5.12 EXERCISES

5.1. If it’s defined, the _OPENMP macro is a decimal int. Write a program that prints its value. What is the significance of the value?

5.2. Download omp_trap_1.c from the book’s website, and delete the critical directive. Now compile and run the program with more and more threads and larger and larger values of n. How many threads and how many trapezoids does it take before the result is incorrect?

5.3. Modify omp_trap_1.c so that
   a. it uses the first block of code on page 222, and
   b. the time used by the parallel block is timed using the OpenMP function omp_get_wtime(). The syntax is

   ```
   double omp_get_wtime(void)
   ```

   It returns the number of seconds that have passed since some time in the past. For details on taking timings, see Section 2.6.4. Also recall that OpenMP has a barrier directive:

   ```
   # pragma omp barrier
   ```

   Now find a system with at least two cores and time the program with
   c. one thread and a large value of n, and
   d. two threads and the same value of n.

5.4. Recall that OpenMP creates private variables for reduction variables, and these private variables are initialized to the identity value for the reduction operator. For example, if the operator is addition, the private variables are initialized to 0, while if the operator is multiplication, the private variables are initialized to 1. What are the identity values for the operators & & , || , & , | , ^ ?

5.5. Suppose that on the amazing Bleeblon computer, variables with type float can store three decimal digits. Also suppose that the Bleeblon’s floating point registers can store four decimal digits, and that after any floating point operation, the result is rounded to three decimal digits before being stored. Now suppose a C program declares an array a as follows:

   ```
   float a[] = {4.0, 3.0, 3.0, 1000.0};
   ```

   a. What is the output of the following block of code if it’s run on the Bleeblon?

   ```
   int i;
   float sum = 0.0;
   ```
for (i = 0; i < 4; i++)
    sum += a[i];
printf("sum = %4.1f\n", sum);

b. Now consider the following code:

```c
int i;
float sum = 0.0;
#pragma omp parallel for num_threads(2) \
    reduction(+:sum)
for (i = 0; i < 4; i++)
    sum += a[i];
printf("sum = %4.1f\n", sum);
```

Suppose that the run-time system assigns iterations $i = 0, 1$ to thread 0 and $i = 2, 3$ to thread 1. What is the output of this code on the Bleeblon?

5.6. Write an OpenMP program that determines the default scheduling of parallel for loops. Its input should be the number of iterations, and its output should be which iterations of a parallelized for loop are executed by which thread. For example, if there are two threads and four iterations, the output might be:

```
Thread 0: Iterations 0 — 1
Thread 1: Iterations 2 — 3
```

5.7. In our first attempt to parallelize the program for estimating $\pi$, our program was incorrect. In fact, we used the result of the program when it was run with one thread as evidence that the program run with two threads was incorrect. Explain why we could “trust” the result of the program when it was run with one thread.

5.8. Consider the loop

```
a[0] = 0;
for (i = 1; i < n; i++)
a[i] = a[i-1] + i;
```

There's clearly a loop-carried dependence, as the value of $a[i]$ can't be computed without the value of $a[i-1]$. Can you see a way to eliminate this dependence and parallelize the loop?

5.9. Modify the trapezoidal rule program that uses a parallel for directive (omp_trap_3.c) so that the parallel for is modified by a schedule(runtime) clause. Run the program with various assignments to the environment variable OMP_SCHEDULE and determine which iterations are assigned to which thread. This can be done by allocating an array iterations of $n$ ints and in the Trap function assigning omp_get_thread_num() to iterations[i] in the $i$th iteration of the for loop. What is the default assignment of iterations on your system? How are guided schedules determined?
5.10. Recall that all structured blocks modified by an unnamed critical directive form a single critical section. What happens if we have a number of atomic directives in which different variables are being modified? Are they all treated as a single critical section?

We can write a small program that tries to determine this. The idea is to have all the threads simultaneously execute something like the following code:

```c
int i;
double my_sum = 0.0;
for (i = 0; i < n; i++)
    # pragma omp atomic
    my_sum += sin(i);
```

We can do this by modifying the code by a parallel directive:

```c
# pragma omp parallel num_threads(thread_count)
{
    int i;
double my_sum = 0.0;
    for (i = 0; i < n; i++)
        # pragma omp atomic
        my_sum += sin(i);
}
```

Note that since `my_sum` and `i` are declared in the parallel block, each thread has its own private copy. Now if we time this code for large `n` when `thread_count = 1` and we also time it when `thread_count > 1`, then as long as `thread_count` is less than the number of available cores, the run-time for the single-threaded run should be roughly the same as the time for the multithreaded run if the different threads’ executions of `my_sum += sin(i)` are treated as different critical sections. On the other hand, if the different executions of `my_sum += sin(i)` are all treated as a single critical section, the multithreaded run should be much slower than the single-threaded run. Write an OpenMP program that implements this test. Does your implementation of OpenMP allow simultaneous execution of updates to different variables when the updates are protected by atomic directives?

