Curtin University, Concurrent Systems

Lab report 3

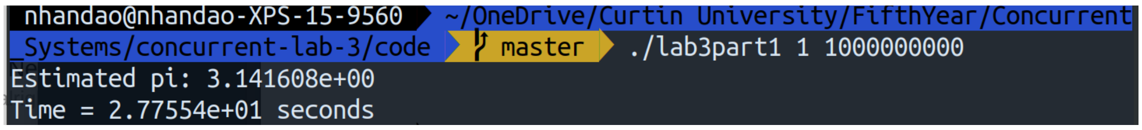
Nhan Dao

November 2, 2020

Contents

1. Monte Carlo method for approximating the value of π in OpenMP

a. Compile and run the template code with a single thread, and thus verify that the serial version is correct.



```
nhandao@nhandao-XPS-15-9560  ~/OneDrive/Curtin University/FifthYear/Concurrent_Systems/concurrent-lab-3/code  master  ./lab3part1 1 1000000000
Estimated pi: 3.141608e+00
Time = 2.77554e+01 seconds
```

Figure 1.1: Terminal output of execution of template code with a single thread

b. Parallelize the code for multi-threaded execution by ONLY adding OpenMP directives

```
# pragma omp parallel num_threads(thread_count) // OMP parallel
   {
      int my_rank = omp_get_thread_num();
      unsigned seed = my_rank + 1;
      long long int toss;
      double x, y, distance_squared;

#     pragma omp for  reduction(+ : number_in_circle) // OMP for
      for(toss = 0; toss < number_of_tosses; toss++) {
         x = 2*my_drand(&seed) - 1;
         y = 2*my_drand(&seed) - 1;
         distance_squared = x*x + y*y;
         if (distance_squared <= 1) number_in_circle++;
      }
   }
```

Listing 1: Parallizing the code for multi-threaded execution (line 34-48 in lab3part1.c)

i. At what point in the code new threads are forked and joined?

The code is serial up until *pragma*. At pragma, multiple threads are forked, line 1 in listing 1. After the structured block (line 2 - 15), the slaves are joined with the master thread and execution resumes on the master thread.

ii. Which part of the code is executed by which threads?

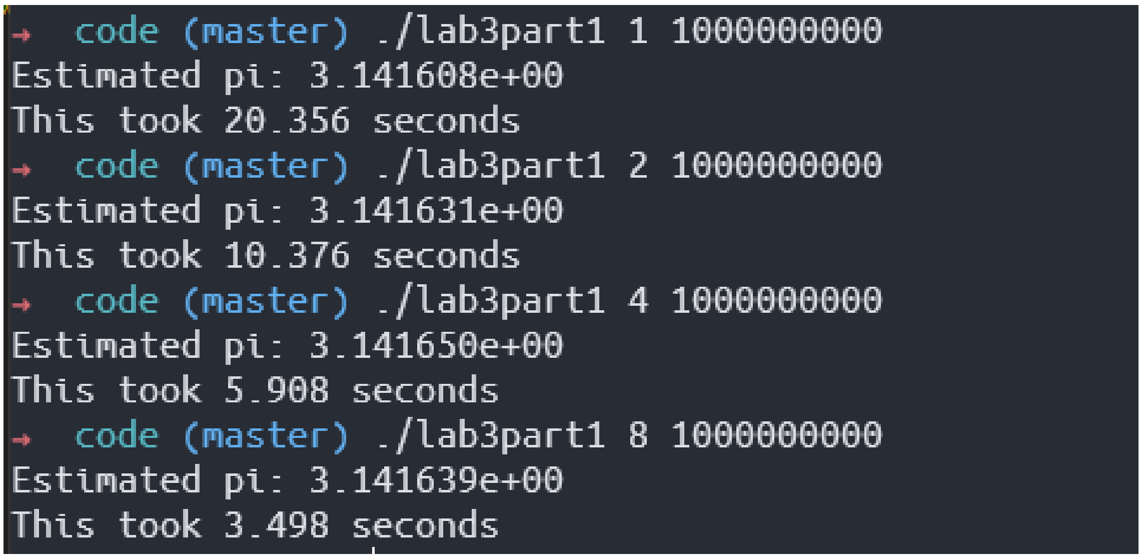
The total number of threads are comprised of one main thread, and the remainder are slave threads. The main thread will run all code outside of the parallel enivornment. Master and slaves thread will run the section of code assigned by the OpenMP directives - line 2 - 15.

iii. Where barrires if any in terms of thread synchronization would be encountered, and hence how the code achives parallelization

Thread synchronization is needed to so reduction can be apply to the variable *number_in_circle* to avoid race condition. OpenMP's reduction basically behaves similarly to a critical section in terms of synchronisation.

c. Tabulate the speedup and efficiency for 1,2,4,8 threads.

