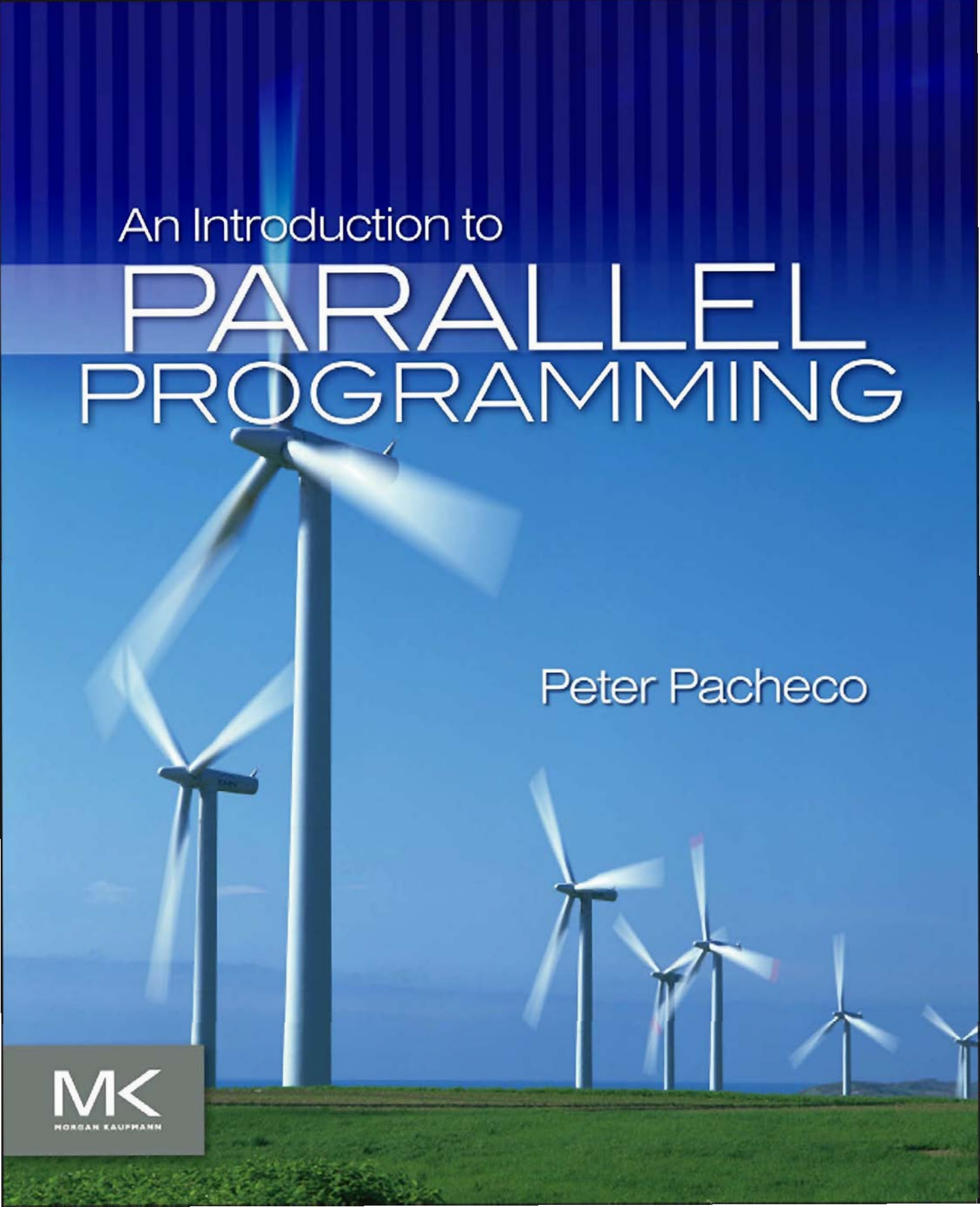
An Introduction to PARALLEL PROGRAMMING



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

Table 4.1 Run-Times (in Seconds) of π Programs Using $n = 10^8$ Terms on a System with Two Four-Core Processors

Threads	Busy-Wait	Mutex
1	2.90	2.90
2	1.45	1.45
4	0.73	0.73
8	0.38	0.38
16	0.50	0.38
32	0.80	0.40
64	3.56	0.38

as each thread only enters the critical section once; so unless the critical section is very long, or the Pthreads functions are very slow, we wouldn't expect the threads to be delayed very much by waiting to enter the critical section. However, if we start increasing the number of threads beyond the number of cores, the performance of the version that uses mutexes remains pretty much unchanged, while the performance of the busy-wait version degrades.

