
Lab report 3

Nhan Dao

November 2, 2020

CONTENTS

1. Monte Carlo method for approximating the value of π in OpenMP	1
a. Compile and run the template code with a single thread, and thus verify that the serial version is correct.	1
b. Parallelize the code for multi-threaded execution by ONLY adding OpenMP directives	1
i. At what point in the code new threads are forked and joined?	1
ii. Which part of the code is executed by which threads?	1
iii. Where barriers if any in terms of thread synchronization would be encountered, and hence how the code achieves parallelization	2
c. Tabulate the speedup and efficiency for 1,2,4,8 threads.	2
i. Do the result match your expectation? Why?	2
2. Trapezoidal rule to approximate area under $f(x) = x^4$ in OpenMP	3
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.	3
b. Parallelize the code for multi-threaded execution by adding to it as few OpenMP directives as possible.	3
i. At what points in the code new threads are forked and joined?	4
ii. Which parts of code is executed by which threads?	4
iii. Where barriers if any in terms of thread synchronization would be encountered, and hence how the code achieves parallelization?	4
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.	4
i. Do the result match the expectation? Why?	5
ii. Explain why the output differs slightly from run to run for multiple thread?	5
3. Parallel histogram generation using OpenMP	6
a. Compile and run the template code with a single thread to verify that the serial version is correct	6
b. Parallelize the code for multi-thread execution by ONLY adding OpenMP directives	6
i. Explain how your code achieves parallelization.	7
ii. Describe at what points new threads are forked and joined and which part of the code are executed by which threads.	7
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	7
4. Parallel Count Sort using OpenMP	8

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

```
1 # pragma omp parallel num_threads(thread_count) // OMP parallel
2 {
3     int my_rank = omp_get_thread_num();
4     unsigned seed = my_rank + 1;
5     long long int toss;
6     double x, y, distance_squared;
7
8 # pragma omp for reduction(+ : number_in_circle) // OMP for
9     for(toss = 0; toss < number_of_tosses; toss++) {
10         x = 2*my_drnd(&seed) - 1;
11         y = 2*my_drnd(&seed) - 1;
12         distance_squared = x*x + y*y;
13         if (distance_squared <= 1) number_in_circle++;
14     }
15 }
```

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 environment. Master and slaves thread will run the section of code assigned by the OpenMP directives - line 2 - 15.

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

Thread synchronization is needed so reduction can be applied 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 π using Monte Carlo's method with 10^9 samples

i. Do the results match your expectation? Why?