Threads	Time (s)	Speedup	Efficiency (%)
1	20.356	N/A	100
2	10.376	1.962	98.10
4	5.908	3.476	86.90
8	3.498	5.819	72.74



```
➜  code (master) ./lab3part1 1 1000000000
Estimated pi: 3.141608e+00
This took 20.356 seconds
➜  code (master) ./lab3part1 2 1000000000
Estimated pi: 3.141631e+00
This took 10.376 seconds
➜  code (master) ./lab3part1 4 1000000000
Estimated pi: 3.141650e+00
This took 5.908 seconds
➜  code (master) ./lab3part1 8 1000000000
Estimated pi: 3.141639e+00
This took 3.498 seconds
```

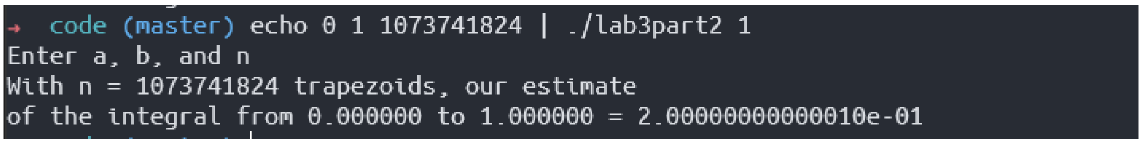
Figure 1.2: Terminal output for execution with 1, 2, 4, 8 threads respectively for estimating pi using Monte Carlo's method with 10^9 samples

i. Do the result match your expectation? Why?

The result matches our expectation. The speedup will increase as the number of threads increase but will saturate at somepoint due to Amdahl's Law which states that the parallel program is bounded by the section of code that can only be executed in serial. Hence, even though we are allocating more threads, the efficiency decreases and we get a diminishing return due to this fact.

2. Trapezoidal rule to approximate area under $f(x) = x^4$ in OpenMP

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.



```
→  code (master) echo 0 1 1073741824 | ./lab3part2 1
Enter a, b, and n
With n = 1073741824 trapezoids, our estimate
of the integral from 0.000000 to 1.000000 = 2.00000000000010e-01
```

Figure 2.1: Terminal output for $a = 0$, $b = 1$, $n = 2^{30}$

The following shows the calculation to compute by hand:

$$\begin{aligned}\text{Area} &= \int_0^1 x^4\\ &= \left[\frac{x^5}{5}\right]_0^1\\ &= \frac{1^5}{5} - \frac{0^5}{5}\\ &= 0.2\end{aligned}$$

The computed integral using the trapezoidal rule equals to the answer worked out by hand analytically using calculus (There is a negligible difference due to base two floating-point numbers limitation to represent 1/20)

b. Parallelize the code for multi-threaded execution by adding to it as few OpenMP directives as possible.

```
/*------------------------------------------------------------------
 * Function:    Trap
 * Purpose:     Use trapezoidal rule to estimate definite integral
 * Input args:
 *    a: left endpoint
 *    b: right endpoint
 *    n: number of trapezoids
 * Return val:
 *    approx:  estimate of integral from a to b of f(x)
 */
double Trap(double a, double b, int n, int thread_count) {
   double  h, approx;
   int  i;

   h = (b-a)/n;
16    approx = (f(a) + f(b))/2.0;
17    #pragma omp parallel for num_threads(thread_count) reduction(+:
         approx)
18    for (i = 1; i <= n-1; i++)
19      approx += f(a + i*h);
20    approx = h*approx;
21
22    return approx;
23 }  /* Trap */
```

Listing 2: Parallizing the code for multi-threaded execution (line 56-78 in lab3part2.c)

i. At what points in the code new threads are forked and joined?

Parallization occurs within the for loop though the use of OpenMP *parallel for*. The *for* pragma do not create a team of threads; instead they take the team of threads that is active, and divide the loop iterations over them.

Since no threads were created prior to the pragma at line 17 in listing 2, this will be the point which new threads are forked. Threads are then joined once the team of threads complete the iterations.

ii. Which parts of code is executed by which threads?

The main thread will execute all the code outside the parallel region, I.E. outside the for loop at line 17 in listing 2. Within the for loop, OpenMP will take the team of thread that is active, and divide the loop iterations over them which include the main thread, and any slaves thread.

iii. Where barriers if any in terms of thread synchronization would be encountered, and hence how the code achieves parallelization?

OpenMP creates a critical section and the values stored in the private variable/s (in this case it is the *approx* variable) are reduced in the critical section. This is synchronization that needs to happen to avoid race conditions between threads.

c. Add timing statement to verify that the parallel solution runs faster than the serial version. Tabulates the speedup and efficiency for 1, 2, 4, 8, threads.

