CHAPTER

1 Why Parallel Computing?

From 1986 to 2002 the performance of microprocessors increased, on average, 50% per year [27]. This unprecedented increase meant that users and software developers could often simply wait for the next generation of microprocessors in order to obtain increased performance from an application program. Since 2002, however, single-processor performance improvement has slowed to about 20% per year. This difference is dramatic: at 50% per year, performance will increase by almost a factor of 60 in 10 years, while at 20%, it will only increase by about a factor of 6.

Furthermore, this difference in performance increase has been associated with a dramatic change in processor design. By 2005, most of the major manufacturers of microprocessors had decided that the road to rapidly increasing performance lay in the direction of parallelism. Rather than trying to continue to develop ever-faster monolithic processors, manufacturers started putting *multiple* complete processors on a single integrated circuit.

This change has a very important consequence for software developers: simply adding more processors will not magically improve the performance of the vast majority of **serial** programs, that is, programs that were written to run on a single processor. Such programs are unaware of the existence of multiple processors, and the performance of such a program on a system with multiple processors will be effectively the same as its performance on a single processor of the multiprocessor system.

All of this raises a number of questions:

1. Why do we care? Aren't single processor systems fast enough? After all, 20% per year is still a pretty significant performance improvement.
2. Why can't microprocessor manufacturers continue to develop much faster single processor systems? Why build **parallel systems**? Why build systems with multiple processors?
3. Why can't we write programs that will automatically convert serial programs into **parallel programs**, that is, programs that take advantage of the presence of multiple processors?

Let's take a brief look at each of these questions. Keep in mind, though, that some of the answers aren't carved in stone. For example, 20% per year may be more than adequate for many applications.

An Introduction to Parallel Programming. DOI: 10.1016/B978-0-12-374260-5.00001-4

1.1 WHY WE NEED EVER-INCREASING PERFORMANCE

The vast increases in computational power that we've been enjoying for decades now have been at the heart of many of the most dramatic advances in fields as diverse as science, the Internet, and entertainment. For example, decoding the human genome, ever more accurate medical imaging, astonishingly fast and accurate Web searches, and ever more realistic computer games would all have been impossible without these increases. Indeed, more recent increases in computational power would have been difficult, if not impossible, without earlier increases. But we can never rest on our laurels. As our computational power increases, the number of problems that we can seriously consider solving also increases. The following are a few examples:

- *Climate modeling*. In order to better understand climate change, we need far more accurate computer models, models that include interactions between the atmosphere, the oceans, solid land, and the ice caps at the poles. We also need to be able to make detailed studies of how various interventions might affect the global climate.
- *Protein folding*. It's believed that misfolded proteins may be involved in diseases such as Huntington's, Parkinson's, and Alzheimer's, but our ability to study configurations of complex molecules such as proteins is severely limited by our current computational power.
- *Drug discovery*. There are many ways in which increased computational power can be used in research into new medical treatments. For example, there are many drugs that are effective in treating a relatively small fraction of those suffering from some disease. It's possible that we can devise alternative treatments by careful analysis of the genomes of the individuals for whom the known treatment is ineffective. This, however, will involve extensive computational analysis of genomes.
- *Energy research*. Increased computational power will make it possible to program much more detailed models of technologies such as wind turbines, solar cells, and batteries. These programs may provide the information needed to construct far more efficient clean energy sources.
- *Data analysis*. We generate tremendous amounts of data. By some estimates, the quantity of data stored worldwide doubles every two years [28], but the vast majority of it is largely useless unless it's analyzed. As an example, knowing the sequence of nucleotides in human DNA is, by itself, of little use. Understanding how this sequence affects development and how it can cause disease requires extensive analysis. In addition to genomics, vast quantities of data are generated by particle colliders such as the Large Hadron Collider at CERN, medical imaging, astronomical research, and Web search engines—to name a few.

These and a host of other problems won't be solved without vast increases in computational power.

1.2 WHY WE'RE BUILDING PARALLEL SYSTEMS

Much of the tremendous increase in single processor performance has been driven by the ever-increasing density of transistors—the electronic switches—on integrated circuits. As the size of transistors decreases, their speed can be increased, and the overall speed of the integrated circuit can be increased. However, as the speed of transistors increases, their power consumption also increases. Most of this power is dissipated as heat, and when an integrated circuit gets too hot, it becomes unreliable. In the first decade of the twenty-first century, air-cooled integrated circuits are reaching the limits of their ability to dissipate heat [26].