We see that when we use busy-waiting, performance can degrade if there are more threads than cores.[4] This should make sense. For example, suppose we have two cores and five threads. Also suppose that thread 0 is in the critical section, thread 1 is in the busy-wait loop, and threads 2, 3, and 4 have been descheduled by the operating system. After thread 0 completes the critical section and sets `flag = 1`, it will be terminated, and thread 1 can enter the critical section so the operating system can schedule thread 2, thread 3, or thread 4. Suppose it schedules thread 3, which will spin in the `while` loop. When thread 1 finishes the critical section and sets `flag = 2`, the operating system can schedule thread 2 or thread 4. If it schedules thread 4, then both thread 3 and thread 4, will be busily spinning in the busy-wait loop until the operating system deschedules one of them and schedules thread 2. See Table 4.2.

4.7 PRODUCER-CONSUMER SYNCHRONIZATION AND SEMAPHORES

Although busy-waiting is generally wasteful of CPU resources, it has the property by which we know, in advance, the order in which the threads will execute the code in the critical section: thread 0 is first, then thread 1, then thread 2, and so on. With

[4]These are typical run-times. When using busy-waiting and the number of threads is greater than the number of cores, the run-times vary considerably.

Table 4.2 Possible Sequence of Events with Busy-Waiting and More Threads than Cores

		Thread				
Time	`flag`	**0**	**1**	**2**	**3**	**4**
0	0	crit sect	busy-wait	susp	susp	susp
1	1	terminate	crit sect	susp	busy-wait	susp
2	2	—	terminate	susp	busy-wait	busy-wait
⋮	⋮			⋮	⋮	⋮
?	2	—	—	crit sect	susp	busy-wait

mutexes, the order in which the threads execute the critical section is left to chance and the system. Since addition is commutative, this doesn't matter in our program for estimating π. However, it's not difficult to think of situations in which we also want to control the order in which the threads execute the code in the critical section. For example, suppose each thread generates an $n \times n$ matrix, and we want to multiply the matrices together in thread-rank order. Since matrix multiplication isn't commutative, our mutex solution would have problems:

```
/* n and product_matrix are shared and initialized by the main
thread */
/* product_matrix is initialized to be the identity matrix  */
void* Thread_work(void* rank) {
   long my_rank = (long) rank;
   matrix_t my_mat = Allocate_matrix(n);
   Generate_matrix(my_mat);
   pthread_mutex_lock(&mutex);
   Multiply_matrix(product_mat, my_mat);
   pthread_mutex_unlock(&mutex);
   Free_matrix(&my_mat);
   return NULL;
}  /* Thread_work */
```

A somewhat more complicated example involves having each thread "send a message" to another thread. For example, suppose we have `thread_count` or t threads and we want thread 0 to send a message to thread 1, thread 1 to send a message to thread 2, ..., thread $t-2$ to send a message to thread $t-1$ and thread $t-1$ to send a message to thread 0. After a thread "receives" a message, it can print the message and terminate. In order to implement the message transfer, we can allocate a shared array of `char*`. Then each thread can allocate storage for the message it's sending, and, after it has initialized the message, set a pointer in the shared array to refer to it. In order to avoid dereferencing undefined pointers, the main thread can set the individual entries in the shared array to `NULL`. See Program 4.7. When we run the program with more than a couple of threads on a dual-core system, we see that some of the messages are never received. For example, thread 0, which is started first,

```
1  /* messages has type char**. It's allocated in main. */
2  /* Each entry is set to NULL in main.                */
3  void* Send_msg(void* rank) {
4     long my_rank = (long) rank;
5     long dest = (my_rank + 1) % thread_count;
6     long source = (my_rank + thread_count - 1) % thread_count;
7     char* my_msg = malloc(MSG_MAX*sizeof(char));
8
9     sprintf(my_msg, "Hello to %ld from %ld", dest, my_rank);
10    messages[dest] = my_msg;
11
12    if (messages[my_rank] != NULL)
13       printf("Thread %ld > %s\n", my_rank, messages[my_rank]);
14    else
15       printf("Thread %ld > No message from %ld\n", my_rank,
               source);
16
17    return NULL;
18 }  /* Send_msg */
```

