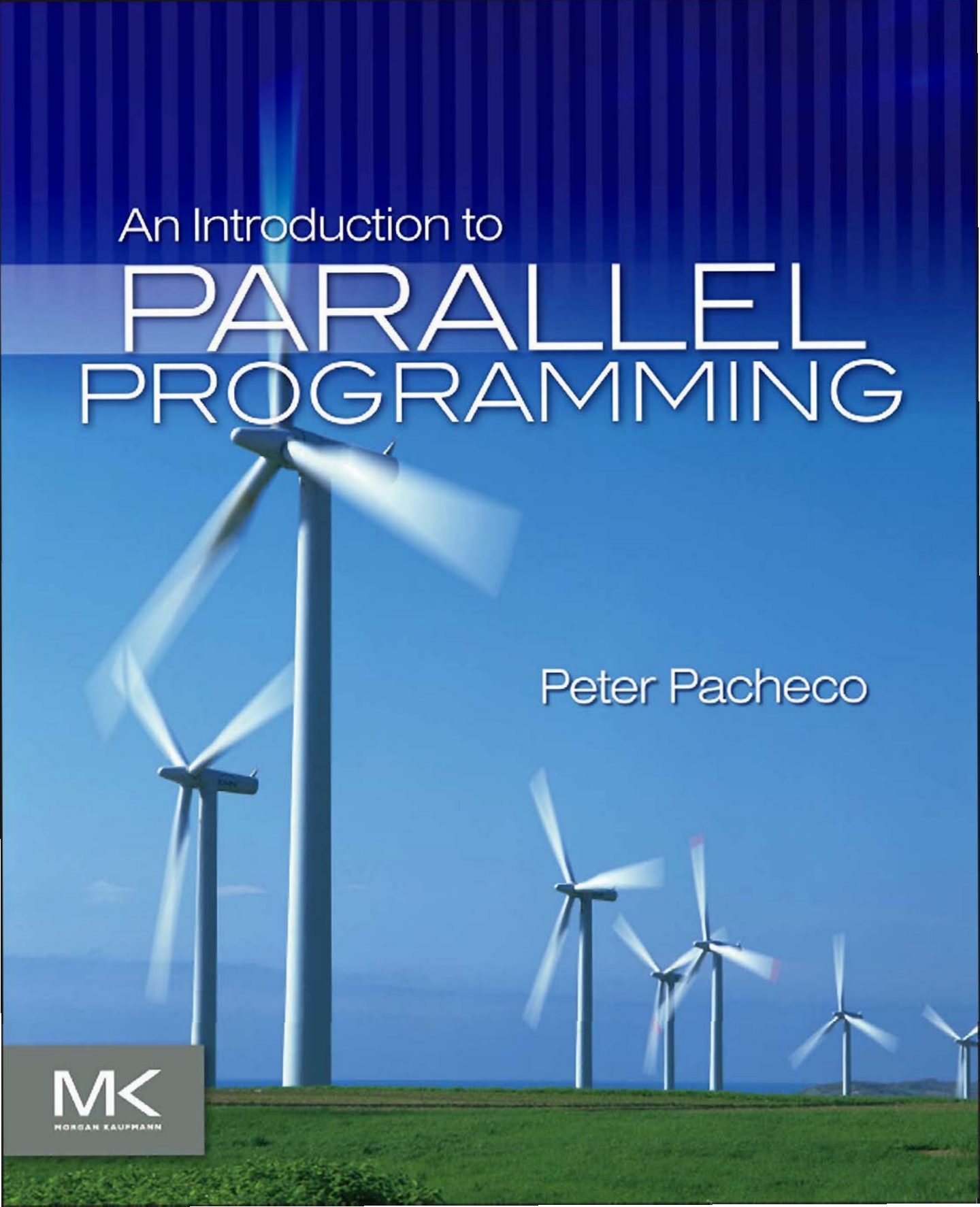
An Introduction to PARALLEL PROGRAMMING



Peter Pacheco

MK
MORGAN KAUFMANN

An Introduction to Parallel Programming

Peter S. Pacheco
University of San Francisco

AMSTERDAM • BOSTON • HEIDELBERG • LONDON
NEW YORK • OXFORD • PARIS • SAN DIEGO
SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO

