Chapter 1:

Introduction: Parallelism = Opportunities + Challenges

The Power and Potential of Parallelism

To begin, consider what parallelism is and how it might be used to advantage.

Parallelism, A Familiar Concept

In a parallel computation multiple instructions are executed simultaneously. This is a familiar concept. Juggling is a parallel task that people can perform; the "instructions" are catching and tossing an object such as a ball. House construction is a parallel activity, because several workers can perform separate tasks simultaneously, such as wiring, plumbing, furnace duct installation, etc. Most manufacturing—cars, hairdryers, frozen dinners—is performed in parallel using a pipeline (assembly line) in which many units of the product are under construction at once. A call center, where many employees are servicing customers at the same time, is also a business that applies parallelism.

Though familiar, these forms of parallelism are different. The call center, for example, differs from home construction in a fundamental way: Calls are generally independent, and can be serviced in any order with little interaction among the workers. In construction, some tasks can be performed simultaneously—wiring and plumbing—while others are ordered—framing must precede wiring. The ordering restricts how much parallelism can be applied at once, limiting the speed at which a construction project can be completed. It also increases the degree of interaction among the workers. Manufacturing pipelines are different still, because they generally have strict ordering constraints with the separate stages often being performed sequentially; the parallelism comes by having many instances of the product in process at once. And juggling is an instance of event-driven parallelism, where the event—a falling ball—causes the execution of instructions—catching, throwing—to respond appropriately to the event. Such familiar properties will also arise in our consideration of parallel computation.

Parallelism in Existing Computer Programs

The main motivation for executing program instructions in parallel is to complete the computation faster. But most programs in existence today are incapable of much improvement through parallelism, because they were written assuming the instructions will be executed in order, one at a time; that is, *sequentially*. The semantics of most programming languages embed sequential execution, and the resulting programs typically rely so heavily on this property for their correctness that it is rare to find significant opportunities for parallel execution. To be sure, there are some opportunities, as when the expression `(a + b) * (c + d)` must be evaluated; assuming these are simple variables, the subexpressions `(a + b)` and `(c + d)` are independent of each other, so they can be computed simultaneously. Such opportunities, known as Instruction Level

Parallelism or ILP, provide small improvements in performance; most modern processors already exploit ILP using hardware mechanisms.

Indeed, one reason that we have continued to write sequential programs is because computer architects have been so successful at applying simple parallel techniques like ILP. (The idea of ILP is simple; its implementation in hardware is not!) They have used the steady improvements in silicon technology to add several kinds of parallelism into sequential processor design. First, architects provide separate wires and caches for instructions and for data. The separation prevents instruction and data memory references from interfering. Second, instruction interpretation uses a pipeline, which fetches and decodes future instructions while the current instruction is being executed, and the results of past instructions are still being written to memory. Further, the processors issue (initiate) more than one instruction at a time, they prefetch instructions and data, they speculatively perform operations in parallel even if they cannot be sure that they will be needed, and they use highly parallel circuits to perform basic arithmetic operations such as addition and multiplication. In short, modern processors are highly parallel systems.

The key point for programmers is that all of this parallelism has been transparently available to sequential programs. Such parallelism, together with increasing clock speeds, has allowed each succeeding generation of processor chip to execute instructions faster, while preserving the illusion of sequential execution. But the prospects for finding new opportunities to apply parallelism while preserving sequential semantics are limited. More seriously, existing parallel techniques have largely reached the point of diminishing returns, in terms of both power consumption and performance, so given current technologies, sequential program execution is nearing its maximum speed.

To continue achieving significant performance improvements, we must move beyond the single sequence of instructions typical of existing programs. We need programs that have multiple separate instruction streams that operate on the computation simultaneously. This approach will require new programming techniques, the topic of this book.

Multi-core Computers, an Opportunity

Though performance improvements for a single processor may be reaching a limit, the self-fulfilling prophesy of Moore’s Law continues to deliver improved transistor densities. Chip manufacturers have used this opportunity to place more than one instruction execution engine, together with its caches, on a single chip. This structure has rapidly acquired the name *core*, because it represents the core components of a typical sequential processor. Early chips had 2, 4 or 8 cores, but this number increases with each generation.

The advent of the first multi-core chips in 2005/2006 prompted a community-wide discussion about the “end of the free lunch.” The key observations of the discussion were

- Software developers have enjoyed steadily improving performance for decades—the “free lunch”—thanks to advances in silicon technology and architecture design, as just described

- Programmers, not needing to be concerned with performance, have changed their techniques and methodologies little over the years; (the object-oriented paradigm is a notable exception)
- Existing software generally cannot exploit multi-core chips directly
- Programmers do not now know how to write parallel programs
- Programs that do not exploit multi-core chips are not now realizing any performance improvements and will not in the future

The uncomfortable conclusion was that programs needed to change, and programmers do, too.

Though the conclusion might be viewed by some as bad news, there was corresponding good news. Specifically, if a computation is rewritten to be parallel and if the parallel program is also *scalable*, meaning that it is capable of using progressively more processors, then as silicon technology advances and more cores are added to future chips, the revised program will stay on the performance curve. Non-scalable parallel programs, though they might exploit a specific number of cores, will not enjoy the continued benefits of silicon technology advances. It is important, therefore, to achieve scalable parallelism.

