MPP - Final Project

Jake Webb

September 9, 2025

1 Overview - Monte Carlo Method

The Monte Carlo method is a computation method that uses randomly generated points to mimic statistical sampling. The idea of the method is to generate a significantly large number of points within a graphical region. Using those points, it is then possible to estimate the area of an object that lies within that region. This is done by taking the ratio of the number of points inside the object to the total number of points in the entire region, here called N. In general, as more points are generated, the estimation becomes more accurate to the true value. The important equations are as follows:

$(Area_{Object}/Area_{Region}) \approx (Points_{Inside}/N)$

$\lim_{N\to\infty}(Points_{Inside}/N) = (Area_{Object}/Area_{Region})$

The following image shows an example of the specific use case that will be demonstrated:

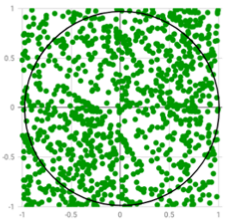

Figure 1: Enter Caption

In this example, the Monte Carlo method is employed to estimate the value of π using the area of a circle inside bounded by a square. The region bounds in this case are from x = -1 to 1 and y = -1 to 1, assuming x is the horizontal axis and y is the vertical axis. Therefore, the equation would be as follows:

$$(Area_{Circle}/Area_{Square}) \approx (Points_{Inside}/N)$$

The following algebra can then be done to solve for the value of π:

$$(Area_{Circle}/Area_{Square}) = ((\pi * r^2)/(4 * r^2)) \approx (Points_{Inside}/N)$$

$$\pi \approx 4 * (Points_{Circle}/N)$$

This will be the foundational method for estimating π.

2 Outline

Based on the method given in Section 1, the following strategy will be used:

- Generate simple serial code
- Analyze where time and memory could be used more optimally
- Iterate on serial code
- Analyze advanced serial code
- Generate simple CUDA code
- Analyze simple CUDA code
- Generate final CUDA code
- Seek future improvements

3 Methods

3.1 Method 1 - Simple Serial Code

The goal of the simple serial code is only to achieve an estimate for π that is reasonable. It is okay for this method to be imperfect as long as there is room for iterations to be made. Upon planning and analyzing, the following parameters were deemed necessary to consider: user input, random variables, calculating circle bounds, and counting methods.

In general, the program will be run using methods described in class, including a Makefile and an mpprun shell method.

3.1.1 Input

There must be a way for the user to input their desired value of N. Because the estimated value for π theoretically becomes more accurate as N approaches infinity, it is important that the user be able to utilize the desired number of sampling points, thus essentially (though not necessarily due to statistics) choosing their accuracy. This input method will not be iterated upon during following methods due to the lack of necessity and relevance.

The selected method was chosen to have a default value for N predetermined in the code, while allowing the user to alter this value if decided. This was done by using the method demonstrated in class:

```
mpprun monte-carlo.sh m
```

The main program function has a character argument that is able to be accessed by the user when the shell script is run. If there is no argument inputted by the user, the value will be set to N=100,000,000 points. If an argument is inputted by the user, here called m, N will be set equal to m. N is an unsigned integer, so this will be valid as long as m<($2^3$2-1). If a user inputs more than one argument, the program will generate an error and terminate the program. The shell script must be modified using the nano command in order to make this possible. The following code was used inside the shell script, monte-carlo.sh to achieve this:

```
/home/webbj/code/MPPFinal/monte-carlo
/home/webbj/code/MPPFinal/monte-carlo <m>
```

In this case, <m> was used as the user input argument.

