School of Electrical Engineering, Computing and Mathematical Sciences
Curtin University

CMPE 4003/6004 – Concurrent Systems
Laboratory Assignment 3 of 3

DUE DATE: MON 02/11/2020 10.00 AM

This task involves parallel programming with OpenMP.

For this you will need access to a C compiler and the associated OpenMP libraries.

It is *required* that you use Ubuntu 18.04.4 LTS to complete this laboratory assignment, as the native gcc compiler supports OpenMP directives.

You are required to complete the assessment tasks at your own pace, either in the lab on campus or at home, and then **submit a full report complete with source codes for grading.**

Ensure that you submit a *single* PDF of your report (filename SURNAME_STUDENTID_lab3report.pdf) plus a *separate* ZIP of all source codes generated (filename SURNAME_STUDENTID_lab3source.zip).

Follow the sequence of steps in each part by downloading the template codes for each part `lab3partX.c` and making the necessary modifications to the code as indicated in each question.

For each part the template has specified the structure for the code but certain sections of the code have been replaced with pseudocode comments indicating what should be executed in its place.

Each of the templates implements a serial version of the programme and your task is to implement the parallel version of the programme.

In preparing your lab report, ensure that where the problem:

- *requests an explanation, that it is given in sufficient detail, and as succinctly as possible,*
- *requests changes or additions to the code, to explicitly indicate the relevant line numbers as well as the modified or inserted code(s); DO NOT SIMPLY CUT AND PASTE YOUR ENTIRE SCRIPT*
- *requests execution of code, that your report shows the output, e.g. a screenshot or terminal session.*

P1. Recall the Monte Carlo method for approximating the value of π by generating random samples in a square and then counting how many samples fall within the circle which is inscribed just inside the square.

(a) Compile and run the template code with a single thread, and thus verify that the serial version is correct. Note that for this part you should compile with OpenMP support but only run with a single thread.
(b) Parallelize the code for multi-threaded execution by ONLY adding OpenMP directives. DO NOT modify any of the source code, EXCEPT for adding OpenMP directives. Your solution should attempt to minimize overheads in terms of forking, using, joining threads and be faster than the serial version. In your lab report, indicate your additions to the code, and explain at a conceptual level, at what points in the code new threads are forked and joined, which parts of code is executed by which threads, where barriers if any in terms of thread synchronization would be encountered, and hence how the code achieves parallelization.
(c) Now add timing statements to verify that your solution runs faster than the serial version for 10^9 samples. Tabulate the speedup and efficiency for 1,2,4,8 threads. Do the results match your expectations? Why?

P2. Recall the trapezoidal rule as a numerical integration scheme to approximate the area under a function. The template code is a serial implementation which is hardwired for the function $f(x)=x^4$.

(a) Compile and run the template code with a single thread, for $a=0,\ b=1,\ n=2^{30}$, and thus verify that the serial version is correct, also checking the calculation by hand.
(b) Parallelize the code for multi-threaded execution by adding to it as few OpenMP directives as possible. DO NOT modify any of the source code, EXCEPT for adding OpenMP directives. In your lab report, indicate your additions to the code, and explain at a conceptual level, at what points in the code new threads are forked and joined, which parts of code is executed by which threads, where barriers if any in terms of thread synchronization would be encountered, and hence how the code achieves parallelization.
(c) Now add timing statements to verify that your parallel solution runs faster than the serial version. Tabulate the speedup and efficiency for 1,2,4,8 threads. Do the results match your expectations? Why?
Can you also explain why the output differs *slightly* from run to run for *multiple threads*?

P3. Recall the simple histogram programme that simply counts the number of points falling within fixed bins. The template code is a serial implementation which has all I/O, debug and data generation routines builtin. The main function has some redundancy when run with a single thread, but has been written this way to facilitate parallelization of the programme.

(a) Compile and run the template code with a single thread, using a data range between 0 and 10, a total of 100 data points, and 10 equally spaced bins. Verify by other means that the serial version is correct. Note that for this part you should compile with OpenMP support but only run with a single thread.