Some observers, especially in the graphics community, no doubt find the discussion of the need for parallel computation curious, because they have been using parallelism for years. Graphics processing units (GPUs, a.k.a. graphics cards) have been the standard technique for accelerating the rendering pipeline. Though the GPU might seem like a niche co-processor of little interest for general computer applications, advances in silicon technology have enabled the GPGPU, the general-purpose graphics processing unit. With a generation cycle of roughly 18 months, the GPU has steadily morphed into a GPGPU. The generality has increased with each generation, and parallel programmers have applied them to a long list of compute-intensive non-graphics applications. Like multi-core chips, exploiting the potential of GPGPUs requires knowledge of parallel programming.

Even More Opportunities to Use Parallel Hardware

The opportunities to use parallelism discussed so far involve a small number of processors. But there are many opportunities that are more ambitious.

The Fastest Supercomputers. The Top 500 List of Fastest Supercomputers describes the latest speed records, together with a description of architectures and analyses of supercomputer trends. This list can be found at www.top500.org.

Supercomputers

The problems of interest at the national research labs, the military and large corporations have traditionally required supercomputers, which by definition are the world’s fastest computers. Twenty years ago, supercomputers were custom-made single processor systems, but single-processor systems last appeared on the Top 500 List over ten years ago in November of 1996, when there were just three of them, ranking #265, #374, and

#498. Today, the Top 500 List is dominated by parallel computers with many thousands of processors. In many ways, supercomputer programmers form the largest, most experienced community of parallel programmers.

Clusters

It is often observed that no matter how fast a single computer is, connecting two or more of them together produces a faster computer in the sense that the combined machine can execute more instructions per unit time. Of course, well written parallel programs are needed to exploit the added speed. Clusters have been popular since the 1990s, because they are relatively inexpensive to build from commodity parts. The low price not only makes them attractive for small groups—labs or small firms—it gives them a tremendous price/performance advantage over other forms of high-end computing. In the latest Top 500 List of fastest computers (November 2006), clusters represented 72.2% of the list.

Server Farms

The expansion of the Internet and the popularity of remote services, such as searching, have created huge installations of networked computers. In terms of total number of instructions executed per second, these centers represent a huge computational resource. The typical computations—the processing of search queries, for example—are independent of each other; further they use distributed—as opposed to parallel—programming techniques (see the section after next). Nevertheless, these huge networked systems are being used to analyze the features of their workload; the solutions apply parallel programming techniques.

Grid Computing

Generalizing still further, the collection of computers need not be in the same location, nor administered by the same organization; after all, the computers connected by the Internet represent an enormous computing resource. By analogy with the power grid, a computing grid seeks to provide a single convenient computing service, even though the underlying computer typically consists of physically dispersed machines governed by multiple administrative organizations. Many technical issues remain before grids become commonplace, but they are a topic of active research.

We see then, that there are ample opportunities to use a parallel program beyond the few processors on a single silicon chip. These large computer systems motivate us to write scalable parallel programs.

Parallelism vs. Distributed Computing

As suggested above, distributed computing and parallel computing are different.
The goal of parallel computing has traditionally been to provide performance—either in terms of processor power or memory—that a single processor cannot provide; thus the goal is to solve a single problem on multiple processors. The goal of distributed computing is to provide convenience, where convenience includes high availability, reliability, and physical distribution (being able to access the distributed system from many different locations).

In parallel computation the interaction between processors is frequent, is typically fine grained with low overhead, and is assumed to be reliable. In distributed computation the interaction is infrequent, is heavier weight, and is assumed to be unreliable. Parallel computation values short execution time; distributed computation values long up time.

Obviously, parallel and distributed computing are closely related. Some features are a matter of degree—frequency of interaction between processors—and we haven't specified the crossover point. Some features are a matter of emphasis—speed versus reliability—and we know that both properties are important to both types of systems. It follows then, that the two kinds of computing represent distinct, but nearby points in a multidimensional space. The more one knows about parallel computation (or distributed computation, but that's not the emphasis of this book), the more easily one can move around in the parallel-distributed space. Learning the basics of parallel computation will be valuable even for programmers with no need to improve speed.

Finally, there is one reason besides speed to exploit parallelism: Some computations are more easily expressed as parallel computations. For example, user interfaces are typically best written as a collection of threads, with one thread responsible for interacting with the user: it lops waits for user input and dispatches other threads to respond appropriately. With such an organization, the code that displays a widget is greatly simplified because it needn't concern itself with the responsibility of polling for a user mouse click that might come at any point in time.

As we shall see, the abstractions used to organize and manage parallel computations make using multiple instruction streams convenient and safe. When flow of control is unpredictable, parallelism can help even when the resulting instruction sequences are not executed simultaneously.

Thus, while we emphasize fast solution of one problem, we acknowledge that there are other uses of parallelism.

System Level Parallelism

Return for a moment to our earlier argument that to enjoy the benefits from parallelism we must move beyond a single sequence of instructions. This argument is relevant within a single application. When we view a desktop computer's software from a system level, however, we see many tasks executing at once. The operating system orchestrates their concurrent execution, which heretofore meant that several tasks were in process at once, but with only one executing at a time, that is, multitasking.[1] An obvious question to ask is, "Why not simply run these separate tasks on the extra processors?" It makes sense because their concurrent design ensures that whatever interactions are required among them will be handled in a safe way.