5.11. Recall that in C, a function that takes a two-dimensional array argument must specify the number of columns in the argument list, so it is quite common for C programmers to only use one-dimensional arrays, and to write explicit code for converting pairs of subscripts into a single dimension. Modify the OpenMP matrix-vector multiplication so that it uses a one-dimensional array for the matrix.

5.12. Download the source file `omp_mat_vect_rand_split.c` from the book’s website. Find a program that does cache profiling (e.g., Valgrind [49]) and compile the program according to the instructions in the cache profiler documentation. (For example, with Valgrind you will want a symbol table and full optimization. (With gcc use, `gcc -g -O2 . . .`). Now run the program according to the instructions in the cache profiler documentation, using input `k \times (k \cdot 10^6)`,...
\((k \cdot 10^3) \times (k \cdot 10^3)\), and \((k \cdot 10^6) \times k\). Choose \(k\) so large that the number of level 2 cache misses is of the order \(10^6\) for at least one of the input sets of data.

a. How many level 1 cache write-misses occur with each of the three inputs?
b. How many level 2 cache write-misses occur with each of the three inputs?
c. Where do most of the write-misses occur? For which input data does the program have the most write-misses? Can you explain why?
d. How many level 1 cache read-misses occur with each of the three inputs?
e. How many level 2 cache read-misses occur with each of the three inputs?
f. Where do most of the read-misses occur? For which input data does the program have the most read-misses? Can you explain why?
g. Run the program with each of the three inputs, but without using the cache profiler. With which input is the program the fastest? With which input is the program the slowest? Can your observations about cache misses help explain the differences? How?

5.13. Recall the matrix-vector multiplication example with the \(8000 \times 8000\) input. Suppose that thread 0 and thread 2 are assigned to different processors. If a cache line contains 64 bytes or 8 doubles, is it possible for false sharing between threads 0 and 2 to occur for any part of the vector \(y\)? Why? What about if thread 0 and thread 3 are assigned to different processors; is it possible for false sharing to occur between them for any part of \(y\)?

5.14. Recall the matrix-vector multiplication example with an \(8 \times 8,000,000\) matrix. Suppose that doubles use 8 bytes of memory and that a cache line is 64 bytes. Also suppose that our system consists of two dual-core processors.

a. What is the minimum number of cache lines that are needed to store the vector \(y\)?
b. What is the maximum number of cache lines that are needed to store the vector \(y\)?
c. If the boundaries of cache lines always coincide with the boundaries of 8-byte doubles, in how many different ways can the components of \(y\) be assigned to cache lines?
d. If we only consider which pairs of threads share a processor, in how many different ways can four threads be assigned to the processors in our computer? Here, we’re assuming that cores on the same processor share cache.
e. Is there an assignment of components to cache lines and threads to processors that will result in no false-sharing in our example? In other words, is it possible that the threads assigned to one processor will have their components of \(y\) in one cache line, and the threads assigned to the other processor will have their components in a different cache line?
f. How many assignments of components to cache lines and threads to processors are there?
g. Of these assignments, how many will result in no false sharing?
5.15. a. Modify the matrix-vector multiplication program so that it pads the vector \( y \) when there’s a possibility of false sharing. The padding should be done so that if the threads execute in lock-step, there’s no possibility that a single cache line containing an element of \( y \) will be shared by two or more threads. Suppose, for example, that a cache line stores eight doubles and we run the program with four threads. If we allocate storage for at least 48 doubles in \( y \), then, on each pass through the for \( i \) loop, there’s no possibility that two threads will simultaneously access the same cache line.

b. Modify the matrix-vector multiplication program so that each thread uses private storage for its part of \( y \) during the for \( i \) loop. When a thread is done computing its part of \( y \), it should copy its private storage into the shared variable.

c. How does the performance of these two alternatives compare to the original program. How do they compare to each other?

5.16. Although \texttt{strtok} is thread-safe, it has the rather unfortunate property that it gratuitously modifies the input string. Write a tokenizer that is thread-safe and doesn’t modify the input string.

5.13 PROGRAMMING ASSIGNMENTS

5.1. Use OpenMP to implement the parallel histogram program discussed in Chapter 2.

5.2. Suppose we toss darts randomly at a square dartboard, whose bullseye is at the origin, and whose sides are 2 feet in length. Suppose also that there’s a circle inscribed in the square dartboard. The radius of the circle is 1 foot, and it’s area is \( \pi \) square feet. If the points that are hit by the darts are uniformly distributed (and we always hit the square), then the number of darts that hit inside the circle should approximately satisfy the equation

\[
\frac{\text{number in circle}}{\text{total number of tosses}} = \frac{\pi}{4},
\]

since the ratio of the area of the circle to the area of the square is \( \pi/4 \).

We can use this formula to estimate the value of \( \pi \) with a random number generator:

```c
number_in_circle = 0;
for (toss = 0; toss < number_of_tosses; toss++) {
    x = random double between -1 and 1;
    y = random double between -1 and 1;
    distance_squared = x*x + y*y;
    if (distance_squared <= 1) number_in_circle++;
}
pi_estimate = 4*number_in_circle/((double) number_of_tosses);
```
This is called a “Monte Carlo” method, since it uses randomness (the dart tosses).

