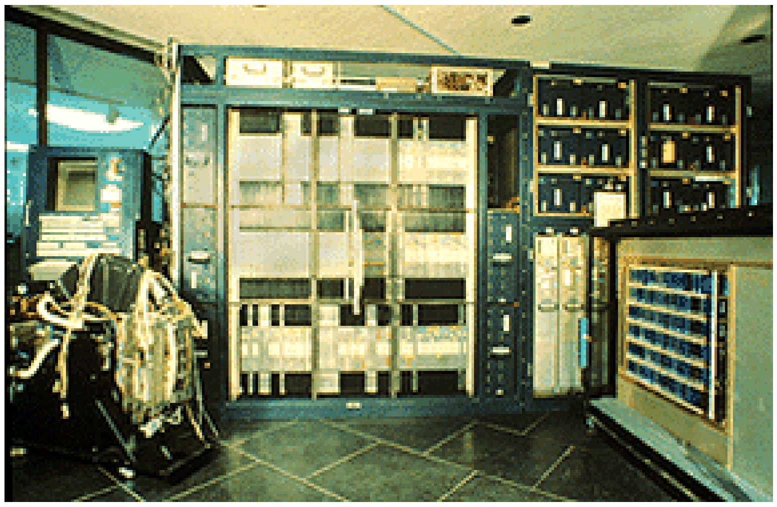
Foundations of Parallel Programming

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.

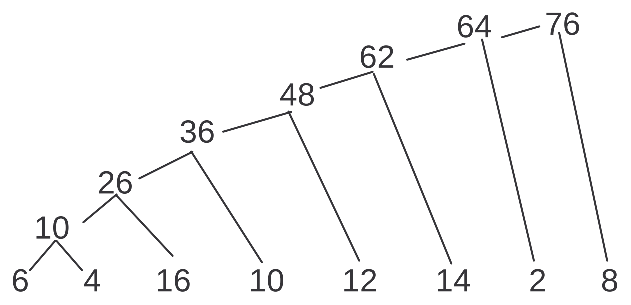


Figure 1.1. Summing in sequence. The order of combining a sequence of numbers (6, 4, 16, 10, 12, 14, 2, 8) when adding them to an accumulation variable.

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.

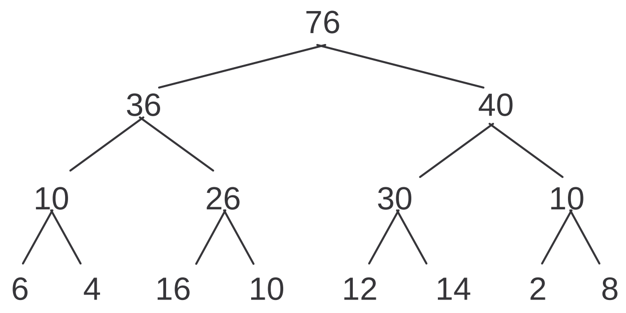


Figure 1.2. Summing in pairs. The order of combining a sequence of numbers (6, 4, 16, 10, 12, 14, 2, 8) by (recursively) combining pairs of values, then pairs of results, etc.

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,

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

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.)

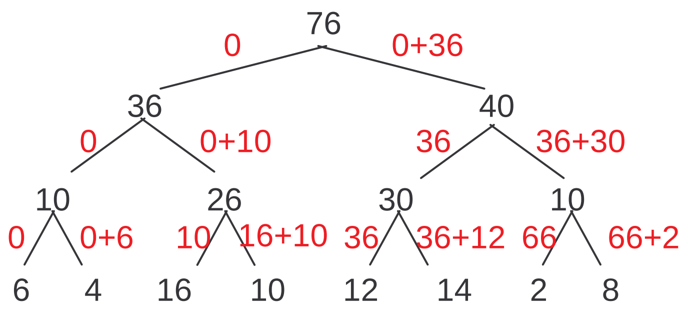


Figure 1.3. Computing the prefix sum. The black values, computed going up the tree, are from the summing by pairs algorithm; the red values, the prefixes, are computed going down the tree by a simple rule: send the value from the parent to the left; add the node's intermediate sum to the value from the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

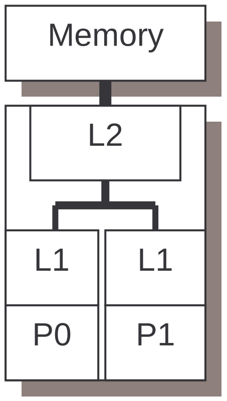


Figure 1.4. Organization of a multi-core chip. Two processors, P0 and P1, have a private L1 cache and share an L2 cache.

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:

```
int *array;
int length;
int count;

int count3s ()
{
   int i;
   count = 0;
   for (i=0; i<length; i++)
   {
      if (array[i] == 3)
      {
         count++;
      }
   }
   return count;
}
```

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.