[1] Because certain I/O devices like disk controllers are typically separate from the main execution engine, there has long been true parallel execution between the processor and its external devices.

The first answer is that for the large scale parallelism just described, there are not nearly enough tasks to keep large processors busy. But, for small amounts of parallelism as is typical of today's multi-core chips, the separate tasks can be run on separate processors. Indeed, it has been suggested that continuously executing tasks like security software (or the OS itself!) would be good candidates for the extra processors. But there may be fewer opportunities than might at first appear. First, many applications don't stress the hardware now—word processors perform spell checking continuously in the background and never fall behind the typist. Second, much of the multitasking in an operating system comes from switching to a new task when the currently executing task requests a time-consuming, external operation, such as a page fault, disk or network I/O, etc. The task that made the request, call it task A, must wait for that operation to complete no matter what. If the task that would have been run in a single processor system, call it task B, is already completed thanks to the parallel execution of independent tasks, then there may be nothing to do but idle; task B completed sooner, but task A didn't because it was I/O constrained.

The main reason that running multiple tasks on the separate parallel processors is not the silver bullet, however, is because it doesn't usually speed an individual application. And in those cases where improved performance is needed, it is essential that there be multiple streams of instructions that work on the application and use multiple processors effectively.

Examining Sequential and Parallel Programs

The emphasis of previous sections has stressed the potential advantages available in hardware. And it has been asserted that existing sequential programs cannot take advantage of, say, multi-core computers. It is time to consider how to realize the benefits of parallel hardware.

Parallelizing Compilers

Knowing that a compiler translates the programs we write into the machine instructions of the computer we use, and not knowing (at least for most of us) how this magical translation is done, it is reasonable to wonder why someone doesn't just write a compiler that translates existing programs into a form suitable for parallel execution. After all, the sequential program specifies the computation, and all that needs to be done is to transform the same operations into a parallel form. The idea to compile sequential programs for parallel machines was among the first approaches tried, and it continues to be a dream. Unfortunately, the dream seems beyond reach, despite over three decades of intense research.

The reason for pessimism is that scalable parallel algorithms are generally qualitatively different from the sequential algorithms found in existing programs. We will describe this situation by saying that solving a problem in parallel requires *a paradigm shift* in the solution approach. Since compilers transform programs in ways that preserve their correctness, they do not change the essential features of the algorithm. (Figure 1.1 illustrates the phases of a generic compiler.) Compilers change the form of the program code; they can remove unnecessary instructions, as for example, when 0 is added to a variable; they can add helpful instructions, say, to check that array indices are in bounds; they can move instructions around, say hoisting them out of loops when the value computed isn't affected by the iteration; and they can perform other amazing transformations. But, the general operation of the algorithm is preserved. Whether it was sequential or parallel in the source form, it will be the same in the object form.

Thus, although automatic parallelization by compiler would be wonderful, we must consider other approaches[2]. First, consider how algorithms for the same task might differ.