3.1.2 Generating Random Variables

In order to do the calculations, points must be generated within the regions. This will be done using the C command rand(), which is a part of the <stdlib.h> library. In C, rand() is used to generate an integer between 0 and a built-in integer called RANDMAX. The following code is a typical way to generate a random floating-point value between 0 and 1:

```
x=(float)rand() / (float)RANDMAX;
```

Now the values for x are in the boundary [0,1]. Next, this value must be multiplied by 2 in order to have it match our desired range of 2. Range can be calculated using the following equation:

$$range = max_{bounds} - min_{bounds}$$

Now the values for x are in the boundary [0,2]. To match our real bounds of [-1,1], a value of 1 must be subtracted from it. The following steps are shown here:

```
x=(float)rand() / (float)RANDMAX;
x=2*x;
```

```
x=x-1;
```

These steps can then be combined to create the following equation:

```
x=((float)rand() / (float)RANDMAX) * 2 - 1;
```

One other thing must be considered when discussing rand(), and that is seeding. In order for this to truly be a helpful simulation, the values for rand() must not be exactly the same every time the program is run. This can be done using the srand() command. A typical way to generate a different seed each time a program is run is to include the <time.h< library and use the following code:

```
srand(time(0));
```

This sets the seed to the current time on every iteration of the program.

3.1.3 Storing Generated Random Variables

In order to do work on these generated random variables, the variables must be saved somewhere. In this method, an array of floats that has length N is created for both x and y. The generated random variables equation occurs N*2 times, once for x and once for y, and on each iteration, the value is stored into the array. An example of this is shown for purposes of clarity:

```
for(int i=0; i<N; i++){
    x[i]=((float)rand() / (float)RANDMAX) * 2 - 1;
    y[i]=((float)rand() / (float)RANDMAX) * 2 - 1;
}
```

These values will then be used to calculate the number of points that fall within the bounds of the circle.

It is also helpful to have an array of boolean values that state whether or not the pair of points are inside the circle. This array should be initialized with values that are all false, so they can be updated to true if the point is inside the circle. This will be shown in the next step.

3.1.4 Points Inside Circle

The next step is to consider whether or not the generated point is within the circle. This can be done using the following equation to calculate the distance of the generated point from the origin:

d=$\sqrt{(x^2 + y^2)}$

Because the radius of the chosen circle is 1 unit, it can be determined that the point is inside the circle if d<1. If this is true, then the array of boolean values can be updated. An example of this is shown:

```
d=x*x + y*y;
    if (d < 1) {
boolarray[i]=true;
}
```

Next, a counting variable will be iterated in order to calculate the number of points inside the circle.

3.1.5 Counting

A counting variable will be used in this method to be used in the calculation. This is a similar strategy to the boolean array previously mentioned. A loop will be iterated N times, and whenever the boolean array is true, the counting variable will be incremented. An example of this is shown:

```
for (int i=0; i<N; i++) {
    if (boolarray[i]==true){
        count+=1;
    }
}
```

This value can then be used in the final step.

3.1.6 Estimation

Recall the equation that is used to estimate π:

$$\pi \approx 4 * (Points_{Circle}/N)$$

This value can then be output to the user.

3.1.7 Summary and Time Analysis

The output that was generated from these steps was the following:

```
–Simple Serial–
Setting up... 0.000014 seconds
Computing... 1.640576 seconds
Pi Estimate: 3.141745
```

Upon analysis, many strategies can be improved. Currently the following significant lines of code are run:

- Generate Random Points: 2*N
- Check Point Distance: 2*N (counting the if statement)
- Increment Count: 2*N

In number of floating operations, this can be shown as the following:

- Generate Random Points: 8*N
- Check Point Distance: 6*N

Currently, most of the time being lost is clearly in the computing step. The best strategy to improve this is to simply use less for loops. Currently, there are 3 for loops in the method. If this could be minimized, the number of lines of code could be lessened. The next method will explain the steps taken to achieve this.

It is also possible to be more efficient with the memory that is used. In method 1, an array is used for the points generated for x and y, and an array is used to check if those points are within the circle. The next method will improve on these strategies.

3.2 Method 2 - Advanced Serial Code

3.2.1 Improvements

As mentioned, there are two obvious ways to improve upon method 1.

