An Introduction to Parallel Programming Solutions, Chapter 4

Krichaporn Srisupapak and Peter Pacheco

May 22, 2011

1. The code will be unchanged if the number of columns isn't evenly divisible by the number of threads. However, if the number of rows isn't evenly divisible by the number of threads we need to decide who will be responsible for the rows left over after integer division of m, the number of rows, by t, the number of threads.

   For example, if $m = 10$ and $t = 4$, then there are two "extra" rows, since `10 % 4 = 2`. Clearly, there are a number of possibilities here. One is to give the first two threads each an extra row. So we'd assign three rows to threads 0 and 1 and two rows to threads 2 and 3.

   Here's an algorithm that implements this:

```
int quotient = m/t;  /* Every thread gets at least m/t rows */
int remainder = m % t;
if (my_rank < remainder) { /* I get m/t + 1 rows */
   local_m = quotient + 1;
   my_first_row = my_rank*local_m;
   my_last_row = my_first_row + local_m - 1;
} else {  /* I get m/t rows */
   local_m = quotient;
   /* Each of the threads 0, 1, . . . , remainder - 1 gets   */
   /* an extra row.  So add in these rows to get my_firs_row */
   my_first_row = my_rank*local_m + remainder;
   my_last_row = my_first_row + local_m - 1;
}
```

2. See `ex4.2_pth_mat_vect.c`. We scheduled the input and output using semaphores. In most cases this version — with A and y distributed — is somewhat faster.

3. - How does the result of the multithreaded calculation compare to the single-threaded calculation with optimization turned off (`-O0`)?

     The single-threaded and the multithreaded estimates of π are identical to about 14 decimals.

     The ratio of the run-time of the single-threaded program to the multithreaded program is equal to the number of threads, as long as the number of threads is no greater than the number of cores.

   - What happens if you try running the program with optimization `O2`?

     The program hangs if it's run with more than one thread.

   - Which variables should be made volatile in the π calculation?

     `flag` and `sum`

   - Change these variables and rerun the program with and without optimization. How do the results compare to the single-threaded program?

     The results are the same with and without optimization.

4. This isn't entirely clear. It might suggest that threads that are descheduled while they're in calls to `pthread_mutex_lock` are not preventing other threads from accessing the critical section. In other words, threads that are descheduled don't seem to be acquiring the mutex — which would prevent other threads from acquiring it until the descheduled thread had been rescheduled.

5. See `ex4.5_pth_pi_mutex.c` for the modified program.

   The performance of the busy-wait version is roughly comparable to this version. On some systems the busy-wait version is slightly faster. On others it's slightly slower.

   In both versions the threads are probably spending much of the execution time waiting to enter the critical section.

6. See `ex4.6_pth_pi_mutex.c`.

   The two programs have similar run-times.

7. For an implementation that uses two threads see `ex4.7_pth_producer_consumer.c`.

   For an implementation that uses an even number of threads with even threads producing and odd threads consuming, see `ex4.7_pth_pc_odd_even.c`.

   For an implementation in which each threads both produces and consumes see `ex4.7_pth_pc_both.c`.

   These programs do use busy-waiting. Until a message becomes available, the consumer threads will repeatedly acquire the mutex, check for a message, and relinquish the mutex.

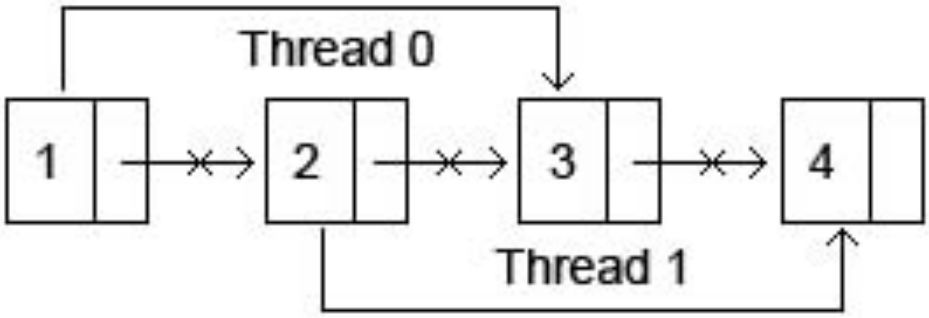

Figure 1: Two deletes that might cause problems

8. (a) The threads will deadlock: thread 0 will be waiting to acquire `mut1` and thread 1 will be waiting to acquire `mut0`, but since thread 0 is waiting for `mut1` it can't relinquish `mut0`, and since thread 1 is waiting to acquire `mut0`, it can't relinquish `mut1`.