Morgan Kaufmann Publishers is an imprint of Elsevier

Acquiring Editor: Todd Green
Development Editor: Nate McFadden
Project Manager: Marilyn E. Rash
Designer: Joanne Blank

Morgan Kaufmann Publishers is an imprint of Elsevier.
30 Corporate Drive, Suite 400
Burlington, MA 01803, USA

Notices

Knowledge and best practice in this field are constantly changing. As new research and experience broaden our understanding, changes in research methods, professional practices, or medical treatment may become necessary.

Practitioners and researchers must always rely on their own experience and knowledge in evaluating and using any information, methods, compounds, or experiments described herein. In using such information or methods they should be mindful of their own safety and the safety of others, including parties for whom they have a professional responsibility.

To the fullest extent of the law, neither the Publisher nor the authors, contributors, or editors, assume any liability for any injury and/or damage to persons or property as a matter of products liability, negligence or otherwise, or from any use or operation of any methods, products, instructions, or ideas contained in the material herein.

Library of Congress Cataloging-in-Publication Data
Pacheco, Peter S.
An introduction to parallel programming / Peter S. Pacheco.
p. cm.
ISBN 978-0-12-374260-5 (hardback)
1. Parallel programming (Computer science) I. Title.
QA76.642.P29 2011
005.2′75–dc22

2010039584

British Library Cataloguing-in-Publication Data
A catalogue record for this book is available from the British Library.

For information on all Morgan Kaufmann publications,
visit our web site at *www.mkp.com* or *www.elsevierdirect.com*

Printed in the United States
11 12 13 14 15 10 9 8 7 6 5 4 3 2 1

2.4 PARALLEL SOFTWARE

Parallel hardware has arrived. Virtually all desktop and server systems use multicore processors. The same cannot be said for parallel software. Except for operating systems, database systems, and Web servers, there is currently very little commodity software that makes extensive use of parallel hardware. As we noted in Chapter 1, this is a problem because we can no longer rely on hardware and compilers to provide a steady increase in application performance. If we're to continue to have routine increases in application performance and application power, software developers must learn to write applications that exploit shared- and distributed-memory architectures. In this section we'll take a look at some of the issues involved in writing software for parallel systems.

First, some terminology. Typically when we run our shared-memory programs, we'll start a single process and fork multiple threads. So when we discuss shared-memory programs, we'll talk about *threads* carrying out tasks. On the other hand, when we run distributed-memory programs, we'll start multiple processes, and we'll talk about *processes* carrying out tasks. When the discussion applies equally well to shared-memory and distributed-memory systems, we'll talk about *processes/threads* carrying out tasks.

2.4.1 Caveats

Before proceeding, we need to stress some of the limitations of this section. First, here, and in the remainder of the book, we'll only be discussing software for MIMD systems. For example, while the use of GPUs as a platform for parallel computing continues to grow at a rapid pace, the application programming interfaces (APIs) for GPUs are necessarily very different from standard MIMD APIs. Second, we stress that our coverage is only meant to give some idea of the issues: there is no attempt to be comprehensive.

Finally, we'll mainly focus on what's often called **single program, multiple data**, or SPMD, programs. Instead of running a different program on each core, SPMD programs consist of a single executable that can behave as if it were multiple different programs through the use of conditional branches. For example,

```
if (I'm thread/process 0)
   do this;
else
   do that;
```

Observe that SPMD programs can readily implement data-parallelism. For example,

```
if (I'm thread/process 0)
   operate on the first half of the array;
else /* I'm thread/process 1 */
   operate on the second half of the array;
```

Recall that a program is **task parallel** if it obtains its parallelism by dividing tasks among the threads or processes. The first example makes it clear that SPMD programs can also implement **task-parallelism**.