The results match our expectation. The speedup will increase as the number of threads increases but will saturate at some point 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 = 2.000000000000010e-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.**

```
1 /*
2  * Function:    Trap
3  * Purpose:    Use trapezoidal rule to estimate definite integral
4  * Input args:
5  *   a: left endpoint
6  *   b: right endpoint
7  *   n: number of trapezoids
8  * Return val:
9  *   approx: estimate of integral from a to b of f(x)
10 */
11 double Trap(double a, double b, int n, int thread_count) {
12     double h, approx;
13     int i;
14
15     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.

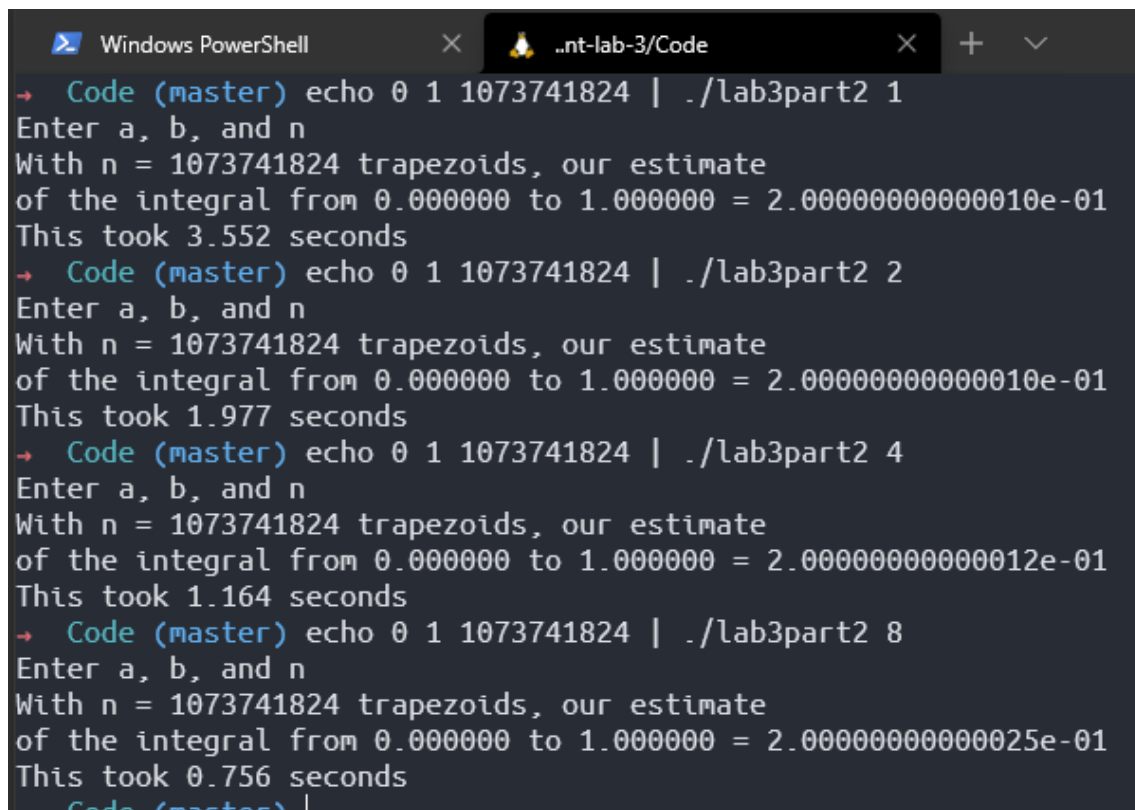
Threads	Time (s)	Speedup	Efficiency (%)
1	3.439	N/A	100
2	1.920	1.791	89.557
4	1.252	2.747	68.67
8	0.754	4.561	57.013

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 there 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 multiple thread?

A screenshot of a Windows PowerShell terminal window. The window title bar shows 'Windows PowerShell' and the current directory is '..nt-lab-3/Code'. The terminal displays the execution of a program named './lab3part2' with different thread counts. Each run prompts for 'a, b, and n' (which are 0, 1, and 1073741824 respectively) and shows the estimated integral value and execution time. The results are as follows:

Threads	Estimate	Time (seconds)
1	2.000000000000010e-01	3.552
2	2.000000000000010e-01	1.977
4	2.000000000000012e-01	1.164
8	2.000000000000025e-01	0.756

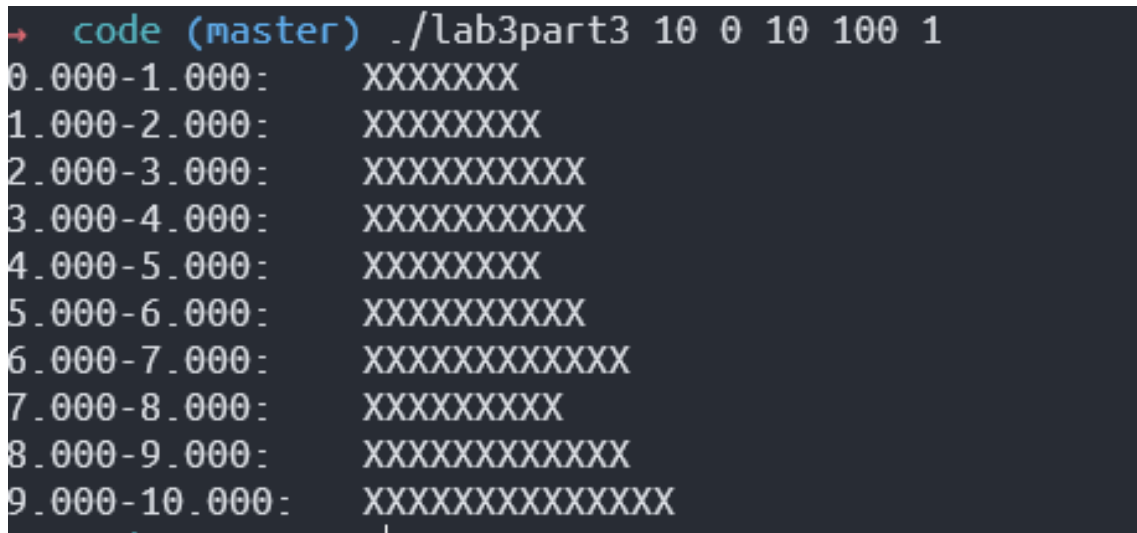
Figure 2.2: Terminal output of execution for varying number of threads

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:  XXXXXXX
2.000-3.000:  XXXXXXXXX
3.000-4.000:  XXXXXXXXX
4.000-5.000:  XXXXXXX
5.000-6.000:  XXXXXXXXX
6.000-7.000:  XXXXXXXXXX
7.000-8.000:  XXXXXXX
8.000-9.000:  XXXXXXXXXX
9.000-10.000: XXXXXXXXXX
```

Figure 3.1: Terminal output for execution with a single thread, using a data range between 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 3 shows the changes within the template code to parallelize by only adding OpenMP directives.

```
1  /* Count number of values in each bin */
2  #pragma omp parallel num_threads(thread_count)
3  {
4      int j, my_rank = omp_get_thread_num();
5      int my_offset = my_rank*bin_count;
6
7      #pragma omp for private(bin)
8      for (i = 0; i < data_count; i++) {
9          my_rank = omp_get_thread_num();
10         bin = Which_bin(data[i], bin_maxes, bin_count, min_meas);
11         #ifdef DEBUG
12             printf("Th %d > i = %d, data = %4.3f, bin = %d\n", my_rank, i,
13                    data[i], bin);
14         #endif
15         loc_bin_counts[my_offset + bin]++;
16     }
```



```

17     #pragma omp for
18     for (i = 0; i < bin_count; i++)
19         for (j = 0; j < thread_count; j++) {
20 #         ifdef DEBUG
21             printf("Th %d > Adding %d to bin_counts[%d] = %d\n",
22                 my_rank, loc_bin_counts[j*bin_count + i], i,
23                 bin_counts[i]);
24 #         endif
25             bin_counts[i] += loc_bin_counts[j*bin_count + i];
26         }
27     }

```

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 3. 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 histogram counter in a form of an array. Here, the variable *bin* must be declared private which makes a separate 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 3) and they are joined once the parallel environment ends (line 25 listing 3). 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 between 0 and 10, a total of 10^7 data points, and 10 equally spaced bins.

4. PARALLEL COUNT SORT USING OPENMP