Therefore, it is becoming impossible to continue to increase the speed of integrated circuits. However, the increase in transistor density *can* continue—at least for a while. Also, given the potential of computing to improve our existence, there is an almost moral imperative to continue to increase computational power. Finally, if the integrated circuit industry doesn't continue to bring out new and better products, it will effectively cease to exist.

How then, can we exploit the continuing increase in transistor density? The answer is *parallelism.* Rather than building ever-faster, more complex, monolithic processors, the industry has decided to put multiple, relatively simple, complete processors on a single chip. Such integrated circuits are called **multicore** processors, and **core** has become synonymous with central processing unit, or CPU. In this setting a conventional processor with one CPU is often called a **single-core** system.

1.3 WHY WE NEED TO WRITE PARALLEL PROGRAMS

Most programs that have been written for conventional, single-core systems cannot exploit the presence of multiple cores. We can run multiple instances of a program on a multicore system, but this is often of little help. For example, being able to run multiple instances of our favorite game program isn't really what we want—we want the program to run faster with more realistic graphics. In order to do this, we need to either rewrite our serial programs so that they're *parallel*, so that they can make use of multiple cores, or write translation programs, that is, programs that will automatically convert serial programs into parallel programs. The bad news is that researchers have had very limited success writing programs that convert serial programs in languages such as C and C++ into parallel programs.

This isn't terribly surprising. While we can write programs that recognize common constructs in serial programs, and automatically translate these constructs into efficient parallel constructs, the sequence of parallel constructs may be terribly inefficient. For example, we can view the multiplication of two $n \times n$ matrices as a sequence of dot products, but parallelizing a matrix multiplication as a sequence of parallel dot products is likely to be very slow on many systems.

An efficient parallel implementation of a serial program may not be obtained by finding efficient parallelizations of each of its steps. Rather, the best parallelization may be obtained by stepping back and devising an entirely new algorithm.

As an example, suppose that we need to compute n values and add them together. We know that this can be done with the following serial code:

```
sum = 0;
for (i = 0; i < n; i++) {
   x = Compute_next_value(. . .);
   sum += x;
}
```

Now suppose we also have p cores and p is much smaller than n. Then each core can form a partial sum of approximately n/p values:

```
my_sum = 0;
my_first_i = . . . ;
my_last_i = . . . ;
for (my_i = my_first_i; my_i < my_last_i; my_i++) {
   my_x = Compute_next_value(. . .);
   my_sum += my_x;
}
```

Here the prefix `my_` indicates that each core is using its own, private variables, and each core can execute this block of code independently of the other cores.

After each core completes execution of this code, its variable `my_sum` will store the sum of the values computed by its calls to `Compute_next_value`. For example, if there are eight cores, $n = 24$, and the 24 calls to `Compute_next_value` return the values

$$1,4,3,\ 9,2,8,\ 5,1,1,\ 6,2,7,\ 2,5,0,\ 4,1,8,\ 6,5,1,\ 2,3,9,$$

then the values stored in `my_sum` might be

Core	0	1	2	3	4	5	6	7
my_sum	8	19	7	15	7	13	12	14

Here we're assuming the cores are identified by nonnegative integers in the range $0, 1, \ldots, p-1$, where p is the number of cores.

When the cores are done computing their values of `my_sum`, they can form a global sum by sending their results to a designated "master" core, which can add their results:

```
if (I'm the master core) {
   sum = my_x;
   for each core other than myself {
      receive value from core;
      sum += value;
   }
```

```
} else {
   send my_x to the master;
}
```

In our example, if the master core is core 0, it would add the values $8 + 19 + 7 + 15 + 7 + 13 + 12 + 14 = 95$.

