Homework 05- Monte Carlo Neutron Transport

Ryan McCormick
Department of Computing Sciences
Coastal Carolina University
Conway, SC, USA
rlmccormi@coastal.edu

***Abstract*—Evaluating the speedup of parallel computing, over serial, on a problem that is "embarrassingly parallel", also called "perfectly parallel." This is when the parallel portion of a problem does not need to pass information between processes, and each one can simply run their portion of the problem. There is some computational overhead for the summation of the data, but this is minimal compared to the overall problem set. This paper dives into what the runtime vs process count, speedup, and efficiency of looks like for a problem being solved by the Monte Carlo method. This is a brute force method that lends itself very well to parallelism. Throughout the report that follows, there will be a walk-through of the problem that was being tested, an explanation of how these tests were conducted and a presentation of the results. This will include a breakdown of the formulas used and how the data was gathered and processed to create the graphs featured in this paper.**

***Index Terms*—efficiency, speedup, performance, MPI, Monte Carlo Method**

I. Introduction

Parallel systems, when used with code that is properly parallelized, can take a problem that would normally take a long time to run serially and accomplish the same task much faster. Some problems lend themselves very well to being parallelized, while others do not. When a problem can be parallelized, this can lead to huge reductions in compute time. Problems that see huge speedups from parallelization because they do not require any information to be passed between processes are called "embarrassingly parallel" or "perfectly parallel." A Monte Carlo simulation is a great way to see this kind of speedup. This type of experimentation method uses a process of repeated tests that leverages the law of large numbers to approximate the solution to a given problem. This project was looking at neutron transport and evaluating it through the use of Monte Carlo simulations. These simulations were done in a 2D environment to see if the neutrons would be absorbed, transmitted, or reflected by the plate. The next few equations were used in the simulations to evaluate the interaction of neutrons on a plate (H).

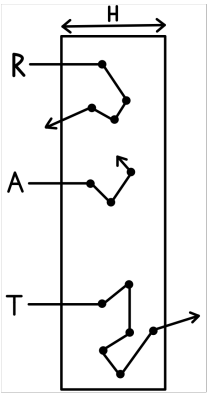
$$C = C_c + C_s \tag{1}$$

C : Total number of neutrons
C_c : Neutrons captured.
C_s : Neutrons scattered.

$$L = -(\frac{1}{C})ln(u) \tag{2}$$

L : Distance traveled
C : Number of neutrons interacting with the plate
u : A random number from the uniform distribution between- $(0, 1)$

Since the simulation was in a 2D space, the testing for the position of a neutron as it interacted with the plate was evaluated between 0 and π. There was no need to evaluate the up and down movement, only the horizontal.



[1]
This was done through the following equation:

$$x = x + Lcos(D) \tag{3}$$

x : position along the x-axis
L : Distance traveled
D : Given direction

This allowed for the evaluation of if a neutron was reflected, absorbed or transmitted, using the logic in Algorithm 1.

This paper will continue into greater detail about how the rest of this research was conducted and what process was followed. This paper is organized as follows. Procedure II covers the process and theory behind the experimentation, while Section III provides an outline for the software that was used. In Section IV, the outcomes of the research are

Algorithm 1 Logic for State of Neutron

Initialize var. D to 0
Initialize var. x to 0
Initialize var. u to random number in $(0, 1)$
Initialize var. B to TRUE
while B == TRUE **do**
  $L \leftarrow -(\frac{1}{C})\ln(u)$
  $x \leftarrow x + L \cdot \cos(D)$
  if $x < 0$ **then**
    $r \leftarrow r + 1$
    $B \leftarrow$ FALSE {REFLECTED}
  else if $x \geq H$ **then**
    $t \leftarrow t + 1$
    $B \leftarrow$ FALSE {TRANSMITTED}
  else if $u < C_c/C$ **then**
    $a \leftarrow a + 1$
    $B \leftarrow$ FALSE {ABSORBED}
  else
    $D \leftarrow u \cdot \pi$
  end if
  $u \leftarrow$ new random number in $(0, 1)$
end while

shown and discussed. Section V concludes the paper.