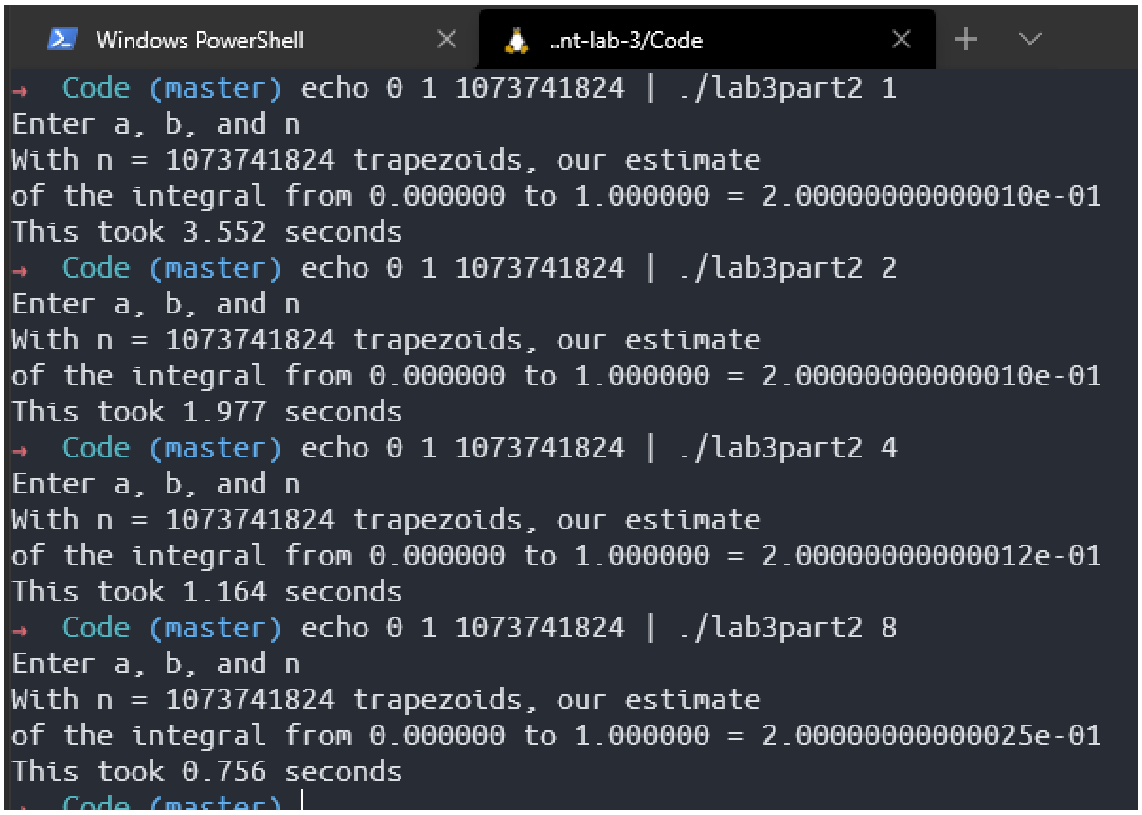
Threads	Time (s)	Speedup	Efficiency (%)
1	3.439	N/A	100
2	1.920	1.791	89.557
4	1.252	2.747	68.67
8	0.754	4.561	57.013

i. Do the result match the expectation? Why?

As expected, the more threads will decrease the execution time, however, there is a diminishing return such that the gain in speed from an increase in thread diminishes as the thread number increase.

This is due to Amdahl's Law, which effectively state that their is an upper bound in the speedup regardless the number of threads available unless virtually all of a serial program is parallelized. Since only the *for* loop to compute the trapezoidal area is parallelized - the speed of the program is going to be very much bounded by the serial execution of the non-parallel portion.

ii. Explain why the output differs slightly from run to run for mulitple thread?



```
→  Code (master) echo 0 1 1073741824 | ./lab3part2 1
Enter a, b, and n
With n = 1073741824 trapezoids, our estimate
of the integral from 0.000000 to 1.000000 = 2.00000000000010e-01
This took 3.552 seconds
→  Code (master) echo 0 1 1073741824 | ./lab3part2 2
Enter a, b, and n
With n = 1073741824 trapezoids, our estimate
of the integral from 0.000000 to 1.000000 = 2.00000000000010e-01
This took 1.977 seconds
→  Code (master) echo 0 1 1073741824 | ./lab3part2 4
Enter a, b, and n
With n = 1073741824 trapezoids, our estimate
of the integral from 0.000000 to 1.000000 = 2.00000000000012e-01
This took 1.164 seconds
→  Code (master) echo 0 1 1073741824 | ./lab3part2 8
Enter a, b, and n
With n = 1073741824 trapezoids, our estimate
of the integral from 0.000000 to 1.000000 = 2.00000000000025e-01
This took 0.756 seconds
→  Code (master) |
```

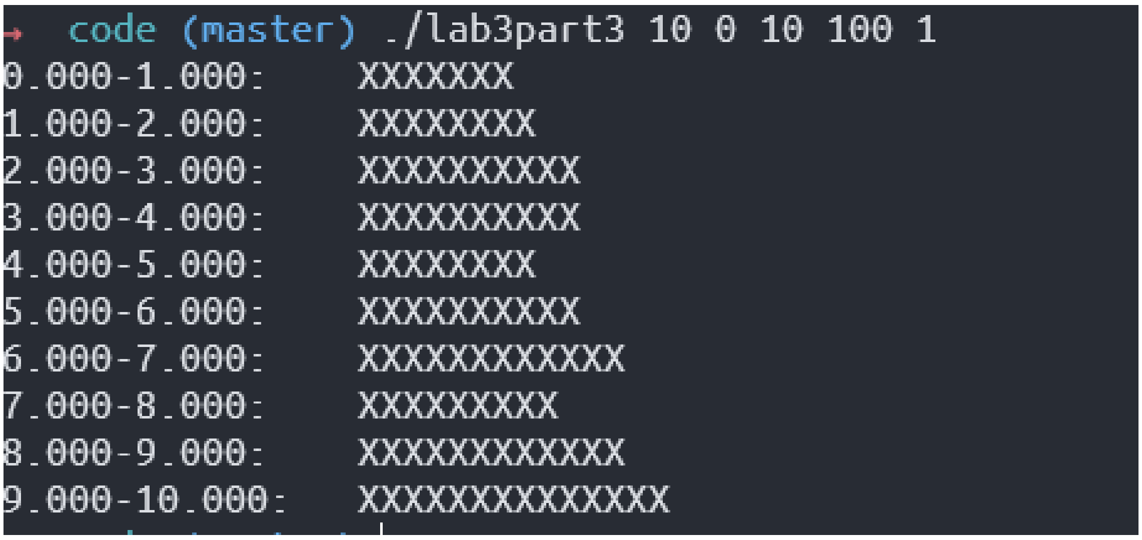
Figure 2.2: Terminal output of execution for varying number of threads

