technology, and so it requires that distinct but contending communication operations be serialized. Unlike SMPs, however, the departure from the model cannot be ignored.

Specifically, the CTA models computers that have parallel communication capabilities. A practical way to think about parallel communication without knowing anything about the interconnection network, is to imagine a listing of the P processors $p_0, p_1, p_2, \ldots, p_{P-1}$, and notice that the communication properties of the CTA would permit each processor to communicate with the next processor in line simultaneously. This is possible for all of the topologies in Figure 2.3, and for almost all interconnection networks ever proposed; at worst, it is possible in as few as three λ times. A bus does not have this property, of course. The P communication operations would have to be serialized.

In the SMP case, small *P* and engineering considerations ensured that the non-local communication would only be a small factor slower than local communication time, well within out 2-5 orders of magnitude guideline for λ. For clusters, λ is large. Networked clusters are nevertheless well modeled. Ethernet clusters, however, must serialize their contending communication operations, so they do not meet the specifications of the CTA. Performance predictions for computations involving considerable communication will be low.

We will accept predictions by the CTA in the case of SMPs, because when they are wrong, the performance will be better than expected. Further, programs that accord well with the CTA will emphasize locality, which the cache-centric SMP design can exploit. But the CTA is not a good model for Ethernet clusters.

Ethernet Clusters. To get good performance from an Ethernet cluster, it is best to run programs with the characteristic that each processor is assigned a large amount of communication-free computation to perform, say λ*P* instructions worth or more, between each communication operation. Such compute-intensive problems are common. They have the property that although there can be contending communication, it will be sufficiently infrequent to give good utilization.

Applying the CTA Model

Recall that in Chapter 1 we solved the Count 3s problem. We began with a straightforward solution (Try1), found that it had a race and corrected that (Try 2), found that the terrible performance was due to a common `count` variable and corrected that (Try 3), and found that performance wasn't yet good enough due to false sharing. The final program (Try 4) is achieved our performance goal, though in Chapter 4 we'll find one more improvement to make to it.

Would the CTA have been a good guide to programming Count 3s? Yes. The CTA, being independent of the actual communication mechanism (shared memory) or caching, would not have guided us with Try 2 or Try 4, but it would have directed us to avoiding the mistake that was fixed with Try 3. The problem was the single global variable `count`, and the lock contention caused my making updates to it. The model would have told us that using a single global variable means that nearly all references will be non-local, and therefore incur λ overhead just to update the count; we would know that a better scheme would be to form a local count to be combined later. Guided by the model, the error would not have occurred and we would have written a better program in the first place.

Notice that the model predicted the problem (single global variable) and the fix (local variables), but not the exact cause. The model worried about the high cost of referencing the global variable, while the actual problem was lock contention. The different explanations are not a problem as long as the model identifies the bad cases and directs us to the correct remedy, which it did. The CTA is not a real machine. It generalizes a huge family of machines, and so cannot possibly match the implementation of each one. But to give enough information for writing quality programs, it provides general guidance as to

the operation of a parallel computer. Some implementations do have a memory latency problem referencing the global variable; some don't, but they have other problems, like contention or even stranger problems. Different implementations will manifest the fundamental behaviors of parallel computation in different ways. The CTA models behavior; it doesn't describe a physical machine.

Summary

Parallel computers are quite diverse, as the five computer profiles indicated It would be impossible to know the hardware details of all parallel machines and to write portable programs capable of running well on any platform. To solve the problem, we adopted the CTA, an abstract parallel machine, as the basis for our programming activities. Thinking of the abstract machine as executing our programs (in the same way we think of the RAM (von Neumann machine) executing our sequential programs) lets us write programs that can run on all machines modeled by the CTA, which represent virtually all multiprocessor computers.

Exercises

1. Suppose four threads performed the computations illustrated in Figure 1.1 and 1.2. (Assume a lock protected global variable permanently allocated to one thread for 1.1.) What is the communication cost, λ, predicted by the CTA for adding 1024 numbers for each computation?

   Answer. For algorithm 1.1, 256, because three of the threads make non-local references. For algorithm 1.2, 2, because all work is local until the final combining, which has two levels.
2. Like Ex. 1, but revising the Figure 1.1 algorithm so each thread keeps a local copy of the count.

   Answer. For algorithm 1.1, 1, because each three threads must update the global count.

Chapter 3: Understanding Parallelism

Introduction

The advantages of parallelism have been understood since Babbage's attempts to build a mechanical computer. Almost from the beginning of electronic computation parallel hardware has been used in the implementation of sequential computers. Efforts to build true parallel computers began in the 1970's and have continued at an accelerating pace, driven by advances in silicon technology. Industrial and academic researchers have studied every imaginable aspect of parallel computation. There is much to learn, and it cannot all be presented in complete detail in a single chapter. So, we begin with an informal tour of almost the entire parallel landscape, knowing that many sights will demand further attention in later chapters. For now, it suffices to gain an appreciation of the opportunities and challenges of parallel computation.

We look at parallelism from different perspectives. The first is performance, since improving performance is the point of parallel computation. The second perspective concerns the structural features of an algorithm that contribute to or hinder performance. Finally, we discuss general parallel problem solving approaches.

Opportunities For Performance Improvement

As the add-a-vector-of-numbers example of Chapter 1 indicates, programs can embody different amounts of parallelism despite requiring the same amount of work (in that case the same number of additions). The naïve summation loop produced a sequential specification, which if executed as specified, requires $O(n)$ time because no provision was made for other processes to contribute to the solution. The tree summation was described in a way that allows sub-computations to be performed simultaneously, which with sufficient processing capacity, would lead to an $O(\log_2 n)$ time execution. Is this the best solution available? What limitations might prevent the best performance? Are there opportunities that are not being exploited? We discuss such issues in this section.

> **Inherently Sequential**. There are computations that are inherently sequential, meaning that all algorithms to solve them have limited parallelism. One such computation is the *circuit value problem*, which takes a circuit specification over logical operators OR, AND and NOT taking *m* inputs, and an *m*-length binary sequence, and evaluates the circuit on the input sequence.

Parallelism vs. Performance

Ideally, a problem that takes T time to execute on a single processor can be solved in T/P time if we can formulate a solution to the problem that exhibits P-fold parallelism. Thus, it is tempting to think that our goal is simply to maximize parallelism, but this is not true.

Consider again the summation of Chapter 1 chapter. For n values, we maximize parallelism by using $P=n/2$ processors, which allows us in each step to perform all pair-wise additions simultaneously. The total algorithm takes $O(\log_2 n)$ time using P processors.

Now consider a variant of the algorithm, which we call the *Schwartz algorithm*. It makes each processor responsible for $log_2\ n$ data items instead of 2 items. (In Figure 3.1, the leaves, which represent data stored on the parent processor, are a total of $log_2\ n$ items.) The idea is that because the height of the summation tree is $log_2\ n$, the tree height defines the computation time; by beginning with each processor finding the sum of $log_2\ n$ local elements, the execution time is only doubled over the naïve solution. That is, in essentially the same time a significantly larger problem can be solved.

Because we are looking at this idea somewhat "backwards," let's put it into numerical terms. Adding a 1000 items using the original tree-based summation takes 10 steps (log_2 1000) using 500 threads of concurrency. If each leaf, rather than being a singleton, were a sequence of 10 items, then a 10,000 item summation could be performed by the same number of threads in 28 steps (9 for each local sum, and 10 to combine them). Using the original summation solution would have required 5,000 threads of concurrency and completed the task in 14 steps. Often, the amount of available parallelism is very small compared to the amount of data, making the idea very attractive.

Schwartz's algorithm shows that trying to maximize parallelism is not always smart. In our original algorithm to process $nlog_2\ n$ data, we would use $P= (n\ log_2\ n)/2$ processors, and we would get a running time of $O(log_2\ (nlog_2\ n)) = O(log_2\ n + loglog_2\ n)$ time. In essence, we use a larger tree having greater depth with the original algorithm. Schwartz's algorithm is not only a simple way to see that maximizing parallelism is not always smart, but it is an excellent solution technique. We will apply it often in Chapter 4.

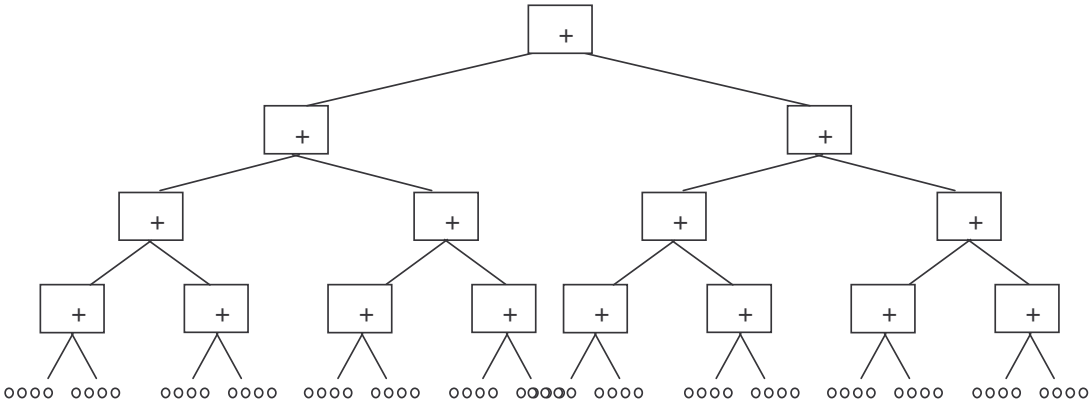


Figure 3.1. Schwartz's approach to the summation computation. Processing nodes are indicated by boxes; the leaves each represent O($log_2\ n$) items.

Our discussion of Schwartz's algorithm makes two points. First, parallelism alone is not the goal. Instead, we need to consider the resources used to exploit this parallelism. Second, when performance is the goal, we need to understand what performance means. The next two sections describe these two topics in turn.

Threads and Processes

To help us reason about the resources needed to exploit parallelism, we will use two common abstractions for encapsulating resources—threads and processes.

A *thread* refers to a thread of control, logically consisting of program code, a program counter, a call stack, and some modest amount of thread-specific data including a set of general purpose registers. Threads share access to the memory, so threads can communicate with other threads by reading from or writing to memory that is visible to them all. (Threads also share access to the file system.) Programming with threads is known as *thread-based parallel programming* or *shared memory parallel programming*.

A *process* is a thread of control that has its own private address space. When multiple processes execute concurrently, they require some mechanism for communicating with each other, since they do not share memory. One cumbersome mechanism might be to communicate through the file system, but a more direct approach is to send messages from one process to another. Parallel programming with processes is often referred to as *message passing parallel programming* or *non-shared memory parallel programming*. A key issue in message passing parallel programming is problem decomposition, since portions of the computation's data structures must be allocated to the separate process memories, that is, they usually cannot be wholly replicated within each process.

In addition to the obvious difference between threads and processes—the distinction between shared and separate memory spaces—there are also distinctions of "weight" and "agility." Threads are usually seen as "lighter weight," being created and completing dynamically throughout a computation. Processes, by contrast, are "heavier weight," taking more time to setup and tear down. Though created dynamically, usually in response to input conditions, they often persist throughout most or all of a computation. Processes can "come and go," but with the (memory) setup time being much greater, they tend to be longer lived.

Latency and Bandwidth

Since performance is the goal, it is important to agree upon what performance means. We often speak of speeding up a computation, but realize that there are two possible goals: latency and bandwidth.

Latency. Latency refers to the amount of time it takes to complete a given piece of work.

Bandwidth. Bandwidth instead refers to the amount of work that can be completed per unit time.

Thus, latency is measured in terms of time or some derivative of time, such as clock cycles. Bandwidth is measured in terms of work per unit time. The distinction between latency and bandwidth is important because they represent different issues with different solutions. For example, consider a web server that returns web pages. The web server's bandwidth can be increased by using multiple processors that allow multiple requests to be served simultaneously, but such parallelism does not reduce the latency of any

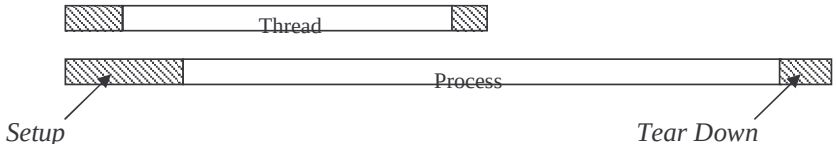
individual request. Alternatively, a web server could employ multiple physically distributed caches that can both decrease the latency of individual requests—for clients that are close to one of the caches—and increase the server's overall bandwidth. In many cases, latency can be reduced at the cost of increased bandwidth. For example, to hide long latencies to memory, modern microprocessors often perform data prefetching to speculatively bring data to caches, where its latency to processors is lower. However, because prefetching invariably brings in some data that is not used, it increases the demand for memory system bandwidth. This idea of trading bandwidth for latency is not new: The Multics operating system used the idea in the 1960s when it introduced the notion of context switching to hide the latency of expensive disk I/O.

The use of latency and bandwidth is common in some, but not all, parallel computation subcommunities, so our use of it throughout this book somewhat broadens its application. We will use latency to refer to the length of execution time or the duration of the computation, and bandwidth to refer to the capacity of a processor, its instruction execution rate. We have slightly expanded the scope of latency and bandwidth to unify terminology. There should be little confusion when encountering alternate terms in the literature.

Sources of Reduced Performance

While we ideally would hope that *P* processors could speed up a computation by a factor of *P*, there are many reasons why this might not be the case. We explore these factors in this section.

Overhead. Any cost that is incurred in the parallel solution but not in the serial solution is considered *overhead*. There is overhead in setting up threads and processes to execute concurrently and also some for tearing them down, as the following schematic indicates.



Because memory allocation and its initialization are expensive, processes incur greater setup overhead than threads. After the first process is set up, all subsequent thread and process setups incur overhead not present in a sequential computation. These costs must be charged against the benefits of parallelism; see the section, Measuring Performance below.

***Communication*.** Communication among threads and processes is a major component of overhead. Since a sequential computation doesn't have to (cannot!) communicate, all communication is a charge against the benefits of parallelism. These costs have been described in detail in Chapter 2, and though they are different depending on the communication mechanism chosen—shared memory, 1-sided or message passing—they are all substantial compared to a local memory reference. To be clear, there is always a

communication charge unless the data is local; the components of the charge are given in Table 3.1.

Synchronization. Synchronization is a form of overhead that arises when one thread or process must wait for another. Synchronization is implicit in many forms of message passing, while synchronization is often explicit when programming with threads.

Table 3.1: Sources of communication overhead by communication mechanism.

Mechanism	**Components of Communication Cost**
Shared Memory	Transmission delay, coherency operations, reference protection, unavailability
1-sided	Transmission delay, reference protection, unavailability
Message Passing	Transmission delay, data marshalling, message formation, demarshalling, unavailability

Contention. Contention is the degradation of system throughput caused by competition for a shared resource. For example, we saw in Chapter 1 how lock contention can reduce network throughput by creating excessive network traffic, and we saw how false sharing can degrade performance by causing data values to bounce back and forth among different caches.

Idle Time. When we conceptualize a parallel computation, we imagine that the processors are all working all of the time, but they might not be. The main reason is that a process or thread cannot proceed because there is no work to do or because the needed data is not yet available. As the next section on Dependences demonstrates, idle time manifests itself in many ways.

Load Imbalance. One common source of idle time is an uneven distribution of work to processors, which is known as load imbalance. For example, the Schwartz algorithm has an advantage over the standard prefix summation because the former keeps all processors busy with useful work much of the time, thereby allowing larger (by a factor of $log_2 n$) problems to be solved with the same number of processors.

Balancing load is straightforward for easy tasks like summation, but most computations are much more complex. We sometimes display the allocation of array computation, especially for the process model, by showing the array and its decomposition among processors; Figure 3.2 shows a schematic example for the LU Decomposition computation, a widely used algorithm for solving systems of linear equations. As shown in Figure 3.2(a) the LU computation builds a lower (black) and upper (white) triangle beginning at left; the area of the computation is shown in gray, and after every iteration of the computation one row and one column are added to the completed portion of the array. Figure 3.2(b) shows sixteen processors logically arranged as a grid, and (c) shows how the array might be allocated to processor memories in a process model of the computation. Though the allocation of data is balanced, i.e. each processor is assigned roughly the same number of array elements, the work is not balanced. For example, after the first 25% of the rows and columns have been added to the result arrays, there is no

more work to do for the seven processors on the left and top sides of the array. That is, nearly half of the processors will be idle after one quarter of the rows/columns have been processed. Though it is true that the amount of work per iteration diminishes as the active (gray) portion of the array shrinks, this allocation of work is still quite unbalanced. Indeed, the last 25% of the rows/columns are computed by processor P_F. Or putting it another way, the last 25% of the rows/columns are computed sequentially.

Redundant Computation. *P* processors will not speed up a sequential computation by a factor of *P* if the parallel version of the computation requires more instructions. But extra instructions are almost always required. For example, if the sequential computation requires the program to loop k times, and if the parallel computation also requires each process to loop k times, then the loop overhead instructions—initialization, incrementing, testing for termination—are not sped up by parallelism. As another example, recall the example of generating a random number from Chapter 2; although it was smart to repeat the computation to avoid non-local communication, having each process generate its own random number means there will be no parallel improvement of that portion of the computation. Of course, the programmer's goal is to make most of the computation non-redundant.

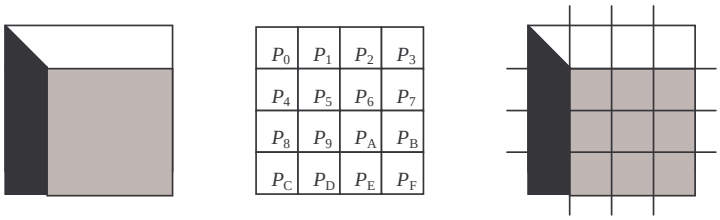


Figure 3.2: Schematic diagram of (a) the LU Decomposition algorithm, (b) sixteen processors (indexed in hexadecimal) arranged in a logical grid, and (c) the allocation of the array elements to the processors, e.g. processor P_0 is assigned that part of the array in the upper left that has completed.

Parallel Structure

By the end of the chapter we will conclude that the ideal parallel computation is one that has large blocks of independent computation that can be executed concurrently. With separate parts of the problem being performed on different processors, there will be little idle time and the solution will be found fast. To prepare to embrace "blocks of independent" computation, we must understand what "dependent" computation is. That is, our ideal case will be formed from normal computation in which we avoid certain performance limiting characteristics of programming. In this section we discuss such features in terms of the concept of dependences.

Dependences

A *dependence* is an ordering relationship between two computations. Dependences can arise in different ways in different contexts. For example, a dependence can occur between two processes when one process waits for a message to arrive from another

process. Dependences can also be defined in terms of read and write operations. Consider a program that requires that a particular memory location be read after an update (write) to the same memory location; as an example, recall the `count` variable in Figure 1.7. In this case, there is a dependence between the write operation and the read operation. If the order of the two operations is swapped, the value read would not reflect the update, so the dependence would be violated by the swap and the semantics of the program would be altered. Any execution ordering that obeys all dependences will produce the same result as the originally specified program. Thus, the notion of dependences allows us to distinguish those execution orderings that are necessary for preserving program correctness from those that are not.

Dependences provide a general way to describe limits to parallelism, so they are not only useful for reasoning about correctness, but they also provide a way to reason about potential sources of performance loss. For example, a data dependence that crosses a thread or process boundary creates a need to synchronize or communicate between the two threads or processes. By knowing the data dependence exists we can understand the consequences for parallelism even if we don't know what aspect of the computation caused the ordering relationship in the first place. To make this point more concrete, let us consider a specific type of dependence, known as data dependences.

Data dependence. A data dependence is an ordering on a pair of memory operations that must be preserved to maintain correctness. There are three kinds of data dependences:

- *Flow dependence:* read after write
- *Anti dependence:* write after read
- *Output dependence:* write after write

Flow dependences are also called *true dependences* because they represent fundamental orderings of memory operations. By contrast, anti and output dependences are collectively referred to as *false dependences* because they arise from the re-use of memory rather than from a fundamental ordering of the operations.

To understand the difference between true and false dependences, consider the following program sequence:

```
1. sum = a + 1;
2. first_term = sum * scale1;
3. sum = b + 1;
4. second_term = sum * scale2;
```

There are flow dependences (via `sum`) relating lines 1 and 2, and there are flow dependences relating lines 3 and 4. Further, there is an anti dependence on `sum` between line 2 and line 3. This anti dependence prevents the first pair of statements from executing concurrently with the second pair. But we see that by renaming `sum` in the first pair of statements as `first_sum` and by renaming the `sum` in the second pair of statements as `second_sum`,

```
1. first_sum = a + 1;
2. first_term = first_sum * scale1;
3. second_sum = b + 1;
4. second_term = second_sum * scale2;
```

the pairs can execute concurrently. Thus, at the cost of increasing the memory usage by a word, we have increased the program's concurrency. By contrast, flow dependences cannot be removed by renaming variables. It may appear that the flow dependences can be removed simply by substituting for sum in the second and fourth lines,

```
1. first_term = (a + 1) * scale1;
2. second_term = (b + 1) * scale2;
```

but this doesn't eliminate the dependence because no matter how it is expressed the addition must precede the multiplication for both terms. The flow—the write of the sum (possibly to an internal register) to the read as an operand (possibly from an internal register)—remains.