The first obvious way is to decrease the number of instructions that are used. This can be done by combining the "Points Inside Circle" section with the "Counting" section. The desire here is to eliminate the array of boolean values completely. This can be done by simply increasing the count value rather than setting a value true when the distance from the origin is determined to be less than 1.

The next obvious way is to improve the memory usage of the program. This can be done by eliminate the array of x and y points by simply replacing the previous value for x on iteration. This requires the combination of the two previously mentioned sections with the "Storing Generated Random Variables Section."

These two ideas combine for the following code:

```
for(int i=0; i<N; i++) {
    x=((float)rand()/(float)RANDMAX)*2-1;
    y=((float)rand()/(float)RANDMAX)*2-1;

    d=(x*x+y*y);
```

```
        if (d<1) {
            count+=1;
        }
    }
```

3.2.2 Summary and Time Analysis

The output that was generated from these steps was the following:

```
–Advanced Serial–
Setting up... 0.000000 seconds
Computing... 1.172259 seconds
Pi Estimate: 3.141628
```

Upon analysis, the strategies seen can quickly be seen. There is less time in the setup portion of the code because the program doesn't have to allocate a large amount of memory for the three arrays. The computing step has also sped up because the number of instructions has decreased. This can be seen by the following analysis in terms of significant lines of code, as the total number of instructions adds up to 4*N rather than 6*N. When working with significantly large values, this can matter. The analysis is shown:

- Generate Random Points: 2*N
- Check Point Distance: 1*N
- Increment Count: 1*N

Unfortunately, floating point operations do not decrease. The analysis for floating point operations remains the same:

- Generate Random Points: 8*N
- Check Point Distance: 6*N

Since setup time has now been improved, computation time will be increased next using CUDA.

3.3 Method 3 - Simple CUDA Code

In method 3, some of the setup improvements mentioned will have to be abandoned in order to significantly improve calculation time. The strategy in this method is to use threading in order to run all of the checks for d¡1 at the same time. To do this, an array for x, y, and count must be reimplemented. This is because rand() is not a part of the CUDA library, it is a part of the ¡stdlib.h¿ library. That means that it is not possible to generate random variables in the kernel using this method.

First, memory will be allocated. Then, random variables will be generated and stored. Next, all of the checks for d¡1 will be done at once. Finally, the count array will be iterated through in a similar fashion to the first method. This is done for simplicity and will be improved later.

3.3.1 CUDA

The standard procedure for CUDA code is the following:

- Allocate host memory
- Allocate device memory
- Copy data from host to device
- Implement kernel
- Copy data from device to host
- Free memory

This procedure was implemented in both methods involving CUDA code.

3.3.2 Kernel

The code provided in the kernel was similar the following:

```
global void montecarlo(float *xarray, float *yarray, int N, int* countarray) {
    int index=blockIdx.x*blockDim.x + threadIdx.x;

    float x=xarray[index];
    float y=yarray[index];
    d = x*x + y*y;

    if (index < N) {
        if (d<1) {
            countarray[index]=1;
        } // d<1
    }// if in scope
}// function
```

Once the program returns to the main function, it follows a similar procedure to the “Counting” and “Estimation” sections in Method 1.

3.3.3 Summary and Time Analysis

The output that was generated from these steps was the following:

```
–Simple CUDA–
Setting up... 1.263448 seconds
```

```
Allocating device memory... 0.154688 seconds
Copying data from host... 0.048824 seconds
Launching kernel... 0.012823 seconds
Copying data from device... 0.015728 seconds
Verifying... 0.011825 seconds
Pi Estimate: 3.141690
```

Upon analysis, a few things stick out. First of all, significant time has been lost in the setup stage. So much time is lost that this method is actually slower than Method 2. That being said, the kernel time is significantly faster. It is less than .013 seconds, when it was over a second long in Method 2. This is at the very least an upgrade in this area.

Unfortunately, there isn't a simple fix to the setup problem. Because random numbers cannot be generated inside the kernel using the method described, the most appropriate next step to lower the setup time is to remove the count array used in this step. This procedure is explained in the Method 4.

