An Introduction to Parallel Programming Solutions, Chapter 5

Krichaporn Srisupapak and Peter Pacheco

June 21, 2011

1. The value of `_OPENMP` is a date having the form *yyyymm,* where *yyyy* is a 4-digit year and *mm* is a 2-digit month. For example, 200505. The OpenMP standard states that when the macro is defined, it will be the year and month of the version of the OpenMP standard that has been implemented. See `ex5.1_omp_macro.c`.

2. The answer here will depend on the system, the number of trapezoids, and the number of threads. On one of our systems, with 1000 trapezoids, we didn't start seeing nondeterministic results until the number of threads was 250, while for another system, we saw nondeterministic results with 100 threads. With only 100 trapezoids, the first system produced unpredictable results with 50 threads, while the second produced unpredictable results with 10. See `ex5.2_omp_trap1_no_crit.c`

3. See `ex5.3_omp_trap1_slow.c` for the version of `omp_trap1.c` that puts the call to `Local_trap` in a critical section. Note that in order to avoid problems with two critical sections (one for the trapezoidal rule result and one for the maximum elapsed time) we simply put the calls to `omp_get_wtime` outside the parallel block.

   When this version is run on one of our systems with 1 thread and $n = 10^6$, the run-time is 2.7 milliseconds, while with 2 threads its run-time is 2.8 milliseconds.

   With the same input both `omp_trap2a_time.c` and `omp_trap2b_time.c` take 2.7 millisecond with 1 thread and 1.4 milliseconds with 2 threads.

   The version that puts the call to `Local_trap` in a critical section forces the threads to execute their parts of the trapezoidal rule sequentially: first one thread executes `Local_trap`, then the other. On the other hand, both of the other two versions allow the threads to do most of their work simultaneously, and they obtain much better performance.

4. The following table shows the identity values for the various operators:

Operator	Identity Value
`&&`	1
`||`	0
`&`	$11\ldots11_2$
`|`	0
`^`	0

5. Recall that floating point numbers are stored in the computer in something that's similar to scientific notation. So it's convenient to think of the array `a` as

   ```
   a[] = {2.00e+00, 2.00e+00, 4.00e+00, 1.00e+03}
   ```

   When a value is stored in a register, we can add an additional digit. For example, when `a[0]` is loaded into a register, we can think of it as `2.000e+00`.

   (a) When the values are added using the serial `for` loop, the value stored in `sum` will be

   ```
   After i = 0:  sum = 2.00e+00
   After i = 1:  sum = 4.00e+00
   After i = 2:  sum = 8.00e+00
   ```

   When `i = 3`, the value `1.008e+03` will be stored in a register, and when it's stored in main memory it will be rounded to `1.01e+00`. So the output will be

   ```
   sum = 1010.0
   ```

   (b) When the values are added using the parallel `for`, recall that the run-time system will create a private variable for each thread. This private variable will be used to store that thread's partial sum. Let's call these private variables `local_sum0` on thread 0 and `local_sum1` on thread 1. Then after thread 0 has completed its iterations, `local_sum0 = 4.00e+00`. After thread 1 has completed its iterations `local_sum1 = 1.00e+03` since the register sum `1.004e+03` will be rounded down. Now when the two private variables are added the value stored in the register will be `1.004e+03` and when it's stored in main memory, we'll have `sum = 1.00e+03`. So the output of the code will be

   ```
   sum = 1000.0
   ```

6. See `ex5.6_omp_schedule.c`.

7. When the program is run with one thread, the `parallel for` directive has no effect, and the program is effectively the same as the preceding serial program. In particular, there's no loop-carried dependence, since there's only one thread.