Dependences Limit Parallelism

To understand how dependences limit parallelism, recall the following code from Chapter 1, which specifies the summation of a set of *n* numbers:

```
sum = 0
for (i=0; i<n; i++) {
    sum += x[i];
}
```

This program, which we described as sequential, is abstracted in Figure 3.3(a); the more parallel tree solution is shown in 3.3(b). In the figure, an edge not involving a leaf represents a flow dependence, because the computation of the lower function will write into memory, and the upper function will read that memory. The key difference between the two algorithms is now evident. In Figure 3.3(a) the sequential solution defines a sequence of flow dependences; they are true dependences whose ordering must be respected. By contrast Figure 3.3(b) specifies shorter chains of flow dependences, imposing fewer ordering constraints and permitting more concurrency. In effect, when we gave the C specification for adding the numbers, we were specifying more than just which numbers to add. (We needed the extra fact of associativity of addition to know that the two solutions produce the same result.)

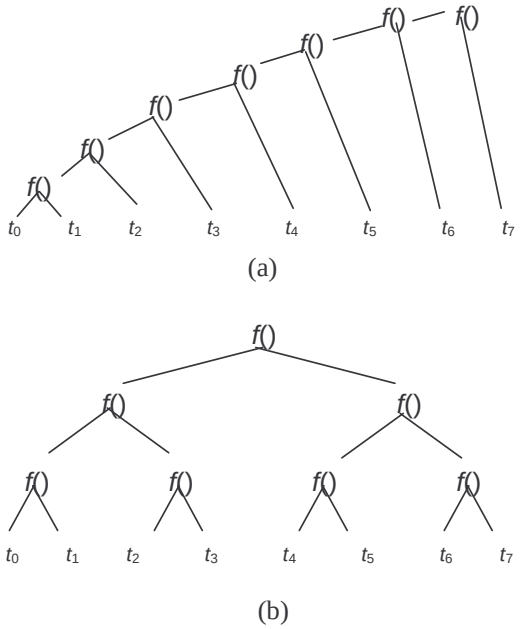


Figure 3.3. Schematic diagram of sequential and tree-based addition algorithms.

The point is that care must be exercised, when programming, to avoid introducing dependences that do not matter to the computation, because such dependences will unnecessarily limit parallelism. (Knowing that *f*() is addition allows powerful compiler techniques to transform this code into a more parallel form, but such technology has a limited scope of application.)

Granularity

A key concept for managing the constraints imposed by dependences is the notion of granularity. We identify and explain two closely related ways in which this term is used:

- Granularity of work
- Granularity of interactions

Notice that grain size is usually described using terms *coarse* and *fine*, though large and small are also used.

Granularity of Interaction. Interaction measures the frequency of dependences crossing the boundaries of threads or processes, where frequency is measured in number of useful instructions separating the interactions. Thus, coarse grain refers to threads and processes that only infrequently depend on data or events in other threads or processes, and conversely, fine grain interactions are those that occur often. As mentioned earlier dependences that cross thread or process boundaries introduce communication with its

associated overhead. Further, frequent interactions imply that waiting time can accumulate as threads and processes stall. For threads sharing through memory the cost for communicating is lower and the amount of work between interactions may be similar, suggesting that fine grain interactions may be worthwhile, especially if used in abundance. Because the overhead of message passing is typically large, processes work best with coarse grain interactions.

Granularity of Work. Work is usually measured by such things as number of instructions executed, or number of data values assigned to a thread or process. Accordingly, a coarse grain computation has a large time and/or memory footprint. Conversely, a fine grain computation has only few values processed locally and contributes mainly by being used in large quantity. Consistent with earlier points, threads often support fine grain parallelism and processes support course grained parallelism. Other semantic nuances include the sense that fine grain computations are more flexible, being available for smaller opportunities for parallelism. By contrast, coarse grain computations can provide better opportunities for amortizing overhead and hiding latency, as we discuss below.

Applying Granularity Concepts. The key point is that no fixed granularity is best for all situations. Instead, it is important to match the granularity of the computation with both the underlying hardware's available resources and the solution's particular needs. For example, the original prefix summation described in Chapter 1 was a fine grain computation involving a small amount of work and fine grain interactions with the adjacent threads. The Schwartz variant of the computation increased the grain size at the start of the computation, performing much more work before communicating. This larger granularity led to better performance. Notice that the fine grain interaction remains in the "accumulation" part of the Schwartz computation. To "coarsen" this part of the computation, the degree of the tree must be increased, where the degree, presently 2, is the number of children of each parent. For other problems a coarse granularity might lead to poor load balance.

In the limit the coarsest computations involve huge amounts of computation and no interaction. SETI@home is such an example. Subproblems are distributed to personal computers and solved entirely locally; the only communication comes at the end to report the results. In this setting the parallel computer can be an Internet-connected collection of PCs. Such super-coarse grain is essential because of the huge cost of communication.

At the other end of the spectrum are threads running on Chip Multiprocessors (CMPs) that provide low latency communication among processors that reside on the same chip, making fine grain threads practical.

Most parallel computation falls between these extremes.

Locality

A concept that is closely related to granularity is that of locality. Computations can exhibit both temporal locality—memory references that are clustered in time—and spatial locality—memory references that are clustered by address. Recall that locality is an

important phenomenon in computing, being the reason why caches work, so improving locality in a program is always a good thing. Of course, the processors of parallel machines also use caches, so all of the benefits of temporal and spatial locality are available. Keeping references local to a thread or process ensures that these benefits will be realized. Indeed, algorithms like the Schwartz approach that operate on blocks of data rather than single items, virtually always exploit spatial locality, and are preferred.

In the parallel context, locality has the added benefit of minimizing dependences among threads or processes, thereby reducing overhead and contention. As outlined above, non-local references imply some form of data communication, which is pure overhead that limits parallel performance. Furthermore, by making non-local references, the threads or processes will often contend with each other somewhere in the execution, either colliding on the shared variable in the case of threads or colliding in the interconnection network in the case of processes. Thus, non-locality has the potential of introducing two kinds of overhead.

A simple example makes both parts clear: Consider a set of threads Counting 3s in a large set of numbers using the scalable algorithm (Try 4 in Chapter 1); by working on a contiguous block of memory, a thread exploits spatial locality; by making the intermediate additions to a local accumulation variable, it benefits from temporal locality. Moreover, by combining with the global variable at the end rather than with each addition, it reduces the number of dependences among threads until the communication is absolutely necessary to achieve the final result. With the reduced number of dependences, locality is improved while overhead and contention are reduced. Note that this use of a local accumulation variable is another example of using a small amount of extra memory to break false dependences.

Forms of Parallelism

Though we have distinguished between thread-based parallelism and process-based parallelism, we have done so to focus on implementation differences, such as granularity and communication overhead. Now we are concerned with understanding where the parallelism can be found at the algorithmic level. We recognize three general types:

- data parallelism
- task parallelism
- pipelining

We now consider each, realizing that there is overlap among the categories.

Data Parallelism

Data parallelism refers to a broad category of parallelism in which the same computation is applied to multiple data items, so the amount of available parallelism is proportional to the input size, leading to tremendous amounts of potential parallelism. For example, the first chapter's "counting the 3s" computation is a data parallel computation: Each element must be tested equal to 3, which is a fully parallel operation. Once the individual outcomes are known, the number of "trues" can be accumulated using the tree summation technique. Notice that the tree add applies to all result elements only for its initial step

and has logarithmically diminishing parallelism thereafter. Still, the parallelism is generally proportional to the input size, so global sum is considered to be a data parallel operation.

As we observed in our discussion of locality and granularity above, the availability of full concurrency does not imply that the best algorithms will use it all. The Schwartz algorithm showed that foregoing concurrency to increase locality and reduce dependences with other threads produces a better result. Indeed, one of the best features of data parallelism is that it gives programmers flexibility in writing scalable parallel programs: The potential parallelism scales with the size of the input, and since, usually, $n >> P$, programs must be designed to process more data per processor than one item. That is, the program should be able to accommodate whatever parallelism is available. (It has been claimed that writing programs as if $n == P$ leads to effective programs because processors can be virtualized, i.e. the physical processors can simulate any number of logical processors, leading to code—it's claimed—that adapts well to any number of processors. This is not our experience. Virtualizing processors leads to extremely fine grain specifications that miss both the benefits of locality and the "economies of scale" of processing a batch of data. We prefer solutions like Schwartz's that explicitly handle batches of data.)

Task Parallelism

The broad classification of task parallelism applies to solutions where parallelism is organized around the functions to be performed rather than around the data. The term "task" in this case is not to be contrasted necessarily to "thread" as we normally do, because the emphasis is on the functional decomposition, which could be implemented with either tasks or threads.

For example, a client-server system employs task parallelism by assigning some tasks the job of making requests and others the job of servicing requests. As another example, the sub-expressions of a functional program can be evaluated in any order, so functional programs naturally exhibit large amounts of task parallelism. Though it is common for task parallel computations to apply an operation to similar data, as data parallel computations do, the task parallel approach becomes desirable when the context in which the data is evaluate matters significantly.

The challenges to task parallelism are to balance the work and to insure that all the work contributes to the result. In many cases, task parallelism does not scale as well as data parallelism.

Pipelining

Pipelined parallelism is a special form of task parallelism where a problem is divided into sub-problems, which can each be operated on independently, and where there are multiple problem instances to be solved. At any point in time, multiple processes can be busy, each working on a sub-problem of a different problem instance. As is familiar with bucket brigades, assembly lines, and pipelined processors, the solution is to run the operations concurrently, but on different problem instances. As the pipeline fills and

drains, there is less than full parallelism, as the opportunities for concurrency increase (fill) and then diminish (drain). A more crucial issue is the balancing of work of each operation. For pipelining to be maximally effective, the operations (stages) must complete in the same amount of time. Pipeline performance is determined—even for pipelines that are not clocked—by the longest running stage. Balancing the stages equals out the work, allowing all stages to process at the maximum prevailing rate.

Though pipelining is frequently thought of as a parallelism approach for cases defined by only a fixed length sequence of operations, it arises more generally. The number of (potential) stages is often determined by the input size. In such cases data dependences entail receiving input value(s) from one or more neighbors, computing, and then yielding the result(s) to opposite neighbor(s). The schematic in Figure 3.4 illustrates the idea. Clearly, in addition to maximizing the use of the processors, such computations are challenging in terms of avoiding stalls caused by fine grain interactions.

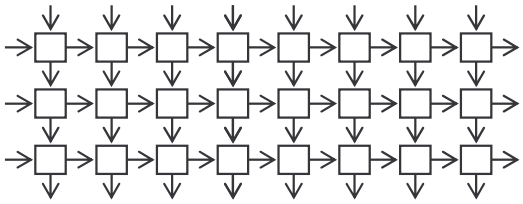


Figure 3.4. Schematic of a 2-dimensional pipelined computation, showing computation (boxes) and data flow (arrows). External data is presumed to be initially present; on the first step only the upper-left computation is enabled.

Summary

In this chapter we have introduced many concepts briefly. The goal has been to become aware of opportunities and challenges to parallel programming. Because the concepts interact in complex ways, it is not possible to understand them completely when treated in isolation. Rather, we have introduced them all in a quick, albeit limited, overview of the issues, and have prepared ourselves for the next chapter where we will develop algorithms and see first hand the consequences of these complexities.

Exercises

1. In transactional memory systems, a thread optimistically assumes that it makes no references to shared data. The transaction either *commits* successfully if there was no shared access detected, or the transaction *rolls back* if there was. Identify the sources of performance loss in a transactional memory system, classifying each as overhead, contention, or idle time.

2. Should contention be considered a special case of overhead? Can there be contention in a single-threaded program? Explain.

3. Should idle time be considered a special case of overhead? Can there be idle time in a single-threaded program? Explain.

4. Does a chess program provide data parallelism or task parallelism?

5. Does quicksort provide data parallelism or task parallelism?

6. Describe a program whose speedup does not increase with increasing problem size.

Bibliographic Notes

Schwartz's algorithm has been discovered later by theoreticians who have given it a different name. [Need to look up this name.] J.T. Schwartz, "Ultracomputers", ACM Transactions on Programming Languages and Systems, 2(4):484-521, 1980

Chapter 4: General Algorithmic Techniques

To become effective programmers, we need to learn a programming language and how to use it to express basic problem solving techniques like building data structures. We must learn how to analyze programs to determine their running time and memory usage. These will be topics for future chapters. For now, perhaps the most important understanding to acquire is the ability to "think about the computation 'right'." That is, we want to think about solving problems in a way that matches well the languages and computers available to us. In this chapter we learn the 'right' way to think about parallel computation.

What Is The Opposite of Sequential?

Many researchers have claimed that the best way to think about parallel computation is to think about the most parallel solution imaginable assuming an unlimited number of processors. They acknowledge that unlimited capacity is not realistic, but—their argument goes—it is possible to "scale back" parts of the computation to be sequential, and arrive at an ideal solution.

So, for example, return to the problem from Chapter 1 in which we want to count the number of 3s in an array `A`. Using the maximum parallelism approach, we expect a solution in which one processor initializes the count value

```
count = 0;
```
Performed by p_0

and then processor i, assigned to the i^{th} data element, performs the operations

```
if (A[i] == 3) count = count + 1;
```
Performed by p_i

Such a specification makes sense from an individual data element's point of view, perhaps, but not when viewed more globally, because processors can collide when referencing `count`. Though advocates of the unlimited parallelism approach have addressed the issue of collisions with everything from "It's an error" to "It's OK, thanks to special (Fetch & Add) hardware," we know from our discussion in Chapters 1 and 2 that there are difficulties that can arise with existing parallel computers:

- races can occur caused by the action of other processors changing `count` between the time processor p_i accesses it to get its value and the time p_i updates it
- the possibility of races implies that `count` must be protected by a lock
- the need for a lock implies the potential for lock contention when `A` contains many 3s and many processors attempt to update it simultaneously
- lock contention results in lock access being serialized
- serializing locks implies that for an array of mostly 3's the execution time is $O(n)$ regardless of the number of processors available.

There may be different "unlimited parallelism" solutions, but this is an obvious one; it does not lead to a very parallel result.

The great body of literature on unlimited parallelism comes from a study of parallel models of computation collectively known as PRAMs, acronymic for Parallel Random Access Machines, though many other unlimited parallelism approaches have been invented as well. The problem with these approaches is that finding parallelism is usually *not* the difficult aspect of parallel programming. Rather—and this is our motive for introducing the topic here—parallel programming is generally concerned with *the consequences of parallel threads interacting*, as the bulleted items just illustrated. These are the dependences discussed in Chapter 3. They arise when processors must access shared resources and when processors contend for, and therefore must wait on, shared resources. Thread interaction influences performance as much as the amount of concurrent work embodied in a problem, often more so. To be effective parallel programmers, we need to focus on the right part of the problem, and that is on the interactions between parallel threads.

Blocks of Independent Computation

If dependences between interacting threads are a significant problem, then the ideal parallel computation must be one composed of large blocks of independent computation with no interactions at all. Such computations exist: SETI@home, the Search for Extra Terrestrial Intelligence, is typical; independent computational tasks are downloaded to participants' idle PCs, computed, and the results returned to the server, which compiles the results. Other tasks from Monte Carlo simulations to integer factorization have these same features. They may be ideal, but they are not typical; nearly all parallel computations require that threads interact, and the amount of interaction is correlated with the amount of parallelism.

General parallel computations, though more complicated, still benefit whenever they can exploit the blocks-of-independent-computation strategy. Our Count 3s solution from Chapter 1 used this approach. The final solution (Try 4) partitioned the array among several threads, allocated a local variable `private_count` to each thread to record intermediate progress, and at the end combined the local results to compute the global result. The application of the principle of blocks-of-independent-computation is evident. Further, our initial tries at solving the problem were largely aimed at neutralizing the consequences of thread-to-thread dependences: races were avoided with locks, contention was removed with the private variables, false sharing was avoided with padding, etc. And as the experimental data showed, the program performed. This is one example of many that we will see of an important principle:

Guideline #1. Parallel programs are better designed when they emphasize (large) blocks of independent computation that minimize the interthread dependences (interactions).

Though our final Count 3s result was quite satisfactory, it was not actually *scalable*; that is, capable of executing well for any amount of parallelism. True, the number of threads was a parameter, so if the number of parallel processors P is greater than one, then the solution partitions the array into blocks, and P of these can execute concurrently. It is fully parallel during the scan of the data array. But there is potential for lock contention

during the final step of combining the `private_count` variables. If *P* is not likely to be a large number, then any serialization due to lock contention is not likely to be a serious problem; if *P* could be large, however, lock contention could harm performance. So, to make the solution more scalable, we combine the `private_count` variables pairwise in a tree, using the tree addition algorithm. This solution gets good performance using any number of processors, though when $P > n/\log n$, the final combining tree may be deeper than necessary, implying that using so many processors is not making the computation faster. (We discuss such tuning issues later in Chapter 5.)

Guideline #2. Just as algorithms are written to be independent of the number of input values, *n*, parallel algorithms should also be written to be independent of the number of parallel threads, *P*, and be capable of improved performance using additional processors.

Finally, reviewing this last version of Count 3s computation, notice that it is really just a simple variation of the Schwartz computation. Recall that the Schwartz algorithm was designed to add array elements, but testing and tallying those elements that equal 3 is a trivial variation. The Schwartz algorithm processes a block of elements locally, as our Count 3s program does. And to produce the final result the Schwartz algorithm uses a tree to combine the intermediate results, as our revised Count 3s solution does. Finally, the range of values over which *P* can vary is the same, as are the considerations of using more or fewer processors.

In summary, as we create parallel algorithms, we will attempt to formulate them as blocks of maximally independent computation, where "maximally independent" means that we try to reduce the interactions (dependences) among the threads. This is a challenging task, and we will often find that our best attempts do not attain our performance goals. Fortunately, there are many techniques like Schwartz's approach that give us direction and ideas for solving problems in parallel.

Assigning Work To Processors Statically

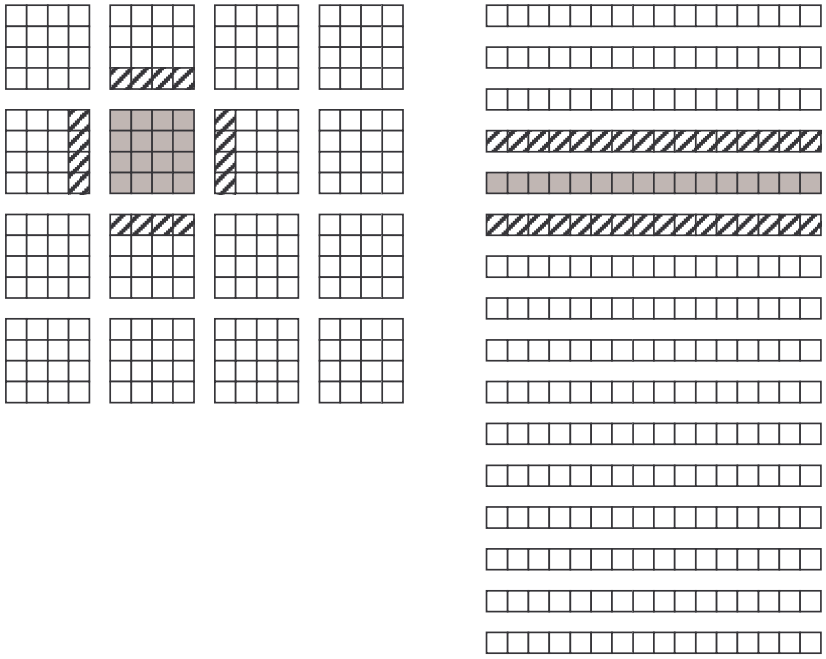
The basic way to assign work is to statically assign data to processors, and require each processor to compute on the data it "owns." This technique works for a wide variety of situations, and is the subject of this section. This is the data parallel approach, because we use the data as the basis for organizing the computation.

Basic Block Allocations

Since our goal is to exploit locality, it follows that contiguous portions of a data structure should be allocated together on the same processor. (The exceptions to this thinking are treated below.) Thus, 1-dimensional arrays are assigned to processors in blocks of consecutive indices. For 2-dimensional arrays, allocating by 2-dimensional blocks, that is, consecutive indices in both dimensions, generally leads to efficient solutions. The reason 2-dimensional blocks tend to make more sense than allocating, say, whole rows, is that blocks can often reduce communication. For example, for computations that rely on neighboring values, the so called *stencil computations* such as

```
B[i,j] = (A[i-1,j] + A[i,j+1] + A[i+1,j] + A[i, j-1])/ 4.0;
```

there is a surface-to-volume advantage, as can be seen in Figure 4.1. That is, a squarish block of array values has the property that the elements that must be referenced by other processors for the stencil computation are on the edge (surface), and as the size (volume) of the block increases, the number of edge elements grows much more slowly, reducing communication costs. This small example isn't very dramatic, but the difference for a 32x32 block is 128 versus 2048 values referenced by (communicated to) other processors. For higher *d*-dimensional arrays, allocating as *d*-dimensional blocks is frequently used for the surface to volume advantage, too, but almost as common is to allocate only two of the dimensions and keep the other dimension(s) allocated locally. The latter choice is often the result insufficiently many processors, or extreme aspect ratios.