2.4.2 Coordinating the processes/threads

In a very few cases, obtaining excellent parallel performance is trivial. For example, suppose we have two arrays and we want to add them:

```
double x[n], y[n];
. . .
for (int i = 0; i < n; i++)
   x[i] += y[i];
```

In order to parallelize this, we only need to assign elements of the arrays to the processes/threads. For example, if we have p processes/threads, we might make process/thread 0 responsible for elements $0,\ldots,n/p-1$, process/thread 1 would be responsible for elements $n/p,\ldots,2n/p-1$, and so on.

So for this example, the programmer only needs to

1. Divide the work among the processes/threads
   a. in such a way that each process/thread gets roughly the same amount of work, and
   b. in such a way that the amount of communication required is minimized.

   Recall that the process of dividing the work among the processes/threads so that (a) is satisfied is called **load balancing**. The two qualifications on dividing the work are obvious, but nonetheless important. In many cases it won't be necessary to give much thought to them; they typically become concerns in situations in which the amount of work isn't known in advance by the programmer, but rather the work is generated as the program runs. For an example, see the tree search problem in Chapter 6.

   Although we might wish for a term that's a little easier to pronounce, recall that the process of converting a serial program or algorithm into a parallel program is often called **parallelization**. Programs that can be parallelized by simply dividing the work among the processes/threads are sometimes said to be **embarrassingly parallel**. This is a bit unfortunate, since it suggests that programmers should be embarrassed to have written an embarrassingly parallel program, when, to the contrary, successfully devising a parallel solution to *any* problem is a cause for great rejoicing.

   Alas, the vast majority of problems are much more determined to resist our efforts to find a parallel solution. As we saw in Chapter 1, for these problems, we need to coordinate the work of the processes/threads. In these programs, we also usually need to

2. Arrange for the processes/threads to synchronize.
3. Arrange for communication among the processes/threads.

These last two problems are often interrelated. For example, in distributed-memory programs, we often implicitly synchronize the processes by communicating among

them, and in shared-memory programs, we often communicate among the threads by synchronizing them. We'll say more about both issues below.

2.4.3 Shared-memory

As we noted earlier, in shared-memory programs, variables can be **shared** or **private**. Shared variables can be read or written by any thread, and private variables can ordinarily only be accessed by one thread. Communication among the threads is usually done through shared variables, so communication is implicit, rather than explicit.

Dynamic and static threads

In many environments shared-memory programs use **dynamic threads**. In this paradigm, there is often a master thread and at any given instant a (possibly empty) collection of worker threads. The master thread typically waits for work requests—for example, over a network—and when a new request arrives, it forks a worker thread, the thread carries out the request, and when the thread completes the work, it terminates and joins the master thread. This paradigm makes efficient use of system resources since the resources required by a thread are only being used while the thread is actually running.

An alternative to the dynamic paradigm is the **static thread** paradigm. In this paradigm, all of the threads are forked after any needed setup by the master thread and the threads run until all the work is completed. After the threads join the master thread, the master thread may do some cleanup (e.g., free memory) and then it also terminates. In terms of resource usage, this may be less efficient: if a thread is idle, its resources (e.g., stack, program counter, and so on.) can't be freed. However, forking and joining threads can be fairly time-consuming operations. So if the necessary resources are available, the static thread paradigm has the potential for better performance than the dynamic paradigm. It also has the virtue that it's closer to the most widely used paradigm for distributed-memory programming, so part of the mindset that is used for one type of system is preserved for the other. Hence, we'll often use the static thread paradigm.