But you can probably see a better way to do this—especially if the number of cores is large. Instead of making the master core do all the work of computing the final sum, we can pair the cores so that while core 0 adds in the result of core 1, core 2 can add in the result of core 3, core 4 can add in the result of core 5 and so on. Then we can repeat the process with only the even-ranked cores: 0 adds in the result of 2, 4 adds in the result of 6, and so on. Now cores divisible by 4 repeat the process, and so on. See Figure 1.1. The circles contain the current value of each core's sum, and the lines with arrows indicate that one core is sending its sum to another core. The plus signs indicate that a core is receiving a sum from another core and adding the received sum into its own sum.

For both "global" sums, the master core (core 0) does more work than any other core, and the length of time it takes the program to complete the final sum should be the length of time it takes for the master to complete. However, with eight cores, the master will carry out seven receives and adds using the first method, while with the second method it will only carry out three. So the second method results in an improvement of more than a factor of two. The difference becomes much more

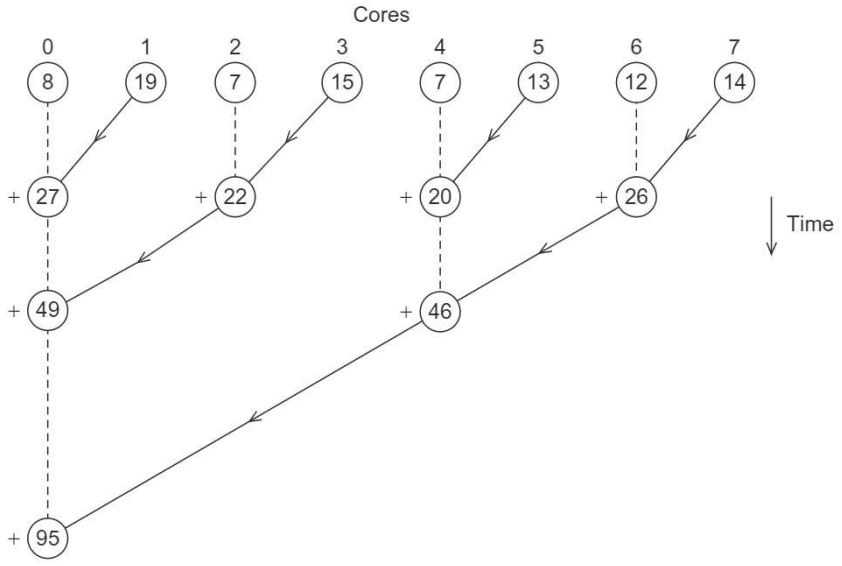

FIGURE 1.1

Multiple cores forming a global sum

dramatic with large numbers of cores. With 1000 cores, the first method will require 999 receives and adds, while the second will only require 10, an improvement of almost a factor of 100!

The first global sum is a fairly obvious generalization of the serial global sum: divide the work of adding among the cores, and after each core has computed its part of the sum, the master core simply repeats the basic serial addition—if there are p cores, then it needs to add p values. The second global sum, on the other hand, bears little relation to the original serial addition.

The point here is that it's unlikely that a translation program would "discover" the second global sum. Rather there would more likely be a predefined efficient global sum that the translation program would have access to. It could "recognize" the original serial loop and replace it with a precoded, efficient, parallel global sum.

We might expect that software could be written so that a large number of common serial constructs could be recognized and efficiently **parallelized**, that is, modified so that they can use multiple cores. However, as we apply this principle to ever more complex serial programs, it becomes more and more difficult to recognize the construct, and it becomes less and less likely that we'll have a precoded, efficient parallelization.

Thus, we cannot simply continue to write serial programs, we must write parallel programs, programs that exploit the power of multiple processors.

1.4 HOW DO WE WRITE PARALLEL PROGRAMS?

There are a number of possible answers to this question, but most of them depend on the basic idea of *partitioning* the work to be done among the cores. There are two widely used approaches: **task-parallelism** and **data-parallelism**. In task-parallelism, we partition the various tasks carried out in solving the problem among the cores. In data-parallelism, we partition the data used in solving the problem among the cores, and each core carries out more or less similar operations on its part of the data.

As an example, suppose that Prof P has to teach a section of "Survey of English Literature." Also suppose that Prof P has one hundred students in her section, so she's been assigned four teaching assistants (TAs): Mr A, Ms B, Mr C, and Ms D. At last the semester is over, and Prof P makes up a final exam that consists of five questions. In order to grade the exam, she and her TAs might consider the following two options: each of them can grade all one hundred responses to one of the questions; say P grades question 1, A grades question 2, and so on. Alternatively, they can divide the one hundred exams into five piles of twenty exams each, and each of them can grade all the papers in one of the piles; P grades the papers in the first pile, A grades the papers in the second pile, and so on.