(a) (b)

Figure 4.1: Two allocations of a 16x16 array to 16 processors: (a) 2-dimensional blocks and (b) rows. For the processor with shaded values to compute a 4-nearest neighbor computation requires communication with other processors to transmit the hatched values. The row allocation requires twice as many values to be transmitted, and because of the surface to volume advantage, the blocked allocation improves as the number of local items increases.

The Specifics of Block Layouts

Our goal, when allocating the array's blocks, is to balance the data assigned to each processor, because the work tends to be proportional to the number of data items. Occasionally, everything "divides perfectly," and each processor is assigned the identical amount of work; and sometimes partitioning can be simplified by making the array

dimensions multiples of the number of processors. But more often the problem size and therefore the size of the arrays, $r \times c$, is dictated by other considerations. In such cases there are various ways to allocate the arrays in blocks.

Assume $P = uv$, that u does not divide r, and that v does not divide c; that is, the divisions

$$r = a_1 u + e_1 \text{ and } e_1 > 0$$
$$c = a_2 v + e_2 \text{ and } e_2 > 0$$

have nonzero remainders. To discuss allocations, let

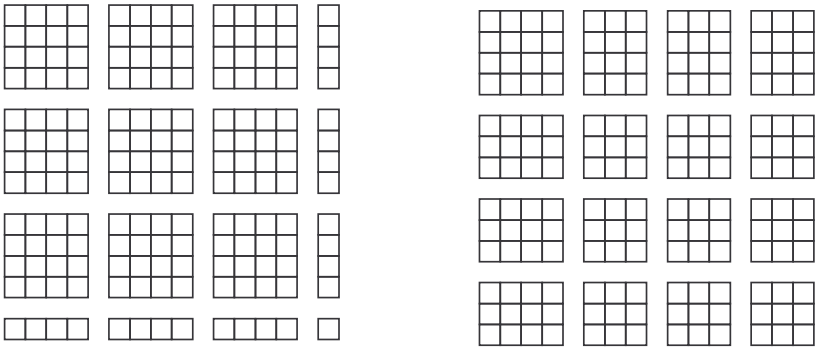
$$r = a_1'(u\text{-}1) + e_1' \text{ and } e_1' > 0$$
$$c = a_2'(v\text{-}1) + e_2' \text{ and } e_2' > 0$$

The two most obvious schemes can be called “Direct Division” and “Ceiling-Floor,” see Figure 4.2:

> **Direct Division:** Allocate blocks of $a_1' \times a_2'$ elements to $(u\text{-}1)(v\text{-}1)$ processors, allocate blocks of $e_1' \times a_2'$ to $(v\text{-}1)$ processors, allocate blocks of $a_1' \times e_2'$ to $(u\text{-}1)$ processors, and allocate a block of $e_1' \times e_2'$ to one processor.
> **Ceiling - Floor:** Allocate blocks of $a_1' \times a_2'$ elements to $e_1'e_2'$ processors, allocate blocks of $a_1' \times (a_2' - 1)$ elements to $(v - e_2')$ processors, allocate blocks of $(a_1' - 1) \times a_2'$ to $(u - e_1')$ processors, and allocate blocks $(a_1' - 1) \times (a_2' - 1)$ elements to the remaining $(u - e_1')(v - e_2')$ processors.

The allocations are the same in their most important respect, the size of the largest block, $a_1' \times a_2'$. This means that if a computation is strictly proportional to the number of local data items, both schemes have the same worst case. However, the Direct Division is likely to have $u+v\text{-}1$ processors that have significantly less work to do than the others, and so are more likely to be idle, waiting on others to complete. Such imbalance often wastes parallel resources. The Ceiling - Floor allocation has the advantage that the number of elements in each dimension differs by only one, making the quantity of data assigned more balanced compared to the Direct Division approach. The work is distributed somewhat better. Without additional information about the characteristics of the problem, the Ceiling - Floor is slightly better.

Sensitivity to Processor p_0: Independent of which allocation is chosen, it is sensible to assign the minimum allocation of data to processor p_0. This is because p_0 is often given additional tasks, such as managing I/O, serving as the root of a combining tree, etc. (These can generally be thought of as controller functions, relative to the CTA model of Chapter 2.) By allocating it the least work, p_0 is available to perform the additional duties.



(a) (b)

Figure 4.2: Two array-to-processor allocations for a 13×13 array on 16 processors; (a) Direct Division, (b) Ceiling - Floor.

Fluff or Halo Buffers: Computations like the 4-point nearest neighbor stencil

```
B[i,j] = (A[i-1,j] + A[i,j+1] + A[i+1,j] + A[i, j-1])/ 4.0;
```

require values stored on other processors. For the block allocations of the type being discussed Figure 4.1(a) shows that for `i` and `j` in the top row of the allocation, references to `A[i-1,j]` are off processor to the north, and similarly for the other edge elements of the block. Such nearest neighbor references, which are quite common in scientific computations, are best solved with the following approach:

- get the necessary values of the adjacent array blocks from the other processors
- store the values in in-position buffers arranged around the local data block
- perform the computation on the (now) entirely local data values

The buffers are known as *fluff* or *halo buffers*, and are allocated in their proper position relative to the other elements in the array block; see Figure 4.3.

Several advantages recommend this approach. First, once the fluff is filled, all references in the computation are local. This means that the many processor-dependences implied by a loop performing the stencil computation over an array block have been merged into *b* dependences, if the computation references *b* neighboring processors. The result is a large block of dependence-free code to execute. Further, the statement uses the same index calculations for all references to A; that is, they can be performed in a single loop having no special edge cases. Finally, moving non-local data to the local thread at one time offers the opportunity (generally available) to batch the data movement; that is, the whole row or column of an adjacent processor, perhaps stored in one or few cache lines, might be moved at once. This is a significant advantage, as noted in Chapter 2, because typically data transmission takes $t_o + dt_b$ seconds to transmit d bytes, where t_o is overhead and t_b is the time per byte. Batching communication saves the multiple overhead charges from multiple transmissions, and any additional waiting times caused by them.

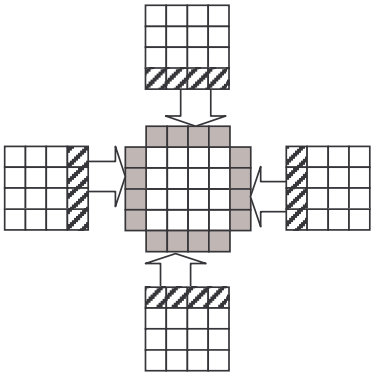


Figure 4.3: The fluff (shaded) for an array block showing the non-local values on adjacent processors that must be moved to fill the fluff; once the fluff is filled, the stencil computation is entirely local. (The "missing corners" are not used, but they are allocated to simplify array index calculations.)

Cyclic and Block Cyclic

As effective as block allocations are, they are not optimal for all algorithms, because of load balancing considerations. For example, as mentioned in Chapter 3, the well-known LU decomposition algorithm begins with all rows and columns participating in the initial step. As the computation proceeds, however, a row and column are completed with each iteration, leading to the schematic shown in Figure 4.4. When all of the rows and columns allocated to a processor are completed, it becomes idle. In the figure, 3 of the 4 processors are idle for half the computation. What should be done?

One solution is to reallocate the data periodically during the computation, but this requires moving nearly all of the active values to other processors, which is considered extremely expensive. The more practical solution is to use a cyclic or block cyclic distribution. The "cyclic" idea is to assign consecutive items to processors in a round-robin order, or as it's often described, as if dealing out cards. Thus a cyclic allocation of array elements proceeds through the array in, say, row-major order, allocating elements to processors. Because keeping track of individual array elements is burdensome, it is more common to "deal out" consecutive subarrays, a strategy called *block-cyclic* allocation.

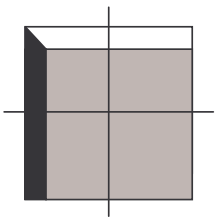


Figure 4.4: Schematic diagram of the LU Decomposition algorithm block-assigned to four processors; the final result is a lower (black) and an upper (white) triangular matrices; active computation is gray; a column and row are completed and added to each result matrix, respectively, in each iteration.

Figure 4.5 shows a block-cyclic allocation in which consecutive array blocks are assigned to separate processors cyclically. The figure shows several features of block-cyclic allocations: A block's dimension (called the *chunk size*) does not have to divide the array's dimension; the block is simply truncated. Each processor receives blocks from throughout the array, implying that as the computation proceeds, completed portions will be resident on each processor, as will not-yet-completed portions. Figure 6 shows the schematic allocation of Figure 5 as it would appear part way through an LU-type computation. Notice how well the remaining work is balanced across the processors.

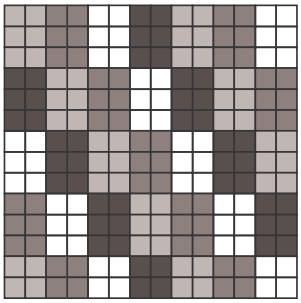


Figure 4.5: Block-cyclic allocation of 3×2 blocks to a 14×14 array distributed to four processors (colors).

The block-cyclic approach has the great advantage of balancing the work across the processors (see Figure 4.6), but this does not come without costs. The most obvious cost is the potential of block-cyclic allocation to complicate the algorithm. If the computation uses the spatial properties of an array—for example, rows—then because block-cyclic breaks some of these relationships, special cases may have to be added to the algorithm. This effect can sometimes be neutralized by picking chunk sizes to create easy-to-work-with patterns in the allocation. Another common recommendation is to allocate relatively large blocks, say 64×64, as a means of amortizing the overhead of the special checking. Of course, allowing the blocks to become too large probably means that the work will be less evenly allocated, and that the unbalanced case at the "end" of the computation will occur sooner and last longer. It is a delicate matter to balance the competing goals of a block-cyclic allocation.

Finally, notice that the block allocations discussed above and the block-cyclic allocations discussed here do not exploit locality equally well, even when block-cyclic uses large blocks. In general, for a given number of processors and array size, block-cyclic will use many smaller blocks whereas the block approach will use a single larger block per processor. This means for computations requiring nearest neighbor communication, e.g. stencils, the surface to volume advantage of blocks will result in much less communication. (The extremely small blocks of Figure 4.5 emphasize this point since every element is on the surface!) Of course, for computations compatible with a single allocation strategy, it is an easy matter to choose the right one. But, for cases where different phases of the computation would benefit from different allocations, it can be difficult to find the right compromise.

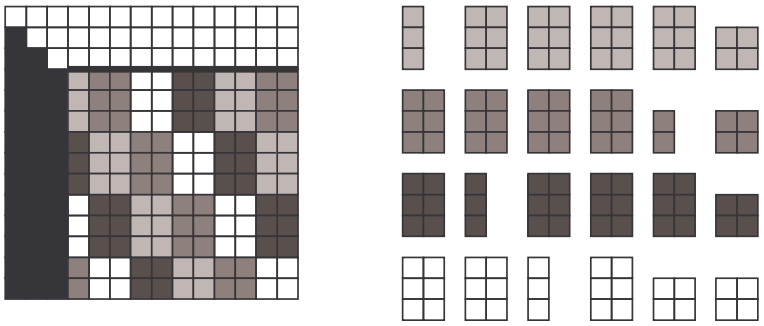


Figure 4.6: The block-cyclic allocation of Figure 4.5 midway through the computation; the blocks to the right summarize the active values for each processor.

Assigning Work to Processors Dynamically

In many cases it is not possible to adopt a fixed work assignment and stay with it. Examples include algorithms that create tasks as they proceed, algorithms whose tasks have highly variable execution times, adaptive algorithms that apply their computing power where the solution needs the greatest work, etc. In these and similar cases the work queue strategy may lead to the best assignment of computation.

Work Queue

A work queue is a first-in first-out list of task descriptors. If as a computation proceeds new work is generated, it is packaged into a task descriptor and is appended to the work queue; if as computation proceeds work is completed and a processor becomes available, it removes a task descriptor from the queue and begins work on it. The commonly used names for these two roles are *producer* and *consumer*.

As an example of a trivial task that can be expressed in work queue form, consider the $3n+1$ conjecture (or Collatz Conjecture), which proposes an affirmative answer to the question "For any positive integer a_0, does the process defined by

$$a_i = \begin{cases} 3a_{i-1} + 1 & a_{i-1} \text{ odd} \\ a_{i-1} / 2 & a_{i-1} \text{ even} \end{cases}$$

converge to 1?" (See http://mathworld.wolfram.com/CollatzProblem.html.) Though this conjecture is known to be true for all integers less than $3 \cdot 2^{53}$, we will program a search of the integers as an example, because it illustrates several aspects of work queue technique.

Our solution postulates a work queue containing the next integers to test. We initialize the queue to the first P positive integers:

```
void init(work_q: q) {     //setup globally-allocated queue array
      int i;
```

```
      for (i=0; i<P; i++) {
        q[i] = i+1;
      }
)
```

The integers are our task descriptors. As a general principle, it is wise to make the task descriptors as small as possible, while making them self-contained.

A worker thread, of which we assume there are *P*, will consume the first item from the work queue, add *P* to it, and append the result to the work queue. The rationale for adding *P* is that *P* threads will be checking integers at once, so advancing by *P* has the effect of skipping those that are (logically) being processed by other threads. (The other threads may not all be computing yet, but they will be.) The worker thread has the following logic:

```
int tester(int: limit) {    //test the conjecture
      int a;                //test number
      int n;                //count number of rounds of testing
      while (n < limit) {
        n = 0;              //initialize
        a = consume();      //remove the first element of work q
        produce(a+P);       //place new item on work q
        while (a != 1 && n < limit) {
           if (even(a))
               a = a / 2;
           else
               a = 3*a + 1;
        n++;
        }                   // exiting w/a==1 confirms converge
      }                     // exiting means limit was exceeded
      return a;             // tell what number is the culprit
)
```

The tester takes a `limit` as a parameter. The processing loop is controlled by `a != 1` and by `n < limit`. Because we think the conjecture is true, we expect the processing loop to exit with `a == 1`, but if there were an integer for which the conjecture is false, the loop will exceed the `limit`. If the `limit` is reached the worker will stop and return; otherwise it continues checking until it is eventually preempted by the parent routine. Of course, the parent, receiving a number back from a worker, cannot know if it is a counter-example to the conjecture, but it can test further to see if the number is actually convergent, but that the `limit` was set too low. This simply requires increasing the size of `limit` and rechecking.

Consider the behavior of the work queue. First, notice that in effect *P* subsequences are being checked simultaneously, but they will not remain in lock step because the amount of work required of each check is different. For example, with four processors the queue might transition through the states at the beginning

Work Queue Active Processors [task]

1, 2, 3, 4	--
2, 3, 4, 5	$\boldsymbol{p_0}$**[1]**
3, 4, 5, 6	$\boldsymbol{p_0}$**[2]**
4, 5, 6, 7	p_0[2], $\boldsymbol{p_1}$**[3]**
5, 6, 7, 8	p_0[2], p_1[3], $\boldsymbol{p_2}$**[4]**
6, 7, 8, 9	$\boldsymbol{p_0}$**[5]**, p_1[3], p_2[4]
7, 8, 9, 10	p_0[5], p_1[3], p_2[4], $\boldsymbol{p_3}$**[6]**

illustrating that because processing 1 is trivial, p_0 might return to the queue to get the next value, 2, before any of the other processors start; processing 2 is also easy, so it is back again quickly. Further, notice that workers do not necessarily process the same subsequence. Indeed, if the timing works out all of the processors could be working on the same subsequence at once. Summarizing, although our work queue is regimented, the way its tasks are processed can accommodate any timing characteristics.

One important detail has been ignored in the worker code. It is the matter of races resulting from multiple threads referencing the work queue simultaneously. We have assumed that the two procedures, `consume()` and `produce()` contain the appropriate synchronization apparatus. Exact details of how to construct the appropriate protection are presented in the next chapter.

The Reduce & Scan Abstractions

The success of Schwartz's algorithm for addition and Count 3s computations is not accidental. Operations of adding and counting array elements are instances of a general form of computation known as *reduce* and *scan*. They are well understood, and so, efficient solutions are known. By recognizing such computations, we can avoid working out their details each time and simply apply standard solutions, such as Schwartz's algorithm. We can save our serious thinking for those parts of a computation that need brain power.

Reduce, which can be thought of as short for "reduce the operand data structure to a single value by combining with the given operator," has been a part of programming languages since the creation of the array language APL. The operators are often limited to the associative and commutative operations *add*, *multiply*, *and*, *or*, *max* and *min*, but many operations work. Though some languages have built-in reduce operations, many do not, and so we will create our own general routine based on the Schwartz approach. To simplify our discussion of reduce, we adopt the notation *operator* **//** *operand*, as in `+ // A`, to describe A's elements being reduced using addition. We might write the Count 3s computation `+ // (A == 3)`, and expect to implement it by instantiating a general Schwartz solution with implementations of these operations.

Scan is a synonym for "parallel prefix," mentioned in Chapter 1. Scan is related to reduce in that scan is the reduce computation in which the intermediate values are saved, assuming a specific order of evaluation. Thus, for array A with values

```
4   2   5   6   1
```

the plus-scan of A is

```
4   6   11   17   18
```

Reduce is simply the final value produced in a scan. (We adopt the notation *operator* **** *operand*, as in + \\ A.) As with reduce the associative and commutative operations *add*, *multiply*, *and*, *or*, *max* and *min* are common, but many other also computations work.

To illustrate using the reduce and scan abstractions to compute the bounding-box enclosing points in the plane. Let A be an array of n points of the form,

```
type planePt = record
        x : int;
        y : int;
     end;
```

then the sequential computation

```
maxX = A[0].x;
for (i = 1, i < n, i++)
{
     if (maxX < A[i].x)
         maxX = A[i].x;
}
maxY = A[0].y;
for (i = 1, i < n, i++)
{
     if (maxY < A[i].y)
         maxY = A[i].y;
}
minX = A[0].x;
for (i = 1, i < n, i++)
{
     if (minX > A[i].x)
         minX = A[i].x;
}
minY = A[0].y;
for (i = 1, i < n, i++)
{
     if (minY > A[i].y)
         minY = A[i].y;
}
```

for computing the four defining values for the bounding box is equivalent to four applications of reduce,

```
maxX = max//A.x
maxY = max//A.y
minX = min//A.x
minY = min//A.y
```

which as we saw in Chapter 1 can be efficiently implemented in parallel. Of course, these four can be merged into one implementing procedure, so that there is only one pass over the data and only one combining tree.

Generalized Reduce, Scan and Vector Operations

Because reduce and scan are such effective abstractions for thinking about parallel computation, we advocate using them, and developing tools for their convenient application. Because there are so many programming systems in use, we describe how to construct an implementation, rather than giving a specific one. We begin the section with reduce, and move on to scan. Finally, we observe that the concepts can be applied to general vector operations.

Structure for Generalized Reduce

To build a general reduce or scan implementation, visualize the Schwartz algorithm, as abstracted in Figure 4.7. Overall, the figure shows local computation performed at the leaves of a combining tree, which emits the reduction result at its root. Looking more closely at the diagram, we see that a data structure called the tally is used together with four functions, two applied on each processor:

> `init()` initializes the process on each processor, setting up the *tally* data structure recording the local result as the reduce performs the accumulation operation
> `accum(tal, val)` performs the actual accumulation by combining the running tally with the operand value

Once the local results are found, they must be combined to form the global result, using two more functions:

> `combine(left, right)` combines two tally values to create a new tally value
> `reduceGen(root)` takes the global tally value and outputs the correct result for reduce.

For example, to compute `+//A`, the `init()` routine would initialize a tally variable to 0; the `accum()` routine would add its tally to an operand value; the `combine()` would add two local tallies and `reduceGen()` is a noop that simply returns the result.