II. Procedure

When using Monte Carlo simulations for approximation and analysis of a problem, the benefit of implementing parallelism is that very little to no information needs to be passed once each process has its work. When considering the neutron transport problem, the approximation is performed by a for loop that contains the logic from the pseudo code in Algorithm 1. This loop then runs for a given number of times, incrementing the total of each result. The math can then be done to find the percentages of each result are these can be evaluated.

Monte Carlo simulations rely on the law of large numbers. This law is stated as follows:

> "...The LLN states that given a sample of independent and identically distributed values, the sample mean converges to the true mean" [2]

The LLN allows for testing of theories or approximation by repeating the test/experiment many times and averaging the results. The simulation that was tested in this project, as mentioned before, was the neutron transport of neutrons passed through a plate with given values for the following variables:

- H - Thickness of plate
- n - Number of neutrons to simulate
- A - Absorption factor for the plate
- C - Mean distance between neutron/atom interactions: $1/C$

These parameters were then passed to a program that could run the simulation serially and one that could run it in parallel. This allowed us to both asses the data from the simulations, but also to evaluate the effects that parallel computing on a problem like this can have on its runtime. This problem lends itself to parallelism because the number of neutrons n can be split between multiple compute nodes and the results added together before the percentages are computed. (The implementation of this will be discussed in the next section.)

The timing, efficiency and speedup were calculated using the following formulas:

$$S = \frac{T_{\text{serial}}}{T_{\text{parallel}}} \tag{4}$$

[3]

T_{serial} : Time to run the program with one processor
T_{parallel} : Time to run the program in parallel

$$E = \frac{S}{p} = \frac{(\frac{T_{\text{serial}}}{T_{\text{parallel}}})}{p} = \frac{T_{\text{serial}}}{p * T_{\text{parallel}}} \tag{5}$$

[3]

T_{serial} : Time to run the program with one processor
T_{parallel} : Time to run the program in parallel
p : The number of processors

These formulas allowed for analysis and graphing of the following metrics, that help when comparing the time, efficiency, and speedup compared to serial.

- Time vs Process Count
- Speedup vs Process Count
- Efficiency vs Process Count

To get the final data that was used in all of the calculations of this project, the program was executed on a compute node of the Expanse system. Expanse is a dedicated Advanced Cyberinfrastructure Coordination Ecosystem: Services and Support (ACCESS) cluster. This system belongs to the San Diego Supercomputer Center at the University of California San Diego and was designed with members of their team and Dell. The compute node that was used in testing had the following specifications:

- CPU Type - AMD EPYC 7742
- Nodes - 728
- Sockets - 2
- Cores/socket - 64
- Clock speed - 2.25 GHz
- Flop speed - 4608 GFlop/s
- Memory capacity - 256 GB DDR4 DRAM
- Local Storage - 1TB Intel P4510 NVMe PCIe SSD
- Max CPU Memory bandwidth - 409.5 GB/s

[4]

Running the program on this system made it possible to test the parallel nature of the program in an environment that is not replaceable on a personal machine. Testing from 1-64

processes on a high quality and isolated node allowed for higher precision than would be possible on a normal computer.

III. Code

This project started with writing a c program that could perform a Monte Carlo simulation for the neutron transport problem that has been discussed above in this paper. This was done using the logic from the pseudo code in Algorithm 1 above, and was implemented as seen in the code below:

```
for(int i = 0; i < n; i++){
        direction = 0.0;
        x_position = 0.0;
        bounce_status = true;
        while(bounce_status == true){
            u_Rnum = erand48(seed);
            L_dist = -(1/C) *
                ↪ log(u_Rnum);
            x_position += (L_dist *
                ↪ cos(direction));
            if(x_position < 0){
                (*local_reflect)++;
                bounce_status = false;
            }
            else if(x_position >= H){
                (*local_transmit)++;
                bounce_status = false;
            }
            else if(u_Rnum < (A/C)){
                (*local_absorb)++;
                bounce_status = false;
            }
            else{
                direction = u_Rnum *
                    ↪ M_PI;
            }
        }
    }
```

