WITS UNIVERSITY

School of Computer Science and Applied Mathematics
UNIVERSITY OF THE WITWATERSRAND, JOHANNESBURG

COMS3008A: Parallel Computing
Exercise 5

2021-9-10

1 Objectives

- Use OpenMP `for`, `parallel`, and synchronization constructs to parallelize a serial program.
- Understanding OpenMP.

2 Problems

1. Continued from Q2 in **Excercise_4**, implement the example of computing the number π in **Lec4** using OpenMP `for` construct. Compare the performance of your implementations using different methods.
2. Given the pseudo code of square matrix multiplication in Listing 1, implement matrix multiplication, $C = AB$, where $C \in \mathbb{R}^{N\times N}$, $A \in \mathbb{R}^{N\times M}$, and $B \in \mathbb{R}^{M\times N}$, in parallel using OpenMP `for` construct. Consider the following questions for your implementation.
   (a) How many dot products are performed in your serial matrix multiplication?
   (b) Use the `OMP_NUM_THREADS` environment variable to control the number of threads and plot the performance with varying number of threads.
   (c) Experiment with two cases in which (i) only the outermost loop is parallelized; (ii) only the second inner loop is parallelized. What are the observed results from these three cases?
   (d) How does `collapse` clause work with `for` construct? Does it improve the performance of your parallel matrix multiplication?
   (e) Experiment with differing matrix sizes, such as $N \gg M$ or $M \gg N$. How does it impact the performance of your parallel implementation?

(f) Experiment with `static` and `dynamic` scheduling clauses to see effects on the performance of your parallel program. For each schedule clause, determine how many dot products each thread performs in your program.

```
1  ......
2  for(int i=0; i<N; i++)
3    for(int j=0; j<N; j++){
4      C[i][j] = 0.0;
5      for(int k=0; k<N; k++)
6        C[i][j] = C[i][j]+A[i][k]*B[k][i];
7    }
8  ......
```

Listing 1: Square matrix multiplication

3. Count sort is a simple serial algorithm that can be implemented as follows.

```
1  void count_sort(int a[], int n) {
2    int i, j, count;
3    int *temp = malloc(n * sizeof(int));
4    for (i = 0; i < n; i++) {
5      count = 0;
6      for (j = 0; j < n ; j++)
7        if (a[j] < a[i])
8          count++;
9        else if(a[j] == a[i] && j < i)
10         count++;
11     temp[count] = a[i];
12   }
13   memcpy (a, temp, n * sizeof(int));
14   free (temp);
15 }
```

Listing 2: 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 index determined by the count. The algorithm also deals with the issue where there are elements with the same value.

- Write a C program that includes both serial and parallel implementations of count-

sort. In your parallel implementation, specify explicitly the data scope of each variable in the parallel region.

- Compare the performance between the serial and parallel versions.

4. In what circumstances do you use the following environment variables? What are the respective equivalent library functions to these environment variables?

   (a) `OMP_NUM_THREADS`

   (b) `OMP_DYNAMIC`

   (c) `OMP_NESTED`

5. What is the difference between the two code snippets (see Listing 3) in terms of memory access?

```
1  //Code snippet 1
2  for(int i=0; i<n; i++)
3    for(int j=0; j<n; j++)
4      sum += a[i][j];

6  //Code snippet 2
7  for(int j=0; j<n; i++)
8    for(int i=0; i<n; j++)
9      sum += a[i][j];
```

Listing 3: Example code snippets

6. Given the code snippet in Listing 4, collapse the nested for loops into a single for loop by hand.

```
1  int m = 4;
2  int n = 1000;
3  ......
4  for(int i=0; i<m; i++)
5    for(int j=0; j<n; j++)
6      a[i][j] = i+j+1;
7  ......
```

Listing 4: Example of a nested for loop

7. Design simple examples to illustrate the functions of various OpenMP synchronization constructs including `barrier`, `critical`, `atomic`, `ordered`.

8. Suppose OpenMP does not have the reduction clause. Show how to implement an efficient parallel reduction for finding the minimum or maximum value in an array.