   (b) Yes, there would still be a problem. Each thread would be waiting for the other to change a flag variable.

   (c) Yes, there would still be a problem. Both threads would be blocked in calls to `sem_wait`, so neither thread could call `sem_post`.

9. For a program that uses Pthreads barriers see `ex4.9_pth_bar.c`.

   On our systems (AMD and Intel processors running Linux), the barrier implemented with semaphores is always the fastest. The relative speed of the other three depends on the particular systems.

10. The program `ex4.10_pth_trap_time.c` is a modified version of the solution to Programming Assignment 4.3, which implements the trapezoidal rule. The barrier is implemented using a Pthreads barrier, and the maximum elapsed time is found using a mutex.

11. (a) Figure 1 shows two deletes that, when executed simultaneously, might cause problems.

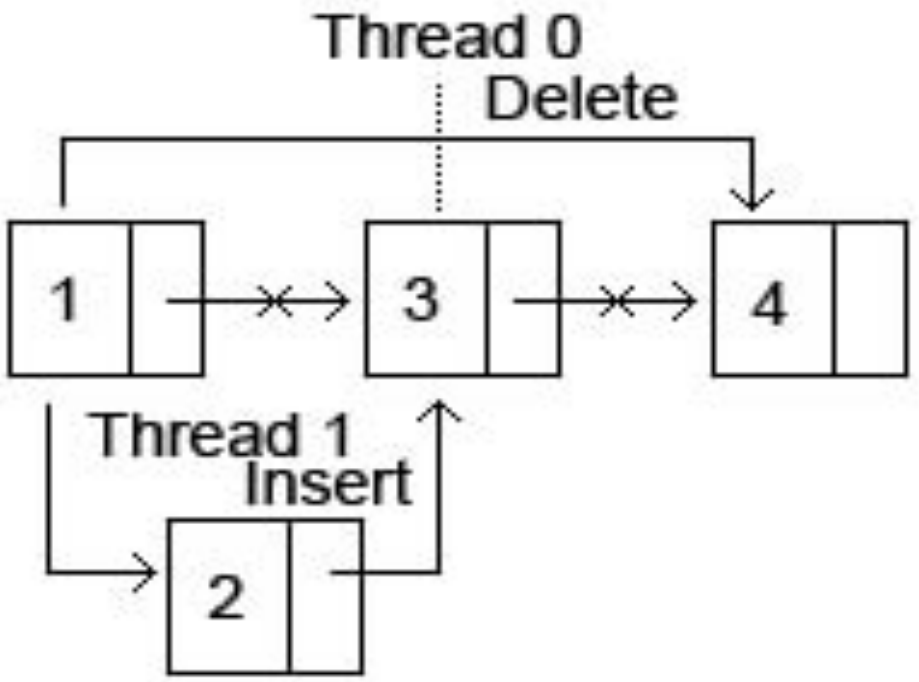

Figure 2: An insert and a delete that might cause problems

(b) Figure 2 shows an insert and a delete that, when executed simultaneously, might cause problems.

(c) If Thread 0 is executing `Member` while Thread 1 is deleting a node, several problems can occur. For example, Thread 0 may return `true` for a value that is no longer in the list because Thread 1 deleted it between the time Thread 0 found it and the time Thread 0 returned. As another example, suppose Thread 0 is visiting a node while Thread 1 is deleting it, and that Thread 0 tries to advance to the next node after Thread 1 has finished the deletion. Then the pointer that Thread 0 dereferences may cause a segmentation violation.

(d) Figure 3 shows two inserts that, when executed simultaneously, might cause problems.