As seen in fig. 2.2, the output differ slightly from run to run. The hypothesis is that the reduction is more accurate for higher thread count due to the fact there is less floating-point precision loss because each thread is approximating a smaller share of the area. In other words, for an integrand of $[1,0]$, a thread count of two will mean that each thread gets 50% of the bound, whereas a thread count of 4 means each thread gets 25% share of the bound. A smaller bound would mean that there is less floating-point precision loss as the reduction function is applied.

3. Parallel histogram generation using OpenMP

a. Compile and run the template code with a single thread to verify that the serial version is correct

Figure 3.1 shows the terminal output for the execution of the template code with one thread.



```
➜  code (master) ./lab3part3 10 0 10 100 1
0.000-1.000:    XXXXXXX
1.000-2.000:    XXXXXXXX
2.000-3.000:    XXXXXXXXXX
3.000-4.000:    XXXXXXXXXX
4.000-5.000:    XXXXXXXX
5.000-6.000:    XXXXXXXXXX
6.000-7.000:    XXXXXXXXXXXX
7.000-8.000:    XXXXXXXXX
8.000-9.000:    XXXXXXXXXXXX
9.000-10.000:   XXXXXXXXXXXXXX
```

Figure 3.1: Terminal output for execution with a single thread, using a data range bewteeen 0 and 10, a total of 100 data points, and 10 equally spaced bins

b. Parallelize the code for multi-thread execution by ONLY adding OpenMP directives

Listing 4 shows the changes within the template code to parallelize by only adding OpenMP directives.

```c
/* Count number of values in each bin */
#pragma omp parallel num_threads(thread_count)
{
    int j, my_rank = omp_get_thread_num();
    int my_offset = my_rank*bin_count;

    #pragma omp for private(bin)
    for (i = 0; i < data_count; i++) {
        my_rank = omp_get_thread_num();
        bin = Which_bin(data[i], bin_maxes, bin_count, min_meas);
#       ifdef DEBUG
        printf("Th %d > i = %d, data = %4.3f, bin = %d\n", my_rank, i,
            data[i], bin);
#       endif
        loc_bin_counts[my_offset + bin]++;
    }

```

```
      #pragma omp for
      for (i = 0; i < bin_count; i++)
         for (j = 0; j < thread_count; j++) {
#           ifdef DEBUG
            printf("Th %d > Adding %d to bin_counts[%d] = %d\n",
               my_rank, loc_bin_counts[j*bin_count + i], i,
               bin_counts[i]);
#           endif
            bin_counts[i] += loc_bin_counts[j*bin_count + i];
         }
   }
```

Listing 3: Parallizing the code for multi-threaded execution (line 92-116 in lab3part3.c)

i. Explain how your code achieves parallelization.

The first OpenMP directive is initializing the parallel environment within the code block from line 3 to 25 in listing 4. This directive will create a team of threads equal to the thread count specified by the user, and all variables within that environment are within their own scope in each thread (I.E stored within the stack of each thread).

The next directive divides the work of determining which data point falls under which bin amongst the threads. It outputs the frequency of data point to a local histrogram counter in a form of an array. Here, the variable *bin* must be declared private which makes a separte copy into each thread's stack. This is done because we want to avoid race conditions for when multiple thread tries to write to *bin* at the same time.

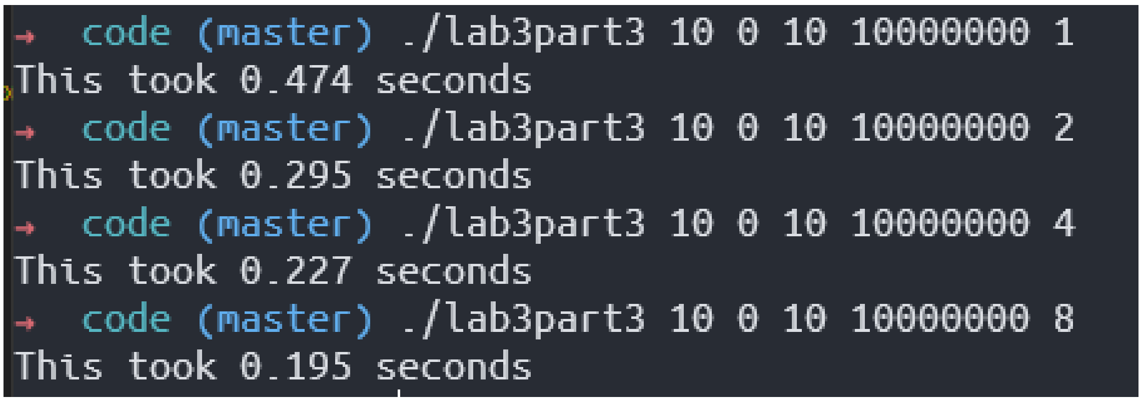
The third and final *for* directive sum up the local histogram counter to the final output histogram. Here a nested for loop is used, because it needs to iterate over each local histogram, and then each bin within the local histogram to sum up the total histogram count.