Listing 1. Neutron Transport

This code, as discussed before, lends itself to parallelism very well because the n can be divided between compute nodes using the Message Passing Interface (MPI). This division of the n was done using the following method, ensuring that each node had an even amount of work and that the extra was taken by the last node.

```
local_n = BLOCK_SIZE(id, np, n);
test_neutrons(C, A, H, local_n,
    ↪ &local_reflect, &local_absorb,
    ↪ &local_transmit);
```

Listing 2. Snippet from $main()$

The macro `BLOCK_SIZE()` was provided in An Introduction to Parallel Programming by Peter S. Pacheco from the University of San Francisco. The macro is as follows:

```
#define BLOCK_LOW(id,p,n)   ((id)*(n)/(p))
#define BLOCK_HIGH(id,p,n)
    ↪ (BLOCK_LOW((id)+1,p,n)-1)
#define BLOCK_SIZE(id,p,n) \
                (BLOCK_HIGH(id,p,n) -
                    ↪ BLOCK_LOW(id,p,n)+1)
```

Listing 3. Quinn Macros

Once the `local_n` had been found for each process, each process had their number of iterations for the simulation. This modified the start to the loop shown above in Listing 1 to be the following: `for(int i = 0; i < local_n; i++)`. After each process gathered its data, the use of `MPI_Reduce()`, as seen bellow, allowed for the values from each node to be consolidated before the values are returned to the user. Bellow is a snippet of the `MPI_Reduce()`:

```
MPI_Reduce(&local_reflect,
    ↪ &total_reflect, 1, MPI_INT,
    ↪ MPI_SUM, 0, MPI_COMM_WORLD);
MPI_Reduce(&local_absorb, &total_absorb,
    ↪ 1, MPI_INT, MPI_SUM, 0,
    ↪ MPI_COMM_WORLD);
MPI_Reduce(&local_transmit,
    ↪ &total_transmit, 1, MPI_INT,
    ↪ MPI_SUM, 0, MPI_COMM_WORLD);
```

All of this code in both the serial and parallel programs was timed so that the time, speedup and efficiency could be calculated and evaluated. The call to the `test_neutrons()` was surrounded in both programs, so that the overhead did not skew the data.