(e) If Thread 0 is executing `Member` while Thread 1 is trying to insert a new node,

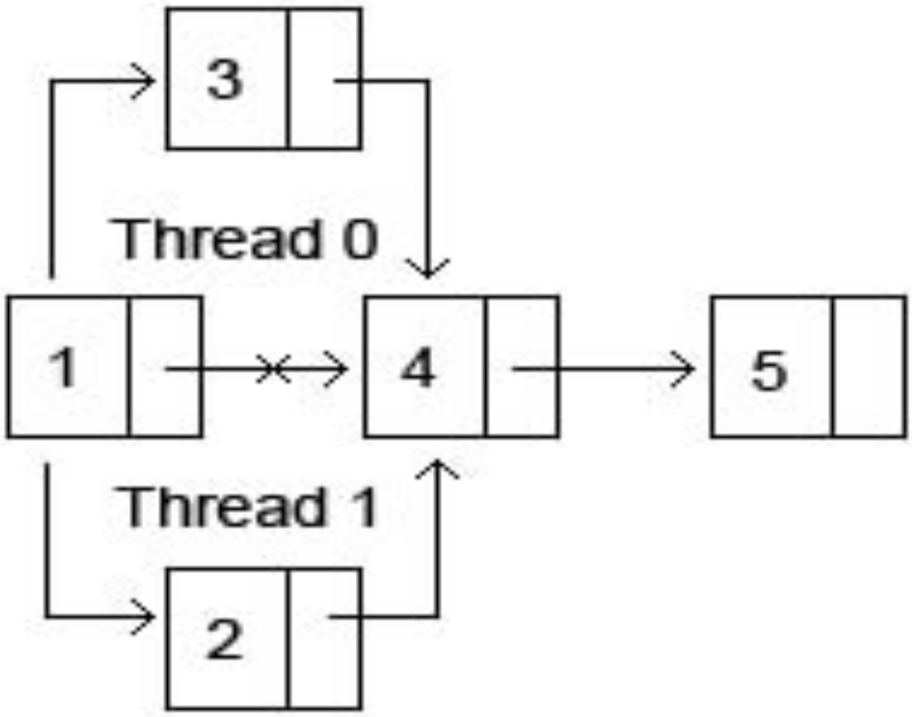

Figure 3: Two inserts that might cause problems

there are several problems that can occur. For example, thread 0 could erroneously return `false`, if thread 1 inserts the searched for value between the time that Thread 0 determines the value isn't in the list and the time it returns.

12. This could be a problem. For example, suppose Thread 0 wants to delete a node from the list. Then it first searches the list with a read-lock. If it finds the node to be deleted, it will need to obtain a write-lock, but in the time that elapses between relinquishing the read-lock and obtaining the write-lock, the state of the list could be changed by another thread. For example, the predecessor of the node to be deleted could be deleted, and the "information" that Thread 0 has about deleting (a pointer to the predecessor node) would no longer be valid.

13. The following table shows the run-times (in seconds) of the three linked-list programs when the initial list has 1000 nodes, and we carry out 100,000 ops. 99.9% of the ops are searches. The times in the "I" column were taken when the remaining 0.1% were insertions, the times in the "D" column were taken when the remaining ops were deletions.

	Number of Threads							
	1		2		4		8	
Implementation	I	D	I	D	I	D	I	D
Read-write locks	0.22	0.21	0.13	0.12	0.099	0.099	0.11	0.11
One mutex	0.22	0.21	0.45	0.42	0.39	0.37	0.46	0.42
One mutex/node	1.8	1.7	6.0	5.6	3.5	3.5	2.7	2.6

The following table shows the run-times of the three linked-list programs when the initial list has 1000 nodes, and we carry out 100,000 ops, 80% of which are searches and 20% are insertions. (We didn't carry out 80% searches and 20% deletions, since there would be about 20,000 deletions and the vast majority of them would be from an empty list.)

	Number of Threads			
Implementation	1	2	4	8
Read-write locks	4.6	8.9	10	10
One mutex	4.8	11	10	11
One mutex/node	35	56	28	19

The run-times with 99.9% searches are roughly comparable to the run-times shown in Table 4.3. When there are 80% searches and 20% insertions, the lists can grow to more than 20 times their original size of 1000. So, not surprisingly, the run-times with 80% searches are *much* greater than the run-times in Table 4.4.

The data with 99.9% searches might seem to indicate that the cost of insertion is slightly greater than the cost of deletion. However, since the average list size for the insertions is slightly greater than the average list size for the deletions, it seems likely that the cost of the two operations is nearly the same.