Program 4.7: A first attempt at sending messages using Pthreads

will typically finish before thread $t-1$ has copied the message into the `messages` array. This isn't surprising, and we could fix the problem by replacing the `if` statement in Line 12 with a busy-wait `while` statement:

```
while (messages[my_rank] == NULL);
printf("Thread %ld > %s\n", my_rank, messages[my_rank]);
```

Of course, this solution would have the same problems that any busy-waiting solution has, so we'd prefer a different approach.

After executing the assignment in Line 10, we'd like to "notify" the thread with rank `dest` that it can proceed to print the message. We'd like to do something like this:

```
. . .
messages[dest] = my_msg;
Notify thread dest that it can proceed;

Await notification from thread source
printf("Thread %ld > %s\n", my_rank, messages[my_rank]);
. . .
```

It's not at all clear how mutexes can be of help here. We might try calling `pthread_mutex_unlock` to "notify" the thread with rank `dest`. However, mutexes are initialized to be *unlocked*, so we'd need to add a call *before* initializing `messages[dest]` to lock the mutex. This will be a problem since we don't know when the threads will reach the calls to `pthread_mutex_lock`.

To make this a little clearer, suppose that the main thread creates and initializes an array of mutexes, one for each thread. Then, we're trying to do something like this:

```
1  . . .
2  pthread_mutex_lock(mutex[dest]);
3  . . .
4  messages[dest] = my_msg;
5  pthread_mutex_unlock(mutex[dest]);
6  . . .
7  pthread_mutex_lock(mutex[my_rank]);
8  printf("Thread %ld > %s\n", my_rank, messages[my_rank]);
9  . . .
```

Now suppose we have two threads, and thread 0 gets so far ahead of thread 1 that it reaches the second call to `pthread_mutex_lock` in Line 7 before thread 1 reaches the first in Line 2. Then, of course, it will acquire the lock and continue to the `printf` statement. This will result in thread 0's dereferencing a null pointer, and it will crash.

There *are* other approaches to solving this problem with mutexes. See, for example, Exercise 4.7. However, POSIX also provides a somewhat different means of controlling access to critical sections: **semaphores**. Let's take a look at them.

Semaphores can be thought of as a special type of `unsigned int`, so they can take on the values 0, 1, 2, ... In most cases, we'll only be interested in using them when they take on the values 0 and 1. A semaphore that only takes on these values is called a *binary* semaphore. Very roughly speaking, 0 corresponds to a locked mutex, and 1 corresponds to an unlocked mutex. To use a binary semaphore as a mutex, you *initialize* it to 1—that is, it's "unlocked." Before the critical section you want to protect, you place a call to the function `sem_wait`. A thread that executes `sem_wait` will block if the semaphore is 0. If the semaphore is nonzero, it will *decrement* the semaphore and proceed. After executing the code in the critical section, a thread calls `sem_post`, which *increments* the semaphore, and a thread waiting in `sem_wait` can proceed.

Semaphores were first defined by the computer scientist Edsger Dijkstra in [13]. The name is taken from the mechanical device that railroads use to control which train can use a track. The device consists of an arm attached by a pivot to a post. When the arm points down, approaching trains can proceed, and when the arm is perpendicular to the post, approaching trains must stop and wait. The track corresponds to the critical section: when the arm is down corresponds to a semaphore of 1, and when the arm is up corresponds to a semaphore of 0. The `sem_wait` and `sem_post` calls correspond to signals sent by the train to the semaphore controller.