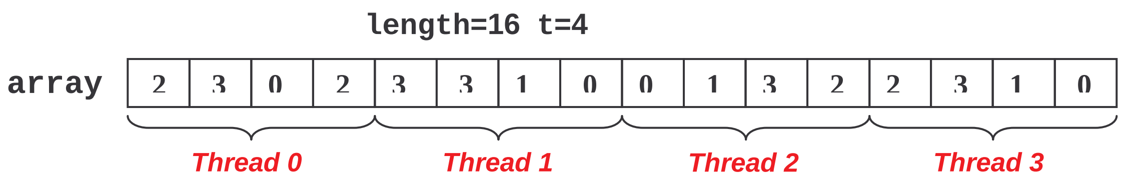


Figure 1.5. Schematic diagram of data allocation to threads. Allocations are consecutive indices.

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 }
```

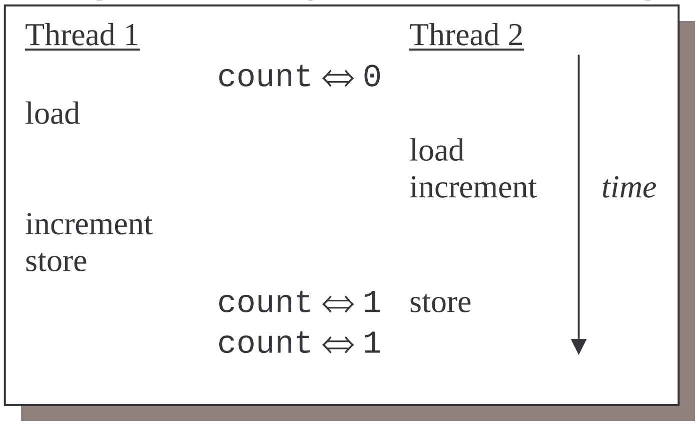
Figure 1.6. The first try at a Count 3s solution using threads.

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.

```
mutex m;

void count3s_thread (int id)
{
  /* Compute portion of the array that this thread should work on */
   int length_per_thread = length/t;
   int start = id * length_per_thread;

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

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.

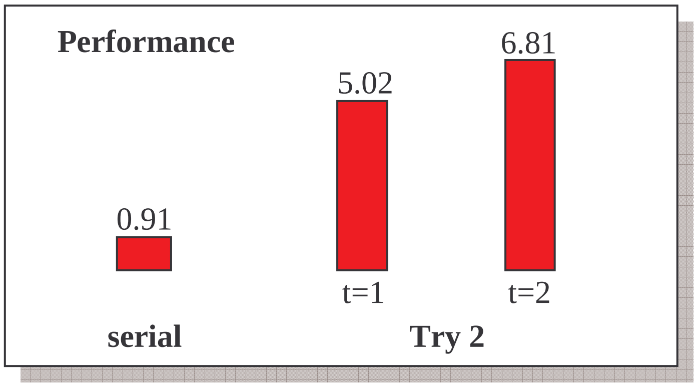


Figure 1.9. Performance of the second Count 3s solution.

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.

```
private_count[MaxThreads];
mutex m;

void count3s_thread (int id)
{
  /* Compute portion of the array that this thread should work on */
   int length_per_thread = length/t;
   int start = id * length_per_thread;

   for (i=start; i<start+length_per_thread; i++)
   {
      if (array[i] == 3)
      {
         private_count[t]++;
      }
   }
   mutex_lock(m);
   count += private_count[t];
   mutex_unlock(m);
}
```

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.

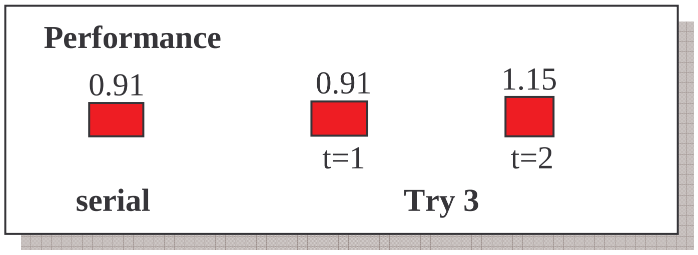


Figure 1.11. Performance results for the third Count 3s solution.

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.