In both approaches the "cores" are the professor and her TAs. The first approach might be considered an example of task-parallelism. There are five tasks to be carried out: grading the first question, grading the second question, and so on. Presumably, the graders will be looking for different information in question 1, which is about

Shakespeare, from the information in question 2, which is about Milton, and so on. So the professor and her TAs will be "executing different instructions."

On the other hand, the second approach might be considered an example of data-parallelism. The "data" are the students' papers, which are divided among the cores, and each core applies more or less the same grading instructions to each paper.

The first part of the global sum example in Section 1.3 would probably be considered an example of data-parallelism. The data are the values computed by `Compute_next_value`, and each core carries out roughly the same operations on its assigned elements: it computes the required values by calling `Compute_next_value` and adds them together. The second part of the first global sum example might be considered an example of task-parallelism. There are two tasks: receiving and adding the cores' partial sums, which is carried out by the master core, and giving the partial sum to the master core, which is carried out by the other cores.

When the cores can work independently, writing a parallel program is much the same as writing a serial program. Things get a good deal more complex when the cores need to coordinate their work. In the second global sum example, although the tree structure in the diagram is very easy to understand, writing the actual code is relatively complex. See Exercises 1.3 and 1.4. Unfortunately, it's much more common for the cores to need coordination.

In both global sum examples, the coordination involves **communication**: one or more cores send their current partial sums to another core. The global sum examples should also involve coordination through **load balancing**: even though we didn't give explicit formulas, it's clear that we want the cores all to be assigned roughly the same number of values to compute. If, for example, one core has to compute most of the values, then the other cores will finish much sooner than the heavily loaded core, and their computational power will be wasted.

A third type of coordination is **synchronization**. As an example, suppose that instead of computing the values to be added, the values are read from `stdin`. Say `x` is an array that is read in by the master core:

```
if (I'm the master core)
   for (my_i = 0; my_i < n; my_i++)
      scanf("%lf", &x[my_i]);
```

In most systems the cores are not automatically synchronized. Rather, each core works at its own pace. In this case, the problem is that we don't want the other cores to race ahead and start computing their partial sums before the master is done initializing `x` and making it available to the other cores. That is, the cores need to wait before starting execution of the code:

```
for (my_i = my_first_i; my_i < my_last_i; my_i++)
   my_sum += x[my_i];
```

We need to add in a point of synchronization between the initialization of `x` and the computation of the partial sums:

```
Synchronize_cores();
```

The idea here is that each core will wait in the function `Synchronize_cores` until all the cores have entered the function—in particular, until the master core has entered this function.

Currently, the most powerful parallel programs are written using *explicit* parallel constructs, that is, they are written using extensions to languages such as C and C++. These programs include explicit instructions for parallelism: core 0 executes task 0, core 1 executes task 1, . . . , all cores synchronize, . . . , and so on, so such programs are often extremely complex. Furthermore, the complexity of modern cores often makes it necessary to use considerable care in writing the code that will be executed by a single core.

There are other options for writing parallel programs—for example, higher level languages—but they tend to sacrifice performance in order to make program development somewhat easier.

1.5 WHAT WE'LL BE DOING

We'll be focusing on learning to write programs that are *explicitly* parallel. Our purpose is to learn the basics of programming parallel computers using the C language and three different extensions to C: the **Message-Passing Interface** or **MPI, POSIX threads** or **Pthreads,** and **OpenMP**. MPI and Pthreads are libraries of type definitions, functions, and macros that can be used in C programs. OpenMP consists of a library and some modifications to the C compiler.

You may well wonder why we're learning three different extensions to C instead of just one. The answer has to do with both the extensions and parallel systems. There are two main types of parallel systems that we'll be focusing on: **shared-memory** systems and **distributed-memory** systems. In a shared-memory system, the cores can share access to the computer's memory; in principle, each core can read and write each memory location. In a shared-memory system, we can coordinate the cores by having them examine and update shared-memory locations. In a distributed-memory system, on the other hand, each core has its own, private memory, and the cores must communicate explicitly by doing something like sending messages across a network. Figure 1.2 shows a schematic of the two types of systems. Pthreads and OpenMP were designed for programming shared-memory systems. They provide mechanisms for accessing shared-memory locations. MPI, on the other hand, was designed for programming distributed-memory systems. It provides mechanisms for sending messages.