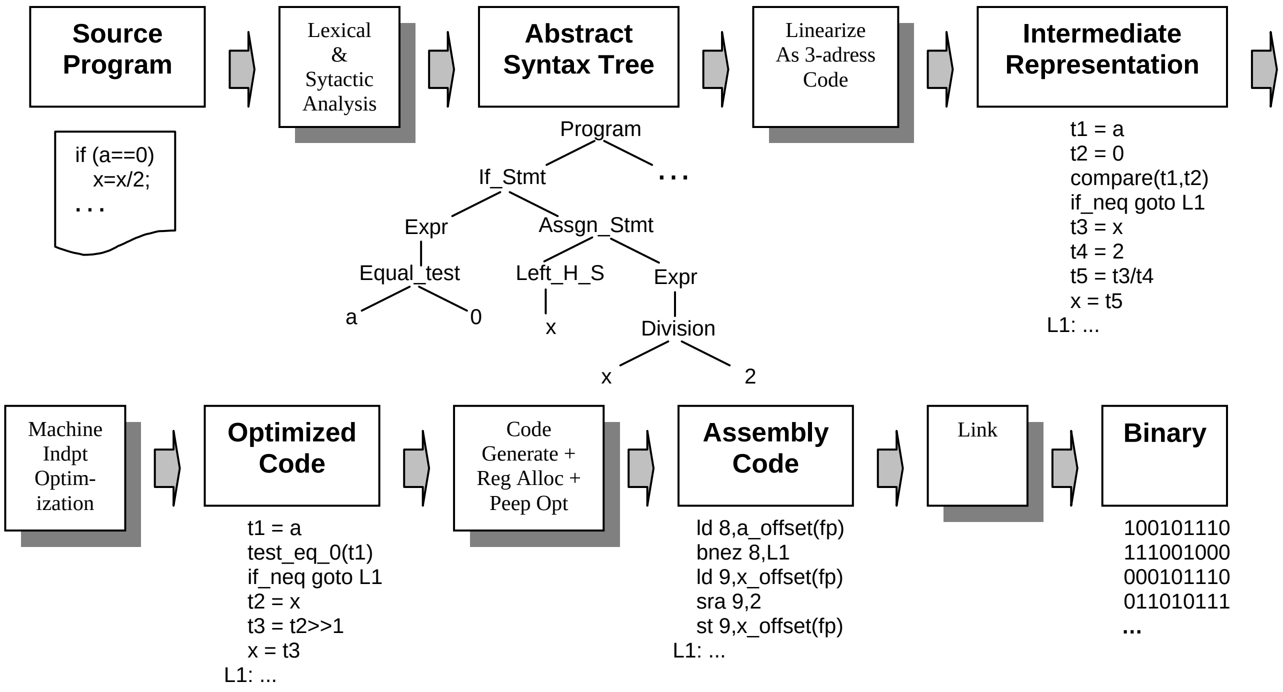

Figure 1.1. Generic compilation process. In the first phases, the familiar source program is scanned (lexical analysis) and parsed (syntactic analysis), resulting in a program representation known as an Abstract Syntax Tree. In this form the program is type-checked to insure, for example, that variables are declared. Next, the program is transformed into a linear sequence of simple instructions known as 3-address code. The resulting intermediate representation is improved (grandly called, optimization). The resulting code is transformed into machine specific assembly code. It is then a trivial matter to transform the result into binary and assign virtual addresses.

A Paradigm Shift

To make clear how we see sequential and parallel algorithms to be different, compare alternative algorithms for finding the sum of a sequence of numbers. This example is sufficiently simple that there *are* compiler techniques to identify it and generate a more parallel solution, but we choose it because it illustrates the conceptual difference between a sequential solution and a parallel solution.

[2] For those who wish to pursue the dream of automatic parallelization, this book should be helpful in pointing out the challenges that must be faced.

Summation

To begin we assume that the sequence has *n* data values,

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

and that these have been stored in an array, `x`.

Iterative Sum

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 1.2. Such solutions might be considered the natural way to think of algorithms.

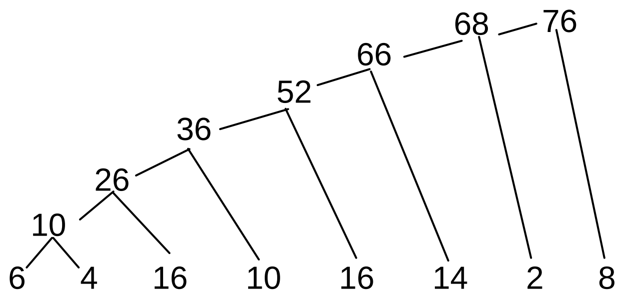

Figure 1.2. Summing in sequence. The order of combining a sequence of numbers (6, 4, 16, 10, 16, 14, 2, 8) when adding them to an accumulation variable.

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 the fixed precision representation of floating point number, because it only approximates the reals. For some sequences of values, adding the numbers in different orders will produce different answers. 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.

Pair-wise Summation

Another, more parallel order of summation is to add even/odd pairs of data values yielding the 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, which are also added in pairs, 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 1.3.

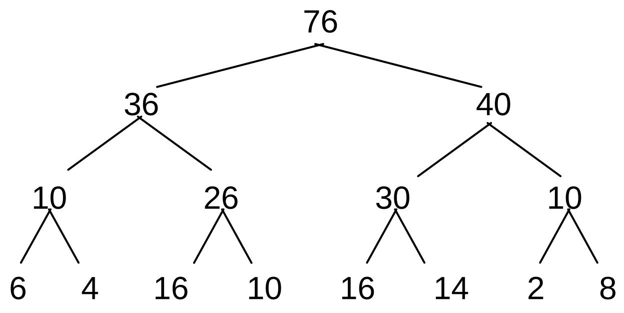

Figure 1.3. Summing in pairs. The order of combining a sequence of numbers (6, 4, 16, 10, 16, 14, 2, 8) by (recursively) combining pairs of values, then pairs of results, etc.

Comparing Figures 1.2 and 1.3, 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 at least $P=n/2$ 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. The strategy significantly improves the linear-in-n sequential time. Like the sequential solution the pair-wise approach is a very intuitive way to think about the computation.

Expressing the Summation

The iterative summation was illustrated using C code, but not the pair-wise summation. We might have written it as

```
1  for (s=0; s<log2(n); s++)
2  {
3      for (i=0; i<n; i=i+2^(s+1))
4      {
5          x[i] = x[i] + x[i+2^s];
6      }
7  )
8  sum = x[0];
```

which can be verified to add the eight numbers of our example with strides (`s`) of 1, 2 and 4. The solution has the unfortunate property of trashing the `x` array, and it presumes *n* is a power of two, but there is a more serious problem from the perspective of parallel computation. The parallel portion of the summation—the inner loop (lines 3-6)—is expressed iteratively, that is, *as a sequence of operations*. We want them to be performed simultaneously.

Of course, C is a sequential programming language with no easy way to express parallelism. Often, the construction `forall` is added to a language to indicate that a set of operations can be performed in any order, including in parallel. We express the indices for the index variable `i` as a triple, `l:u:b`, where `l` is the lower limit, `u` is the upper limit and `b` is the stride. Thus, we can write

```
1  for (s=0; s<log2(n); s++)
2  {
3      forall i in 0:n-1:2^(s+1) )
4      {
5          x[i] = x[i] + x[i+2^s];
6      }
7  )
8  sum = x[0];
```

By using `forall`, we relax the semantics of order implied by the conventional iteration. The outer loop should remain a `for`, because we need the successive levels of the tree to be executed in order.

