Shared Memory Programming with OpenMP
Programming Assignment – 7

Exercise – 1 [2 points] [OpenMP Implementation]

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. [2 points]

b) Modify the matrix-vector multiplication 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. [1 point]

c) How does the performance of these two alternatives compare to the original program? How do they compare to each other? [1 point]

Exercise – 2 [4 points]

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

```
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?

Exercise – 3 [4 points]

Recall that when we solve a large linear system, we often use Gaussian elimination followed by *backward substitution*. Gaussian elimination converts an n × 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 non-zero 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 :

$$\begin{aligned} 2x_0 &- 3x_1 &&= 3 \\ 4x_0 &- 5x_1 + x_2 &&= 7 \\ 2x_0 &- x_1 - 3x_2 &&= 7 \end{aligned}$$

can be reduced to the upper triangular form :

$$\begin{aligned} 2x_0 - 3x_1 &&&= 3 \\ x_1 + x_2 &&&= 1 \\ - 5x_2 &&&= 0 \end{aligned}$$

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

```
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:

```
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?