ii. Describe at what points new threads are forked and joined and which part of the code are executed by which threads.

New threads are forked when the parallel environment begins (line 2 in listing 4) and they are joined once the parallel environment ends (line 25 listing 4). Within the parallel environment, work are divided amongst the threads, I.E, through the use of the *for* pragma. Any code outside of the parallel environment are run in serial by the main thread.

c. Tabulate the speedup and efficiency for 1,2,4,8 threads using a data range between 0 and 10, a total of 10^7 data points, and 10 equally spaced bins

Threads	Time (s)	Speedup	Efficiency (%)
1	0.474	N/A	100
2	0.295	1.607	80.34
4	0.227	2.088	52.20
8	0.195	2.431	30.38



```
→  code (master) ./lab3part3 10 0 10 10000000 1
This took 0.474 seconds
→  code (master) ./lab3part3 10 0 10 10000000 2
This took 0.295 seconds
→  code (master) ./lab3part3 10 0 10 10000000 4
This took 0.227 seconds
→  code (master) ./lab3part3 10 0 10 10000000 8
This took 0.195 seconds
```

Figure 3.2: Terminal output for execution with 1, 2, 4, 8 threads respectively, using a data range bewteeen 0 and 10, a total of 10^7 data points, and 10 equally spaced bins.

4. PARALLEL COUNT SORT USING OPENMP

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

The variables *temp, a* should be shared and, *i, j, count* should be private.

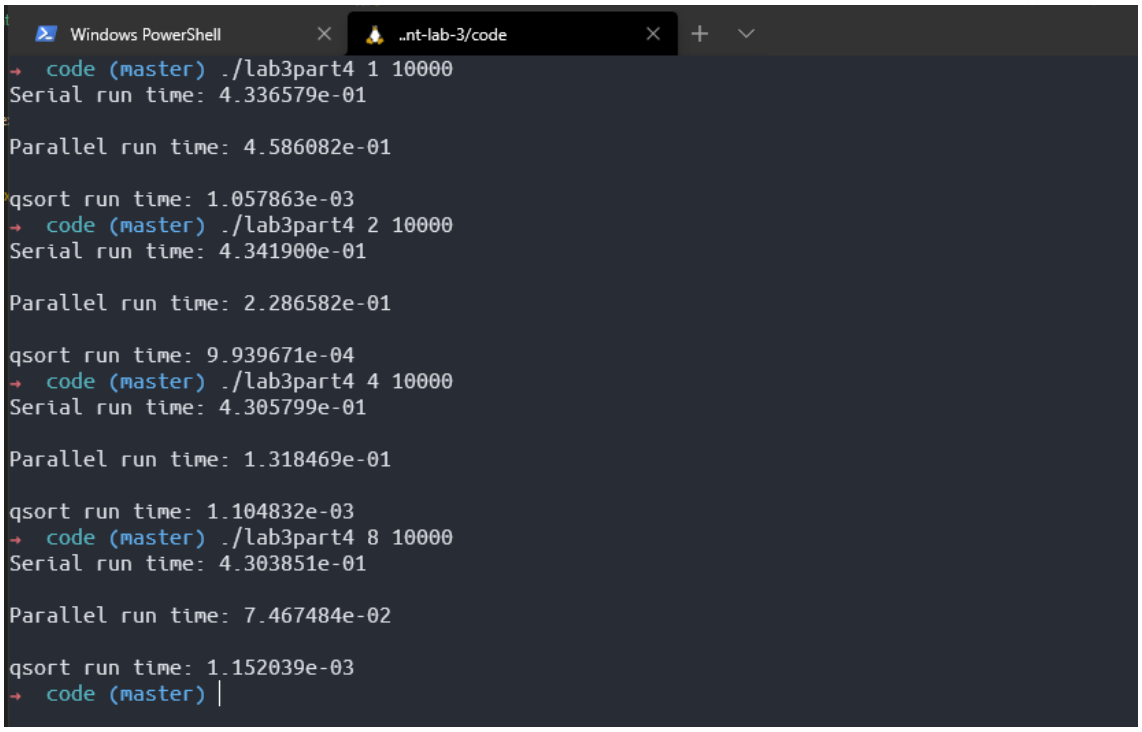
b. Parallelize the code using OpenMP directives to modify the Count_sort_parallel subroutine

```c
/*-----------------------------------------------------------------------
 * Function:     Count_sort_parallel
 * Purpose:      sort elements in an array using parallel count sort
 * In arg:       n: number of elements
 * In/out arg:   a: array of elements
 */

void Count_sort_parallel(int a[], int n, int thread_count) {
   int i, j, count;
   int* temp = malloc(n*sizeof(int));

      #pragma omp parallel for num_threads(thread_count) shared(temp, a)
          private(i, j, count)
      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_parallel */
```
Listing 4: Parallizing the code for multi-threaded execution (line 167-191 in lab3part3.c)