Nondeterminism

In any MIMD system in which the processors execute asynchronously it is likely that there will be **nondeterminism**. A computation is nondeterministic if a given input can result in different outputs. If multiple threads are executing independently, the relative rate at which they'll complete statements varies from run to run, and hence the results of the program may be different from run to run. As a very simple example, suppose we have two threads, one with id or rank 0 and the other with id or rank 1. Suppose also that each is storing a private variable `my_x`, thread 0's value for `my_x` is 7, and thread 1's is 19. Further, suppose both threads execute the following code:

```
. . .
printf("Thread %d > my_val = %d\n", my_rank, my_x);
. . .
```

Then the output could be

```
Thread 0 > my_val = 7
Thread 1 > my_val = 19
```

but it could also be

```
Thread 1 > my_val = 19
Thread 0 > my_val = 7
```

In fact, things could be even worse: the output of one thread could be broken up by the output of the other thread. However, the point here is that because the threads are executing independently and interacting with the operating system, the time it takes for one thread to complete a block of statements varies from execution to execution, so the order in which these statements complete can't be predicted.

In many cases nondeterminism isn't a problem. In our example, since we've labelled the output with the thread's rank, the order in which the output appears probably doesn't matter. However, there are also many cases in which nondeterminism—especially in shared-memory programs—can be disastrous, because it can easily result in program errors. Here's a simple example with two threads.

Suppose each thread computes an `int`, which it stores in a private variable `my_val`. Suppose also that we want to add the values stored in `my_val` into a shared-memory location `x` that has been initialized to 0. Both threads therefore want to execute code that looks something like this:

```
my_val = Compute_val(my_rank);
x += my_val;
```

Now recall that an addition typically requires loading the two values to be added into registers, adding the values, and finally storing the result. To keep things relatively simple, we'll assume that values are loaded from main memory directly into registers and stored in main memory directly from registers. Here is one possible sequence of events:

Time	Core 0	Core 1
0	Finish assignment to `my_val`	In call to `Compute_val`
1	Load `x = 0` into register	Finish assignment to `my_val`
2	Load `my_val = 7` into register	Load `x = 0` into register
3	Add `my_val = 7` to `x`	Load `my_val = 19` into register
4	Store `x = 7`	Add `my_val` to `x`
5	Start other work	Store `x = 19`

Clearly this is not what we want, and it's easy to imagine other sequences of events that result in an incorrect value for `x`. The nondeterminism here is a result of the fact that two threads are attempting to more or less simultaneously update the memory location `x`. When threads or processes attempt to simultaneously access a

resource, and the accesses can result in an error, we often say the program has a **race condition**, because the threads or processes are in a "horse race." That is, the outcome of the computation depends on which thread wins the race. In our example, the threads are in a race to execute `x += my_val`. In this case, unless one thread completes `x += my_val` before the other thread starts, the result will be incorrect. A block of code that can only be executed by one thread at a time is called a **critical section**, and it's usually our job as programmers to insure **mutually exclusive** access to the critical section. In other words, we need to insure that if one thread is executing the code in the critical section, then the other threads are excluded.

The most commonly used mechanism for insuring mutual exclusion is a **mutual exclusion lock** or **mutex** or **lock**. A mutex is a special type of object that has support in the underlying hardware. The basic idea is that each critical section is *protected* by a lock. Before a thread can execute the code in the critical section, it must "obtain" the mutex by calling a mutex function, and, when it's done executing the code in the critical section, it should "relinquish" the mutex by calling an unlock function. While one thread "owns" the lock—that is, has returned from a call to the lock function, but hasn't yet called the unlock function—any other thread attempting to execute the code in the critical section will wait in its call to the lock function.

Thus, in order to insure that our code functions correctly, we might modify it so that it looks something like this:

```
my_val = Compute_val(my_rank);
Lock(&add_my_val_lock);
x += my_val;
Unlock(&add_my_val_lock);
```

This insures that only one thread at a time can execute the statement `x += my_val`. Note that the code does *not* impose any predetermined order on the threads. Either thread 0 or thread 1 can execute `x += my_val` first.

Also note that the use of a mutex enforces **serialization** of the critical section. Since only one thread at a time can execute the code in the critical section, this code is effectively serial. Thus, we want our code to have as few critical sections as possible, and we want our critical sections to be as short as possible.