For our current purposes, the crucial difference between semaphores and mutexes is that there is no ownership associated with a semaphore. The main thread can initialize all of the semaphores to 0—that is, "locked," and then any thread can execute a `sem_post` on any of the semaphores, and, similarly, any thread can execute `sem_wait` on any of the semaphores. Thus, if we use semaphores, our `Send_msg` function can be written as shown in Program 4.8.

```
1  /* messages is allocated and initialized to NULL in main   */
2  /* semaphores is allocated and initialized to 0 (locked) in
        main */
3  void* Send_msg(void* rank) {
4     long my_rank = (long) rank;
5     long dest = (my_rank + 1) % thread_count;
6     char* my_msg = malloc(MSG_MAX*sizeof(char));
7
8     sprintf(my_msg, "Hello to %ld from %ld", dest, my_rank);
9     messages[dest] = my_msg;
10    sem_post(&semaphores[dest])
            /* ''Unlock'' the semaphore of dest */
11
12    /* Wait for our semaphore to be unlocked */
13    sem_wait(&semaphores[my_rank]);
14    printf("Thread %ld > %s\n", my_rank, messages[my_rank]);
15
16    return NULL;
17 }  /* Send_msg */
```

Program 4.8: Using semaphores so that threads can send messages

The syntax of the various semaphore functions is

```
int sem_init(
      sem_t*    semaphore_p   /* out */,
      int       shared        /* in  */,
      unsigned  initial_val   /* in  */);

int sem_destroy(sem_t*  semaphore_p  /* in/out */);
int sem_post(sem_t*     semaphore_p  /* in/out */);
int sem_wait(sem_t*     semaphore_p  /* in/out */);
```

We won't make use of the second argument to `sem_init`: the constant 0 can be passed in. Note that semaphores are *not* part of Pthreads. Hence, it's necessary to add the following preprocessor directive to any program that uses them:[5]

```
#include <semaphore.h>
```

Finally, note that the message-sending problem didn't involve a critical section. The problem wasn't that there was a block of code that could only be executed by one thread at a time. Rather, thread `my_rank` couldn't proceed until thread `source` had finished creating the message. This type of synchronization, when a thread can't

[5]Some systems (e.g., some versions of Mac OS X) don't support this version of semaphores. They support something called "named" semaphores. The functions `sem_wait` and `sem_post` can be used in the same way. However, `sem_init` should be replaced by `sem_open`, and `sem_destroy` should be replaced by `sem_close` and `sem_unlink`. See the book's website for an example.

proceed until another thread has taken some action, is sometimes called **producer-consumer synchronization**.

4.8 BARRIERS AND CONDITION VARIABLES

Let's take a look at another problem in shared-memory programming: synchronizing the threads by making sure that they all are at the same point in a program. Such a point of synchronization is called a **barrier** because no thread can proceed beyond the barrier until all the threads have reached it.

Barriers have numerous applications. As we discussed in Chapter 2, if we're timing some part of a multithreaded program, we'd like for all the threads to start the timed code at the same instant, and then report the time taken by the last thread to finish, that is, the "slowest" thread. We'd therefore like to do something like this:

```
/* Shared */
double elapsed_time;
. . .
/* Private */
double my_start, my_finish, my_elapsed;
. . .
Synchronize threads;
Store current time in my_start;
/* Execute timed code */
. . .
Store current time in my_finish;
my_elapsed = my_finish - my_start;

elapsed = Maximum of my_elapsed values;
```

Using this approach, we're sure that all of the threads will record `my_start` at approximately the same time.

Another very important use of barriers is in debugging. As you've probably already seen, it can be very difficult to determine *where* an error is occuring in a parallel program. We can, of course, have each thread print a message indicating which point it's reached in the program, but it doesn't take long for the volume of the output to become overwhelming. Barriers provide an alternative:

```
point in program we want to reach;
barrier;
if (my_rank == 0) {
   printf("All threads reached this point\n");
   fflush(stdout);
}
```

Many implementations of Pthreads don't provide barriers, so if our code is to be portable, we need to develop our own implementation. There are a number of options; we'll look at three. The first two only use constructs that we've already studied. The third uses a new type of Pthreads object: a *condition variable.*