CS 435/535 – Fall 2024

Homework 3

The deadline for this homework is **December 29th at 23:59 (midnight).** Please upload your solutions to LMS by the deadline.

Question 1: Loop-carried Data Dependencies (20 points)

Consider the following 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] cannot be computed without the value of a[i−1]. How to eliminate this dependence and parallelize the loop?

Question 2: Critical Sections vs Atomic Operations (20 points)

Recall that all blocks modified by an unnamed critical directive form a single critical section in openmp. 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?

To test this, you can use a code similar to below. The idea is to have all the threads simultaneously execute something like the following code:

```
#pragma omp parallel num threads( thread_count)
{
        int i;
        double sum = 0.0;

        for ( i = 0; i < n; i ++){
                #pragma omp atomic
                sum += sin( i);
        }
}
```

Note that since *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 *sum += sin(i)* are treated as different critical sections. On the other hand, if the different executions of *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 (you can extend the given code above). Does your implementation of OpenMP allow simultaneous execution of updates to different variables when the updates are protected by atomic directives?

Question 3: (20 points)

Count sort is a simple serial sorting algorithm (known as count sort) 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 );
}
```

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) [6 points] If you try to parallelize the for i loop (the outer loop), which variables should be private and which should be shared?

b) [4 points] If you parallelize the for i loop using the scoping you specified in part (a), are there any loop-carried data dependencies? Explain your answer.

c) [10 points] Write a parallel version of count sort with OpenMP.

Question 4: OpenMP Parallelization Directives (20 points)

Please ***explain*** the behavior of threads running the for loops 1 and 2, and indicate what would be the output.

```
int main(int argc, char* argv[]) {

   int x = 0;
   int y = 0;
#  pragma omp parallel num_threads(2) firstprivate(x,y)
 {
   int my_rank = omp_get_thread_num();
   printf("Thread: %d\n", my_rank);
   //first loop
   for (int i = 0 ; i < 4; i++){
      x += i;
   }
   printf("Thread %d running first loop, result: %d\n", my_rank, x);

   #  pragma omp for
   //second loop
   for (int i = 0 ; i < 4; i++){
      y += i;
   }
   printf("Thread %d running second loop, result: %d\n", my_rank, y);
 }
   return 0;
}
```

Question 5: OpenMP Programming and Schedules (20 points)

Implement parallel matrix multiplication using OpenMP for AxB where both A and B are size of 1024×1024. You can use an auxilry function to fill these matrices automatically with random numbers. When you implement your matrix multiplication with OpenMP, you should experiment with the following scheduling policies: static, dynamic, and guided. You should provide your code along with the results of the following analysis.

a) Compare the performance of these policies with 2,4,8,16 threads with default chunk size.

b) Compare the performance of these policies with 16 threads when the chunk size varies to 4, 16, 32, and 64.

c) Analyze the impact of the chunk size when using dynamic and guided scheduling.

d) Discussion on why certain policies perform better for specific scenarios.