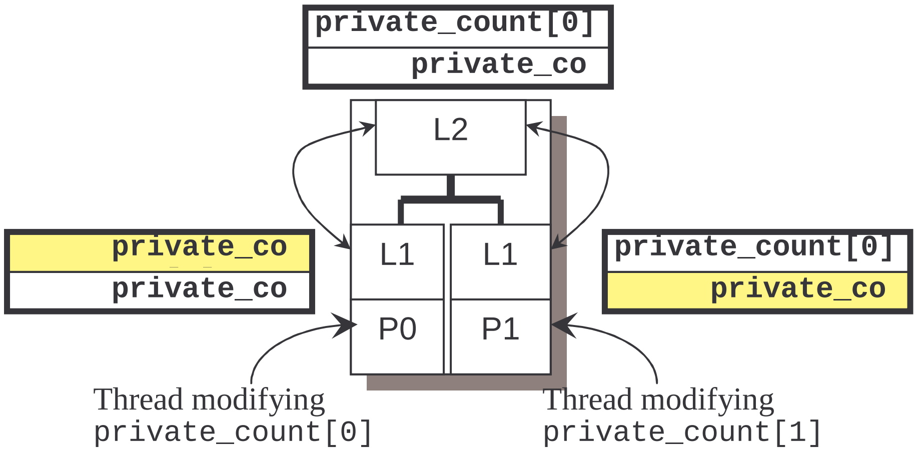


Figure 1.12. False Sharing. A cache line ping-pongs between the L1 caches and the L2 cache, because although the references to private_count don't collide, they use the same cache line.

```
struct padded_int
{
   int value;
   char padding[32];
} private_count[MaxThreads];

void count3s_thread (int id)
{
/*Compute portion of the array this thread should work on */
   int length_per_thread = length/t;
   int start = id * length_per_thread;

   for (i=start; i<start+length_per_thread; i++)
   {
      if (array[i] == 3)
      {
         private_count[t]++;
      }
   }
   mutex_lock(m);
   count += private_count[t].value;
   mutex_unlock(m);
}
```

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.

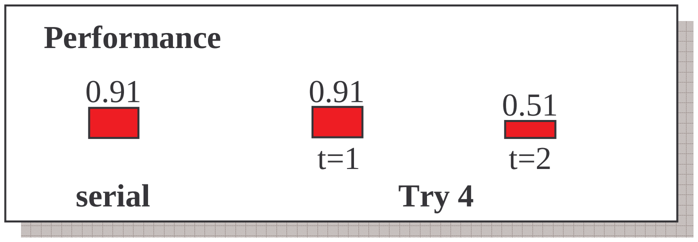


Figure 1.14. Performance for the fourth solution to the Count 3s problem shows that one processor has performance equivalent to the standard sequential solution, and two processors improve the computation time by a factor off 2.

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.

```
location = -1;
for (j = 0; j < n; j++)
{
     if (A[j] == searchee)
     {
       location = j;
       break;
     }
}
```

```
location = -1;
hi = n-1;
lo = 0;
while (lo != hi)
{
  mid=lo+floor((hi–lo+1)/2);
 if (A[mid] == searchee)
     break;
 if (A[mid] < searchee)
     hi = mid;
 else
    lo = mid+1;
}
```

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.

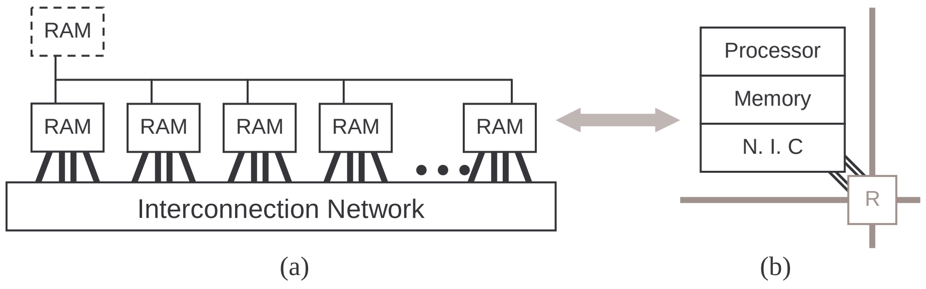
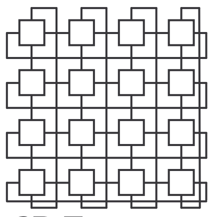


Figure 2.2. The CTA parallel computer model. (a) The schematic shows the CTA as composed of *P* sequential computers connected by a interconnection network; the distinguished (by dashed lines) computer is the controller, and serves such clerical functions as initiating the processing. (b) Detail of a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

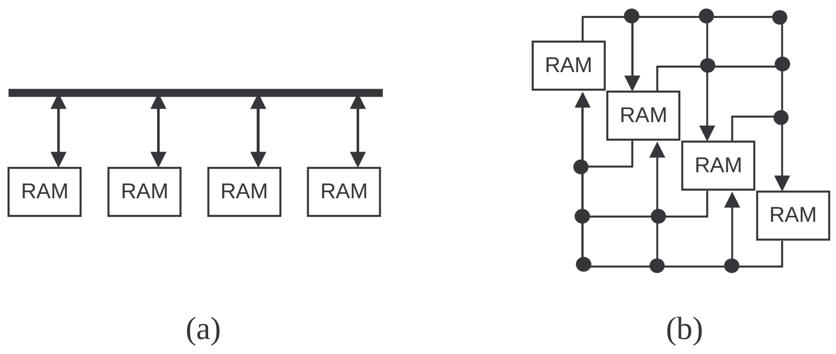


Figure 2.4. Schematics for directly connect parallel computers; (a) bus-based, (b) crossbar-based, where a solid circle can be set to connect one pair of incident wires.

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, …, 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.