14. Here's code for the matrix-vector multiplication using a one-dimensional array:

```
void *Pth_mat_vect(void* rank) {
   long my_rank = (long) rank;
   int i, j;
   int local_m = m/thread_count;
   int my_first_row = my_rank*local_m;
   int my_last_row = (my_rank+1)*local_m - 1;

   for (i = my_first_row; i <= my_last_row; i++) {
      y[i] = 0.0;
      for (j = 0; j < n; j++) {
        //y[i] += A[i][j]*x[j];
          y[i] += A[i*n+j]*x[j];
      }
   }

   return NULL;
}  /* Pth_mat_vect */
```

There is little, if any, difference in the run-times of the two versions.

15. **Notes:**

- We've only looked at data cache misses. There are differences in the total number of instructions executed, but the number of instruction cache misses is relatively small and about the same for all three inputs.
- The initialization of A and x will substantially increase the number of misses. It will also make it difficult to determine the state of the cache when the matrix-vector multiplication begins. There are a number of possible solutions to this. We chose the simplest: we omitted initialization and used the default values assigned by the system (0).

We chose $k = 8$. So the orders of the three matrices are 8×8 million, 8000×8000, and 8 million $\times$ 8, respectively. All of the data were taken with one thread. The following table shows the number of *data* cache misses. M = million(s), K = thousand(s). The

numbers in parentheses are percentages of the total number of data-reads or writes. The system on which these data were collected has a 64 Kbyte L1 data-cache and a 1024 Kbyte L2 cache.

Matrix	Data Cache Misses							
	L1-Write		L2-Write		L1-Read		L2-Read	
$8 \times 8M$	16K	(0.0)	65	(0.0)	16M	(12.5)	16M	(12.5)
$8K \times 8K$	2K	(0.0)	1K	(0.0)	16M	(12.5)	8M	(6.2)
$8M \times 8$	1M	(1.4)	1M	(1.4)	8M	(6.2)	8M	(6.2)

c. The largest number of write-misses occur with the $8M \times 8$ system. This makes sense, since for this system the array y has order 8,000,000, while for the other two systems, it has order 8000 and 8, respectively. Furthermore, among the variables A, x, and y, y is the only one that is written by the matrix-vector multiplication code.

f. The largest number of read-misses occur with the $8 \times 8M$ system. This also makes sense. Each element of the array A will be read exactly once with all three inputs, and for each input A has 64,000,000 entries. Also, the updates `y[i] += . . .` will be executed exactly 64,000,000 times for each set of input, and these are unlikely to cause read-misses since before the inner loop is executed, `y[i]` is initialized, and hence is probably already in cache. On the other hand, the reads of `x[j]` may cause cache misses, and in the $8 \times 8M$ system x has 8,000,000 entries as opposed to only 8000 and 8 for the other two systems.

g. From the one-thread run-times given in Table 4.5 on page 192 we see that the program is slowest with the $8 \times 8M$ input and fastest with the $8K \times 8K$ input. Read-misses tend to be more expensive than write-misses. When a program needs data to carry out a computation, it must either try executing another computation or wait for the data. On the other hand, the data created by a write can often be simply queued up and the computation can proceed. So it's not surprising that the program is slowest with the data that results in the most read misses.

On the other hand the program with the $8M \times 8$ input has vastly more write misses than the program with the $8K \times 8K$ data. Furthermore the number of L2 read-misses for the two programs is identical. They do differ in the number of L1 read-misses, but these are substantially less expensive than L2 read-misses. So it's not surprising that the program is fastest with the $8K \times 8K$ input.

16. With 8000 elements `y` will be partitioned as follows

```
Thread 0:     y[0],    y[1], . . . , y[1999]
```

```
Thread 1:  y[2000], y[2001], . . . , y[3999]
Thread 2:  y[4000], y[4001], . . . , y[5999]
Thread 3:  y[6000], y[6001], . . . , y[7999]
```

In order for false-sharing to occur between thread 0 and thread 2, there must be elements of y that belong to the same cache line, but are assigned to different threads. On thread 0, the cache line that's "closest" to the elements assigned to thread 2 is the line that contains y[1999]. But even if this is the first element of the cache line, the highest possible index for an element of y that belongs to this line is 2006:

y[1999]	y[2000]	y[2001]	y[2002]	y[2003]	y[2004]	y[2005]	y[2006]

Since the least index of an element of y assigned to thread 2 is 4000, there can't possibly be a cache line that has elements belonging to both thread 0 and thread 2. Similar reasoning applies to threads 0 and 3.