8. Observe that

```
a[0] = 0
a[1] = a[0] + 1 = 0 + 1
a[2] = a[1] + 2 = 0 + 1 + 2
a[3] = a[2] + 3 = 0 + 1 + 2 + 3
a[4] = a[3] + 4 = 0 + 1 + 2 + 3 + 4
```

   etc. In general, then

$$\mathtt{a}[i] = \sum_{j=0}^{i} j.$$

   But

$$\sum_{j=0}^{i} j = \frac{i(i+1)}{2}.$$

   So we can rewrite the code as

```
for (i = 0; i < n; i++)
   a[i] = i*(i+1)/2;
```

   In this loop, the result of any iteration isn't used again. So the code can be parallelized with a `parallel for` directive:

```
#  pragma omp parallel for num_threads(thread_count) \
       default(none) private(i) shared(a, n)
   for (i = 0; i < n; i++)
      a[i] = i*(i+1)/2;
```

9. See the source file `ex5.9_omp_trap3_schedule.c`.

   On our system if no `schedule` clause is used, the default schedule is approximately a block partition. However, the run-time system seems to invariably assign slightly more iterations to thread 0 and slightly fewer to thread `thread_count-1`.

   If the `schedule` clause is included, but the environment variable `OMP_SCHEDULE` isn't defined, then the assignment of iterations to threads seems to be dynamic: the exact assignment changes from run to run.

If the `schedule` clause is included and `OMP_SCHEDULE` is defined to be `guided`, then roughly $n/$`thread_count` iterations are assigned to one thread, and successive blocks of consecutive iterations are roughly half the size of the previous block. In this case also, the exact assignment changes from run to run.

10. See the file `ex5.10_omp_atomic.c`.

    Our implementation doesn't enforce exclusive access across all atomic regions: updates in independent atomic regions can occur simultaneously on different threads. This behavior isn't guaranteed by the standard, however. Also if two different atomic regions update the same variable, then the OpenMP standard requires that a thread executing either update must be given exclusive access to the variable. In our example, if the private variable `my_sum` is replaced by a shared variable `sum`, then only one thread at at time will execute the update to `sum`.

11. The following code will do the job:

```
#  pragma omp parallel for num_thread(thread_count) \
      default(none) private(i, j) shared(A, x, y, m, n)
   for (i = 0; i < m; 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];
   }
```

12. **Notes:**

    - Weve 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 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)	570	(0.0)	16M	(12.5)	16M	(12.4)
$8K \times 8K$	4K	(0.0)	2K	(0.0)	16M	(12.5)	8M	(6.2)
$8M \times 8$	1M	(1.4)	1M	(1.3)	8M	(5.2)	8M	(5.2)

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 $8 \times 8M$ ystem. 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. The table on page 253 has the following run-times (in seconds) when the program is run with one thread on one of our systems:

Threads	$8 \times 8M$	$8K \times 8K$	$8M \times 8$
1	0.33	0.26	0.32

So we see that the program is slowest (although not by much) 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 its 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 its not surprising that the program is fastest with the $8K \times 8K$ input.

13. With 8000 elements y will be partitioned (approximately) 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 thats "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 cant possibly be a cache line that has elements belonging to both thread 0 and thread 2. Similar reasoning applies to threads 0 and 3.

14. 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.

15. (a) See the file `ex5.15_omp_mat_vect_pad.c`.

(b) See the file `ex5.15_omp_mat_vect_private.c`. Also see the file `ex5.15_omp_mat_vect_scalar.c` for an implementation that uses a temporary private scalar instead of a subvector.

(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 almost every case. In particular, it seems to do the best job in avoiding problems with false sharing when the matrix has order $8 \times 8M$.

Threads	Impl	Matrix Order		
		8M × 8	8K × 8K	8 × 8M
1	Orig	0.32	0.26	0.33
	Pad	0.33	0.26	0.33
	Priv Vect	0.42	0.26	0.33
	Priv Scal	0.30	0.25	0.32
2	Orig	0.22	0.19	0.30
	Pad	0.22	0.19	0.26
	Priv Vect	0.28	0.19	0.26
	Priv Scal	0.21	0.18	0.26
4	Orig	0.14	0.12	0.30
	Pad	0.14	0.12	0.23
	Priv Vect	0.18	0.12	0.25
	Priv Scal	0.14	0.12	0.24

16. See the file `ex5.16_omp_tokenize_r.c`.