When it comes to instructions that are executed, the analysis is the following:

- Generate Random Points: 3*N
- Check Point Distance: 1
- Increment Count: 1*N

As is clear, this value of 4*N + 1 is 1 instruction larger than in Method 2. The goal for the next step is to reduce this.

When it comes to floating point operations, the analysis is the following:

- Generate Random Points: 8*N
- Check Point Distance: 4

Clearly, the computation step has decreased significantly. The number of floating point operations has decreased from 12*N to 8*N + 4. This will remain the same for Method 4.

3.4 Method 4 - Advanced CUDA Code

3.4.1 Kernel

The only difference between Method 3 and Method 4 is the use of the count array in Method 3 versus the use of atomicAdd() in Method 4. The atomicAdd() function allows for the ability to decrease setup time by not having to allocate memory for an array the size of N. Rather, the program can use an integer pointer that is updated by atomicAdd() and incremented whenever the case of d¡1 is true in the kernel. An updated snippet of the code is shown:

```
global void montecarloatomic(...,int* count) {
    ...
    if (index < N) {
        if (d<1) {
            atomicAdd(count, 1);
        }
    }
}
```

Note that ellipses are used to show code that is the same as Method 3.

Method 4 also has the advantage of now skipping the “Copy Data from Device” section because atomicAdd() utilizes integer pointers rather than copying data back and forth between the host and device.

3.4.2 Summary and Time Analysis

The output that was generated from these steps was the following:

```
–Advanced CUDA–
Setting up... 1.176481 seconds
Allocating device memory... 0.000554 seconds
Copying data from host... 0.032767 seconds
Launching kernel... 0.002325 seconds
Copying data from device... 0.000001 seconds
Verifying... 0.000064 seconds
Pi Estimate: 3.141595
```

Compared to Method 3, this method certainly speeds up the setup, though not significantly enough for it to be particularly viable. It is in fact slower than Method 2, though, this shows promise for future applications that require longer kernel times. Such applications will be discussed in the next section.

When it comes to instructions that are executed, the analysis is the following:
- Generate Random Points: 2*N
- Check Point Distance: 1
- Increment Count: 1

This is an improvement to two of the three sections for instructions executed. This is because the count variable is able to be used rather than the array. The “Generate Random Points” section decreased from 3*N to 2*N, while the “Increment Count” section decreased from 1*N to 1.

When it comes to floating point operations, the analysis is the following:
- Generate Random Points: 8*N

- Check Point Distance: 4

As noted before, these values did not change.

4 Taking it Further

First of all, there is a CUDA library called cudaRAND. Using this toolkit, it seems like it is potentially possible to do the desired task of generating random values inside of the kernel. Though it was not deeply analyzed in this project due to a lack of understanding of this library, further diving into this could be greatly beneficial in this area. Because this was by far the biggest drawback of Method 3 and Method 4, that would certainly be the next step taken if this is further pursued.

Another example of this would be to use the Monte Carlo method for a more computationally significant application. This application was to reflect on research that has already been enacted and to attempt to build further skill in CUDA programming. For that reason, it was mostly elementary in its application of the actual method. A more advanced application of the Monte Carlo like complicated integrals, statistical simulations, or any type of mathematical operation that involves integrals would likely show the desired improved time output.

5 Summary

In all, 4 implementations of the Monte Carlo method were discussed and experimented. Method 1 was a simple serial code, Method 2 was an advanced serial code, Method 3 was a simple CUDA code, and Method 4 was an advanced CUDA code. In the end, Method 2 was the best in this specific application, but Method 4 showed lots of promise for other Monte Carlo method examples or future implementations with newer CUDA libraries.

The task of estimating π was also deemed to be successful. The calculated values were within reasonable error of the “real” value, and all 4 methods showed similar results.

6 Works Cited

https://www.geeksforgeeks.org/dsa/estimating-value-pi-using-monte-carlo/
https://ggcarvalho.dev/posts/montecarlo/