Joshua Catoe

CSCI 473

4/1/20

PTH07 – Report

**I. Implementation**

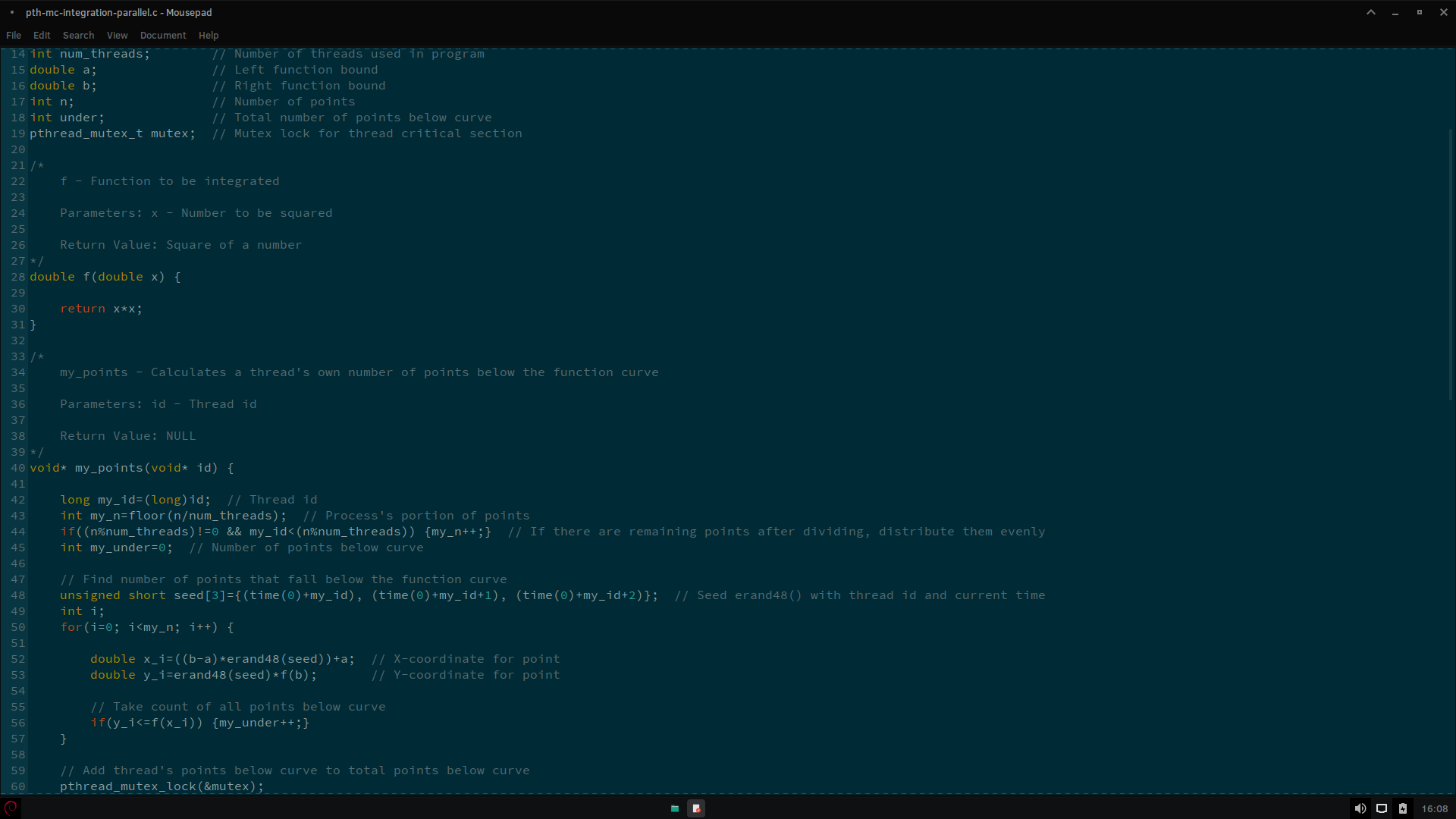
The program pth-mc-integration-parallel (**Figure 1**) implements the Monte Carlo integration method to evaluate . To review briefly, Monte Carlo integration is done by taking a sampling of random (x,y) coordinate points between the bounds of integration, equating the ratio of the area under the function curve over the total area of the function plane to the ratio of the number of points below the function curve over the total number of points taken and solving for the area under the function curve. Compared to the prior program mc-integration-parallel, the implementation of Monte Carlo is largely identical, however here it is done using Pthreads threads rather than MPI processes.