c. Tabuate the speedup and efficiency for 1,2,4,8 threads and n=1000 elements

Threads	Time (s)	Speedup	Efficiency (%)
1	0.459	N/A	100
2	0.229	1.899	94.95
4	0.132	3.266	81.65
8	0.075	5.764	72.05



```
→  code (master) ./lab3part4 1 10000
Serial run time: 4.336579e-01

Parallel run time: 4.586082e-01

qsort run time: 1.057863e-03
→  code (master) ./lab3part4 2 10000
Serial run time: 4.341900e-01

Parallel run time: 2.286582e-01

qsort run time: 9.939671e-04
→  code (master) ./lab3part4 4 10000
Serial run time: 4.305799e-01

Parallel run time: 1.318469e-01

qsort run time: 1.104832e-03
→  code (master) ./lab3part4 8 10000
Serial run time: 4.303851e-01

Parallel run time: 7.467484e-02

qsort run time: 1.152039e-03
→  code (master) |
```

Figure 4.1: Terminal output for execution with 1, 2, 4, 8 threads respectively and n=10000

i. How does the performance of the parallelized code compare to the serial version?

The performance of the parallelize code is better than the serial version in terms of speed as seen in the table above. However, there is a diminishing return in the speed gain vs the resources (threads) allocated.

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?

The time complexity of serial Count_sort is O(n), and the time complexity of serial qsort is O(nlogn). Which means theoretically, qsort should be slower than serial Count_sort, however, this is not the case when looking at the run times generated by the program in fig. 4.1.

5. Parallel Gaussian elimination using OpenMP

a. Add OpenMP directives to parallelize the *Guassian_elim* function

```c
/*----------------------------------------------------------------------
 * Function:     Gaussian_elim
 * Purpose:      Convert A to an upper triangular system
 * In args:      n, thread_count
 * In/out args:  A, b
 * Note:         It's assumed that row-swaps aren't necessary
 */
void Gaussian_elim(double A[], double b[], int n, int thread_count){
   int i, j, k;
   double fact;

#pragma omp parallel num_threads(thread_count) private(i, k, j, fact)
   for (i = 0; i < n-1; i++)
#pragma omp for
      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 */
```

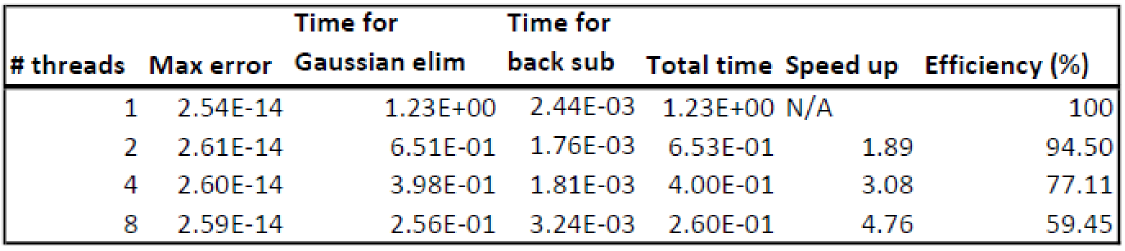
Listing 5: Parallelizing Gaussian_elim (line 117-138 in lab3part5.c)