Given the availability of this new programming statement, we might have simply rewritten the original iteration using `forall` rather than `for`. But there are benefits in having discovered the tree solution.

Prefix Summation

Closely related to the sum is the prefix sum, also known as *scan* in many parallel 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 to, 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 (see Figure 1.3), and if we save that information, we can figure out the prefixes. We start at the root, whose prefix—that is, the sum of all elements before the elements of the 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 at the root by pair-wise sum, 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 computed on the way up; 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 1.4, where downward moving prefix values are shown in the white square.)

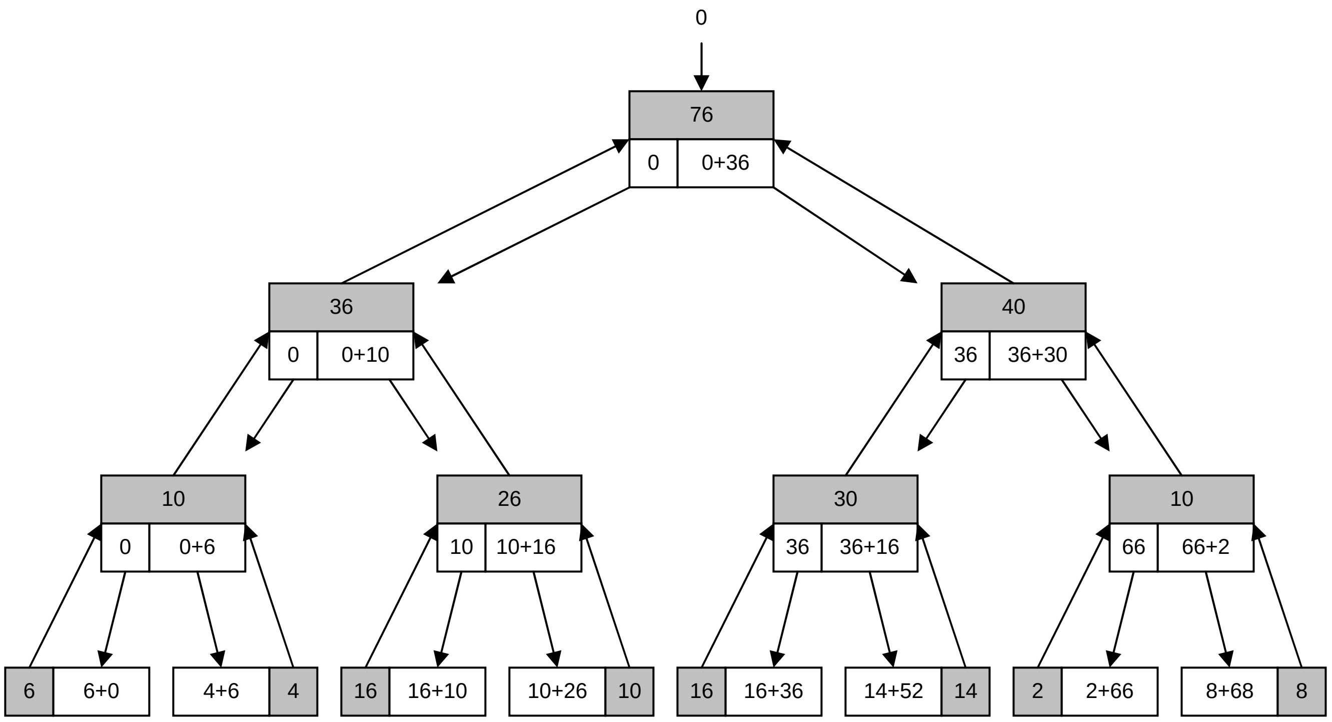

Figure 1.4. Computing the prefix sum. The gray node values, computed going up the tree, are from the pair-wise sum algorithm; the white values, the prefixes, are computed going down the tree by a simple rule: send the value from the parent to the left child; add the sum from the left child to the value from the parent and send it to the right child.

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 in a sweep 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.

An essential difference illustrated between the sequential and parallel algorithms is that we organized the parallel algorithms to remove order in the computation.

Parallelism Using Multiple Instruction Streams

In this section, we illustrate the complexities of parallel programming by developing a parallel program that solves a trivial problem. It will take us four tries to get a satisfactory result.

We begin by describing one way to conceptualize an instruction stream.

The Concept of a Thread

A *thread*, or thread of execution, is a unit of parallelism. As we will discuss in Chapter 3, a thread has everything that it needs to execute a stream of instructions-- a private program text, a call stack, and a program counter-- but it shares access to memory with other threads. Thus, multiple threads can cooperate to compute on some global data.

For example, the iterative summation loop discussed above could be the basis for a thread if we rewrite it as follows

```
for (i=start; i<end; i++)          Caution: Incomplete Solution
{
    sum += x[i];
}
```

The loop index `i` would be local and the array `x` would be shared. (Whether the other names are local or global depends on how the thread is completed.) This allows multiple streams of instructions to work on a problem at once, introducing one form of parallelism.