Toward the top of the program, the global variables *num\_threads, a, b, n, under* and *mutex* are defined, as they need to be accessed in both *main()* and the thread function *my\_points()*. The function to be integrated, *f(x),* is defined as taking in a double value and returning the square of that value. The function *my\_points()* is defined as taking in a void pointer and returning a void pointer (in this case NULL), in accordance with the specification of the *pthread\_create()* function. This is the routine that each thread runs during the course of program execution. The void pointer parameter *id* is converted into the long integer value *my\_id*; the variable name *id* is used instead of *rank* to more easily differentiate a thread-based program from an MPI-based program. Each thread’s specific number of points to test, *my\_n*, is determined by taking the floor function of the total number of points divided by the number of threads. If the total number of points is not evenly divisible by the number of threads, each thread whose ID is less than the remainder of that division will receive an extra point. The random number generator is then seeded using an array consisting of three instances of the current time plus the thread’s ID plus an extra number (only in the last two instances) to ensure that each thread will generate different random numbers. For each point, random (x,y) coordinates are determined using *erand48()* instead of the typical *drand48()*, in order to be thread safe; if the y-coordinate lies on or below the function curve, the count of points below the function curve is increased. Each thread then updates the global count of points under the function curve by adding their own count to it. This process of updating the global count is enclosed in a mutex to ensure that only one thread will update the count at a time, avoiding race conditions and other issues that may arise.

In *main()*, space is allocated for an array of threads, followed by initializing the mutex that will be used in the *my\_points()* function. The threads themselves are then created, looping through the *threads* array to create each one; after they have finished their respective tasks, they are joined back together. Next, the main process calculates the total area of the function plane followed by the area under the function curve, using the *under* variable that was calculated by the threads. Finally, the mutex is destroyed and the memory used by the *threads* array is freed. Since the *MPI\_Wtime()* function was not available, the *GET\_TIME()* macro was used to time the program.

It should be noted that converting to/from a void pointer was done using a long integer. The reason for this was simply to avoid a compiler warning dealing with converting integers of different sizes. On 64-bit systems, a void pointer is the same size as a long integer, thus doing this causes the warning to disappear. The program ran even with the warning, however this might not be the case on other systems, so this also adds safety to the program.

**Figure 1: pth-mc-integration-parallel**

****

**A screenshot of a computer

Description automatically generated**

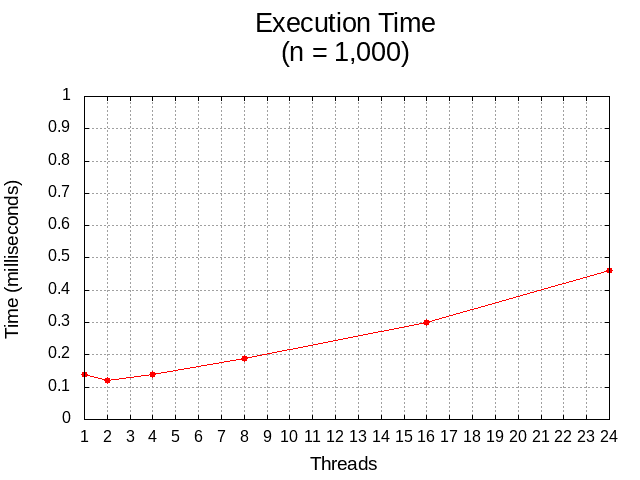
**Figure 1 (2)**

**Figure 1 (3)**

**A screenshot of a computer

Description automatically generated**

**II. Data Analysis**

**A close up of a map

Description automatically generated**

**A close up of a map

Description automatically generatedA close up of a map

Description automatically generated**

**Figure 2: Execution Time (Threads vs. Processes)**

**A close up of a map

Description automatically generatedA close up of a map

Description automatically generated**

A close up of a map

Description automatically generatedA close up of a map

Description automatically generated

A close up of a map

Description automatically generatedA close up of a map

Description automatically generated

A close up of a map

Description automatically generatedA close up of a map

Description automatically generated

**Figure 3: Speedup (Threads vs. Processes)**

A close up of a map

Description automatically generatedA close up of a map

Description automatically generated

A close up of a map

Description automatically generatedA close up of a map

Description automatically generated

**Figure 4: Efficiency (Threads vs. Processes)**

A close up of text on a white background

Description automatically generatedA close up of text on a white background

Description automatically generated

The trends in execution time (**Figure 2**), speedup (**Figure 3**) and efficiency (**Figure 4**) were almost identical to those found in the previous program; those plots are included here to illustrate this. Execution time was as expected, increasing as the number of threads increased at a small problem size (n = 1,000) and decreasing as the number of threads increased at larger problem sizes (n = 1,000,000 and 1,000,000,000). The speedup at n = 1,000 was normal, decreasing as the thread count increased. At n = 1,000,000, speedup drifted slightly away from the ideal until it plateaued at 16 threads. At n = 1,000,000,000, speedup was ideal on 1, 2, 4 and 8 threads and started to rise slightly above the ideal at 16 and 24 threads. Efficiency at n = 1,000 was typical, decreasing with an increasing thread count. At n = 1,000,000 efficiency dropped from 100% on one thread to just above 90% on four threads where it stayed until dropping much further to around 60% on twenty-four threads. The program was extremely efficient at n = 1,000,000,000, staying near 100% until eight threads and rising to just under 102% on sixteen threads.

Regarding accuracy, pth-mc-integration-parallel was just as accurate as the previous program. Using the integral as an example, the analytical solution is or approximately 333.333. At n = 1,000 on twenty-four threads, the result was approximately 365. At n = 1,000,000 on twenty-four threads, the result was approximately 333.226. At n = 1,000,000,000 on twenty-four threads, the result was approximately 333.319, nearly the same as the analytical solution.