(b) Parallelize the code for multi-threaded execution by ONLY adding OpenMP directives. Do this by adding OpenMP directives to the section of the code within the `main` function which is shown by the comment `/* Count number of values in each bin */` since this is the "functional" part which oversees the generation of the local bin counts for each thread and subsequently adds these together to obtain the total bin counts for the final answer. DO NOT modify any of the source code, EXCEPT for adding OpenMP directives. In your lab report, indicate your additions to the code. Explain how your code achieves parallelization. Describe at what points new threads are forked and joined and which parts of the code are executed by which threads. Pay particular attention as to whether or not you need to add explicit barriers for thread synchronization to avoid race conditions. Be careful with your declaration of private and shared variables between threads.

(c) Now add timing statements to verify that your parallel solution runs faster than the serial version. Again use a data range between 0 and 10, a total of 10^7 data points, and 10 equally spaced bins. Tabulate the speedup and efficiency for 1,2,4,8 threads. You can comment and disable the output routine for this part.

P4. 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 numberof 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.

The template code has all the necessary functionality including serial code, timing, and output routines, *except* for the parallel sorting subroutine `Count_sort_parallel` which you will need to write, but has been temporarily filled with serial code to temporarily allow compilation and execution.

(a) If we attempt to parallelize the `for i` loop (the outer loop), which variables should be private and which should be shared?
(b) Hence parallelize the code using OpenMP directives to modify the `Count_sort_parallel` subroutine (which in the template code is a just copy of the serial code) ensuring that your new parallel code is faster than the original serial code (at least for a sufficient number of elements).
(c) Run your programme with 1,2,4,8 threads and *n=10000* elements. Check that your programme runs correctly using the checker in the template code! Calculate the speedup and efficiency of the new parallelized version. How does the performance of your parallelization of compare to the serial version?
(d) What is the complexity of serial `Count_sort`? What is the complexity of serial `qsort`? Does your answer match the run times generated by the programme?

P5. Recall that when we solve a large linear system $\mathbf{A}\boldsymbol{x}=\boldsymbol{b}$, we often use Gaussian elimination followed by backward substitution. Gaussian elimination converts an *n x n* linear system into an upper or lower triangular linear system by using so called 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 simplicity in the code we will ignore the need for row swaps and focus on the conversion to upper triangular form followed by back substitution:

- Solve for the last variable using the last equation;
- Solve for the next variable using the next equation given the value of the previous solution(s);
- Iterate until all variables are solved.

A serial implementation for Gaussian elimination is given as follows:

```
void Gaussian_elim(double A[], double b[], int n){
   int i, j, k;
   double fact;

   for (i = 0; i < n-1; i++)
      for (k = i+1; k < n; k++) {
         fact = -A[k*n + i]/A[i*n + i];
         A[k*n + i] = 0;
         for (j = i+1; j < n; j++)
            A[k*n + j] += fact*A[i*n + j];
         b[k] += fact*b[i];
      }
}  /* Gaussian_elim */
```

Once we have an upper triangular form for the linear system, a serial implementation for back substitution is:

```
void Row_solve(double A[], double b[], double x[], int n) {
   int i, j;
   double tmp;

   for (i = n-1; i >= 0; i--) {
      tmp = b[i];
      for (j = i+1; j < n; j++)
         tmp += -A[i*n+j]*x[j];
         x[i] = tmp/A[i*n+i];
   }
}  /* Row_solve */
```

We will parallelize this method for solving a system of linear equations. To make things a little simpler and to focus on the parallelization, the template code supplies the input/output and timing routines. In particular the code generates the $n \times n$ matrix $\mathbf{A}$ such that the diagonal elements are $n/10$ and the remaining elements are random doubles between 0 and 1. The constraint vector $\boldsymbol{b}$ is then generated such that the solution to the system of equations $\mathbf{A}\boldsymbol{x}=\boldsymbol{b}$ is always $\boldsymbol{x}=\mathbf{1}$ (the vector of all ones). Thus your task will be to parallelize the Gaussian elimination and back substitution routine using OpenMP. In the template code these target functions have been supplied via the serial version, except that an extra argument for the number of threads has been added for use in your parallelization. Now:

(a) Add OpenMP directives to parallelize the `Gaussian_elim` function.
Ensure that your parallelization is reasonably efficient

(b) Add OpenMP directives to parallelize the `Row_solve` function
Consider the use of a reduction operator and ensure an efficient parallelization

(c) Compile and run your code.
Use a matrix size $10^3 \times 10^3$. Tabulate the performance and speedup for 1,2,4,8 threads.
At what point does the speedup stop increasing?
Is this consistent with the number of cores?

Be careful to take into account race conditions and the scoping of variables for parallelization.