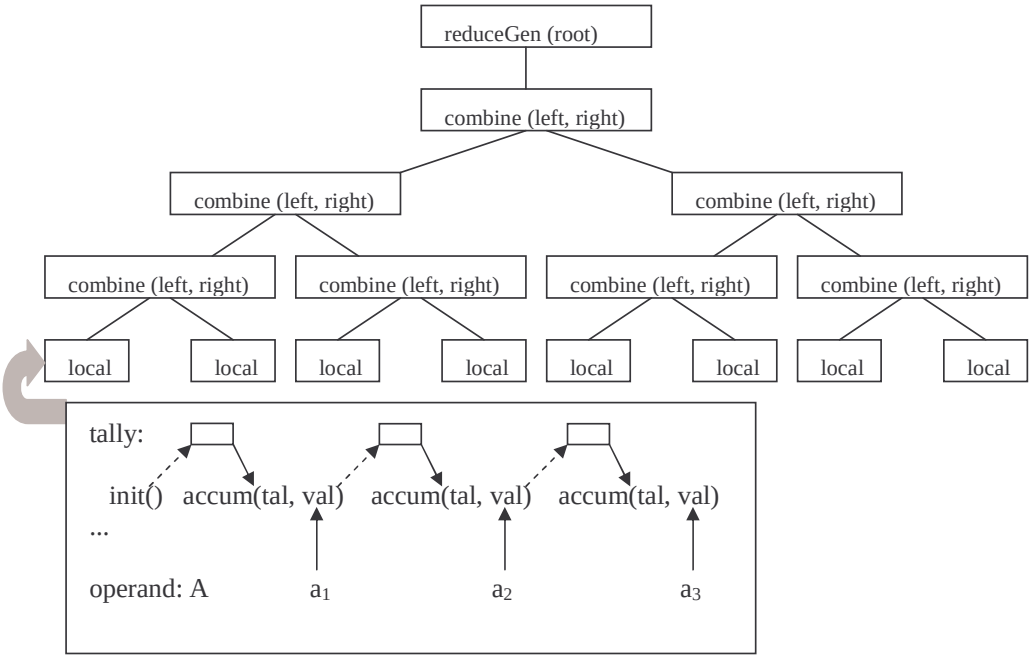


Figure 4.7: Schematic diagram of the Schwartz algorithm used to implement user-defined reduce. The local operations are abstracted in the box; function `init()` sets up the tally; `accum()`combines the tally and the operand data structure; outside the box, in the combining tree, `combine()` forms a new tally from two others, and `reduceGen()` produces the final answer from the global tally.

For the simple operation of `+//A`, the four-function formulation is excessively general, but this structure is essential in more complex (and more powerful) cases, as illustrated in a moment. The key to understanding the roles of the four reduce functions is first to recognize that the tally data structure need not have the same type as the operand data, and second that the four routines take different argument types. So, for example, imagine a user-defined `secondMin // A` that finds the second smallest element in an array, useful perhaps for an array of non-negative numbers with many 0s. In this case the tally data structure would have to be a two-element record or array storing the smallest and second smallest values. The `init()` function would set-up the tally, initialized to the + infinity for the operand data type; `accum()` would compare an operand value with the tally elements recording the two smallest; `combine()` would find the two smallest of its two tally arguments, and the `reduceGen()` would return the second smallest value as the result. When called at the right points in a parallel procedure implementing Schwartz's algorithm, they produce a parallel `secondMin()` solution. Figure 4.8 makes this logic precise.

```
type tally = record
       sm1 : float;                    //smallest element
       sm2 : float;                    //second smallest
     end;

void init(tally: tal) {       //setup globally-allocated tally
      tal.sm1 = MAX_FLOAT;
      tal sm2 = MAX_FLOAT;
)
void accum(float: elem, tally: tal) {   //local accumulation
      if (tal.sm1 > elem) {
          tal.sm2 = tal.sm1;
          tal.sm1 = elem;
      }
      elseif (tal.sm2 > elem) {
          tal.sm2 = elem;
      }
}
void combine(tally: left, tally: right) { //combine into "left"
      accum(right.sm1, left);             //by accumulating right
      accum(right.sm2, left);             //values one at a time
}
float reduceGen(tally: tal) {
      return tal.sm2;
}
```

Figure 4.8: The four user-defined reduce functions implementing `secondMin` reduce. The tally, globally defined on each processor, is a two-element record.

Structure for Generalized Scan

Generalized scan applies the same concepts as generalized reduce. The primary difference is that after the combining is complete the "parallel prefix" values must be passed back down the combining tree. That is, in order to complete the prefix computation on the local values, an intermediate value from the combining tree will be needed by each processor. (Refer to the parallel prefix discussion in Chapter 1 and review Figure 1.2.)

The generalized scan begins like the generalized reduce, and there is no conceptual difference in the three functions `init()`, `accum()` and `combine()` for the two algorithms. However, the scan is not finished when the global tally value has been computed. Rather, tally values must be propagated down the tree subject to the constraint that

> *for any node, the value it receives from its parent is the tally for the values that are left of its leftmost leaf*

which because we are computing on blocks, means the first item in the leftmost leaf's block. This causes us to call `init()` to create the value as the input from the logical parent of the root, because there is no tally for the items to the left of those covered by the root. Each node, receiving a value from the parent, relays it to its left child; for its right

child, it combines the value received from the left child on the upsweep with the value received from the parent, and sends the result to the right.

When the tally value is received at a leaf, it must be combined with the values stored in the operand array to compute the prefix totals, which are stored in the operand position. In Figure 4.9 these operands are shown schematically in the box at the bottom. Thus, the `scanGen()` procedure produces the final result.

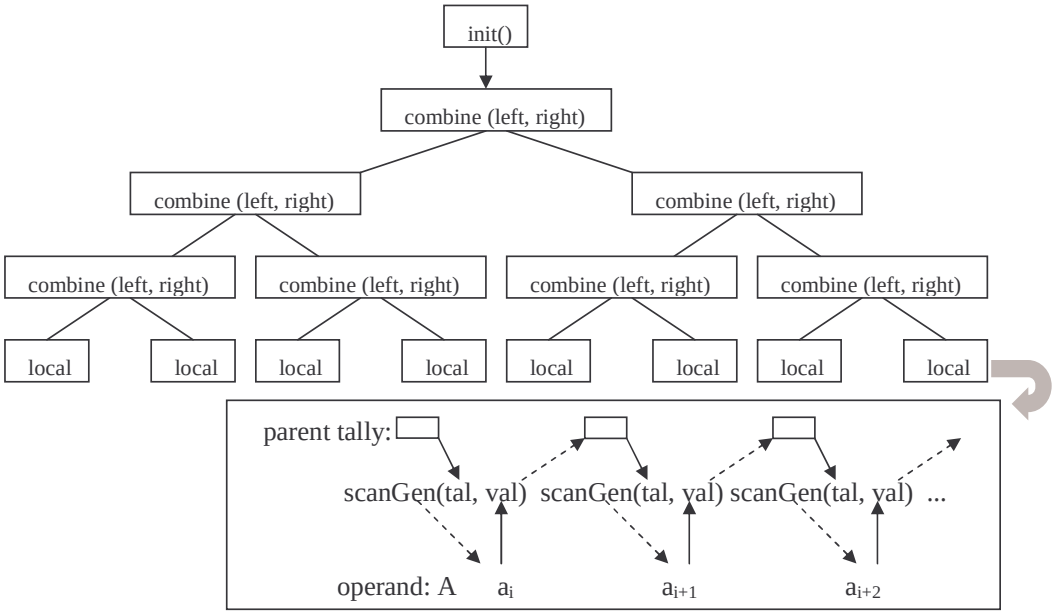


Figure 4.9: Schematic of the scan operation. The first part of the algortihm is simply the generalized reduce, schematized in Figure 4.7. Once the global tally is found, prefixes are propagated down the tree. When a prefix arrives at a leaf, the local operation applies the `scanGen()` function, and stores the result in the operand item.

To illustrate the operation of user-defined scan, imagine an array `A` of integers from the sequence 0, …, *k*-1. The scan `sameAs \\ A` records in position `A[i]` the number of elements in the first *i* matching `A[i]`. We use as a tally an array of *k* elements, which is initialized to 0s; the accumulate function increments the array item count for the operand value; combine function adds the two arrays; and the scan generator performs an accumulate (initialized this time by the prefix received from the parent), and stores the count for the item found. Figure 4.10 shows the functions that realize this result. (Notice that the tally array at the end is a histogram for the array.)

```
void init(tally: tal) {         //setup globally-allocated array
      for (i=0; i<k; i++) {
         tal[i]= 0;
      }
)
void accum(int: elem, tally: tal) {       //local accumulation
      tal[elem]++;
}
void combine(tally: left, tally: right) { //combine into "left"
      for (i=0; i<k; i++) {
           left[i] += right[i];
      }
}
int scanGen(int: elem, tally: tal) {      //finalizing scan
      accum(elem, tal);                   //accum w/parent tally
      return tal[elem];                   //store running count
}
```

Figure 4.10: User-defined scan functions to return the running count of *k* items; the tally is a globally allocated array of *k* elements.

Structure for Gneralized Vector Operations

The foregoing discussion shows that instantiating the basic structure of reduce and scan with custom functions can create efficient parallel solutions. But the idea is even more general. There is no need that the operations "accumulate from left to right;" computations that can be performed on blocks of data that can be merge to produce a larger solution are good candidates for applying this same structure. We illustrate the idea by computing the longest run of positive values stored an array. So, for the sequence

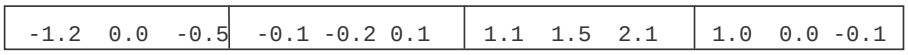
```
-1.2  0.0  0.5 -0.1 -0.2 0.1  1.1  1.5  2.1  1.0  0.0 -0.1
```

the computation would return 5.

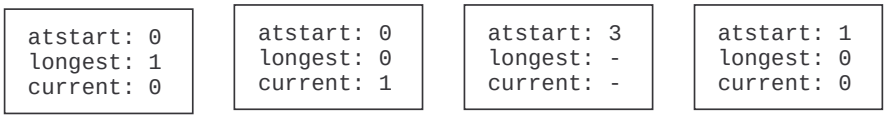
To formulate a local block computation to find the longest run of positive values, observe that the run could straddle the boundary (or boundaries) between local blocks. For that reason, we select a tally that has three values

```
type runLen = record
       atstart : int;   // count of positives from left
       longest : int;   // longest (interior) run found so far
       current : int;   // length of current run
     end;
```

that will count the number of positive values beginning at the start of a block, `atstart`, if any, the longest run properly contained in the block, `longest`, and the length of the current run, `current`. This last variable will have the effect of recording the length of the positive run extending to the right end of the block, if any. Because the block will be processed left to right, it will be convenient to treat a sequence that completely spans a block as having an undefined `longest` and `current` values. So for example, dividing the foregoing example among four processors

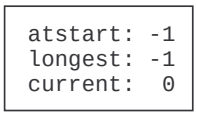


results in the tallies



Notice, the third thread has the undefined values in its tally.

To build the four reduce functions, initialize the tally items to undefined (represented as -1), but to simplify the combining logic later, set `current` to 0.



Accumulating begins by counting positive items in `atstart` until the sequence is broken; thereafter, it counts positive items in `current`, and records the length of the runs in `longest`. Thus, `accum()` must separate the initial sequence from the others, and for that it requires a cascade of logic as shown in Figure 4.11.

Given two tallies, their combined tally must handle four cases. These are

- both blocks span only positive elements: the result spans positive elements, so add the right block's `atfirst` to the left block's `atfirst`, the blocks' `longest` and `current` are the same
- the left block spans only positive elements: the right block's `atfirst` adds to left's `atfirst`, and the right block's `longest` and `current` apply
- the right block spans only positive elements: add the right block's `atfirst` to the left block's `current`, and the left block's `atfirst` and `longest` apply
- both blocks have non-positives; the left block's `current` plus the right block's `atfirst` could be longer than either `longest`, the left block's `atfirst` and the right block's `current` apply

We use `longest == -1` as our test for a positive only block. This logic is implemented as a cascade of `if`-statements.

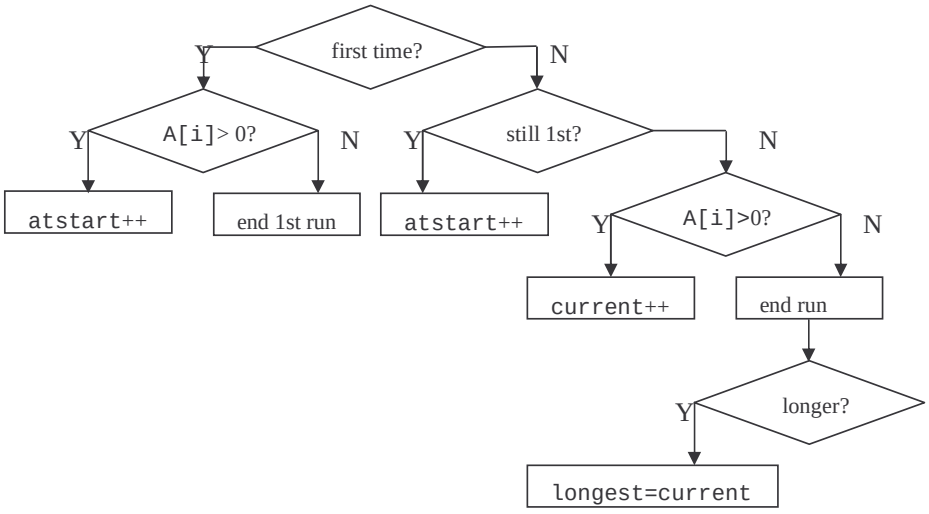


Figure 4.11: The accumulate logic. The "first time?" test is true when `atfirst == -1`; the "still 1st?" test is true when `longest == -1`.

Finally, the global result must pick the largest of its `atfirst`, `longest` and `current` values. If the longest sequence starts at the beginning, then the preceding logic ensures that `atfirst` will record its length; if the longest sequence extends to the end, the `current` will record the length. Otherwise the longest sequence is somewhere in the middle, and `longest` will record the value.

See Figure 4.12 for the exact logic.

These are powerful techniques that have efficient parallel implementations.

```
void init(runLen: zero) {     //setup globally-allocated record
      zero.atstart = -1;
      zero.longest = -1;
      zero.current = 0;
)
void accum(int: elem, runLen: z) {   //local accumulation
      if (z.atstart == -1) {         //first time?
         if (elem > 0)
             z.atstart = 1;
         else {
             z.atstart = 0;
             z.longest = 0;
         }
      }
      else {
         if (z.longest == -1) {      //still first time?
             if (elem > 0)
                  z.atstart++;
```

```
             else
                  z.longest = 0;
         }
         else {
            if (elem > 0)
               z.current++;
            else {
              if (z.longest > z.current)
                  z.longest = z.current;
              z.current = 0;
            }
         }
      }
}
void combine(runLen: left, runLen: right) { //combine into "left"
      if ((left.longest == -1) && (right.longest == -1)) //spans
          left.atstart = left.atstart + right.atstart;
      else {
         if (left.longest == -1) {
             left.atstart = left.atstart + right.atstart;
             left.longest = right.longest;
             left.current = right.current;
         }
         else {
            if (right.longest == -1)
                left.current = left.current + right.atfirst;
            else {
               left.longest = MAX(left.longest,
                                  left.current + right.atstart,
                                  right.longest);
               left.current = right.current;
            }
         }
      }
}

int reduceGen(runLen: z)   {
      if (z.longest < z.atfirst)
          z.longest = z.atfirst;
      if (z.longest < z.current)
          z.longest = z.current;
      return z.longest;
}
```

Figure 4.12: The four reduce/scan functions for the “longest positive run” computation.

Trees

After arrays, trees must be the most import way to represent a computation. They present challenges in parallel computation for several reasons. First, trees are usually constructed using pointers, and in many parallel computation situations, pointers are local only to one processor. Second, we typically use trees for their dynamic flexibility, but dynamic

behavior often implies performance-bashing communication. Third, trees complicate allocation-for-locality. But, challenging or not, trees are too useful to ignore.

Representation of Trees

Begin by noticing that we have already used trees in several computations to perform accumulation and parallel prefix operations. They were implicit in that they derive from the communication patterns used. So, in the reduce and scan primitives above, the `combine()` operations were performed pairwise, with the intermediate results also being combined pairwise, etc. inducing a tree, as shown in Figure 4.13. There are no pointers; processors simply perform the appropriate tree roles, and the result is achieved. By this technique we use trees to perform global operations even when they are not the base data structure of the computation.

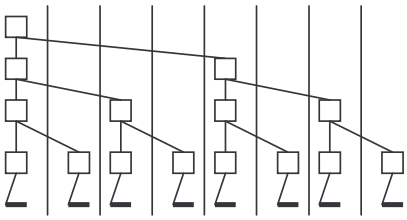
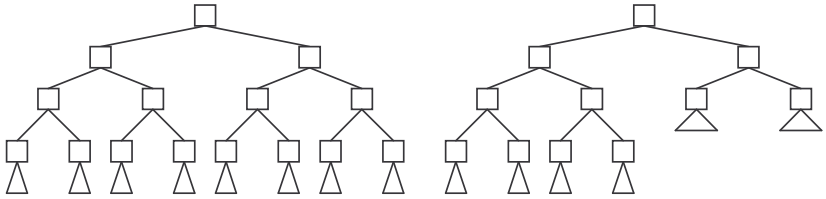


Figure 4.13: Induced tree. Each processor computes on a sequence of values (heavy lines), and then combines the results pairwise, inducing a tree; notice processor 0 participates at each level in the tree.

Our Guideline #1 rule, to maximize the number of large blocks of independent computation, motivates us to use the implicit tree idea even in cases where the base data structure is a tree. This means that we separate the local and the global paradigms: Locally, we may choose to use pointers in our implementation, but at the higher levels of the tree where the edges are non-local, we use the implicit solution – each node simply performs its proper tree role. Though it can be inconvenient to shift processing paradigms, the advantage may be that it allows us to write a single-threaded solution to the subproblem (it might already exist), and then incorporate those subproblems into the global solution.

Breadth First. Consider first trees that can naturally be enumerated breadth first, that is, all nodes of a level can be generated given their parent nodes. In this case we conceptually generate the complete tree down to the level, or pair of levels, having P nodes, one corresponding to each processor. So, in the binary tree case, if $P = 2^l$, generate to level l. For example, for $P = 8$, we generate a binary tree down to level 3, as shown in Figure 4.14(a). When P does not equal the number of nodes on a level, pick the greatest level less than P and then expand enough of the nodes to the next level to equal P, as shown in Figure 4.14(b). Then, assuming the tree extends much more deeply below each of these nodes, allocate to each processor corresponding to a node the entire subtree enumerated from the node. The computation is conceptually local for the whole subtree.



(a) (b)

Figure 4.14: Logical tree representations: (a) a binary tree where $P = 2^l$; (b) a binary tree where $P = 6$.

Example. This technique works well for problems that can be recursively partitioned into subproblems. For example, suppose we are searching a game tree for Tic-Tac-Toe (Naughts and Crosses) games on $P = 4$ processors. When symmetries are considered, there are only three initial positions, and we expand one of these to fill out the 4 search tasks, see Figure 4.15. That is, each processor will search the game tree descendant from the indicated board position.

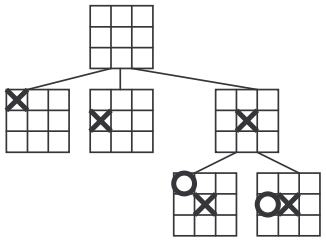


Figure 4.15: Enumerating the Tic-Tac-Toe game tree; a processor is assigned to search the games beginning with each of the four initial move sequences.

Depth First. Trees that should be enumerated by depth can be implemented in parallel using a work queue approach. The queue is initialized with the root; a processor removes a node and if that action leaves the queue empty, the processor expands the node, taking one descendant as its task and appending the others to the queue. Such an approach corresponds to standard iterative depth first traversal, and has a structure as shown in Figure 4.16.

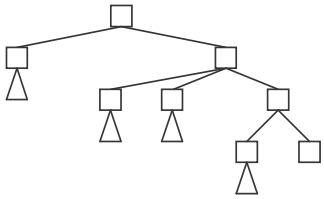


Figure 4.16: Basic depth allocation of a tree to P=4 processors, which are each responsible for the subtree rooted at the node; the right-most node remains in the queue.

Having assigned a subtree to each processor, consider the main aspect of depth first enumerations, the feedback from the early parts of the enumeration to the later parts. This

takes various forms including a list of nodes visited, alpha-beta limits on productive subtrees to consider, and other measures of "progress". A parallel algorithm will have to adapt such feedback to the processing of independent subtrees. For example, consider a packing algorithm trying to minimize the area occupied by the arrangement of several objects. A global variable recording the area of the best arrangement found so far can be referenced by all processors and so be used in each subtree to prune the enumeration.

Full Enumeration. Certain trees must be expanded in their entirety. A common example is the family of K-D Trees used in gravitational simulations and the closely related Barnes-Hut trees. Such trees are used for rapid lookup of related elements. In the case of gravitation simulation, 3D space is partitioned into octants, which are in turn partitioned, etc. until each region contains only one point. This allows the points physically near a given point to be quickly located by tree traversal. (Advanced algorithms allow groups of points acting at a distance to be approximated as a single meta-point.) Areas of high concentration can lead to locally deep trees.

Perhaps the best allocation for such a tree is the so-called "cap allocation," as shown in Figure 4.17. In the allocation the *P* nodes nearest the root, the cap, are redundantly allocated to each processor. Additionally, a processor is also allocated one of the subtrees rooted at the bottom of the cap. As the computation proceeds, the cap portion of the tree must be maintained coherently. That is, all processors must "see" the same state, and a locking protocol must be respected. Interaction among the subtrees can use a messaging protocol.

