Joshua Catoe

CSCI 473

3/15/20

MPI06 – Report

**I. Monte Carlo**

A Monte Carlo method, in the context of computational science, is an algorithm that aims to solve a deterministic problem using an estimation found through the use of repetitive random sampling (Pease). The specific method discussed here is Monte Carlo integration. This type of integration is done by taking a number of random (x,y) coordinate points within the bounds of integration, determining how many of those points fall below the function curve, taking the total area of the rectangle formed by the integration bounds, and finally solving for the area under the curve by equating the ratios of total area to area under the curve and the number of points under the curve to the total number of points taken. See **Figure 1** for a visual representation.

(xi,yi)…

(xi,yi)…

A

T

b

a

**Figure 1: Monte Carlo Integration**

The programmatic implementation of this integration was done in both a serial and parallel fashion. **Figures 2** and **3** show the serial implementation, while **Figure 4** shows the parallel implementation. The parsing and printing of command-line arguments will not be discussed here, as they are rather straightforward and not as important as the main idea of these programs.

A screenshot of a computer

Description automatically generated

**Figure 2: Serial Implementation**

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

**Figure 3: Monte Function**

The serial implementation, mc-integration-serial, was done mainly through a function called monte(), defined in utilities.c and shown in **Figure 3**. This function takes in as parameters the bounds of integration, the number of points to test, the function to be integrated and a pointer to where the final area value will be stored. First, the total area is calculated by multiplying the width (right bound minus left bound) by the height (where f(x) is at the right bound). Next, the random number generator is seeded using the current time. Then, the main loop of Monte Carlo integration is executed; for each point, a random number is chosen using drand48() for both the x and y coordinate of the point, and if the point lies on or below the function curve, the count of points below the curve is increased by one. Finally, the area under the curve is calculated using the equation from **Figure 1**. In the main program, the Monte Carlo integration was calculated by calling the monte() function. The serial version of the program was timed using the macro GET\_TIME to obtain the start and end times, subtracting the start time from the end time to get the total program run time.

A screenshot of a computer

Description automatically generated

**Figure 4: Parallel Implementation**

The parallel implementation of the program, mc-integration-parallel, shown in **Figure 4**, is largely the same as the serial version, with a few parallel-specific changes. The first change was using the MPI function MPI\_Wtime() to time the program rather than the GET\_TIME macro. This was done since MPI was already being used, making it more natural to use a pre-defined function rather than a custom macro. The second and most substantial change was to the Monte Carlo portion of the program. Instead of one process testing all of the points, multiple processes tested their own portion of the points and reported their count of points under the curve to the root process, in this case process zero. Each process’s portion of points was calculated by taking the floor function of the total number of points divided by the number of processes. If the total number of points was not evenly distributed among the processes, then each process whose rank was less than the total number of processes received one extra point in order to distribute the remaining points as evenly as possible. The main Monte Carlo loop stayed mostly the same, except that the random number generator was seeded with the current time plus each processes rank to ensure different random numbers for each process, and each process’s local count of points under the curve was incremented instead of the total count. The total number of points under the curve was calculated by using the MPI\_Reduce() function and MPI\_SUM operator to send each process’s count of points under the curve to the root process and having the root process sum them all. The root process then calculates the area under the curve the same way as in the serial version, using the sum that was just calculated. It should be noted that the monte() function was not used in this parallel implementation, as it would have been much more complicated to implement parallelism into said function, and would have involved using another MPI environment, thus it was deemed more efficient to implement the Monte Carlo process directly in main().

**II. Comparison**

Although the methods of Monte Carlo integration and standard analytical integration are very different, they both aim to reach the same conclusion, thus a comparison between the two is appropriate. The integral in question in both of the aforementioned programs is . The bounds of integration are arbitrary, so for this comparison a=0 and b=10, thus the integral is now . The analytical solution to this integral is . While Monte Carlo integration may not be one-hundred percent accurate to the analytical solution, it should be fairly close, especially with a high number of points. This is to be expected, as Monte Carlo is an estimation, not an exact answer. Running the parallel program with the same integral and the number of points n=1,000 results in an approximation of 327, an error of 6.333; this is not extremely inaccurate, but not extremely accurate, either. Using the number of points n=1,000,000 results in an approximation of 333.055, an error of 0.278 and almost exactly the same as the analytical solution. Using the number of points n=1,000,000,000 results in an approximation of 333.376, an error of -0.043 and even more accurate to the analytical solution. As stated previously, the Monte Carlo integration method is different from the analytical method, but these results show that it is a very viable option.

**III. Data Analysis**

Overall, the data collected from several runs of the program, using several different values of n, yielded very interesting results. For n=1,000 (**Figure 5)**, the program’s execution time stayed very low and near-zero until sixteen processes were used, where it took a huge leap to 2.8 milliseconds, only to drop to near-zero once again at twenty-four processes. This pattern is seen again to a much lesser degree for n=1,000,000 (**Figure 6**), where the execution time falls only to rise again at sixteen processes and fall again to twenty-four processes. A possible explanation for this is that sixteen processes did not lend itself to the internal mathematics, thus taking more time for calculations. The execution time for n=1,000,000,000 (**Figure 7**) was as expected, falling as the number of processes increased.

The speedup for n=1,000 (**Figure 8**) was strikingly low, dropping below 1x after four processes and rising to around 1x at twenty-four processes. For n=1,000,000 (**Figure 9**), the speedup expectedly rises until sixteen processes, dropping from 7x to just below 5x, and rises again at twenty-four processes, thus reflecting the pattern seen in the first two execution time plots. The speedup for n=1,000,000,000 (**Figure 10**) was excellent, staying almost exactly at the ideal speedup as the number of processes rose.

Efficiency at n=1,000 (**Figure 11**) was extremely low, plummeting from 100% at two processes and reaching an abysmal 0.1% at sixteen processes. Such low efficiency suggests that a small problem size is not ideal for a large number of processes. The efficiency at n=1,000,000 (**Figure 12**) fares a bit better, staying at 100% until eight processes, dropping to just under 30% at sixteen and rising to just under 60% at twenty-four processes. These two plots also reflect the sixteen processes pattern seen in the execution time and speedup. The efficiency at n =1,000,000,000 (**Figure 13**) was outstanding, staying nearly at 100% the entire time.

**A close up of a map

Description automatically generated**

**Figure 5: Execution Time (n=1,000)**

**A close up of a map

Description automatically generated**

**Figure 6: Execution Time (n=1,000,000)**

**A close up of a map

Description automatically generated**

**Figure 7: Execution Time (n=1,000,000,000)**

**A close up of a map

Description automatically generated**

**Figure 8: Speedup (n=1,000)**

**A close up of a map

Description automatically generated**

**Figure 9: Speedup (n=1,000,000)**

**A close up of a map

Description automatically generated**

**Figure 10: Speedup (n=1,000,000,000)**

**A close up of a map

Description automatically generated**

**Figure 11: Efficiency (n=1,000)**

**A close up of a map

Description automatically generated**

**Figure 12: Efficiency (n=1,000,000)**

**A close up of text on a white background

Description automatically generated**

**Figure 13: Efficiency (n=1,000,000,000)**

\* **Plots created using** gnuplot**.**

# Works Cited

Pease, Christopher. "An Overview of Monte Carlo Methods." *Towards Data Science*. 6 September 2018. 14 March 2020. <https://towardsdatascience.com/an-overview-of-monte-carlo-methods-675384eb1694>.