b. Add OpenMP directives to parallelize the *Row_solve* function

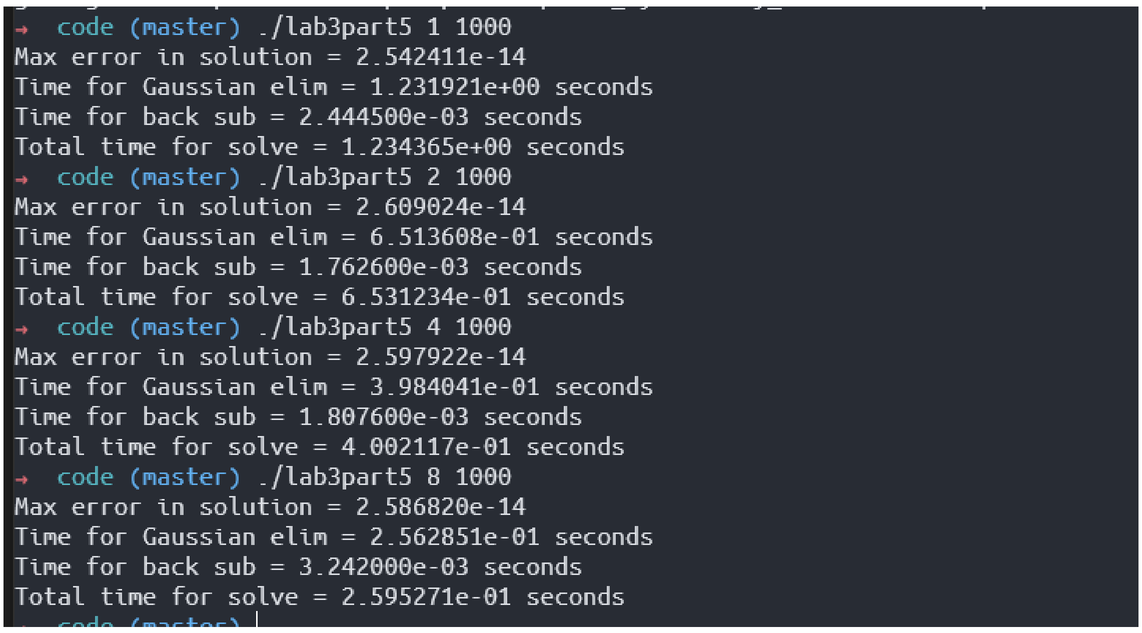
```c
/*----------------------------------------------------------------------
 * Function:   Row_solve
 * Purpose:    Solve a triangular system using the row-oriented algorithm
 * In args:    A, b, n, thread_count
 * Out arg:    x
 */
void Row_solve(double A[], double b[], double x[], int n, int
    thread_count) {
   int i, j;
   double tmp;

#pragma omp parallel num_threads(thread_count) private(i, j) shared(tmp)
   for (i = n-1; i >= 0; i--) {
      #pragma omp single
      tmp = b[i];
      #pragma omp for reduction(+: tmp)
      for (j = i+1; j < n; j++)
         tmp += -A[i*n+j]*x[j];
      #pragma omp single
      {
         x[i] = tmp/A[i*n+i];
#        ifdef DEBUG
         printf("x[%d] = %.1f\n", i, x[i]);
#        endif
      }
   }
} /* Row_solve */
```

Listing 6: Parallelizing Row_solve (line 140-165 in lab3part5.c)

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

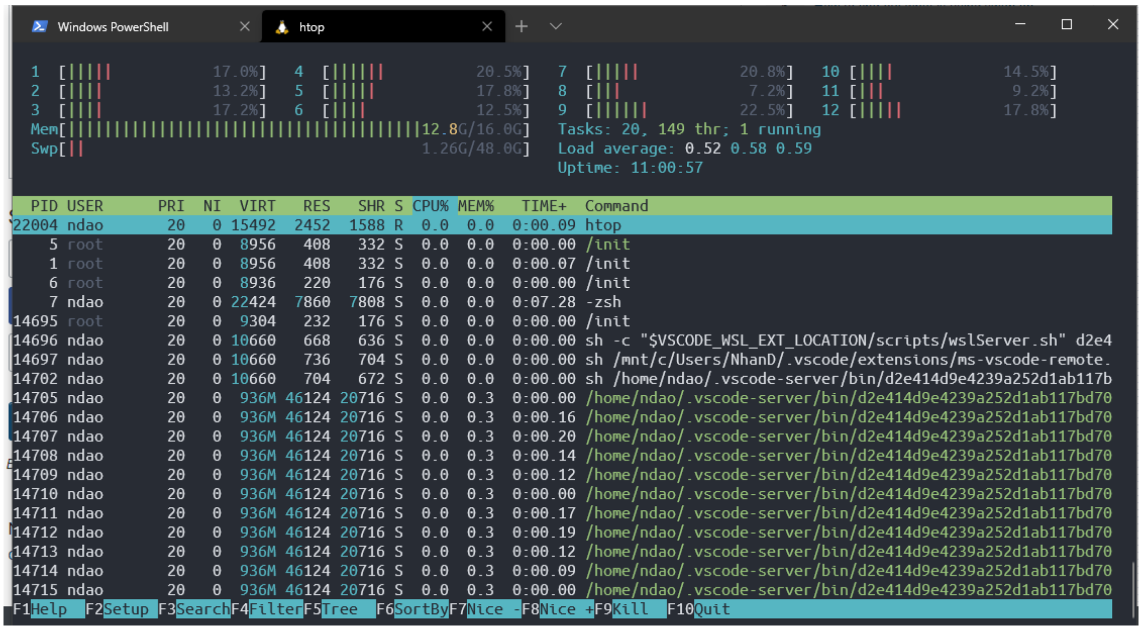


# threads	Max error	Time for Gaussian elim	Time for back sub	Total time	Speed up	Efficiency (%)
1	2.54E-14	1.23E+00	2.44E-03	1.23E+00	N/A	100
2	2.61E-14	6.51E-01	1.76E-03	6.53E-01	1.89	94.50
4	2.60E-14	3.98E-01	1.81E-03	4.00E-01	3.08	77.11
8	2.59E-14	2.56E-01	3.24E-03	2.60E-01	4.76	59.45



```
→ code (master) ./lab3part5 1 1000
Max error in solution = 2.542411e-14
Time for Gaussian elim = 1.231921e+00 seconds
Time for back sub = 2.444500e-03 seconds
Total time for solve = 1.234365e+00 seconds
→ code (master) ./lab3part5 2 1000
Max error in solution = 2.609024e-14
Time for Gaussian elim = 6.513608e-01 seconds
Time for back sub = 1.762600e-03 seconds
Total time for solve = 6.531234e-01 seconds
→ code (master) ./lab3part5 4 1000
Max error in solution = 2.597922e-14
Time for Gaussian elim = 3.984041e-01 seconds
Time for back sub = 1.807600e-03 seconds
Total time for solve = 4.002117e-01 seconds
→ code (master) ./lab3part5 8 1000
Max error in solution = 2.586820e-14
Time for Gaussian elim = 2.562851e-01 seconds
Time for back sub = 3.242000e-03 seconds
Total time for solve = 2.595271e-01 seconds
```