A Multithreaded Solution to Counting 3's

To understand the obstacles to writing correct, efficient and scalable threaded 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; what is required to solve it using threads?

The Parallel Computer

To simplify matters, let's assume that we will execute our parallel program on a multi-core chip with two processors, see Figure 1.5. The processors are labeled P0 and P1. They are shown adjacent to their (private) Level 1 caches, labeled L1. A cache is fast (compared to the RAM) memory for storing instructions and data while a program runs. The Level 2 cache is memory intermediate in speed between the fast L1and slower RAM. It is also intermediate in size between the smaller L1 and the larger RAM. Information shared by both processors is exchanged in the L2 cache.

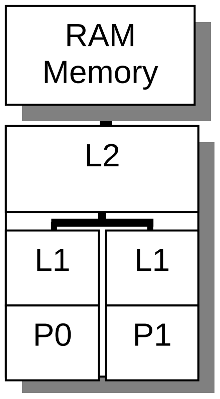

Figure 1.5. Organization of a multi-core chip. Two processors, P0 and P1, have a private L1 cache, and share an L2 cache.

First Solution

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 1.6 shows graphically how we might divide the work for *t*=4 threads and *length*=16.

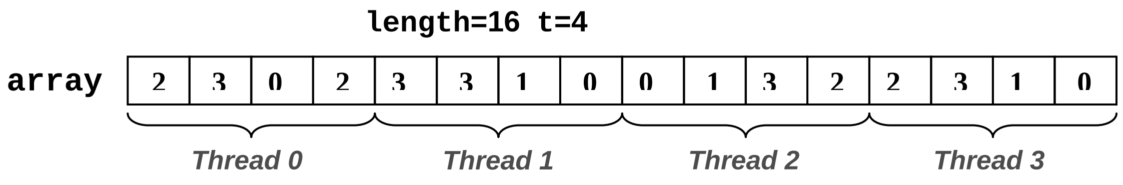

Figure 1.6. Schematic diagram of data allocation to threads. Allocations are consecutive indices.

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 1.7.

```
int t;         /* number of threads */
int *array;
int length;
int count;

void count3s ()
{
 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 1.7. 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* or *race condition* in the statement that increments the value of `count` on line 29. A race 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 1.8. 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.

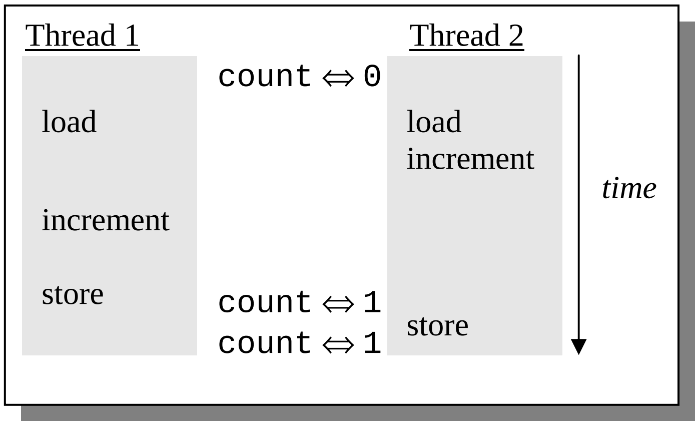

Figure 1.8. One of several possible interleaving in time of references to the unprotected variable `count` illustrating a race.

Second Solution: Try 2

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 1.9.

```
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 1.9. 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 1.10, our parallel program is much slower than our original sequential 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.

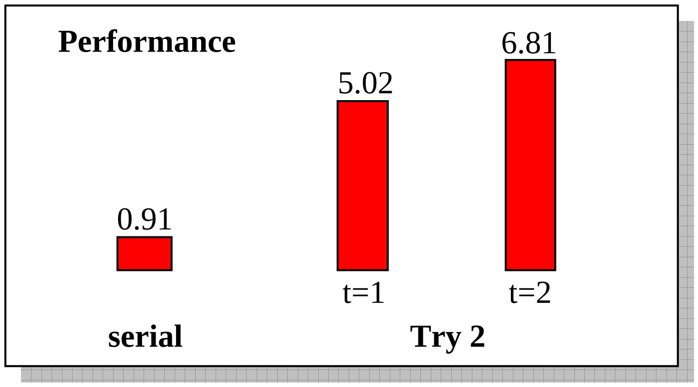

Figure 1.10. Performance of the second Count 3s solution.