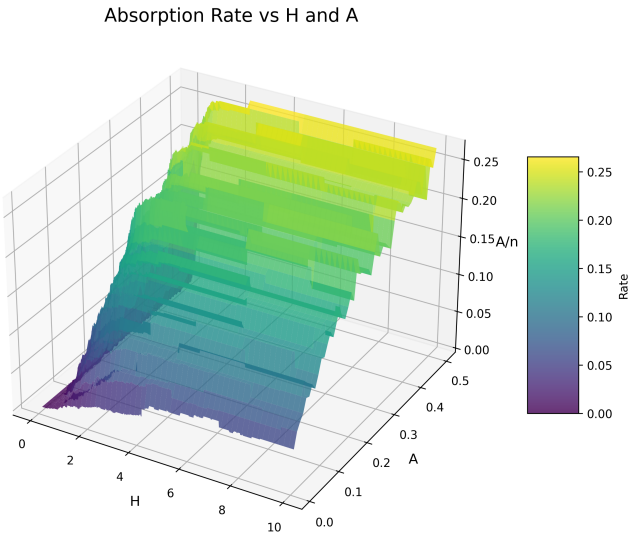
IV. Results

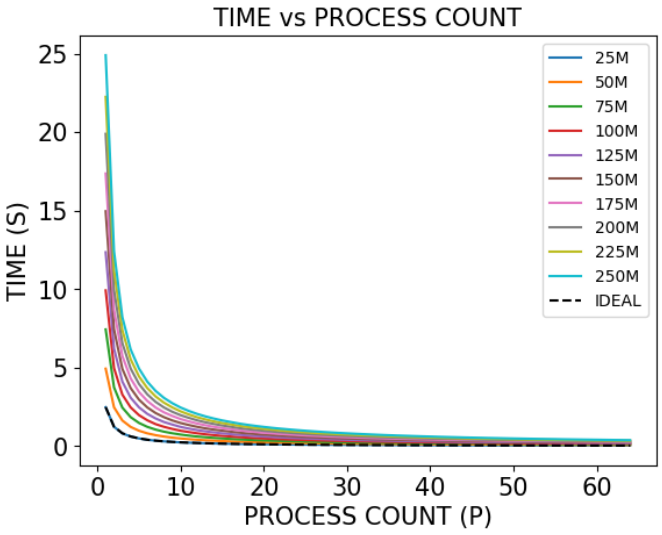
The first data that was collected was for the actual simulation that was being conducted. This data can be seen in the following graphs, (see left column of next page).

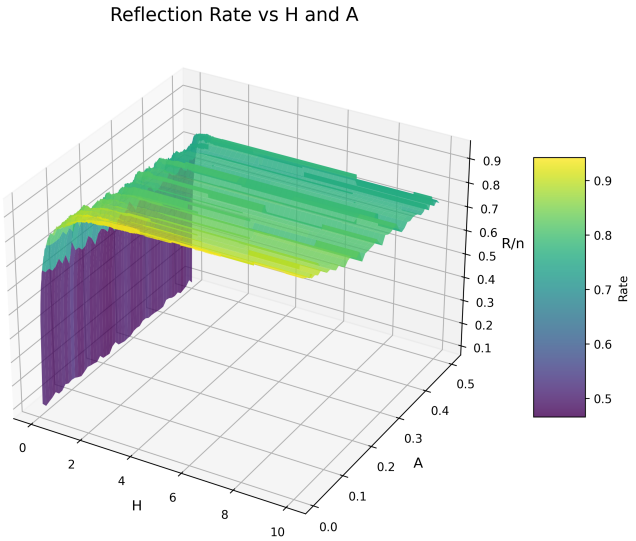
The next set of data what was evaluated was to calculate the time, speedup, and efficiency of the program between running it serially, in parallel, and then evaluating the difference in parallel with different numbers of processors. The processor range went from $(1-64)$ and the values of n ranged from 25 million to 250 million. See the graphs below (right side of the next page).