Write an OpenMP program that uses a Monte Carlo method to estimate \( \pi \).
Read in the total number of tosses before forking any threads. Use a reduction clause to find the total number of darts hitting inside the circle. Print the result after joining all the threads. You may want to use `long long int`s for the number of hits in the circle and the number of tosses, since both may have to be very large to get a reasonable estimate of \( \pi \).

5.3. Count sort is a simple serial sorting algorithm that can be implemented as follows:

```c
void Count_sort(int a[], int n) {
    int i, j, count;
    int* temp = malloc(n*sizeof(int));

    for (i = 0; i < n; i++) {
        count = 0;
        for (j = 0; j < n; j++)
            if (a[j] < a[i])
                count++;
            else if (a[j] == a[i] && j < i)
                count++;
        temp[count] = a[i];
    }

    memcpy(a, temp, n*sizeof(int));
    free(temp);
} /* Count_sort */
```

The basic idea is that for each element \( a[i] \) in the list \( a \), we count the number of elements in the list that are less than \( a[i] \). Then we insert \( a[i] \) into a temporary list using the subscript determined by the count. There’s a slight problem with this approach when the list contains equal elements, since they could get assigned to the same slot in the temporary list. The code deals with this by incrementing the count for equal elements on the basis of the subscripts. If both \( a[i] == a[j] \) and \( j < i \), then we count \( a[j] \) as being “less than” \( a[i] \).

After the algorithm has completed, we overwrite the original array with the temporary array using the string library function `memcpy`.

a. If we try to parallelize the `for i` loop (the outer loop), which variables should be private and which should be shared?

b. If we parallelize the `for i` loop using the scoping you specified in the previous part, are there any loop-carried dependences? Explain your answer.

c. Can we parallelize the call to `memcpy`? Can we modify the code so that this part of the function will be parallelizable?

d. Write a C program that includes a parallel implementation of `Count_sort`.

e. How does the performance of your parallelization of `Count_sort` compare to serial `Count_sort`? How does it compare to the serial `qsort` library function?
5.4. Recall that when we solve a large linear system, we often use Gaussian elimination followed by backward substitution. Gaussian elimination converts an $n \times n$ linear system into an upper triangular linear system by using the “row operations.”

- Add a multiple of one row to another row
- Swap two rows
- Multiply one row by a nonzero constant

An upper triangular system has zeroes below the “diagonal” extending from the upper left-hand corner to the lower right-hand corner.

For example, the linear system

\[
\begin{align*}
2x_0 - 3x_1 &= 3 \\
4x_0 - 5x_1 + x_2 &= 7 \\
2x_0 - x_1 - 3x_2 &= 5
\end{align*}
\]

can be reduced to the upper triangular form

\[
\begin{align*}
2x_0 - 3x_1 &= 3 \\
x_1 + x_2 &= 1, \\
-5x_2 &= 0
\end{align*}
\]

and this system can be easily solved by first finding $x_2$ using the last equation, then finding $x_1$ using the second equation, and finally finding $x_0$ using the first equation.

We can devise a couple of serial algorithms for back substitution. The “row-oriented” version is

```c
for (row = n-1; row >= 0; row--) {
    x[row] = b[row];
    for (col = row+1; col < n; col++)
        x[row] -= A[row][col]*x[col];
    x[row] /= A[row][row];
}
```

Here the “right-hand side” of the system is stored in array $b$, the two-dimensional array of coefficients is stored in array $A$, and the solutions are stored in array $x$. An alternative is the following “column-oriented” algorithm:

```c
for (row = 0; row < n; row++)
    x[row] = b[row];

for (col = n-1; col >= 0; col--)
    x[col] /= A[col][col];
    for (row = 0; row < col; row++)
        x[row] -= A[row][col]*x[col];
```

a. Determine whether the outer loop of the row-oriented algorithm can be parallelized.
b. Determine whether the inner loop of the row-oriented algorithm can be parallelized.

c. Determine whether the (second) outer loop of the column-oriented algorithm can be parallelized.

d. Determine whether the inner loop of the column-oriented algorithm can be parallelized.

e. Write one OpenMP program for each of the loops that you determined could be parallelized. You may find the single directive useful—when a block of code is being executed in parallel and a sub-block should be executed by only one thread, the sub-block can be modified by a #pragma omp single directive. The threads in the executing team will block at the end of the directive until all of the threads have completed it.

f. Modify your parallel loop with a schedule(runtime) clause and test the program with various schedules. If your upper triangular system has 10,000 variables, which schedule gives the best performance?

5.5. Use OpenMP to implement a program that does Gaussian elimination. (See the preceding problem.) You can assume that the input system doesn’t need any row-swapping.

5.6. Use OpenMP to implement a producer-consumer program in which some of the threads are producers and others are consumers. The producers read text from a collection of files, one per producer. They insert lines of text into a single shared queue. The consumers take the lines of text and tokenize them. Tokens are “words” separated by white space. When a consumer finds a token, it writes it to stdout.