Third Solution: Try 3

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 or unit 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 1.11.

```
private_count[MaxThreads];
mutex m;

void count3s_thread (int id)
{
  /* Compute portion of array for this thread to 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 1.11. 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 1.12.

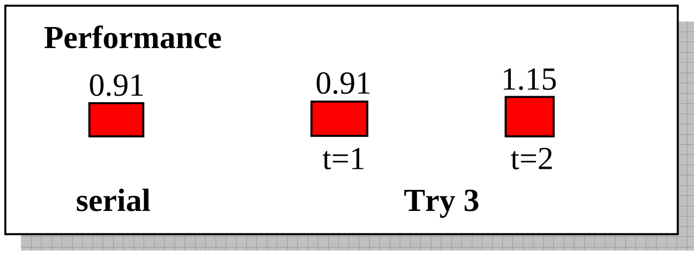

Figure 1.12. Performance results for the third Count 3s solution.

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 accidentally 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.

Fourth Solution: Try 4

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 1.13.) 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 1.14.

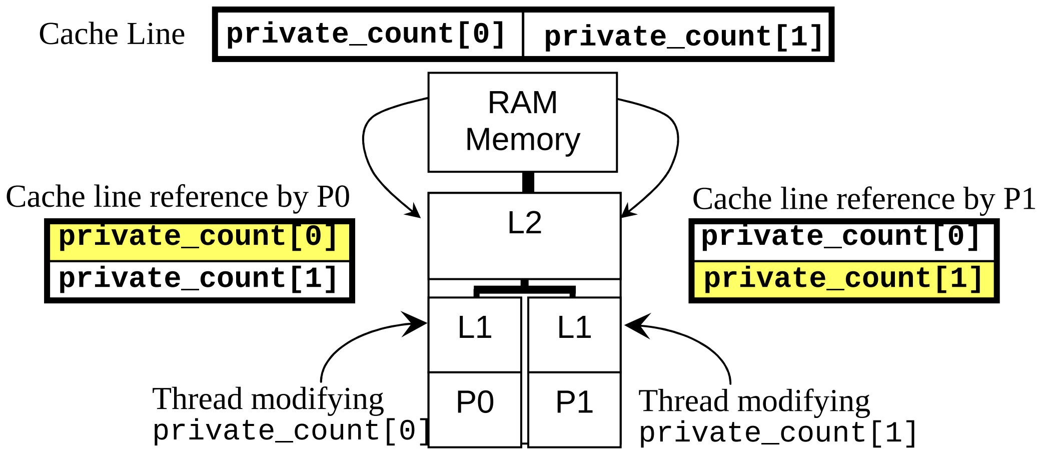

Figure 1.13. False Sharing. A cache line moves from RAM to the L2 cache, and then to the L1 cache when a thread references its `private_count`. When the other thread references its `private_count`, the copy in the other L1 is invaldiated, writen back to the L2 cache, and then fetched into the other L1 cache. The 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 1.14. 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 1.15.

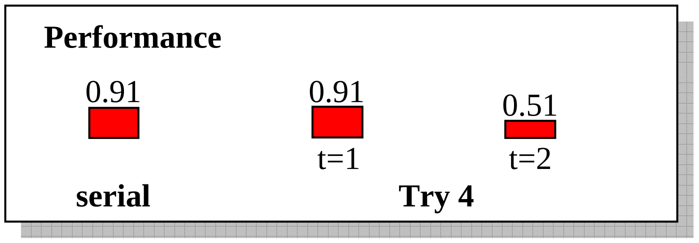

Figure 1.15. Performance for the fourth solution to the Count 3s problem shows that one processor has performance equivalent to the standard sequential solution, and two processors improve the computation time by a factor off 2.

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 Goals: Scalable Performance and Portability

The Count 3s program illustrates both that performance can be achieved through parallelism and that achieving it can be complicated. Having mastered some of the issues facing CMP's—race conditions, issues of granularity, and false sharing—it's tempting to think that parallel programming is concerned only with issues of correctness and performance. In fact, the goals of this book are broader. Our goal is to help you write good parallel programs, by which we mean parallel programs with four characteristics:

- They are correct
- They achieve good performance
- They are scalable to large numbers of processors
- They are portable across a wide variety of parallel platforms.

The first goal does not require explanation, other than to notice that correctness can often be more difficult to achieve in a parallel program than in a sequential program. The second goal seems pretty clear, but as we will see in Chapter 3, defining what we mean by "good performance" is filled with subtleties.

The third and fourth goals, however, require some elaboration because they appear to be overly lofty and often unnecessary. For example, someone who programs for a CMP with a few cores has little interest in a parallel supercomputer with many thousands of processors. Indeed, there will always be some markets where the extreme desire for performance will dictate low-level non-portable solutions. But for the vast majority of programmers, scalability and portability are important because the landscape of parallel hardware is changing rapidly. For example, the first CMP's had only two cores per chip, but Intel has already announced a new chip with 80 cores. Of course, as the number of cores increases, other micro-architectural features, such as the memory system, will have to change as well. Given this highly fluid hardware landscape, it's best to not be caught scrambling when new hardware arrives. The solution is to design for scalability and portability from the beginning, so that programs will enjoy a long lifetime, justifying the significant intellectual and economic investment in their creation.

Let’s now briefly consider scalability and portability in more detail.

Scalability

To understand the issue of scalability, consider the consequences of programming decisions when the number of processors increases. For example, the Count 3s program was parameterized so that the number of threads could vary. This flexibility allows us to run the program on a four-core chip with little modification. It would seem that we have produced a general solution that could scale to thousands simply be changing `maxThreads`. But we have not. It’s true that the scan of the array, having been broken into segments, is independent, and therefore, parallel for any number of threads. But the combining of the intermediate results is not, because all threads update the one global sum. For a large number of threads, we would again encounter lock contention. Obviously, our pair-wise sum solution fixes the problem. Scalability requires scalable programming practices.

More generally, as the number of parallel processors increases, physical constraints force design changes that impact how programs perform. For example, communication latency—the delay encountered when transmitting information among processors—necessarily increases as the number of processors grows simply because of speed of light limitations. On a single chip different issues apply, but they still affect communication latency when the number of cores grows large. For a small number of processors, proximity can make certain operations fast that do not remain fast as the size of the system grows. Exploiting these benefits makes sense when possible, but the program must avoid relying on them for its success. Well written parallel programs can exploit the fast components and avoid over-using the slow components of a parallel computer.

Performance Portability

The problem just discussed—that physical constraints impact the characteristics of parallel computers as the number of processors increases—is not limited to slowing down certain operations. The problem is much more insidious.

Architects, grappling with those physical constraints, have created scores of parallel computer designs. These machines can differ from each other dramatically. Unlike the sequential case, where a new computer usually requires only a recompilation of the source code to execute respectably well, a program running well on one parallel machine may have to be rewritten for the next one.