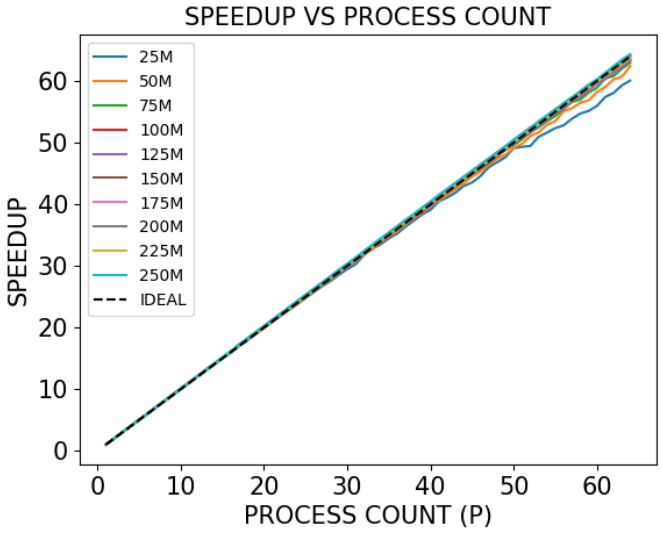
V. Conclusion

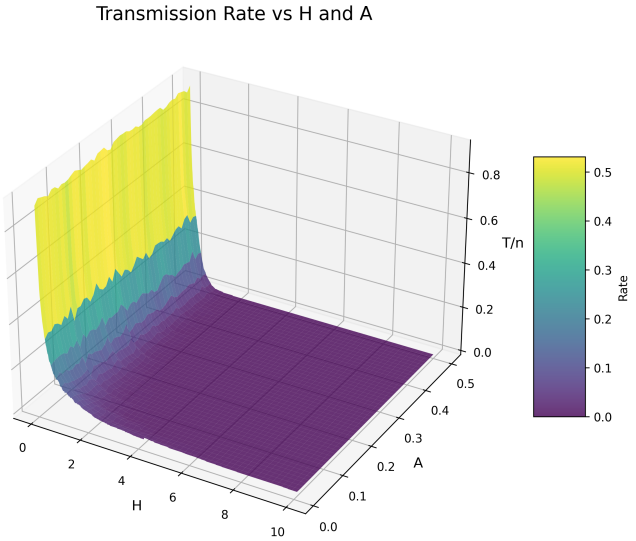
This project presented a great opportunity to test how parallel programs can be affected by compute resources and how the problem size can also affect how well something benefits from parallelization. Using Expanse and getting to run the program in an actual testing environment was also a great experience. The results of this project showed a super-linear speedup of the program when it is run in parallel, because it is ”embarrassingly parallel” or ”perfectly parallel”, as mentioned earlier. The results of the simulation showed what the absorption, transmission, and reflection go towards as the simple size increased.

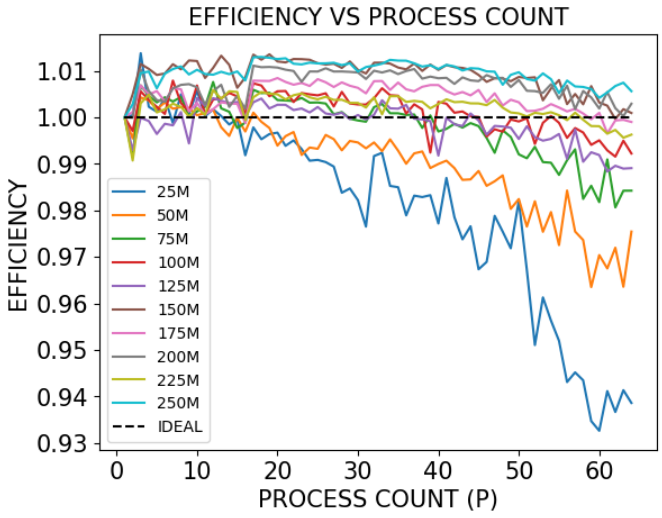

Absorption Rate vs H and A
A/n
H
A
Rate


TIME vs PROCESS COUNT
TIME (S)
PROCESS COUNT (P)
25M
50M
75M
100M
125M
150M
175M
200M
225M
250M
IDEAL


Reflection Rate vs H and A
R/n
H
A
Rate


SPEEDUP VS PROCESS COUNT
SPEEDUP
PROCESS COUNT (P)
25M
50M
75M
100M
125M
150M
175M
200M
225M
250M
IDEAL


Transmission Rate vs H and A
T/n
H
A
Rate


EFFICIENCY VS PROCESS COUNT
EFFICIENCY
PROCESS COUNT (P)
25M
50M
75M
100M
125M
150M
175M
200M
225M
250M
IDEAL

REFERENCES

[1] "Neutron transport image."

[2] Law of large numbers. [Online]. Available: https://en.wikipedia.org/wiki/Law$_o f_l arge_n umbers$

[3] P. S. Pacheco, *An Introduction to Parallel Programming*, 2nd ed. 50 Hampshire Street, 5th Floor, Cambridge, MA 02139, United States: Morgan Kaufmann Publishers, 2022.

[4] Expanse user guide. [Online]. Available: https://www.sdsc.edu/support/user$_g uides/expanse.htmltech_s ummary$