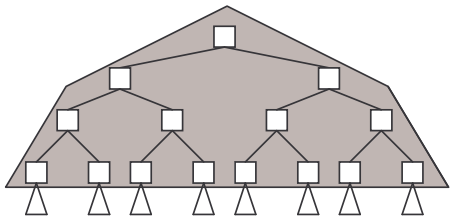


Figure 4.17. Cap allocation for a binary tree on *P*=8 processors; the cap (shaded) and one of the "leaf subtrees" are allocated to each processor.

The cap allocation is effective primarily because most of the activity takes place in the subtrees, and therefore is entirely local to a processor. Further, activity around the root is rare, so there is little likelihood for lock contention. Finally, the availability of the root means that interactions "crossing the root" can be navigated by percolating up in the local tree and crossing the cap locally, so as to identify the correct destination subtree. Navigating in the destination subtree is typically assigned as a task to the owner. As an additional bonus in the advanced algorithms for gravitational simulations in which large regions of the problem are aggregated into meta-points, the points around the root are all meta-points, and therefore are read-only, eliminating races and locking as issues.

Summary

Exercises

Exercise 0. Write a sequential program to perform the operations illustrated in Figure 3. That is, using an imagined library operation GetIt(<direction>, <buffer>) to transmit the data from the neighboring threads, compute a 4-point stencil computation on the interior data array A. Assume that GetIt blocks if the data on the other processor isn't ready; also, assume that buffer is a separate block of memory not related to A, i.e. you must include the fluff buffers in your portion of the data array and fill them manually.

Exercise 1. Generalize the longest positive run program to return a Boolean mask with a 1 in the element positions for every positive value in the longest run, and 0s everywhere else. This computation uses the existing functions slightly modified, but replaces the `reduceGen()` with `scanGen()` to produce the final values. There is also a revision required for the tally data structure. [Hint: Assume in the accumulate function the availability of a variable, called `index`, giving the index of the element being processed.]

Exercise 2. Revise the timing assumptions of the work queue example so that all processors are working on the subsequence: 4, 8, 12, 16. (There are multiple solutions.)

Ex. Revise the tester program so that it exploits the fact that if the threads have established that all number less than k converge, then no thread need check further when $a_i < k$.

Historical Context

Need in the historical section a bunch of things about the PRAM, Fetch and Add, and other one-point-per processor schemes, such as SIMD. Need to cite Anderson/Snyder to emphasize that there are more fundamental reasons to not like PRAMs than lock contention. Cite Blelloch. Ladner and Fischer, Deitz; who thought up block-cyclic—Lennert?

Chapter 5: Achieving Good Performance

Typically, it is fairly straightforward to reason about the performance of sequential computations. For most programs, it suffices simply to count the number of instructions that are executed. In some cases, we realize that memory system performance is the bottleneck, so we find ways to reduce memory usage or to improve memory locality. In general, programmers are encouraged to avoid premature optimization by remembering the 90/10 rule, which states that 90% of the time is spent in 10% of the code. Thus, a prudent strategy is to write a program in a clean manner, and if its performance needs improving, to identify the 10% of the code that dominates the execution time. This 10% can then be rewritten, perhaps even rewritten in some alternative language, such as assembly code or C.

Unfortunately, the situation is much more complex with parallel programs. As we will see, the factors that determine performance are not just instruction times, but also communication time, waiting time, dependences, etc. Dynamic effects, such as contention, are time-dependent and vary from problem to problem and from machine to machine. Furthermore, controlling the costs is much more complicated. But before considering the complications, consider a fundamental principle of parallel computation.

Amdahl's Law

Amdahl's Law observes that if $1/S$ of a computation is inherently sequential, then the maximum performance improvement is limited to a factor of S. The reasoning is that the parallel execution time, T_P, of a computation with sequential execution time, T_S, will be the sum of the time for the sequential component and the parallel component. For P processors we have

$$T_P = 1/S \cdot T_S + (1-1/S) \cdot T_S / P$$

Imagining a value for P so large that the parallel portion takes negligible time, the maximum performance improvement is a factor of S. That is, the proportion of sequential code in a computation determines its potential for improvement using parallelism.

Given Amdahl's Law, we can see that the 90/10 rule does not work, even if the 90% of the execution time goes to 0. By leaving the 10% of the code unchanged, our execution time is at best 1/10 of the original, and when we use many more than 10 processors, a 10x speedup is likely to be unsatisfactory.

The situation is actually somewhat worse than Amdahl's Law implies. One obvious problem is that the parallelizable portion of the computation might not be improved to an unlimited extent—that is, there is probably an upper limit on the number of processors that can be used and still improve the performance—so the parallel execution time is unlikely to vanish. Furthermore, a parallel implementation often executes more total instruction than the sequential solution, making the $(1\text{-}1/S) \cdot T_S$ an under estimate.

Amdahl's Law. The "law" was enunciated in a 1967 paper by Gene Amdahl, an IBM mainframe architect [Amdahl, G.M., Validity of the single-processor approach to achieving large scale computing capabilities. In *AFIPS Conference Proceedings*, AFIPS Press 30:483-485, 1967]. It is a "law" in the same sense that the Law of Supply and Demand is a law: It describes a relationship between two components of program execution time, as expressed by the equation given in the text. Both laws are powerful tools to explain the behavior of important phenomena, and both laws assume as constant other quantities that affect the behavior. Amdahl's Law applies to a program instance.

Many, including Amdahl, have interpreted the law as proof that applying large numbers of processors to a problem will have limited success, but this seems to contradict news reports in which huge parallel computers improve computations by huge factors. What gives? Amdahl's law describes a key fact that applies to an *instance* of a computation. Portions of a computation that are sequential will, as parallelism is applied, dominate the execution time. The law fixes an *instance*, and considers the effect of increasing parallelism. Most parallel computations, such as those in the news, fix the parallelism and expand the instances. In such cases the proportion of sequential code diminishes relative to the overall problem as larger instances are considered. So, doubling the problem size may increase the sequential portion negligibly, making a greater fraction of the problem available for parallel execution.

In summary, Amdahl's law does not deny the value of parallel computing. Rather, it reminds us that to achieve parallel performance we must be concerned with the entire program.

Measuring Performance

As mentioned repeatedly, the main point of parallel computing is to run computations faster. *Faster* obviously means "in less time," but we immediately wonder, "How much less?" To understand both what is possible and what we can expect to achieve, we use several metrics to measure parallel performance, each with its own strengths and weaknesses.

Execution Time

Perhaps the most intuitive metric is *execution time*. Most of us think of the so called "wall clock" time as synonymous with execution time, and for programs that run for hours and hours, that equivalence is accurate enough. But the elapsed wall clock time includes operating system time for loading and initiating the program, I/O time for reading data, paging time for the compulsory page misses, check-pointing time, etc. For

short computations—the kind that we often use when we are analyzing program behavior—these items can be significant contributors to execution time. One argument says that because they are not affected by the user programming, they should be factored out of performance analysis that is directed at understanding the behavior of a parallel solution; the other view says that some services provided by the OS are needed, and the time should be charged. It is a complicated matter that we take up again at the end of the chapter.

In this book we use execution time to refer to the net execution time of a parallel program exclusive of initial OS, I/O, etc. charges. The problem of compulsory page misses is usually handled by running the computation twice and measuring only the second one. When we intend to include all of the components contributing to execution time, we will refer to *wall clock time*.

Notice that execution times (and wall clock times for that matter) cannot be compared if they come from different computers. And, in most cases it is not possible to compare the execution times of programs running different inputs even for the same computer.

FLOPS

Another common metric is FLOPS, short for floating point operations per second, which is often used in scientific computations that are dominated by floating point arithmetic. Because double precision floating point arithmetic is usually significantly more expensive than single precision, it is common when reporting FLOPS to state which type of arithmetic is being measured. An obvious downside to using FLOPS is that it ignores other costs such as integer computations, which may also be a significant component of computation time. Perhaps more significant is that FLOPS rates can often be affected by extremely low-level program modifications that allow the programs to exploit a special feature of the hardware, e.g. a combined multiply/add operation. Such "improvements" typically have little generality, either to other computations or to other computers.

A limitation of both of the above metrics is that they distill all performance into a single number without providing an indication of the parallel behavior of the computation. Instead, we often wish to understand how the performance of the program scales as we change the amount of parallelism.

Speedup

Speedup is defined as the execution time of a sequential program divided by the execution time of a parallel program that computes the same result. In particular, *Speedup* = T_S / T_P, where T_S is the sequential time and T_P is the parallel time running on P processors. Speedup is often plotted on the y-axis and the number of processors on the x-axis, as shown in Figure 5.1.

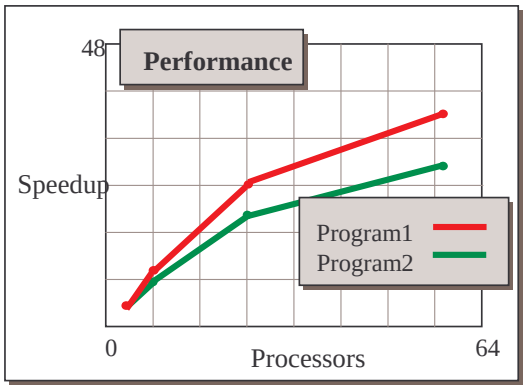


Figure 5.1. A typical speedup graph showing performance for two programs.

The speedup graph shows a characteristic typical of many parallel programs, namely, that the speedup curves level off as we increase the number of processors. This feature is the result of keeping the problem size constant while increasing the number of processors, which causes the amount of work per processor to decrease; with less work per processor costs such as overhead—or sequential computation, as Amdahl predicted—become more significant, causing the total execution not to scale so well.

Efficiency

Efficiency is a normalized measure of speedup: *Efficiency* = *Speedup/P*. Ideally, speedup should scale linearly with *P*, implying that efficiency should have a constant value of 1. Of course, because of various sources of performance loss, efficiency is more typically below 1, and it diminishes as we increase the number of processors. Efficiency greater than 1 represents superlinear speedup.

Superlinear Speedup

The upper curve in the Figure 5.1 graph indicates superlinear speedup, which occurs when speedup grows faster than the number of processors. How is this possible? Surely the sequential program, which is the basis for the speedup computation, could just simulate the *P* processes of the parallel program to achieve an execution time that is no more than *P* times the parallel execution time. Shouldn't superlinear speedup be impossible? There are two reasons why superlinear speedup occurs.

The most common reason is that the computation's working set—that is, the set of pages needed for the computationally intensive part of the program—does not fit in the cache when executed on a single processor, but it does fit into the caches of the multiple processors when the problem is divided amongst them for parallel execution. In such cases the superlinear speedup derives from improved execution time due to the more efficient memory system behavior of the multi-processor execution.

The second case of superlinear speedup occurs when performing a search that is terminated as soon as the desired element is found. When performed in parallel, the search is effectively performed in a different order, implying that the total amount of data searched can actually be less than in the sequential case. Thus, the parallel execution actually performs less work.

Issues with Speedup and Efficiency

Since speedup is a ratio of two execution times, it is a unitless metric that would seem to factor out technological details such as processor speed. Instead, such details insidiously affect speedup, so we must be careful in interpreting speedup figures. There are several concerns.

First, recognize that it is difficult to compare speedup from machines of different generations, even if they have the same architecture. The problem is that different components of a parallel machine are generally improved by different amounts, changing their relative importance. So, for example, processor performance has increased over time, but communication latency has not fallen proportionately. Thus, the time spent communicating will not have diminished as much as the time spent computing. As a result, speedup values have generally decreased over time. Stated another way, the parallel components of a computation have become relatively more expensive compared to the processing components.

The second issue concerns T_S, speedup's numerator, which should be the time for the fastest sequential solution for the given processor and problem size. If T_S is artificially inflated, speedup will be greater. A subtle way to increase T_S is to turn off scalar compiler optimizations for both the sequential and parallel programs, which might seem fair since it is using the same compiler for both programs. However, such a change effectively slows the processors and improves—relatively speaking—communication latency. When reporting speedup, the sequential program should be provided and the compiler optimization settings detailed.

Another common way to increase T_S is to measure the one-processor performance of the *parallel* program. Speedup computed on this basis is called *relative speedup* and should be reported as such. True speedup includes the likely possibility that the sequential algorithm is different than the parallel algorithm. Relative speedup, which simply compares different runs of the same algorithm, takes as the base case an algorithm optimized for concurrent execution but with no parallelism; it will likely run slower because of parallel overheads, causing the speedup to look better. Notice that it can happen that a well-written parallel program on one processor *is* faster than any known sequential program, making it the best sequential program. In such cases we have true speedup, not relative speedup. The situation should be explicitly identified.

Relative speed up cannot always be avoided. For example, for large computations it may be impossible to measure a sequential program on a given problem size, because the data structures do not fit in memory. In such cases relative speedup is all that can be reported. The base case will be a parallel computation on a small number of processors, and the *y*-

axis of the speedup plot should be scaled by that amount. So, for example, if the smallest possible run has P=4, then dividing by the runtime for P=64, will show perfect speedup at y=16.

Another way to inadvertently affect T_S is the "cold start" problem. An easy way to accidentally get a large T_S value is to run the sequential program once and include all of the paging behavior and compulsory cache misses in its timing. As noted earlier it is good practice to run a parallel computation a few times, measuring only the later runs. This allows the caches to "warm up," so that compulsory cache miss times are not unnecessarily included in the performance measure, thereby complicating our understanding of the program's speedup. (Of course, if the program has conflict misses, they should and will be counted.) Properly, most analysts "warm" their programs. But the sequential program should be "warmed," too, so that the paging and compulsory misses do not figure into its execution time. Though easily overlooked, cold starts are also easily corrected.

More worrisome are computations that involve considerable off-processor activity, e.g. disk I/O. One-time I/O bursts, say to read in problem data, are fine because timing measurements can by-pass them; the problem is continual off-processor operations. Not only are they slow relative to the processors, but they greatly complicate the speedup analysis of a computation. For example, if both the sequential and parallel solutions have to perform the same off-processor operations from a single source, huge times for these operations can completely obscure the parallelism because they will dominate the measurements. In such cases it is not necessary to parallelize the program at all. If processors can independently perform the off-processor operations, then this parallelism alone dominates the speedup computation, which will likely look perfect. Any measurements of a computation involving off-processor charges must control their effects carefully.

Performance Trade-Offs

We know that communication time, idle time, wait time, and many other quantities can affect the performance of a parallel computation. The complicating factor is that attempts to lower one cost can increase others. In this section we consider such complications.

Communication vs. computation

Communication costs are a direct expense for using parallelism because they do not arise in sequential computing. Accordingly, it is almost always smart to attempt to reduce them.

Overlap Communication and Computation. One way to reduce communication costs is to overlap communication with computation. Because communication can be performed concurrently with computation, and because the computation must be performed anyway, a perfect overlap—that is, the data is available when it is needed—hides the communication cost perfectly. Partial overlap will diminish waiting time and give partial improvement. The key, of course, is to identify computation that is independent of the communication. From a performance perspective, overlapping is generally a win without

costs. From a programming perspective, overlapping communication and computation can complicate the program's structure.

Redundant Computation. Another way to reduce communication costs is to perform redundant computations. We observed in Chapter 2, for example, that the local generation of a random number, *r*, by all processes was superior to generating the value in one process and requiring all others to reference it. Unlike overlapping, redundant computation incurs a cost because there is no parallelism when all processors must execute the random number generator code. Stated another way, we have increased the total number of instructions to be executed in order to remove the communication cost. Whenever the cost of the redundant computation is less than the communication cost, redundant computation is a win.

Notice that redundant computation *also removes a dependence* from the original program between the generating process and the others that will need the value. It is useful to remove dependences even if the cost of the added computation exactly matches the communication cost. In the case of the random number generation, redundant computation removes the possibility that a client process will have to wait for the server process to produce it. If the client can generate its own random number, it does not have to wait. Such cases complicate the assessing the trade-off.

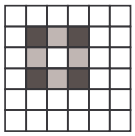
Memory vs. parallelism

Memory usage and parallelism interact in many ways. Perhaps the most favorable is the "cache effect" that leads to superlinear parallel performance, noted above. With all processors having caches, there is more fast memory in a parallel computer. But there are other cases where memory and parallelism interact.

Privatization. For example, parallelism can be increased by using additional memory to break false dependences. One memorable example is the use of `private_count` variables in the Count 3s program, which removed the need for threads to interact each time they recorded the next 3. The effect was to increase the number of count variables from 1 to `t`, the number of threads. It is a tiny memory cost for a big savings in reduced dependences

Batching. One way to reduce the number of dependences is to increase the granularity of interaction. Batching is a programming technique in which work or transmissions are performed as a group. For example, rather than transmitting elements of an array, transmit a whole row or column; rather than grabbing one task from the task queue, get several. Batching effectively raises the granularity (see below) of fine-grain interactions to reduce their frequency. The added memory is simply required to record the items of the batch, and like privatization, is almost always worth the memory costs.

Memoization. Memoization stores a computed value to avoid re-computing later. An example is a stencil optimization: A value is computed based on some combination of the scaled values of its neighbors, shown schematically below,



where color indicates the scaling coefficient; elements such as the corner elements are multiplied by the scale factor four times as the stencil "moves through the array," and memoizing this value can reduce the number of multiplies and memory references. [DETAILED EXAMPLE HERE.] It is a sensible program optimization that removes instruction executions that, strictly speaking, may not result in parallelism improvements. However, in many cases memoization will result in better parallelism, as when the computation is redundant or involves non-local data values.

Padding. Finally, we note that false sharing—references to independent variables that become dependent because they are allocated to the same cache line—can be eliminated by padding data structures to push the values onto different cache lines.

Overhead vs. parallelism

Parallelism and overhead are sometimes at odds. At one extreme, all parallel overhead, such as lock contention, can be avoided by using just one process. As we increase the number of threads the contention will likely increase. If the problem size remains fixed each processor has less work to perform between synchronizations, causing synchronization to become a larger portion of the overall computation. And a smaller problem size implies that there is less computation available to overlap with communication, which will typically increase the wait times for data.

It is the overhead of parallelism that is usually the reason why P cannot increase without bound. Indeed, even computations that could conceptually be solved with a processor devoted to each data point will be buried by overhead before $P=n$. Thus, we find that most programs have an upper limit for each data size at which the marginal value of an additional processor is negative, that is, adding a processor causes the execution time to increase.

Parallelize Overhead. Recall that in Chapter 4, when lock contention became a serious concern, we adopted a combining tree to solve it. In essence, the threads split up the task of accumulating intermediate values into several independent parallel activities.

[THIS SECTION CONTINUES WITH THESE TOPICS]

Load balance vs. parallelism. Increased parallelism can also improve load balance, as it's often easier to distribute evenly a large number of fine-grained units of work than a smaller number of coarse-grained units of work.

Granularity tradeoffs. Many of the above tradeoffs are related to the granularity of parallelism. The best granularity often depends on both algorithmic characteristics, such as the amount of parallelism and the types of dependences, and hardware characteristics,

such as the cache size, the cache line size, and the latency and bandwidth of the machine's communication substrate.

Latency vs. bandwidth. As discussed in Chapter 3, there are many instances where bandwidth can be used to reduce latency.

Scaled speedup vs. Fixed-Size speedup
Choosing a problem size can be difficult**.**

What should we measure?

The kernel or the entire program?
Amdahl's law says that everything is important!

Operating System Costs
Because operating systems are so integral to computation, it is complicated to assess their effects on performance.

Initialization.
How is memory laid out in the parallel computer?

Summary

Exercises

Chapter 6: Programming with Threads

Recall in Chapter 1 that we used threads to implement the count 3's program. In this chapter we'll explore thread-based programming in more detail using the standard POSIX Threads interface. We'll first explain the basic concepts needed to create threads and to let them interact with one another. We'll then discuss issues of safety and performance before we step back and evaluate the overall approach.

Thread Creation and Destruction

Consider the following standard code:

```
 1 #include <pthread.h>
 2 int err;
 3
 4 void main ()
 5 {
 6    pthread_t tid[MAX];   /* An array of Thread ID's, one for each */
 7                          /* thread that is created */
 8
 9    for (i=0; i<t; i++)
10    {
11       err = pthread_create (&tid[i], NULL, count3s_thread, i);
12    }
13
14    for (i=0; i<t; i++)
15    {
16       err = pthread_join_(tid[i], &status[i])
17    }
18 }
```

The above code shows a `main()` function, which then creates—and launches—`t` threads in the first loop, and then waits for the `t` threads to complete in the second loop. We often refer to the creating thread as the ***parent*** and the created threads as ***children***.