There are alternatives to mutexes. In **busy-waiting**, a thread enters a loop whose sole purpose is to test a condition. In our example, suppose there is a shared variable `ok_for_1` that has been initialized to false. Then something like the following code can insure that thread 1 won't update `x` until after thread 0 has updated it:

```
my_val = Compute_val(my_rank);
if (my_rank == 1)
   while (!ok_for_1);  /* Busy-wait loop */
x += my_val;           /* Critical section */
if (my_rank == 0)
   ok_for_1 = true;    /* Let thread 1 update x */
```

So until thread 0 executes `ok_for_1 = true`, thread 1 will be stuck in the loop `while (!ok_for_1)`. This loop is called a "busy-wait" because the thread can be

very busy waiting for the condition. This has the virtue that it's simple to understand and implement. However, it can be very wasteful of system resources, because even when a thread is doing no useful work, the core running the thread will be repeatedly checking to see if the critical section can be entered. **Semaphores** are similar to mutexes, although the details of their behavior are slightly different, and there are some types of thread synchronization that are easier to implement with semaphores than mutexes. A **monitor** provides mutual exclusion at a somewhat higher-level: it is an object whose methods can only be executed by one thread at a time. We'll discuss busy-waiting and semaphores in Chapter 4.

There are a number of other alternatives that are currently being studied but that are not yet widely available. The one that has attracted the most attention is probably **transactional memory** [31]. In database management systems, a **transaction** is an access to a database that the system treats as a single unit. For example, transferring $1000 from your savings account to your checking account should be treated by your bank's software as a transaction, so that the software can't debit your savings account without also crediting your checking account. If the software was able to debit your savings account, but was then unable to credit your checking account, it would *roll-back* the transaction. In other words, the transaction would either be fully completed or any partial changes would be erased. The basic idea behind transactional memory is that critical sections in shared-memory programs should be treated as transactions. Either a thread successfully completes the critical section or any partial results are rolled back and the critical section is repeated.

Thread safety

In many, if not most, cases parallel programs can call functions developed for use in serial programs, and there won't be any problems. However, there are some notable exceptions. The most important exception for C programmers occurs in functions that make use of *static* local variables. Recall that ordinary C local variables—variables declared inside a function—are allocated from the system stack. Since each thread has its own stack, ordinary C local variables are private. However, recall that a static variable that's declared in a function persists from one call to the next. Thus, static variables are effectively shared among any threads that call the function, and this can have unexpected and unwanted consequences.

For example, the C string library function `strtok` splits an input string into substrings. When it's first called, it's passed a string, and on subsequent calls it returns successive substrings. This can be arranged through the use of a static `char*` variable that refers to the string that was passed on the first call. Now suppose two threads are splitting strings into substrings. Clearly, if, for example, thread 0 makes its first call to `strtok`, and then thread 1 makes its first call to `strtok` before thread 0 has completed splitting its string, then thread 0's string will be lost or overwritten, and, on subsequent calls it may get substrings of thread 1's strings.

A function such as `strtok` is not **thread safe**. This means that if it is used in a multithreaded program, there may be errors or unexpected results. When a block of code isn't thread safe, it's usually because different threads are accessing shared

data. Thus, as we've seen, even though many serial functions can be used safely in multithreaded programs—that is, they're *thread safe*—programmers need to be wary of functions that were written exclusively for use in serial programs. We'll take a closer look at thread safety in Chapters 4 and 5.

2.4.4 Distributed-memory

In distributed-memory programs, the cores can directly access only their own, private memories. There are several APIs that are used. However, by far the most widely used is message-passing. So we'll devote most of our attention in this section to message-passing. Then we'll take a brief look at at a couple of other, less widely used, APIs.

Perhaps the first thing to note regarding distributed-memory APIs is that they can be used with shared-memory hardware. It's perfectly feasible for programmers to logically partition shared-memory into private address spaces for the various threads, and a library or compiler can implement the communication that's needed.