Figure 5.1: Terminal output for execution with 1, 2, 4, 8 threads respectively, for a matrix size $10^3 \times 10^3$



```
  1  [|||||             17.0%]   4  [||||||            20.5%]   7  [|||||             20.8%]   10 [||||              14.5%]
  2  [||||              13.2%]   5  [|||||             17.8%]   8  [|||                7.2%]   11 [|||                9.2%]
  3  [||||              17.2%]   6  [||||              12.5%]   9  [|||||||           22.5%]   12 [|||||             17.8%]
  Mem[|||||||||||||||||||||||||||||||||||||12.8G/16.0G]   Tasks: 20, 149 thr; 1 running
  Swp[||                                   1.26G/48.0G]   Load average: 0.52 0.58 0.59
                                                          Uptime: 11:00:57

  PID USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
22004 ndao       20   0 15492  2452  1588 R  0.0  0.0  0:00.09 htop
    5 root       20   0  8956   408   332 S  0.0  0.0  0:00.00 /init
    1 root       20   0  8956   408   332 S  0.0  0.0  0:00.07 /init
    6 root       20   0  8936   220   176 S  0.0  0.0  0:00.00 /init
    7 ndao       20   0 22424  7860  7808 S  0.0  0.0  0:07.28 -zsh
14695 root       20   0  9304   232   176 S  0.0  0.0  0:00.00 /init
14696 ndao       20   0 10660   668   636 S  0.0  0.0  0:00.00 sh -c "$VSCODE_WSL_EXT_LOCATION/scripts/wslServer.sh" d2e4
14697 ndao       20   0 10660   736   704 S  0.0  0.0  0:00.00 sh /mnt/c/Users/NhanD/.vscode/extensions/ms-vscode-remote.
14702 ndao       20   0 10660   704   672 S  0.0  0.0  0:00.00 sh /home/ndao/.vscode-server/bin/d2e414d9e4239a252d1ab117b
14705 ndao       20   0  936M 46124 20716 S  0.0  0.3  0:00.00 /home/ndao/.vscode-server/bin/d2e414d9e4239a252d1ab117bd70
14706 ndao       20   0  936M 46124 20716 S  0.0  0.3  0:00.16 /home/ndao/.vscode-server/bin/d2e414d9e4239a252d1ab117bd70
14707 ndao       20   0  936M 46124 20716 S  0.0  0.3  0:00.20 /home/ndao/.vscode-server/bin/d2e414d9e4239a252d1ab117bd70
14708 ndao       20   0  936M 46124 20716 S  0.0  0.3  0:00.14 /home/ndao/.vscode-server/bin/d2e414d9e4239a252d1ab117bd70
14709 ndao       20   0  936M 46124 20716 S  0.0  0.3  0:00.12 /home/ndao/.vscode-server/bin/d2e414d9e4239a252d1ab117bd70
14710 ndao       20   0  936M 46124 20716 S  0.0  0.3  0:00.00 /home/ndao/.vscode-server/bin/d2e414d9e4239a252d1ab117bd70
14711 ndao       20   0  936M 46124 20716 S  0.0  0.3  0:00.17 /home/ndao/.vscode-server/bin/d2e414d9e4239a252d1ab117bd70
14712 ndao       20   0  936M 46124 20716 S  0.0  0.3  0:00.19 /home/ndao/.vscode-server/bin/d2e414d9e4239a252d1ab117bd70
14713 ndao       20   0  936M 46124 20716 S  0.0  0.3  0:00.12 /home/ndao/.vscode-server/bin/d2e414d9e4239a252d1ab117bd70
14714 ndao       20   0  936M 46124 20716 S  0.0  0.3  0:00.09 /home/ndao/.vscode-server/bin/d2e414d9e4239a252d1ab117bd70
14715 ndao       20   0  936M 46124 20716 S  0.0  0.3  0:00.00 /home/ndao/.vscode-server/bin/d2e414d9e4239a252d1ab117bd70
F1Help  F2Setup F3SearchF4FilterF5Tree  F6SortByF7Nice -F8Nice +F9Kill  F10Quit
```

Figure 5.2: Number of cores using htop

i. At what point does the speedup stop increasing?

Speedup stop increasing at after 10 threads.

ii. Is this consistent with the number of cores?

As seen in fig. 5.2 there are 12 available cores, hence it is consistent with the number of cores as the speedup seems to stop increasing after 10 threads which is close to the number of available cores.