The above code differs from the pseudocode in Chapter 1 in a few details. Line 1 includes the pthreads header file, which declares the various pthreads routines and datatypes. Each thread that is created needs its own thread ID, so these thread ID's are declared on line 6. To create a thread, we invoke the `pthread_create()` routine with four parameters. The first parameter is a pointer to a thread ID, which will point to a valid thread ID when this thread successfully returns. The second argument provides the thread's attributes; in this case, the NULL value specifies default attributes. The third parameter is a pointer to the start function, which the thread will execute once it's created. The fourth argument is passed to the start routine, in this case, it represents a unique integer between `0` and `t-1` that is associated with each thread. The loop on line 16 then calls `pthread_join()` to wait for each of the child threads to terminate. If

instead of waiting for the child threads to complete, the `main()` routine finishes and exits using `pthread_exit()`, the child threads will continue to execute. Otherwise, the child threads will automatically terminate when `main()` finishes, since the entire process will have terminated. See Code Specs 1 and 2.

pthread_create()

```
int pthread_create (              // create a new thread
  pthread_t *tid,                 // thread ID
  const pthread_attr_t *attr,     // thread attributes
  void *(*start_routine) (void *),// pointer to function to execute
  void *arg                       // argument to function
  );
```

Arguments:

- The thread ID of the successfully created thread.
- The thread's attributes, explained below; the `NULL` value specifies default attributes.
- The function that the new thread will execute once it is created.
- An argument passed to the `start_routine()`, in this case, it represents a unique integer between `0` and `t-1` that is associated with each thread.

Return value:

- 0 if successful. Error code from <errno.h> otherwise.

Notes:

- Use a structure to pass multiple arguments to the start routine.

Code Spec 1. pthread_create(). The POSIX Threads thread creation function.

pthread_join()

```
int pthread_join (                // wait for a thread to terminate
  pthread_t tid,                  // thread IT to wait for
  void **status                   // exit status
);
```

Arguments:

- The ID of the thread to wait for.
- The completion status of the exiting thread will be copied into `*status` unless status is `NULL`, in which case the completion status is not copied.

Return value:

- 0 for success. Error code from <errno.h> otherwise.

Notes:

- Once a thread is joined, the thread no longer exists, its thread ID is no longer valid, and it cannot be joined with any other thread.

Code Spec 2. pthread_join(). The POSIX Threads rendezvous function pthread_join().

Thread ID's

Each thread has a unique ID of type `pthread_t`. As with all pthread data types, a thread ID should be treated as an ***opaque type***, meaning that individual fields of the structure should never be accessed directly. Because child threads do not know their thread ID, the two routines allow a thread to determine its thread ID, pthread_self(), and to compare two thread ID's, pthread_equal(), see Code Specs 3 and 4.

pthread_self()

```
pthread_t pthread_self ();         // Get my thread ID
```

Return value:

- The ID of the thread that called this function.

Code Spec 3. pthread_self(). The POSIX Threads function to fetch a thread's ID.

pthread_equal()

```
int pthread_equal (                 // Test for equality
  pthread_t t1,                     // First operand thread ID
  pthread_t t2                      // Second operand thread ID
);
```

Arguments:

- Two thread ID's

Return value:

- Non-zero if the two thread ID's are the same (following the C convention).
- 0 if the two threads are different.

Code Spec 4. pthread_equal(). The POSIX Threads function to compare two thread IDs for equality.

Destroying Threads

There are three ways that threads can terminate.

1. A thread can return from the start routine.
2. A thread can call `pthread_exit()`.
3. A thread can be ***cancelled*** by another thread.

In each case, the thread is destroyed and its resources become unavailable.

void pthread_exit()

```
void pthread_exit (                    // terminate a thread
  void *status                         // completion status
);
```

Arguments:

- The completion status of the thread that has exited. This pointer value is available to other threads.

Return value:

- None

Notes:

- When a thread exits by simply returning from the start routine, the thread's completion status is set to the start routine's return value.

Code Spec 5. pthread_exit(). The POSIX Threads thread termination function pthread_exit().

Thread Attributes

Each thread maintains its own properties, known as attributes, which are stored in a structure of type `pthread_attr_t`. For example, threads can be either ***detached*** or ***joinable***. Detached threads cannot be joined with other threads, so they have slightly lower overhead in some implementations of POSIX Threads. For parallel computing, we will rarely need detached threads. Threads can also be either ***bound*** or ***unbound***. Bound threads are scheduled by the operating system, whereas unbound threads are scheduled by the Pthreads library. For parallel computing, we typically use bound threads so that each thread provides physical concurrency.

POSIX Threads provides routines to initialize thread attributes, set their attributes, and destroy attributes, as shown in Code Spec 6.

Code Spec 6. pthread attributes. An example of how thread attributes are set in the POSIX Threads interface.

Thread Attributes

```
pthread_attr_t  attr;              // Declare a thread attribute
pthread_t       tid;

pthread_attr_init(&attr);          // Initialize a thread attribute
pthread_attr_setdetachstate(&attr, // Set the thread attribute
                            PTHREAD_CREATE_UNDETACHED);

pthread_create (&tid, &attr, start_func, NULL); // Use the attribute
                                                // to create a thread
pthread_join(tid, NULL);
pthread_attr_destroy(&attr);       // Destroy the thread attribute
```

Example

The following example illustrates a potential pitfall that can occur because of the interaction between parent and child threads. The parent thread simply creates a child thread and waits for the child to exit. The child thread does some useful work and then exits, returning an error code. Do you see what's wrong with this code?

```
 1 #include <pthread.h>
 2
 3 void main ()
 4 {
 5    pthread_t tid;
 6    int *status;
 7
 8    pthread_create (&tid, NULL, start, NULL);
 9    pthread_join_(tid, &status);
10 }
11
12 void start()
13 {
14    int errorcode;
15    /* do something useful. . . */
16
17    if (. . . )
18       errorcode = something;
19    pthread_exit(&errorcode);
20 }
```

The problem occurs in the call to `pthread_exit()` on line 17, where the child is attempting to return an error code to the parent. Unfortunately, because `errorcode` is declared to be local to the `start()` function, the memory for `errorcode` is allocated on the child thread's stack. When the child exits, its call stack is de-allocated, and the parent has a dangling pointer to `errorcode`. At some point in the future, when a new procedure is invoked, it will over-write the stack location where `errorcode` resides, and the value of `errorcode` will change.

Mutual Exclusion

We can now create and destroy threads, but to allow threads to interact constructively, we need methods of coordinating their interaction. In particular, when two threads share access to memory, it is often useful to employ a lock, called a ***mutex***, to provide *mutual exclusion* or mutually exclusive access to the variable. As we saw in Chapter 1, without mutual exclusion, race conditions can lead to unpredictable results, because when multiple threads execute the following code, the `count` variable, which is shared among all threads, will not be atomically updated.

```
    for (i=start; i<start+length_per_thread; i++)
    {
      if (array[i] == 3)
      {
          count++;
```

```
    }
  }
```

The solution, of course, is to protect the update of count using a mutex, as shown below:

```
 1 pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
 2
 3 void count3s_thread (int id)
 4 {
 5    /* Compute portion of array that this thread should work on */
 6    int length_per_thread = length/t;
 7    int start = id * length_per_thread;
 8
 9    for (i=start; i<start+length_per_thread; i++)
10    {
11       if (array[i] == 3)
12       {
13          pthread_mutex_lock(&lock);
14          count++;
15          pthread_mutex_unlock(&lock);
16       }
17    }
18 }
```

Line 1 shows how a mutex can be statically declared. Like threads, mutexes have attributes, and by initializing the mutex to `PTHREAD_MUTEX_INITIALIZER`, the mutex is assigned default attributes. To use this mutex, its address is passed to the lock and unlock routines on lines 13 and 15, respectively. The appropriate discipline, of course, is to bracket all ***critical sections***, that is, code that must be executed atomically by only one thread at a time, by the locking of a mutex upon entrance and the unlocking of a mutex upon exit. Only one thread can acquire the mutex at any one time, so a thread will block if it attempts to acquire a mutex that is already held by another thread. When a mutex is unlocked, or ***relinquished***, one of the threads that was blocked attempting to acquire the lock will become unblocked and granted the mutex. The POSIX Threads standard defines no notion of fairness, so the order in which the locks are acquired is *not* guaranteed to match the order in which the threads attempted to acquire the locks.

It is an error to unlock a mutex that has not been locked, and it is an error to lock a mutex that is already held. The latter will lead to ***deadlock***, in which the thread cannot make progress because it is blocked waiting for an event that cannot happen. We will discuss deadlock and techniques to avoid deadlock in more detail later in the chapter.

Acquiring and Releasing Mutexes

```
int pthread_mutex_lock(             // Lock a mutex
   pthread_mutex_t *mutex);

int pthread_mutex_unlock(           // Unlock a mutex
   pthread_mutex_t *mutex);

int pthread_mutex_trylock(          // Non-blocking lock
   pthread_mutex_t *mutex);
```

Arguments:

- Each function takes the address of a mutex variable.

Return value:

- 0 if successful. Error code from <errno.h> otherwise.

Notes:

- The `pthread_mutex_trylock()` routine attempts to acquire a mutex but will not block. This routine returns `EBUSY` if the mutex is locked.

Code Spec 7. The POSIX Threads routines for acquiring and releasing mutexes.

Serializability

It's clear that our use of mutexes provides atomicity: the thread that acquires the mutex `m` will execute the code in the critical section until it relinquishes the mutex. Thus, in our above example, the counter will be updated by only one thread at a time. Atomicity is important because it ensures ***serializability***: A concurrent execution is serializable if the execution is guaranteed to execute in an order that corresponds to *some* serial execution of those threads.

Mutex Creation and Destruction

In our above example, we knew that only one mutex was needed, so we were able to statically allocate it. In cases where the number of required mutexes is not known *a priori*, we can instead allocate and deallocate mutexes dynamically. Code Spec 8 shows how such a mutex is dynamically allocated, initialized with default attributes, and destroyed.

Mutex Creation and Destruction

```
int pthread_mutex_init(             // Initialize a mutex
   pthread_mutex_t *mutex,
   pthread_mutexattr_t *attr);

int pthread_mutex_destroy (         // Destroy a mutex
   pthread_mutex_t *mutex);

int pthread_mutexattr_init(         // Initialize a mutex attribute
   pthread_mutexattr_t *attr);

int pthread_mutexattr_destroy (     // Destroy a mutex attribute
   pthread_mutexattr_t *attr);
```

Arguments:

- The `pthread_mutex_init()` routine takes two arguments, a pointer to a mutex and a pointer to a mutex attribute. The latter is presumed to have already been initialized.
- The `pthread_mutexattr_init()` and `pthread_mutexattr_destroy()` routines take a pointer to a mutex attribute as arguments.

Notes:

- If the second argument to `pthread_mutex_init()` is `NULL`, default attributes will be used.

Code Spec 8. The POSIX Threads routines for dynamically creating and destroying mutexes.

Dynamically Allocated Mutexes

```
pthread_mutex_t *lock;             // Declare a pointer to a lock

lock = (pthread_mutex_lock_t *) malloc(sizeof (pthread_mutex_t));

pthread_mutex_init(lock, NULL);
   /*
    * Code that uses this lock.
    */

pthread_mutex_destroy (lock);
free (lock);
```

Code Spec 9. An example of how dynamically allocated mutexes are used in the POSIX Threads interface.

Synchronization

Mutexes are sufficient to provide atomicity for critical sections, but in many situations we would like a thread to synchronize its behavior with that of some other thread. For example, consider a classic bounded buffer problem in which one or more threads put

items into a circular buffer while other threads remove items from the same buffer. As shown in Figure 1, we would like the producers to stop producing data—to wait—if the consumer is unable to keep up and the buffer becomes full, and we would like the consumers to wait if the buffer is empty.

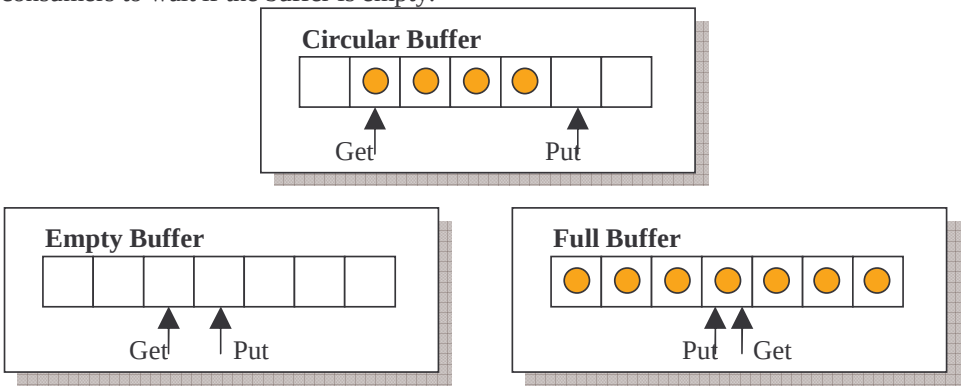


Figure 1. A bounded buffer with producers and consumers. The Put and Get cursors indicate where the producers will insert the next item and where the consumers will remove its next item, respectively. When the buffer is empty, the consumers must wait. When the buffer is full, the producers must wait.

Such synchronization is supported by ***condition variables***, which are a more general form of synchronization than joining threads. A condition variable allows threads to wait until some condition becomes true, at which point one of the waiting threads is non-deterministically chosen to stop waiting. We can think of the condition variable as a gate (see Figure 2). Threads wait at the gate until some condition is true. Other threads open the gate to signal that the condition has become true, at which point one of the waiters is allowed to enter the gate and resume execution. If a thread opens the gate when there are no threads waiting, the signal has no effect.

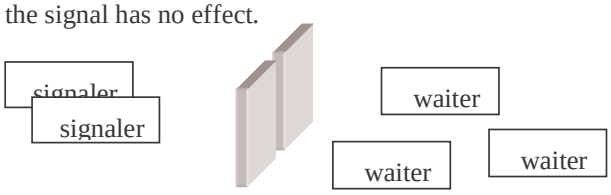


Figure 2. Condition variables act like a gate. Threads wait outside the gate by calling `pthread_cond_wait()`, and threads open the gate by calling `pthread_cond_signal()`. When the gate is opened, one waiter is allowed through. If there are no waiters when the gate is opened, the signal has no effect.

We can solve our bounded buffer problem with two condition variables, `nonempty` and `nonfull`, as shown below.

```
1 pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
2 pthread_cond_t nonempty = PTHREAD_COND_INITIALIZER;
3 pthread_cond_t nonfull= PTHREAD_COND_INITIALIZER;
4 Item buffer[SIZE];
5 int in = 0;                    // Buffer index for next insertion
6 int out = 0;                   // Buffer index for next removal
```

```
 7
 8 void put (Item x)                // Producer thread
 9 {
10    pthread_mutex_lock(&lock);
11    while (in – out) == SIZE)     // While buffer is full
12       pthread_cond_wait(&nonfull, &lock);
13    buffer[in % SIZE] = x;
14    in++;
15    pthread_cond_signal(&nonempty);
16    pthread_mutex_unlock(&lock);
17 }
18
19 Item get()                      // Consumer thread
20 {
21    Item x;
22    pthread_mutex_lock(&lock);
23    while (out – in)              // While buffer is empty
24       pthread_cond_wait(&nonempty, &lock);
25    x = buffer[out % SIZE];
26    out++;
27    pthread_cond_signal(&nonfull);
28    pthread_mutex_unlock(&lock);
29    return x;
30 }
```

Of course, since multiple threads will be updating these condition variables, we need to protect their access with a mutex, so Line 1 declares a mutex. The remaining declarations define a buffer, `buffer`, and its two cursors, `in` and `out`, which indicate where to insert the next item and where to remove the next item. The two cursors wrap around when they exceed the bounds of `buffer`, yielding a circular buffer.

Given these data structures, the producer thread executes the `put()` routine, which first acquires the mutex to access the condition variables. (This code omits the actual creation of the producer and consumer threads, which are assumed to iteratively invoke the `put()` and `get()` routines, respectively.) If the buffer is full, the producer waits on the `nonfull` condition so that it will later be awakened when the buffer becomes non-full. If this thread blocks, the mutex that it holds must be relinquished to avoid deadlock. Because these two events—the releasing of the mutex and the blocking of this waiting thread—must occur atomically, they must be performed by `pthread_cond_wait()`, so the mutex is passed as a parameter to `pthread_cond_wait()`. When the producer resumes execution after returning from the wait on Line 12, the protecting mutex will have been re-acquired by the system on behalf of the producer.

In a moment we will explain the need for the `while` loop on Line 11, but for now assume when the producer executes Line 13, the buffer is not full, so it is safe to insert a new item and to bump the `In` cursor by one. At this point, the buffer cannot be empty because the producer has just inserted an element, so the producer signals that the buffer is nonempty, waking one more consumers that may be waiting on an empty buffer. If there are no waiting consumers, the signal is lost. Finally, the producer releases the

mutex and exits the routine. The consumer thread executes the `get()` routine, which operates in a very similar manner.

pthread_cond_wait()

```
int pthread_cond_wait(
   pthread_cond_t *cond,            // Condition to wait on
   pthread_mutex_t *mutex);         // Protecting mutex

int pthread_cond_timedwait (
   pthread_cond_t *cond,
   pthread_mutex_t *mutex,
   const struct timespec *abstime); // Time-out value
```

Arguments:

- A condition variable to wait on.
- A mutex that protects access to the condition variable. The mutex is released before the thread blocks, and these two actions occur atomically. When this thread is later unblocked, the mutex is reacquired on behalf of this thread.
-

Return value:

- 0 if successful. Error code from <errno.h> otherwise.

Code Spec 10. pthread_cond_wait(): The POSIX Thread routines for waiting on condition variables.

pthread_cond_signal()

```
int pthread_cond_signal(
   pthread_cond_t *cond);           // Condition to signal

int pthread_cond_broadcast (
   pthread_cond_t *cond);           // Condition to signal
```

Arguments:

- A condition variable to signal.

Return value:

- 0 if successful. Error code from <errno.h> otherwise.

Notes:

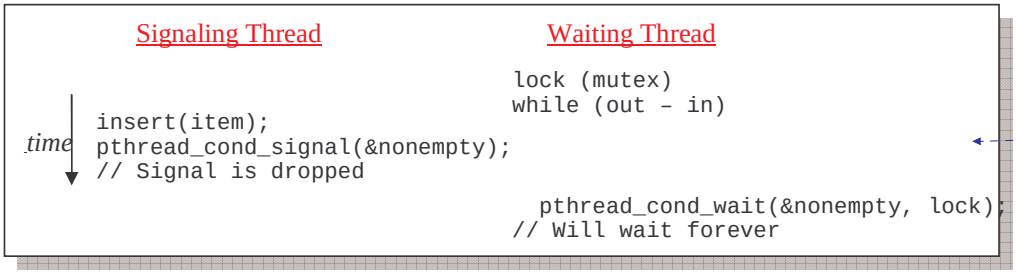
- These routines have no effect if there are no threads waiting on `cond`. In particular, there is no memory of the signal when a later call is made to `pthread_cond_wait()`.
- The `pthread_cond_signal()` routine may wake up more than one thread, but only one of these threads will hold the protecting mutex.
- The `pthread_cond_broadcast()` routine wakes up all waiting threads. Only one awakened thread will hold the protecting mutex.

Code Spec 11. pthread_cond_signal(). The POSIX Threads routines for signaling a condition variable.

Protecting Condition Variables

Let us now return to the `while` loop on Line 11 of the bounded buffer program. If our system has multiple producer threads, this loop is essential because `pthread_cond_signal()` can wake up multiple waiting threads[3], of which only one will hold the protecting mutex at any particular time. Thus, at the time of the signal, the buffer is not full, but when any particular thread acquires the mutex, the buffer may have become full again, in which case the thread should call `pthread_cond_wait()` again. When the producer thread executes Line 13, the buffer is necessarily not full, so it is safe to insert a new item and to bump the `In` cursor.

We see on Lines 15 and 27 that the call to `pthread_cond_signal()` is also protected by the lock. The following example shows that this protection is necessary.



Formatted: Bullets and Numbering

Figure 3. Example of why a signaling thread needs to be protected by a mutex.

In this example, the waiting thread, in this case the consumer, acquires the protecting mutex and finds that the buffer is empty, so it executes `pthread_cond_wait()`. If the signaling thread, in this case the producer, does not protect the call to `pthread_cond_signal()` with a mutex, it could insert an item into the buffer immediately after the waiting thread found it empty. If the producer then signals that the buffer is non-empty before the waiting thread executes the call to `pthread_cond_wait()`, the signal will be dropped and the consumer thread will not realize that the buffer is actually not empty. In the case that the producer only inserts a single item, the waiting thread will needlessly wait forever.

The problem, of course, is that there is a race condition involving the manipulation of the buffer. The obvious solution is to protect both the call to `pthread_cond_signal()` with the same mutex that protects the call to `pthread_cond_wait`(), as shown in the code for our bounded buffer solution. Because both the `Put()` and `Get()` routines are protected by the same mutex, we have three critical sections related to the nonempty buffer, as shown in Figure 4, and in no case can the signal be dropped while a waiting thread thinks that the buffer is empty.

[3] These semantics are due to implementation details. In some cases it can be expensive to ensure that exactly one waiter is unblocked by a signal.