As we noted earlier, distributed-memory programs are usually executed by starting multiple processes rather than multiple threads. This is because typical "threads of execution" in a distributed-memory program may run on independent CPUs with independent operating systems, and there may be no software infrastructure for starting a single "distributed" process and having that process fork one or more threads on each node of the system.

Message-passing

A message-passing API provides (at a minimum) a send and a receive function. Processes typically identify each other by ranks in the range $0, 1, \ldots, p-1$, where p is the number of processes. So, for example, process 1 might send a message to process 0 with the following pseudo-code:

```
char message[100];
. . .
my_rank = Get_rank();
if (my_rank == 1) {
   sprintf(message, "Greetings from process 1");
   Send(message, MSG_CHAR, 100, 0);
} else if (my_rank == 0) {
   Receive(message, MSG_CHAR, 100, 1);
   printf("Process 0 > Received: %s\n", message);
}
```

Here the `Get_rank` function returns the calling process' rank. Then the processes branch depending on their ranks. Process 1 creates a message with `sprintf` from the standard C library and then sends it to process 0 with the call to `Send`. The arguments to the call are, in order, the message, the type of the elements in the message (`MSG_CHAR`), the number of elements in the message (100), and the rank of the destination process (0). On the other hand, process 0 calls `Receive` with the following arguments: the variable into which the message will be received (`message`), the

type of the message elements, the number of elements available for storing the message, and the rank of the process sending the message. After completing the call to `Receive`, process 0 prints the message.

Several points are worth noting here. First note that the program segment is SPMD. The two processes are using the same executable, but carrying out different actions. In this case, what they do depends on their ranks. Second, note that the variable `message` refers to different blocks of memory on the different processes. Programmers often stress this by using variable names such as `my_message` or `local_message`. Finally, note that we're assuming that process 0 can write to `stdout`. This is usually the case: most implementations of message-passing APIs allow all processes access to `stdout` and `stderr`—even if the API doesn't explicitly provide for this. We'll talk a little more about I/O later on.

There are several possibilities for the exact behavior of the `Send` and `Receive` functions, and most message-passing APIs provide several different send and/or receive functions. The simplest behavior is for the call to `Send` to **block** until the call to `Receive` starts receiving the data. This means that the process calling `Send` won't return from the call until the matching call to `Receive` has started. Alternatively, the `Send` function may copy the contents of the message into storage that it owns, and then it will return as soon as the data is copied. The most common behavior for the `Receive` function is for the receiving process to block until the message is received. There are other possibilities for both `Send` and `Receive`, and we'll discuss some of them in Chapter 3.

Typical message-passing APIs also provide a wide variety of additional functions. For example, there may be functions for various "collective" communications, such as a **broadcast**, in which a single process transmits the same data to all the processes, or a **reduction**, in which results computed by the individual processes are combined into a single result—for example, values computed by the processes are added. There may also be special functions for managing processes and communicating complicated data structures. The most widely used API for message-passing is the **Message-Passing Interface** or MPI. We'll take a closer look at it in Chapter 3.

Message-passing is a very powerful and versatile API for developing parallel programs. Virtually all of the programs that are run on the most powerful computers in the world use message-passing. However, it is also very low level. That is, there is a huge amount of detail that the programmer needs to manage. For example, in order to parallelize a serial program, it is usually necessary to rewrite the vast majority of the program. The data structures in the program may have to either be replicated by each process or be explicitly distributed among the processes. Furthermore, the rewriting usually can't be done incrementally. For example, if a data structure is used in many parts of the program, distributing it for the parallel parts and collecting it for the serial (unparallelized) parts will probably be prohibitively expensive. Therefore, message-passing is sometimes called "the assembly language of parallel programming," and there have been many attempts to develop other distributed-memory APIs.