But why two extensions for shared-memory? OpenMP is a relatively high-level extension to C. For example, it can "parallelize" our addition loop

```
sum = 0;
for (i = 0; i < n; i++) {
   x = Compute_next_value(. . .);
   sum += x;
}
```

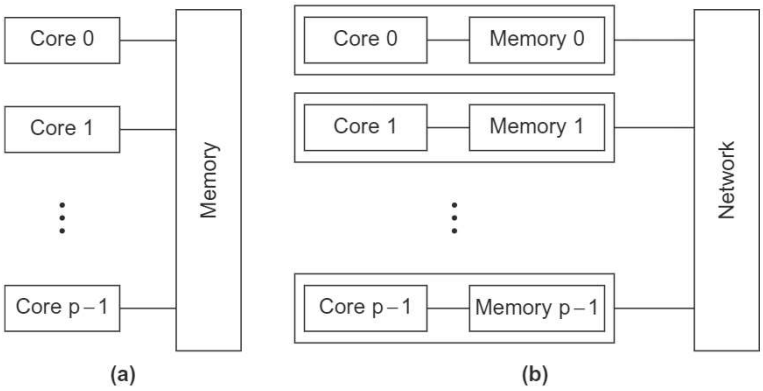

FIGURE 1.2

(a) A shared-memory system and (b) a distributed-memory system

with a single directive, while Pthreads requires that we do something similar to our example. On the other hand, Pthreads provides some coordination constructs that are unavailable in OpenMP. OpenMP allows us to parallelize many programs with relative ease, while Pthreads provides us with some constructs that make other programs easier to parallelize.

1.6 CONCURRENT, PARALLEL, DISTRIBUTED

If you look at some other books on parallel computing or you search the Web for information on parallel computing, you're likely to also run across the terms **concurrent computing** and **distributed computing**. Although there isn't complete agreement on the distinction between the terms parallel, distributed, and concurrent, many authors make the following distinctions:

- In concurrent computing, a program is one in which multiple tasks can be *in progress* at any instant [4].
- In parallel computing, a program is one in which multiple tasks *cooperate closely* to solve a problem.
- In distributed computing, a program may need to cooperate with other programs to solve a problem.

So parallel and distributed programs are concurrent, but a program such as a multitasking operating system is also concurrent, even when it is run on a machine with only one core, since multiple tasks can be *in progress* at any instant. There isn't a clear-cut distinction between parallel and distributed programs, but a parallel program usually runs multiple tasks simultaneously on cores that are physically close to each other and that either share the same memory or are connected by a very high-speed

network. On the other hand, distributed programs tend to be more "loosely coupled." The tasks may be executed by multiple computers that are separated by large distances, and the tasks themselves are often executed by programs that were created independently. As examples, our two concurrent addition programs would be considered parallel by most authors, while a Web search program would be considered distributed.

But beware, there isn't general agreement on these terms. For example, many authors consider shared-memory programs to be "parallel" and distributed-memory programs to be "distributed." As our title suggests, we'll be interested in parallel programs—programs in which closely coupled tasks cooperate to solve a problem.

1.7 THE REST OF THE BOOK

How can we use this book to help us write parallel programs?

First, when you're interested in high-performance, whether you're writing serial or parallel programs, you need to know a little bit about the systems you're working with—both hardware and software. In Chapter 2, we'll give an overview of parallel hardware and software. In order to understand this discussion, it will be necessary to review some information on serial hardware and software. Much of the material in Chapter 2 won't be needed when we're getting started, so you might want to skim some of this material, and refer back to it occasionally when you're reading later chapters.

The heart of the book is contained in Chapters 3 through 6. Chapters 3, 4, and 5 provide a very elementary introduction to programming parallel systems using C and MPI, Pthreads, and OpenMP, respectively. The only prerequisite for reading these chapters is a knowledge of C programming. We've tried to make these chapters independent of each other, and you should be able to read them in any order. However, in order to make them independent, we did find it necessary to repeat some material. So if you've read one of the three chapters, and you go on to read another, be prepared to skim over some of the material in the new chapter.