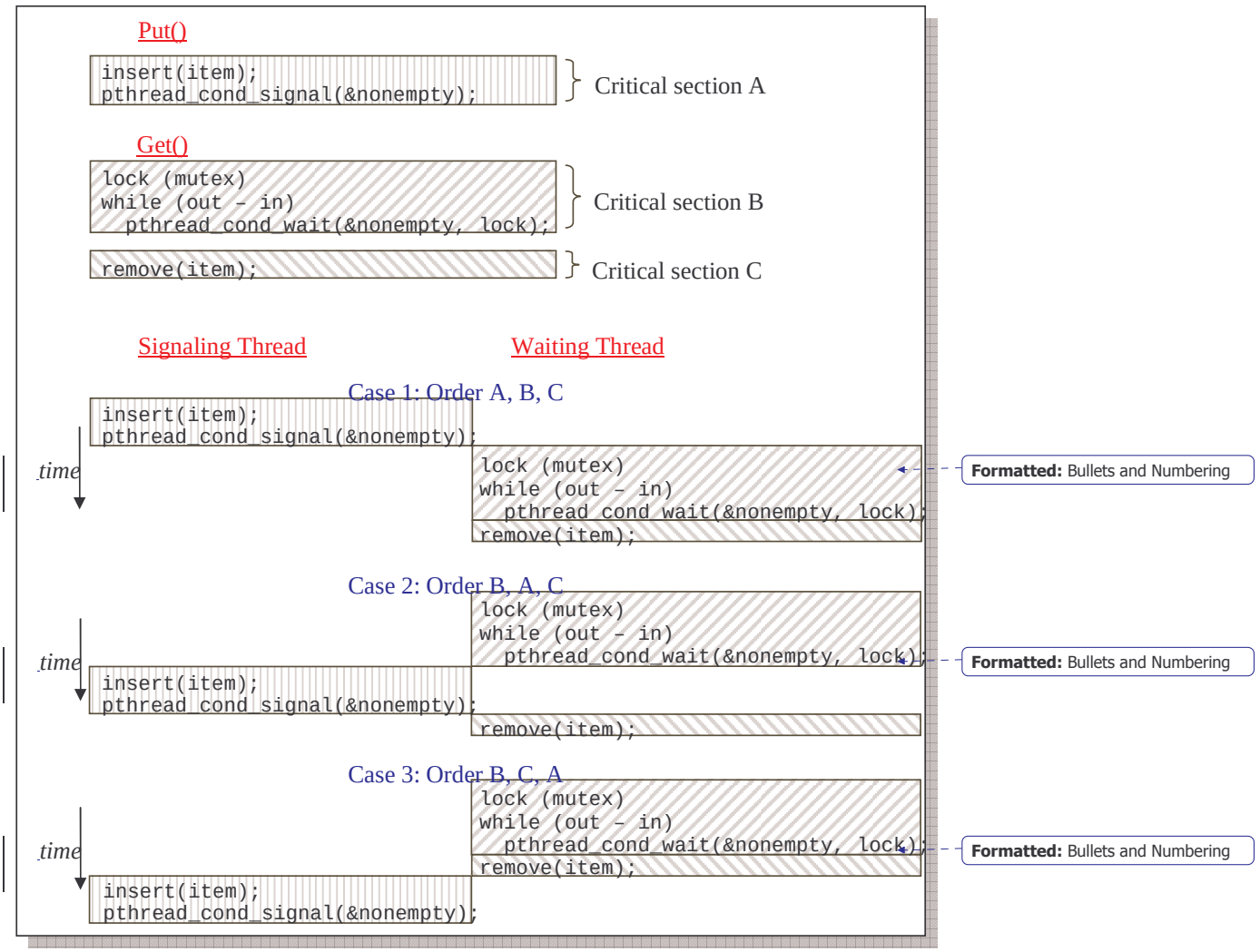


Figure 4. Proper locking of the signaling code prevents race conditions. By identifying and protecting three critical sections pertaining to the nonempty buffer, we guarantee that each of A, B, and C will execute atomically, so our problem from Figure 3 is avoided: There is no way for the `Put()` routine's signal to be dropped while a thread executing the `Get()` routine thinks that the buffer is empty.

We have argued that the call to `pthread_cond_signal()` must be protected by the same mutex that protects the waiting code. However, notice that the race condition occurs not from the signaling of the condition variable, but with the access to the shared buffer. Thus, we could instead simply protect any code that manipulates the shared buffer, which implies that the `Put()` code could release the mutex immediately after inserting an item into the buffer but before calling `pthread_cond_signal()`. This new code is not only legal, but it produces better performance because it reduces the size of the critical section, thereby allowing more concurrency.

Creating and Destroying Condition Variables

Like threads and mutexes, condition variables can be created and destroyed either statically or dynamically. In our bounded buffer example above, the static condition variables were both given default attributes by initializing them to `PTHREAD_COND_INITIALIZER`. Condition variables can be dynamically allocated as indicated in Code Spec 12.

Dynamically Allocated Condition Variables

```
int pthread_cond_init(
   pthread_cond_t *cond,            // Condition variable
   const pthread_condattr_t *attr); // Condition attribute

int pthread_cond_destroy (
   pthread_cond_t *cond);           // Condition to destroy
```

Arguments:

- Default attributes are used if `attr` is `NULL`.

Return value:

- 0 if successful. Error code from <errno.h> otherwise.

Code Spec 12. The POSIX Threads routines for dynamically creating and destroying condition variables.

Waiting on Multiple Condition Variables

In some cases a piece of code cannot execute unless multiple conditions are met. In these situations the waiting thread should test all conditions simultaneously, as shown below.

```
 1 EatJuicyFruit()
 2 {
 3    pthread_mutex_lock(&lock);
 4    while (apples==0 && oranges==0)
 5    {
 6       pthread_cond_wait(&more_apples, &lock);
 7       pthread_cond_wait(&more_oranges, &lock);
 8    }
 9    /* Eat both an apple and an orange */
10    pthread_mutex_unlock(&lock);
11 }
```

By contrast, the following code, which waits on each condition in turn, fails because there is no guarantee that both conditions will be true at the same time. That is, after returning from the first call to `pthread_cond_wait()` but before returning from the second call to `pthread_cond_wait`(), some other thread may have removed an apple, making the first condition false.

```
 1 EatJuicyFruit()
 2 {
```

```
 3    pthread_mutex_lock(&lock);
 4    while (apples==0)
 5       pthread_cond_wait(&more_apples, &lock);
 6    while (oranges==0)
 7       pthread_cond_wait(&more_oranges, &lock);
 8
 9    /* Eat both an apple and an orange */
10    pthread_mutex_unlock(&lock);
11 }
```

Thread-Specific Data

It is often useful for threads to maintain private data that is not shared. For example, we have seen examples where a thread index is passed to the start function so that the thread knows what portion of an array to work on. This index can be used to give each thread a different element of an array, as shown below:

```
1  . . .
2
3    for (i=0; i<t; i++)
4       err = pthread_create (&tid[i], NULL, start_function, i);
5
6 void start_function(int index)
7 {
8    private_count[index] = 0;
9 . . .
```

A problem occurs, however, if the code that accesses `index` occurs in a function, `foo(),` which is buried deep within other code. In such situations, how does `foo()` get the value of `index`? One solution is to pass the `index` parameter to every procedure that calls `foo(),` including procedures that call `foo()` indirectly through other procedures. This solution is cumbersome, particularly for those procedures that require the parameter but do not directly use it.

Instead, what we really want is a variable that is global in scope to all code but which can have different values for each thread. POSIX Threads supports such a notion in the form of ***thread-specific data***, which uses a set of ***keys***, which are shared by all threads in a process, but which map to different pointer values for each thread. (See Figure 4.)

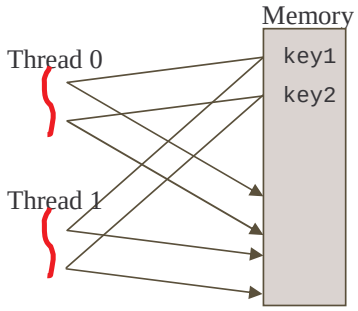


Figure 5. Example of thread-specific data in POSIX Threads. Thread-specific data are accessed by keys, which map to different memory locations in different threads.

As a special case, the error values for POSIX Threads routines are returned in thread-specific data, but such data does not use the interface defined by Code Specs 13-17. Instead, each thread has its own value of `errno`.

Thread-Specific Data

```
pthread_key_t *my_index;
#define index (pthread_getspecific (my_index))

main()
{
   . . .
   pthread_key_create(&my_index, 0);
   . . .
}

void start_routine(int id)
{
   pthread_setspecific (my_index, id);
   . . .
}
```

Notes:

- Avoid accessing index in a tight inner loop because each access requires a procedure call.

Code Spec 13. Example of how thread-specific data is used. Once initialized with this code, any procedure can access the value of `my_index`.

pthread_key_create

```
int pthread_key_create (
   pthread_key_t *key,              // The key to create
   void (*destructor) (void*));     // Destructor function
```

Arguments:

- A pointer to the key to create.
- A destructor function. `NULL` indicates no destructor.

Return value:

- 0 if successful. Error code from <errno.h> otherwise.

Notes:

- Avoid accessing index in a tight inner loop because each access requires a procedure call.

Code Spec 14. pthread_key_create. POSIX Thread routine for creating a key for thread-specific data.

pthread_key_delete

```
int pthread_key_delete (
   pthread_key_t *key);              // The key to delete
```

Arguments:

- A pointer to the key to delete.

Return value:

- 0 if successful. Error code from <errno.h> otherwise.

Notes:

- Destructors will not be called.

Code Spec 15. pthread_key_delete. POSIX Thread routine for deletin a key.

pthread_setspecific

```
int pthread_setspecific (
   pthread_key_t *key,              // Key to set
   void *value));                   // Value to set
```

Arguments:

- A pointer to the key to be set.
- The value to set.

Return value:

- 0 if successful. Error code from <errno.h> otherwise.

Notes:

- It is an error to call `pthread_setspecific()` before the key has been created or after the key has been deleted.

Code Spec 16. pthread_setspecific. POSIX Thread routine for setting the value of thread-specific data.

pthread_getspecific

```
int pthread_getspecific (
   pthread_key_t *key);             // Key to value
```

Arguments:

- Key whose value is to be retrieved.

Return value:

- Value of `key` for the calling thread.

Notes:

- The behavior is undefined if a thread calls `pthread_getspecific()` before the key is created or after the key is deleted.

Code Spec 17. pthread_getspecific. POSIX Thread routine for getting the value of some thread-specific data.

Safety Issues

Many types of errors can occur from the improper use of locks and condition variables. We've already mentioned the problem of double-locking, which occurs when a thread attempts to acquire a lock that it already holds. Of course, problems also arise if a thread accesses some shared variable without locking it, or if a thread acquires a lock and does not relinquish it. One particularly important problem is that of avoiding deadlock. This section discusses various methods of avoiding deadlock and other potential bugs.

Deadlock

There are four necessary conditions for deadlock:

1. ***Mutual exclusion***: a resource can be assigned to at most one thread.
2. ***Hold and wait***: a thread that holds resources can request new resources.
3. ***No preemption***: a resource that is assigned to a thread can only be released by the thread that holds it.
4. ***Circular wait***: there is a cycle in which each thread waits for a resource that is assigned to another thread. (See Figure 6.)

Of course, for threads-based programming, mutexes are resources that can cause deadlock. There are two general approaches to dealing with deadlock: (1) prevent deadlocks, and (2) allow deadlock to occur, but detect their occurrence and then break the deadlock. We will focus on the deadlock avoidance, because POSIX Threads does not provide a mechanism for breaking locks.

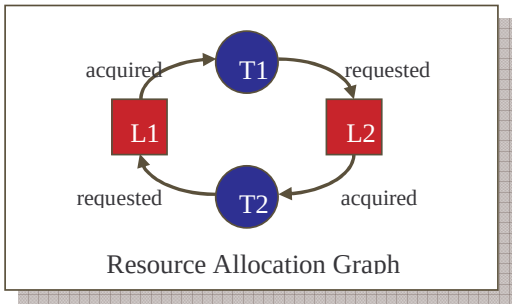


Figure 6. Deadlock example. Threads T1 and T2 hold locks L1 and L2, respectively, and each thread attempts to acquire the other lock, which cannot be granted.

Lock Hierarchies

A simple way to prevent deadlocks is to prevent cycles in the resource allocation graph. We can prevent cycles by imposing an ordering on the locks and by requiring all threads to acquire their locks in the same order. Such a discipline is known as a ***lock hierarchy.***

One problem with a lock hierarchy is that it requires programmers to know *a priori* what locks a thread needs to acquire. Suppose that after acquiring locks L1, L3, and L7, a thread finds that it needs to also acquire lock L2, which would violate the lock hierarchy. One solution would be for the thread to release locks L3 and L7, and then reacquire locks L2, L3, and L7 in that order. Of course, this strict adherence to the lock hierarchy is expensive. A better solution would be to attempt to lock L2 using `pthread_mutex_trylock()`(see Code Spec 7), which either obtains the lock or immediately returns without blocking. If the thread is unable to obtain lock L2, it must resort to the first solution.

Monitors

The use of locks and condition variables is error prone because it relies on programmer discipline. An alternative is to provide language support, which would allow a compiler to enforce mutual exclusion and proper synchronization. A ***monitor*** is one such language construct, and although almost no modern language provides such a construct, it can be implemented in an object oriented setting, as we will soon see. A monitor encapsulates code and data and enforces a protocol that ensures mutual exclusion. In particular, a monitor has a set of well-defined entry points, its data can only be accessed by code that resides inside the monitor, and at most one thread can execute the monitor's code at any time. Monitors also provide condition variables for signaling and waiting, and they ensure that the use of these condition variables obeys the monitor's protocol. Figure 7 shows a graphical depiction of a monitor.

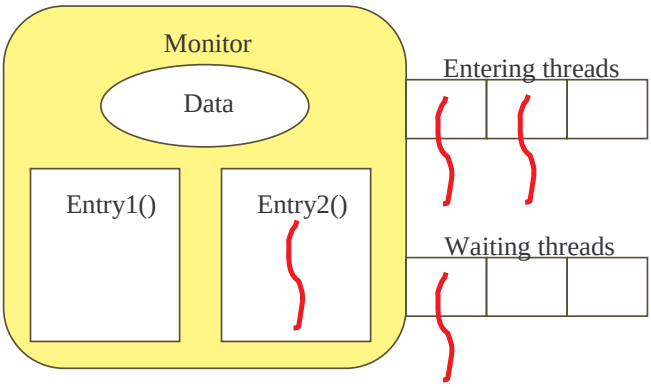


Figure 7. Monitors provide an abstraction of synchronization in which only one thread can access the monitor's data at any time. Other threads are blocked either waiting to enter the monitor or waiting on events inside the monitor.

We can implement monitors in an object oriented language, such as C++, as shown below.

```
 1 class BoundedBuffer
 2 {                                 // Emulate a monitor
 3 private:
 4    pthread_mutex_t lock;          // Synchronization variables
 5    pthread_cond_t nonempty, nonfull;
 6    Item *buffer;                  // Shared data
 7    int in, out;                   // Cursors
 8    CheckInvariant();
 9
10 public:
11    BoundedBuffer(int size);       // Constructor
12    ~BoundedBuffer();              // Destructor
13    void put(Item x);
14    Item get();
15 }
16
17 // Constructor and Destructor
18 BoundedBuffer::Bounded (int size)
19 {
20    // Initialize synchronization variables
21    pthread_mutex_init(&lock, NULL);
22    pthread_cond_init(&nonempty, NULL);
23    pthread_cond_init(&nonfull, NULL);
24
25    // Initialize the buffer
26    buffer = new Item[size];
27    in = out = 0;
28 }
29
30 BoundedBuffer::~BoundedBuffer()
31 {
32    pthread_mutex_destroy(&lock);
33    pthread_cond_destroy(&nonempty);
34    pthread_cond_destroy(&nonfull);
```

```
35    delete buffer;
36 }
37
38 // Member functions
39 BoundedBuffer::Put(Item x)
40 {
41    pthread_mutex_lock(&lock);
42    while (in - out == size)      // while buffer is full
43       pthread_cond_wait(&nonfull, &lock);
44    buffer[in%size] = x;
45    in++;
46    pthread_cond_signal(&nonempty);
47    pthread_mutex_unlock(&lock);
48 }
49
50 Item BoundedBuffer::Get()
51 {
52    pthread_mutex_lock(&lock);
53    while (in = out)              // while buffer is empty
54       pthread_cond_wait(&nonempty, &lock);
55    x = buffer[out%size];
56    out++;
57    pthread_cond_signal(&nonfull);
58    pthread_mutex_unlock(&lock);
59    return x;
60 }
```

Monitors not only enforce mutual exclusion, but they provide an abstraction that can simplify how we reason about concurrency. In particular, the limited number of entry points facilitates the preservation of invariants. Monitors have ***internal*** functions and ***external*** functions. Internal functions assume that the monitor lock is held. By contrast, external functions must acquire the monitor lock before executing, so external functions cannot invoke each other. In this setting ***invariants*** are properties that can are assumed to be true upon entry and which must be restored upon exit. These invariants may be violated while the monitor lock is held, but they must be restored before the monitor lock is released. This use of invariants is graphically depicted in Figure 8.

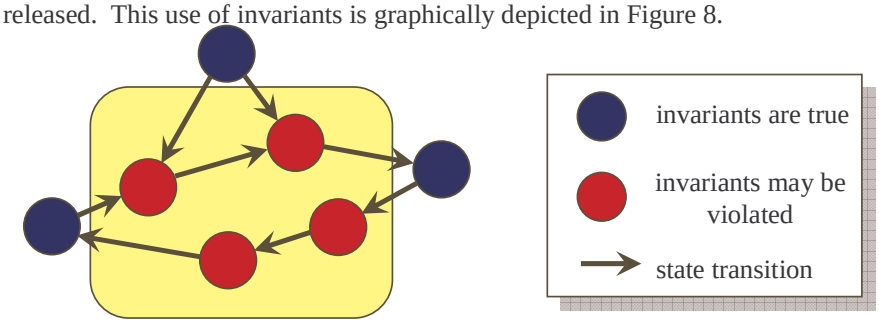


Figure 8. Monitors and invariants. The red circles represent program states in which the invariants may be violated. The blue circles represent program states in which the invariants are assumed to be maintained.

For example, in our bounded buffer example, we have two invariants:

1. The distance between the In and Out cursors is at most the size of the buffer.

2. The In cursor is not left of the Out cursor. (In Figure 1, the Put arrow is not left of the Get arrow.)

Once we have identified our invariants, we can write a routine that checks all invariants, and this routine can be invoked before every entrance to the monitor and after every exit from the monitor. The use of such invariants can be a significant debugging tool. For example, the following code checks these invariants to help debug the monitor's internal routines.

```
 1 BoundedBuffer::CheckInvariant()
 2 {
 3    if (in – out > size)           // Check invariant (1)
 4       return (0);
 5    if (in < out)                  // Check invariant (2)
 6       return (0);
 7    return (1);
 8 }
 9
10 Item BoundedBuffer::Get()
11 {
12    pthread_mutex_lock(&lock);
13    assert(CheckInvariant());     // Check on every entrance
14    while (in = out)              // while buffer is empty
15    {
16       assert(CheckInvariant());  // Check on every exit
17       pthread_cond_wait(&nonempty, &lock);
18       assert(CheckInvariant());
19    }
20    x = buffer[out%size];
21    out++;
22    pthread_cond_signal(&nonfull);
23    assert(CheckInvariant());
24    pthread_mutex_unlock(&lock);
25    return x;
26 }
```

As we have mentioned before, the call to `pthread_cond_wait()` may implicitly release the lock, so it is a potential monitor exit, and the return from `pthread_cond_wait()` will implicitly re-acquire the lock, so it is a monitor entrance.

Re-entrant Monitors

While monitors help enforce a locking discipline, they do not ensure that all concurrency problems go away. For example, if a procedure in a monitor attempts to re-enter the monitor by calling an entry procedure, deadlock will occur. To avoid this problem, the procedure should first restore all invariants, release the monitor lock, and then try to re-enter the monitor. Of course, such a structure means that atomicity is lost. This same problem occurs if a monitor procedure attempts to re-enter the monitor indirectly by calling some external procedure that then tries to enter the monitor, so monitor procedures should invoke external routines with care.

Monitor functions that take a long time or wait for some outside event will prevent other threads from entering the monitor. To avoid such problems, such functions can often be rewritten to wait on a condition, thereby releasing the lock and increasing parallelism. As with re-entrant routines, such functions will need to restore invariants before releasing the lock.

Performance Issues

We saw in Chapter 3 that dependences among threads constrain parallelism. Because locks dynamically impose dependences among threads, the granularity of our locks can greatly affect parallelism. At one extreme, the coarsest locking scheme uses a single lock for all shared variables, which is simple but severely limits concurrency when there is sharing. At the other extreme, fine-grained locks may protect small units of data. For example, in our Count 3's example, we might use a different lock to protect each node of the accumulation tree. As an intermediate point, we might use one lock for the entire accumulation tree. As we reduce the lock granularity, the overhead of locking increases while the amount of available parallelism increases.