17. If we look at the location of y[0] in the first cache line containing all or part of y we see that y can be distributed across cache lines in eight different ways. If y[0] is the first element of the cache line, then we'll have the following assignment of y to cache lines:

first line	y[0]	y[1]	y[2]	y[3]	y[4]	y[5]	y[6]	y[7]

If y[0] is the second element of the cache line, then we'll have the following assignment:

first line	—	y[0]	y[1]	y[2]	y[3]	y[4]	y[5]	y[6]
second line	y[7]	—	—	—	—	—	—	—

As a final example, if y[0] is the last element of the first line, then we'll have the following assignment

first line	—	—	—	—	—	—	—	y[0]
second line	y[1]	y[2]	y[3]	y[4]	y[5]	y[6]	y[7]	—

(a) From our first example, we see it's possible for y to fit into a single cache line.

(b) However, in most cases, y will be split across two cache lines.

(c) There are eight ways the doubles can be assigned: the eight ways correspond to the eight different possible locations for y[0] in the first line.

(d) We can choose two of the threads and assign them to one of the processors: 0 and 1, or 0 and 2, or 0 and 3. Note that this covers all possibilities. For example, choosing 2 and 3 and assigning them to one processor is the same as choosing 0 and 1. So there are three possible assignments of threads to processors.

(e) Yes. Suppose threads 0 and 1 share one processor and threads 2 and 3 share another. Then if `y[0], y[1], y[2],` and `y[3]` are in one cache line and `y[4], y[5], y[6],` and `y[7]` are in another, any write by thread 0 or thread 1 won't invalidate the line storing the data in the cache line of threads 2 and 3. Similarly, writes by 2 and 3 won't invalidate the data in the cache of 0 and 1.

(f) For each of the 3 assignments of threads to processors, there are 8 possible assignments of `y` to cache lines. So we get a total of 24.

(g) Note first that if the execution of the threads is serialized (e.g., first thread 0 runs, then thread 1 runs, etc.), there may not be any false sharing. So we'll assume that all four threads are running simultaneously.

If 0 and 1 are assigned to different processors, then any assignment of `y` that puts `y[1]` and `y[2]` in the same cache line will cause false sharing between 0 and 1. The only way this can fail to happen is if `y[0]` and `y[1]` are in one line and the remainder of `y` is another. But when this happens threads 2 and 3 will be on different processors, and hence there will be false sharing between them.

So 0 and 1 must be assigned to the same processor. Furthermore, if any component of `y` with subscript > 3 is assigned to this processor or if any component with subscript < 4 is assigned to the other processor, there will be false sharing. So the assignment from the previous part is the only one that doesn't result in false sharing.

18. (a) See the file `ex4.18_pth_mat_vec_pad.c`

(b) See the file `ex4.18_pth_mat_vec_private.c`

Note: A third possibility is to simply use a private scalar variable and write to this variable during the inner `for` loop. After the inner for loop is completed, the appropriate element of `y` can be assigned its final value. See file `ex4.18_pth_mat_vec_scalar.c`

(c) The following table shows the run-times (in seconds) of the versions with different input matrices. "Orig" denotes the original implementation; "Pad" is the implementation that pads `y` with extra storage; "Priv Vect" is the implementation that has each thread allocate a private copy of its local part of `y`; and "Priv Scal" is the implementation that has each thread use a private scalar to store the contents of a component of `y` during the computation. In most cases, the private vector implementation performs the worst. No doubt this is due to the costs of

allocating and freeing the temporary storage, and copying the temporary storage into `y`. On the other hand, using a private scalar does as well or better than the other implementations in every case. In particular, it seems to do the best job in avoiding problems with false sharing when the matrix has order $8 \times 8,000,000$.

Threads	Impl	Matrix Order		
		8M × 8	8K × 8K	8 × 8M
1	Orig	0.40	0.36	0.44
	Pad	0.40	0.36	0.44
	Priv Vect	0.49	0.36	0.43
	Priv Scal	0.38	0.33	0.43
2	Orig	0.22	0.19	0.29
	Pad	0.22	0.19	0.26
	Priv Vect	0.28	0.19	0.31
	Priv Scal	0.21	0.18	0.22
4	Orig	0.14	0.12	0.38
	Pad	0.14	0.12	0.24
	Priv Vect	0.19	0.12	0.25
	Priv Scal	0.14	0.12	0.24

19. See the file `ex4.19_pth_tokenize_r.c`