Chapter 6 puts together all we've learned in the preceding chapters, and develops two fairly large programs in both a shared- and a distributed-memory setting. However, it should be possible to read much of this even if you've only read one of Chapters 3, 4, or 5. The last chapter, Chapter 7, provides a few suggestions for further study on parallel programming.

1.8 A WORD OF WARNING

Before proceeding, a word of warning. It may be tempting to write parallel programs "by the seat of your pants," without taking the trouble to carefully design and incrementally develop your program. This will almost certainly be a mistake. Every

parallel program contains at least one serial program. Since we almost always need to coordinate the actions of multiple cores, writing parallel programs is almost always more complex than writing a serial program that solves the same problem. In fact, it is often *far* more complex. All the rules about careful design and development are usually *far* more important for the writing of parallel programs than they are for serial programs.

1.9 TYPOGRAPHICAL CONVENTIONS

We'll make use of the following typefaces in the text:

- Program text, displayed or within running text, will use the following typefaces:

```
/*  This is a short program */
#include <stdio.h>

int main(int argc, char* argv[]) {
   printf("hello, world\n");

   return 0;
}
```

- Definitions are given in the body of the text, and the term being defined is printed in boldface type: A **parallel program** can make use of multiple cores.
- When we need to refer to the environment in which a program is being developed, we'll assume that we're using a UNIX shell, and we'll use a `$` to indicate the shell prompt:

```
$ gcc –g –Wall –o hello hello.c
```

- We'll specify the syntax of function calls with fixed argument lists by including a sample argument list. For example, the integer absolute value function, `abs`, in `stdlib` might have its syntax specified with

```
int abs(int x);  /* Returns absolute value of int x */
```

  For more complicated syntax, we'll enclose required content in angle brackets `<>` and optional content in square brackets `[]`. For example, the C `if` statement might have its syntax specified as follows:

```
if ( <expression> )
   <statement1>
[else
   <statement2>]
```

  This says that the `if` statement must include an expression enclosed in parentheses, and the right parenthesis must be followed by a statement. This statement can be followed by an optional `else` clause. If the `else` clause is present, it must include a second statement.

1.10 SUMMARY

For many years we've enjoyed the fruits of ever faster processors. However, because of physical limitations the rate of performance improvement in conventional processors is decreasing. In order to increase the power of processors, chipmakers have turned to **multicore** integrated circuits, that is, integrated circuits with multiple conventional processors on a single chip.

Ordinary **serial** programs, which are programs written for a conventional single-core processor, usually cannot exploit the presence of multiple cores, and it's unlikely that translation programs will be able to shoulder all the work of **parallelizing** serial programs, meaning converting them into **parallel programs**, which can make use of multiple cores. As software developers, we need to learn to write parallel programs.

When we write parallel programs, we usually need to **coordinate** the work of the cores. This can involve **communication** among the cores, **load balancing**, and **synchronization** of the cores.

In this book we'll be learning to program parallel systems so that we can maximize their performance. We'll be using the C language with either MPI, Pthreads, or OpenMP. MPI is used for programming **distributed-memory** systems, and Pthreads and OpenMP are used for programming **shared-memory** systems. In distributed-memory systems, the cores have their own private memories, while in shared-memory systems, it's possible, in principle, for each core to access each memory location.

Concurrent programs can have multiple tasks in progress at any instant. **Parallel** and **distributed** programs usually have tasks that execute simultaneously. There isn't a hard and fast distinction between parallel and distributed, although in parallel programs, the tasks are usually more tightly coupled.

Parallel programs are usually very complex. So it's even more important to use good program development techniques with parallel programs.

1.11 EXERCISES

1.1 Devise formulas for the functions that calculate `my_first_i` and `my_last_i` in the global sum example. Remember that each core should be assigned roughly the same number of elements of computations in the loop. *Hint*: First consider the case when n is evenly divisible by p.