One-sided communication

In message-passing, one process, must call a send function and the send must be matched by another process' call to a receive function. Any communication requires the explicit participation of two processes. In **one-sided communication**, or **remote memory access**, a single process calls a function, which updates either local memory with a value from another process or remote memory with a value from the calling process. This can simplify communication, since it only requires the active participation of a single process. Furthermore, it can significantly reduce the cost of communication by eliminating the overhead associated with synchronizing two processes. It can also reduce overhead by eliminating the overhead of one of the function calls (send or receive).

It should be noted that some of these advantages may be hard to realize in practice. For example, if process 0 is copying a value into the memory of process 1, 0 must have some way of knowing when it's safe to copy, since it will overwrite some memory location. Process 1 must also have some way of knowing when the memory location has been updated. The first problem can be solved by synchronizing the two processes before the copy, and the second problem can be solved by another synchronization or by having a "flag" variable that process 0 sets after it has completed the copy. In the latter case, process 1 may need to **poll** the flag variable in order to determine that the new value is available. That is, it must repeatedly check the flag variable until it gets the value indicating 0 has completed its copy. Clearly, these problems can considerably increase the overhead associated with transmitting a value. A further difficulty is that since there is no explicit interaction between the two processes, remote memory operations can introduce errors that are very hard to track down.

Partitioned global address space languages

Since many programmers find shared-memory programming more appealing than message-passing or one-sided communication, a number of groups are developing parallel programming languages that allow the user to use some shared-memory techniques for programming distributed-memory hardware. This isn't quite as simple as it sounds. If we simply wrote a compiler that treated the collective memories in a distributed-memory system as a single large memory, our programs would have poor, or, at best, unpredictable performance, since each time a running process accessed memory, it might access local memory—that is, memory belonging to the core on which it was executing—or remote memory, memory belonging to another core. Accessing remote memory can take hundreds or even thousands of times longer than accessing local memory. As an example, consider the following pseudo-code for a shared-memory vector addition:

```
shared int n = . . . ;
shared double x[n], y[n];
private int i, my_first_element, my_last_element;
my_first_element = . . . ;
my_last_element = . . . ;
```

```
/* Initialize x and y */
. . .

for (i = my_first_element; i <= my_last_element; i++)
   x[i] += y[i];
```

We first declare two shared arrays. Then, on the basis of the process' rank, we determine which elements of the array "belong" to which process. After initializing the arrays, each process adds its assigned elements. If the assigned elements of `x` and `y` have been allocated so that the elements assigned to each process are in the memory attached to the core the process is running on, then this code should be very fast. However, if, for example, all of `x` is assigned to core 0 and all of `y` is assigned to core 1, then the performance is likely to be terrible, since each time the assignment `x[i] += y[i]` is executed, the process will need to refer to remote memory.

Partitioned global address space, or PGAS, languages provide some of the mechanisms of shared-memory programs. However, they provide the programmer with tools to avoid the problem we just discussed. Private variables are allocated in the local memory of the core on which the process is executing, and the distribution of the data in shared data structures is controlled by the programmer. So, for example, she knows which elements of a shared array are in which process' local memory.

There are a number of research projects currently working on the development of PGAS languages. See, for example, [7, 9, 45].

2.4.5 Programming hybrid systems

Before moving on, we should note that it is possible to program systems such as clusters of multicore processors using a combination of a shared-memory API on the nodes and a distributed-memory API for internode communication. However, this is usually only done for programs that require the highest possible levels of performance, since the complexity of this "hybrid" API makes program development extremely difficult. See, for example, [40]. Rather, such systems are usually programmed using a single, distributed-memory API for both inter- and intra-node communication.

2.5 INPUT AND OUTPUT

We've generally avoided the issue of input and output. There are a couple of reasons. First and foremost, parallel I/O, in which multiple cores access multiple disks or other devices, is a subject to which one could easily devote a book. See, for example, [35]. Second, the vast majority of the programs we'll develop do very little in the way of I/O. The amount of data they read and write is quite small and