Readers and Writers Example: Granularity Issues

Just as there are different granularities for locking, there are different granularities of condition variables. Consider a resource that can be shared by multiple readers or accessed exclusively by a single writer. To coordinate access to such a resource, we can provide four routines—`AcquireExclusive()`, `ReleaseExclusive()`, `AcquireShared()`, and `ReleaseShared()`—that readers and writers can invoke. These routines are each protected by a single mutex, and they collectively use two condition variables. To acquire the resource in exclusive mode, a thread waits on the `wBusy` condition variable, which ensures that no readers are still accessing the resource. When the last reader is done accessing a resource in shared mode, it signals the `wBusy` condition to allow the writer to proceed. Likewise, when a writer is done accessing the resource in exclusive mode, it signals the `rBusy` condition to allow any readers to have access to the resource; and before accessing the shared resource, threads wait on the `rBusy` condition variable.

```
 1 int readers;                      // Negative value => active writer
 2 pthread_mutex_t lock;
 3 pthread_cond_t rBusy, wBusy;      // Use separate condition variables
 4                                   // for readers and writers
 5 AcquireExclusive()
 6 {
 7    pthread_mutex_lock(&lock);
 8    while (readers != 0)
 9       pthread_cond_wait(&wBusy, &lock);
10    readers = -1;
11    pthread_mutex_unlock(&lock);
12 }
13
14 AcquireShared()
```

```
15 {
16    pthread_mutex_lock(&lock);
17    readWaiters++;
18    while (readers<0)
18       pthread_cond_wait(&rBusy, &lock);
19    readWaiters--;
20    pthread_mutex_unlock(&lock);
21 }
22
23 ReleaseExclusive()
24 {
25    pthread_mutex_lock(&lock);
26    readers = 0;
27    pthread_cond_broadcast(&rBusy);  // Only wake up readers
28    pthread_mutex_unlock(&lock);
29 }
30
31 ReleaseShared(
32 {
33    int doSignal;
34
35    pthread_mutex_lock(&lock);
36    readers--;
37    doSignal = (readers==0)
38    pthread_mutex_unlock(&lock);
39    if (doSignal)                    // Signal is performed outside
40       pthread_cond_signal(&wBusy);  // of critical section
41 }
```

Two points about this code are noteworthy.

First, the code uses two condition variables, but it's natural to wonder if one condition variable would suffice. In fact, one condition variable could be used, as shown below, and the code would be functionally correct. Unfortunately, by using a single condition variable, the code suffers from ***spurious wakeups*** in which writers can be awoken only to immediately go back to sleep. In particular, when `ReleaseExclusive()` is called both readers and writers are signaled, so writers will suffer spurious wakeups whenever any reader is also waiting on the condition. Our original solution avoids spurious wakeups by using two condition variables, which forces exclusive access and shared access to alternate as long as there is demand for both types of access.

```
 1 int readers;                     // Negative value => active writer
 2 pthread_mutex_t lock;
 3 pthread_cond_t busy;             // Use one condition variable to
 4                                  // indicate whether data if busy
 5 AcquireExclusive()
 6 {
 7    pthread_mutex_lock(&lock);    // This code suffers from spurious
 8    while (readers != 0)          // wakeups!!!
 9       pthread_cond_wait(&busy, &lock);
10    readers = -1;
11    pthread_mutex_unlock(&lock);
12 }
13
```

```
14 AcquireShared()
15 {
16    pthread_mutex_lock(&lock);
17    while (readers<0)
18       pthread_cond_wait(&busy, &lock);
19    readers++;
20    pthread_mutex_unlock(&lock);
21 }
22
23 ReleaseExclusive()
24 {
25    pthread_mutex_lock(&lock);
26    readers = 0;
27    pthread_cond_broadcast(&busy);
28    pthread_mutex_unlock(&lock);
29 }
30
31 ReleaseShared(
32 {
33    pthread_mutex_lock(&lock);
34    readers--;
35    if (readers==0)
36       pthread_cond_signal(&busy);
37    pthread_mutex_unlock(&lock);
38 }
```

Second, the `ReleaseShared()` routine signals the `wBusy` condition variable outside of the critical section to avoid the problem of ***spurious lock conflicts***, in which a thread is awoken by a signal, executes a few instructions, and then immediately blocks in attempt to acquire the lock. If the `ReleaseShared()` were instead to execute the signal inside of the critical section, as shown below, then any writer that would be awakened would almost immediately block trying to acquire the lock.

```
31 ReleaseShared(
32 {
33    pthread_mutex_lock(&lock);
34    readers--;
35    if (readers==0)
36       pthread_cond_signal(&wBusy);  // Wake up writers inside of
37    pthread_mutex_unlock(&lock);     // the critical section
38 }
```

The decision to move the signal outside of the critical section represents a tradeoff, because it allows a new reader to enter the critical section before the `ReleaseShared()` routine is able to awaken a waiting writer, allowing readers to again starve out writers, albeit with much less probability than would occur with a single condition variable.

Thread Scheduling

[This might be out of place—perhaps it belongs much earlier.]

POSIX Threads supports two scheduling scopes. Threads in system contention scope are called ***bound*** threads because they are bound to a particular processor, and they are scheduled by the operating system. By contrast, threads in process contentions scope are called ***unbound*** threads because they can execute on any of the Pthreads library's set of processors. These unbound threads are scheduled by the Pthreads library. For parallel computing, we typically use bound threads.

[Need a few more details: what is the default scope?
Are scheduling priorities an optional feature of POSIX Threads?
If not, talk here about scheduling attributes and priority inversion.]

Overlapping Synchronization with Computation

As we mentioned in Chapter 4, it is often useful to overlap long-latency operations with independent computation. For example, in Figure 9 Thread 0 reaches the barrier well before Thread 1, so would be profitable for Thread 0 to do some useful work rather than simply sit idle.

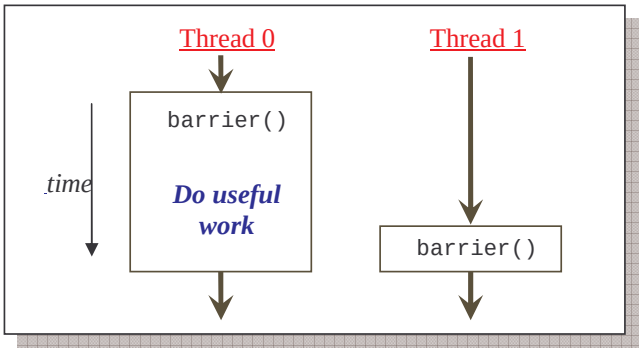


Figure 9. It's often useful to do useful work while waiting for some long-latency operation to complete.

To take advantage of such opportunities, we often need to create ***split-phase operations***, which separate a synchronization operation into two phases: initiation and completion, as shown in Figure 10.

```
// Initiate synchronization
barrier.arrived();

// Do useful work

// Complete synchronization
barrier.wait();
```

Figure 10. Split-phase barrier allows a thread to do useful work while waiting for other threads to arrive at a barrier.

To see a concrete example of how split-phase operations can help, consider a 2D successive relaxation program, which is often used—often in 3D form—to solve systems of differential equations, such as the Navier-Stokes equations for fluid flow. This

Formatted: Bullets and Numbering

computation starts with an array of n+2 values: n interior values and 2 boundary values. At each iteration, it replaces each interior value with the average of its 2 neighbor values,

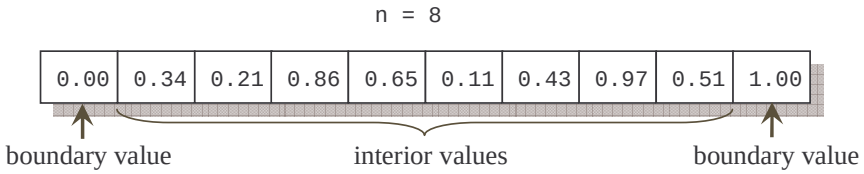


Figure 11. A 2D relaxation replaces, on each iteration, all interior values by the average their two nearest neighbors.

The code for computing a 2D relaxation with a single-phase barrier is shown below. Here, we assume that we have `t` threads, each of which is responsible for computing the relaxation of `n/t` values.

```
 1 double *val, *new;               // Hold n values
 2 int n;                           // Number of interior values
 3 int t;                           // Number of threads
 4 int iterations                   // Number of iterations to perform
 5
 6 thread_main(int index)
 7 {
 8    int n_per_thread = n / t;
 9    int start = index * n_per_thread;
10
11    for (int i=0; i<iterations, i++)
12    {
13       // Update values
14       for (int j=start; j<start+n_per_thread; j++)
15       {
16          new[j] = (val[j-1] + val[j+1]) / 2.0;   // Compute average
17       }

18       swap(new, val);
19       // Synchronize
20       barrier();
21    }
22 }
```

With a split-phase barrier, the main routine is changed as follows:

```
 6 thread_main(int index)
 7 {
 8    int n_per_thread = n / t;
 9    int start = index * n_per_thread;
10
11    for (int i=0; i<iterations, i++)
12    {
13       // Update local boundary values
14       int j = start;
15       val[j] = (val[j-1] + val[j+1]) / 2.0;
```

```
16       j = start+n_pre_thread -1;
17       val[j] = (val[j-1] + val[j+1]) / 2.0;
18
19       // Start barrier
20       barrier.arrived();
21
22       // Update local interior values
23       for (j=start+1; j<start+n_per_thread-1; j++)
24       {
25          new[j] = (val[j-1] + val[j+1]) / 2.0;   // Compute average
26       }
27       swap(new, val);
28
29       // Complete barrier
30       barrier.wait();
31    }
32 }
```

The code to implement the split-phase barrier seems straightforward enough. As shown below, we can implement a Barrier class that keeps a counter of the number of threads that should arrive at the barrier. To initiate the synchronization, each thread calls the `arrived()` routine, which increments the counter. The last thread to arrive at the barrier (line 30) then signals all waiters to wakeup and resume execution; the last thread also sets the counter to 0 in preparation for the next use of the barrier. To complete the synchronization, the `wait()` routine checks to see if the counter is non-zero, in which case it waits for the last thread to arrive. Of course, a lock is used to provide mutual exclusion, and a condition variable is used to provide synchronization.

Deleted: 31

```
 1 class Barrier
 2 {
 3    int nThreads;                 // Number of threads
 4    int count;                    // Number of threads participating
 5    pthread_mutex_t lock;

 6    pthread_cond_t all_here;
 7 public:
 8    Barrier(int t);
 9    ~Barrier(void);
10    void arrived(void);          // Initiate a barrier
11    int done(void);              // Check for completion
12    void wait(void);             // Wait for completion
13 }
14
15 int Barrier::done(void)
16 {
17    int rval;
18    pthread_mutex_lock(&lock);
19
20    rval = !count;               // Done if the count is zero
21
22    pthread_mutex_unlock(&unlock);
23    return rval;
24 }
25
```

```
26 void Barrier::arrived(void)
27 {
28    pthread_mutex_lock(&lock);
29    count++                      // Another thread has arrived
30
31    // If last thread, then wake up any waiters
32    if (count==nThreads)
33    {
34       count = 0;
35       pthread_cond_broadcast (&all_here);
36    }
37
38    pthread_mutex_unlock(&lock);
39 }
40
41 void Barrier::wait(void)
42 {
43    pthread_mutex_lock(&lock);
44
45    // If not done, then wait
46    if (count != 0)
47    {
48       pthread_cond_wait(&all_here, &lock);
49    }
50
51    pthread_mutex_lock(&lock);
52 }
```

Unfortunately, the code presented above does not work correctly! In particular, consider an execution with two threads and two iterations, as shown in Figure 12. Initially, the counter is 0, and Thread 0's arrival increments the value to 1. Thread 1's arrival increments the counter to 2, and because Thread 1 is the last thread to arrive at the barrier, it resets the counter to 0 and wakes up any waiting threads, of which there are none. The problem arises when Thread 1 gets ahead of Thread 0 and executes its next iteration—and hence its next calls to `arrive()` and `wait()`—before Thread 0 invokes `wait()` for its first iteration. In this case, Thread 1 will increment the counter to 1, and when Thread 0 arrives at the wait, it will wait block. At this point, Thread 0 is blocked waiting for the completion of the barrier in the first iteration, while Thread 1 is blocked waiting for the completion of the second iteration, resulting in deadlock. Of course, the first barrier has completed, but Thread 0 is unaware of this important fact.

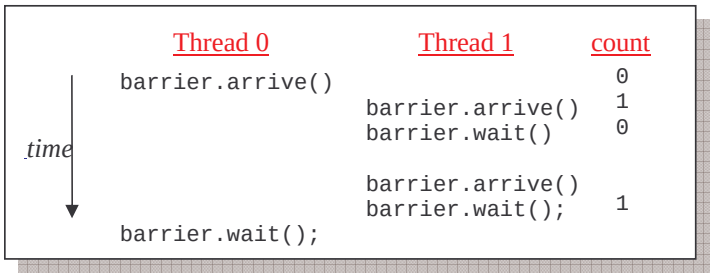


Figure 12. Deadlock with our initial implementation of a split-phase barrier.

Formatted: Bullets and Numbering

Of course, we seem to have become quite unlucky to have Thread 0 execute so slowly relative to Thread 1, but because our barrier needs to work in all cases, we need to handle this race condition.

The problem in Figure 12 occurs because Thread 0 was looking at the state of the counter for the wrong invocation of the barrier. A solution then is to keep track of the current phase of the barrier. In particular, the `arrived()` method returns a phase number, which is then passed to the `done()` and `wait()` methods. The correct code is shown below.

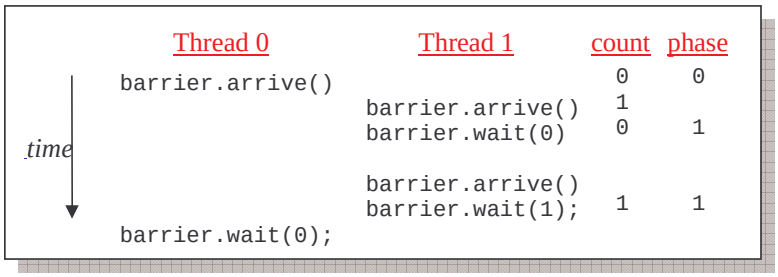
```
 1 class Barrier
 2 {
 3    int nThreads;                 // Number of threads
 4    int count;                    // Number of threads participating
 5    int phase;                    // Phase # of this barrier
 6    pthread_mutex_t lock;
 7    pthread_cond_t all_here;
 8 public:
 9    Barrier(int t);
10    ~Barrier(void);
11    void arrived(void);          // Initiate a barrier
12    int done(int p);             // Check for completion of phase p
13    void wait(int p);            // Wait for completion of phase p
13 }
14
15 int Barrier::done(int p)
16 {
17    int rval;
18    pthread_mutex_lock(&lock);
19
20    rval = (phase != p)          // Done if the phase # has changed
21
22    pthread_mutex_unlock(&unlock);
23    return rval;
24 }
25
26 void Barrier::arrived(void)
26 {
27    int p;
28    pthread_mutex_lock(&lock);
29
30    p = phase;                   // Get phase number
31    count++                      // Another thread has arrived
32
33    // If last thread, then wake up any waiters, go to next phase
34    if (count==nThreads)
35    {
36       count = 0;
37       pthread_cond_broadcast (&all_here);
38       phase = 1 – phase;
39    }
40
41    pthread_mutex_unlock(&lock);
42    return p;
```

```
43 }
44
45 void Barrier::wait(int p)
46 {
47    pthread_mutex_lock(&lock);
48
49    // If not done, then wait
50    while (p == phase)
51    {
52       pthread_cond_wait(&all_here, &lock);
53    }
54
55    pthread_mutex_lock(&lock);
56 }
```

Since the interface to the barrier routines have changed, we need to modify our relaxation code as shown below.

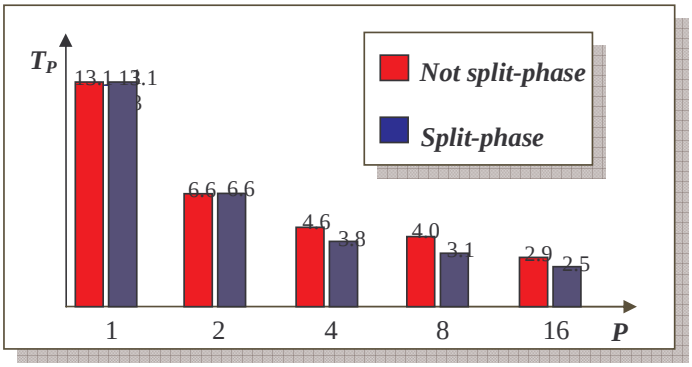
```
 6 thread_main(int index)
 7 {
 8    int n_per_thread = n / t;
 9    int start = index * n_per_thread;
10    int phase;
11
12    for (int i=0; i<iterations, i++)
13    {
14       // Update local boundary values
15       int j = start;
16       val[j] = (val[j-1] + val[j+1]) / 2.0;
17       j = start+n_pre_thread -1;
18       val[j] = (val[j-1] + val[j+1]) / 2.0;
19
20       // Start barrier
21       phase = barrier.arrived();
22
23       // Update local interior values
24       for (j=start+1; j<start+n_per_thread-1; j++)
25       {
26          new[j] = (val[j-1] + val[j+1]) / 2.0;   // Compute average
27       }
28       swap(new, val);
29
30       // Complete barrier
31       barrier.wait(phase);
32    }
33 }
```

With this new barrier implementation, the situation in Figure 12 no longer results in deadlock. As depicted in Figure 13, Thread 0's invocation of `wait(0)` explicitly waits for the completion of the first invocation of the barrier, so when it executes line 50 in the `wait()` routine, it falls out of the while loop and never calls `pthread_cond_wait()`. Thus, deadlock is avoided.



Formatted: Bullets and Numbering

Figure 13. Deadlock does not occur with our new split-phase barrier.



Formatted: Bullets and Numbering

Figure 14. Performance benefit of split-phase barrier on a Sun E4000. n=10,000,000, 10 iterations.

Java Threads

[Discussion of Java threads and a larger discussion of hiding concurrency inside of libraries.

- Nice model: explicit and convenient support for some common cases, but provides the freedom to use lower-level locks and condition variables where necessary. Can also hide concurrency inside of specific classes.
- Synchronized methods and synchronized classes
- Wait() and Notify()

Can we come up with examples where modular decisions about locking and synchronization are sub-optimal? In particular, we need examples where the context in which the data structure is used affects the synchronization policies.]

Critique

[What's good about threads. What's bad about threads.]

Shared virtual memory. Why can't threads-based programs execute on machines that do not support shared memory? Why can't we use software to provide a virtually shared address space on top of such machines? This question was heavily studied in the 1980's and 1990's. The basic issue is that the Shared Virtual Memory system needs to handle all data movement, and it is difficult to do this efficiently without knowledge of the application's sharing behavior. In particular, there is a tradeoff regarding the granularity of sharing: Large units of sharing can amortize inter-processor communication costs, at the expense of false sharing. Small units of sharing reduce false sharing but increase the overhead of moving data. In general, we'd ideally like the shared virtual memory system's granularity of sharing to match the application's logical granularity of sharing. Of course, even if the underlying shared virtual memory system were extremely efficient, there is still the question of whether threads-based programming is the right programming model.

Exercises

1. Our bounded buffer example uses a single mutex to protect both the `nonempty` and `nonfull` condition variables. Could we instead use one mutex for each condition variable? What are the tradeoffs?

   A: Yes, but this would not be a good tradeoff because both the producer and consumer access both condition variables, so both routines would have to acquire both locks instead of just one lock. Thus, there is added locking overhead but no greater concurrency.

2. The pthread_cond_wait() routine takes the address of the protecting mutex as a parameter so that the routine can atomically block the waiting thread and release the lock that is held by the waiting thread. Explain why these two operations must be performed atomically.

   A: If the two operations are not atomic, there are two cases: either (1) the thread is blocked first or (2) the lock is released first. In case (1), we have deadlock. In case (2), the code that blocks the waiting thread must first acquire the lock so that it knows that it is the only thread that is manipulating the queues associated with the condition variable, so the solution is possible but increases the latency of the operation. [Perhaps need to think about this answer some more.]

3.

Bibliographic Notes

The four necessary conditions for deadlock were first identified by Coffman, et al.

Hoare and Brinch Hansen proposed slightly different variations of monitors in the mid-seventies.

E.G. Coffman, M.J. Elphick, and A. Shoshani, "System Deadlocks," *Computing Surveys*, volume 3, pp. 67-78, June 1971.

C.A.R. Hoare, "Monitors, An Operating System Structuring Concept," *Communications of the ACM*, volume17, pp. 549-557, Oct 1974; Erratum in *Communications of the ACM*, volume 18, p. 95, Feb 1975.

P. Brinch Hansen, "The Programming Language Concurrent Pascal," *IEEE Transactions on Software Engineering*, volume SE-1, pp. 199-207, June 1975.