1.2 We've implicitly assumed that each call to `Compute_next_value` requires roughly the same amount of work as the other calls. How would you change your answer to the preceding question if call $i = k$ requires $k + 1$ times as much work as the call with $i = 0$? So if the first call ($i = 0$) requires 2 milliseconds, the second call ($i = 1$) requires 4, the third ($i = 2$) requires 6, and so on.

1.3 Try to write pseudo-code for the tree-structured global sum illustrated in Figure 1.1. Assume the number of cores is a power of two (1, 2, 4, 8, ...).

Hints: Use a variable `divisor` to determine whether a core should send its sum or receive and add. The `divisor` should start with the value 2 and be doubled after each iteration. Also use a variable `core_difference` to determine which core should be partnered with the current core. It should start with the value 1 and also be doubled after each iteration. For example, in the first iteration `0 % divisor = 0` and `1 % divisor = 1`, so 0 receives and adds, while 1 sends. Also in the first iteration `0 + core_difference = 1` and `1 - core_difference = 0`, so 0 and 1 are paired in the first iteration.

1.4 As an alternative to the approach outlined in the preceding problem, we can use C's bitwise operators to implement the tree-structured global sum. In order to see how this works, it helps to write down the binary (base 2) representation of each of the core ranks, and note the pairings during each stage:

	Stages		
Cores	*1*	*2*	*3*
$0_{10} = 000_2$	$1_{10} = 001_2$	$2_{10} = 010_2$	$4_{10} = 100_2$
$1_{10} = 001_2$	$0_{10} = 000_2$	×	×
$2_{10} = 010_2$	$3_{10} = 011_2$	$0_{10} = 000_2$	×
$3_{10} = 011_2$	$2_{10} = 010_2$	×	×
$4_{10} = 100_2$	$5_{10} = 101_2$	$6_{10} = 110_2$	$0_{10} = 000_2$
$5_{10} = 101_2$	$4_{10} = 100_2$	×	×
$6_{10} = 110_2$	$7_{10} = 111_2$	$4_{10} = 100_2$	×
$7_{10} = 111_2$	$6_{10} = 110_2$	×	×

From the table we see that during the first stage each core is paired with the core whose rank differs in the rightmost or first bit. During the second stage cores that continue are paired with the core whose rank differs in the second bit, and during the third stage cores are paired with the core whose rank differs in the third bit. Thus, if we have a binary value `bitmask` that is 001_2 for the first stage, 010_2 for the second, and 100_2 for the third, we can get the rank of the core we're paired with by "inverting" the bit in our rank that is nonzero in `bitmask`. This can be done using the bitwise exclusive or $\wedge$ operator.

Implement this algorithm in pseudo-code using the bitwise exclusive or and the left-shift operator.

1.5 What happens if your pseudo-code in Exercise 1.3 or Exercise 1.4 is run when the number of cores is *not* a power of two (e.g., 3, 5, 6, 7)? Can you modify the pseudo-code so that it will work correctly regardless of the number of cores?

1.6 Derive formulas for the number of receives and additions that core 0 carries out using

a. the original pseudo-code for a global sum, and

b. the tree-structured global sum.

Make a table showing the numbers of receives and additions carried out by core 0 when the two sums are used with $2, 4, 8, \ldots, 1024$ cores.

1.7 The first part of the global sum example—when each core adds its assigned computed values—is usually considered to be an example of data-parallelism, while the second part of the first global sum—when the cores send their partial sums to the master core, which adds them—could be considered to be an example of task-parallelism. What about the second part of the second global sum—when the cores use a tree structure to add their partial sums? Is this an example of data- or task-parallelism? Why?

1.8 Suppose the faculty are going to have a party for the students in the department.

- **a.** Identify tasks that can be assigned to the faculty members that will allow them to use task-parallelism when they prepare for the party. Work out a schedule that shows when the various tasks can be performed.
- **b.** We might hope that one of the tasks in the preceding part is cleaning the house where the party will be held. How can we use data-parallelism to partition the work of cleaning the house among the faculty?
- **c.** Use a combination of task- and data-parallelism to prepare for the party. (If there's too much work for the faculty, you can use TAs to pick up the slack.)

1.9 Write an essay describing a research problem in your major that would benefit from the use of parallel computing. Provide a rough outline of how parallelism would be used. Would you use task- or data-parallelism?