To give one example, parallel computers can mostly be divided into one of three classes: shared memory, typified by multi-core processors, shared address space, typified by various supercomputers, and separately addressed memories (shared nothing), typified by clusters. This distinction affects every memory reference in a program, so it has a tremendous impact on how the program should be written. Programs intended to port to all of these platforms must be robust to these differences in memory structure, which is not easy to do.

The classification by memory capability specifies the variety along one axis. There are many other differences among parallel processors. We could solve the portability problem by simply setting a high enough level of abstraction (high level programming language) that none of these differences are visible; then, a compiler will map the high level specification to the platform. The strategy will make our programs insensitive to the parallel hardware, but it's not a good idea. Generally, though compilers can perform the mapping, they will usually introduce software layers to implement the abstractions; the added software will reduce performance. We cannot divorce ourselves entirely from the underlying hardware if we want high performance. So, we will use a different strategy, described in the next chapter.

Our goal, then, is portability with performance, often called *performance portability*. It's not enough for the program to run on different (parallel) machines. It must run well on all of them.

Principles First

This book does not provide a step-by-step tutorial for writing good parallel programs. Instead, it emphasizes the principles underlying parallel computation, explaining the various phenomena and explaining why they represent opportunities or barriers to successful parallel programming. Our reason for this approach is twofold. First, by focusing on principles, we hope to provide enduring knowledge that will outlive the specifics of the latest hardware or software technology, which as we've pointed out are changing rapidly. Second, and more importantly, the parallel programming community does not yet have all of the answers, so a step-by-step solution is not available. Indeed, one of our goals is to inspire the next generation of researchers to understand the limitations of current technology so that they can build the better solutions of tomorrow.

After presenting these principles, we discuss some popular programming languages and tools used for programming contemporary parallel machines. Again, our goal is more concerned with the principles behind the approach, than it is making the reader into an expert in the language. Our treatments, therefore, are minimal, and readers should expect to consult reference manuals for more complete and detailed information.

Summary

The chapter began with the observation that parallelism—doing two or more things at once to achieve a single goal—is a familiar idea that we encounter in everyday life. Though familiar, parallelism has not been a significant aspect of programming in the past because sequential computer performance has increased steadily for decades. Such improvements have been due to a combination of technology (Moore's Law) and the incorporation of parallelism into sequential processor design by computer architects. Because the architectural opportunities have largely been mined, the continued advancement of technology has made computers with multiple processors standard. This shift is having a profound effect on computer programming.

We noted that existing sequential programs generally cannot take advantage of a parallel computer. The primary reason is that existing programming languages and standard programming techniques strongly incorporate the one-after-another processing of the traditional von Neumann computer architecture. Parallel solutions, as illustrated by several simple computations—summation, parallel prefix and Count 3s—illustrated features of parallel computations. Though they might not have been the first solutions to come to mind, they were still quite intuitive. A change in thinking about computation will be required—we called it a shift in paradigm—before programmers instinctively devise parallel solutions to their computational problems.

In a quick and incomplete survey of parallel hardware, we noted platforms as diverse as chips with two processors to server farms with thousands of processors. Though dramatically different in scale and design, their parallel features rely on a small set of fundamental principles. We committed to focusing on those principles with the goal of empowering programmers to strive for parallel programs that achieve high performance, scalability, and performance-portability.

Historical Context

Parallelism has been applied in the design of sequential computers since the first commercial machines in the 1950s. A landmark parallel machine was the Illiac IV, built in the 1970s by a team at the University of Illinois, Urbana-Champaign. Though the Illiac IV was successfully programmed in low-level assembly-like code, the task of developing a compiler to translate sequential (Fortran) programs into a parallel form was begun by David Kuck and colleagues. Investigators throughout the community pursued the goal to the end of the century.

Exercises

1. Explain the meaning of the following vocabulary related to thread programming:
    a. Thread
    b. Race or Race Condition
    c. Mutex
    d. Lock Contention
    e. Granularity
    f. False Sharing
2. Describe how the pair-wise summation computation can be changed to find the maximum of the elements of `x` array.
3. Reformulate the pair-wise summation program to solve the Count 3s computation in log n time, assuming P=n/2.
4. Reformulate the pair-wise summation program to solve the Count 3s computation assuming that n=1024, but P=8.

5. Rewrite the iterative summation program using forall; remember about races.