Parallel Scientific Computing, ME 471, Boise State University
March 2022

Parallelization of the Monte Carlo Method to Find Pi

Gavin Wale
Boise State University
BS Mechanical Engineering
https://github.com/gavinwale

ABSTRACT

The Monte Carlo method is a statistical approach to approximating integrals through repeated random sampling. Pi, π, is the ratio of the circumference of any circle to the diameter of that circle regardless of size. The irrational number is well defined and well known, which makes it the ideal value to test the power of parallel computation. In this academic project, a simple algorithm was programmed and scaled to account for up to 2^{28} run tests and up to 64 processors on Boise State University's R2 compute cluster. Strong and weak experiments were executed, testing the scalability of the program. It was found that the written algorithm scaled correctly. Attempts to optimize the algorithm are also documented.

NOMENCLATURE

π	Pi
S_p	Speedup on p processes
T_1	Serial execution time
T_p	Parallel execution time
r	Permanently serial code
T_c	Communication execution time
E_p	Efficiency
T_c	Communication execution time

Equations

Amdahl's Law: $S_p = \frac{T_1}{(1-r)\frac{T_1}{p} + rT_1 + T_c}$

Efficiency (strong): $E_p = \frac{S_p}{p} = \frac{T_1}{pT_p}$

Efficiency (weak): $E_p = \frac{T_1}{T_p}$

INTRODUCTION

Background

The Monte Carlo method uses randomly generated numbers as input to generate solutions and gain understanding of complex systems (Ref 1). The statistical method is used across information technology in applications from artificial intelligence to the stock market. With enough trials, strong correlations can be made to complex problems that are otherwise thought to be entirely random.

Project Objectives

The goal of this project is to apply the Monte Carlo method in a C algorithm to accurately predict the value of Pi. The goal is to analyze how the number of trials affects accuracy of the prediction and to test the parallel efficiency of the program itself. As test cases reach 9, 10 and even 11 digits, parallelization and optimization become important as to execute the program in the shortest amount of time possible.

Theory

Ideally, as the number of trials increases, the percent error from the actual value of Pi should decrease. This is a core concept of the monte carlo method. This will be tested serially and will be seen as parallel tests are run. Optimal performance of a parallel program is *linear speedup.* This can be shown by the following equation.

$$T_p = \frac{T_1}{p} \quad \text{thus,} \quad S_p = p$$

In order for scalability to be close to linear, it is necessary that the time spent on processor communication is considerably smaller than the total parallel execution time (shown below).

$$T_c \ll \frac{T_1}{p}$$

EXPERIMENTAL METHODS

Environment

Using a secure shell connection to Boise State's R2 compute cluster, I compiled both my serial and parallel programs in C and wrote bash scripts to run the program with a desired amount of processors, nodes, and trials on the supercomputer. The VSCode IDE was used in my local environment for most development while the VIM text editor was used to touch up on final details while within the virtual machine. Although redundant, as data was collected into text files, I saved the data I needed into convenient CSV files in order to be easily interpreted by MATLAB and plotted to show results.

Procedure

Each time a program is submitted to R2's queue, you can make sure your code was executed using the *squeue* command. After this, you will have an output file with the name you designated followed by the R2 job number. To see how to do this, visit the github repository linked in references. These files can be conveniently downloaded to a local workstation using the *scp* command or a linux tool like MobaXTerm. From there, the data can be analyzed in a spreadsheet, MATLAB, or any other convenient and effective way.

Data Reduction

Due to the nature of the Monte Carlo method and the random seeds of my program's generated numbers, there was no need to run multiple trials for each evaluation. Every set of data was representative of how efficient the program was and how accurately the algorithm could calculate Pi. Data was collected in the form of execution time, accuracy, number of trials, and number of processors and reduced to demonstrate efficiency and speedup.

RESULTS

Experiment Results

All plots and data analyzation will be shown in the following pages for convenience and clarity. The plots and discussions surrounding them were generated with MATLAB mlx files.

REFERENCES

[1] Britannica, The Editors of Encyclopaedia. "Monte Carlo method". *Encyclopedia Britannica*, 10 Jan. 2022, https://www.britannica.com/science/Monte-Carlo-method. Accessed 1 March 2022.

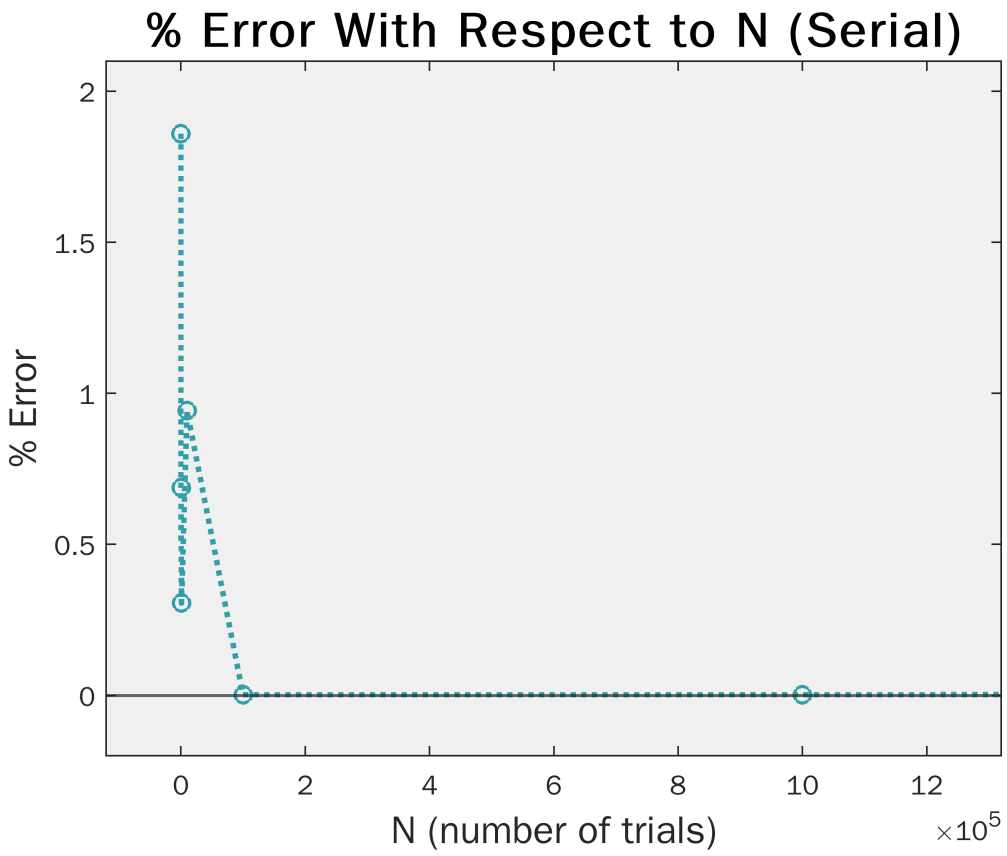
[2] "C Programming Language." 1 March 2022. https://devdocs.io/c/.

[3] "What is Monte Carlo Simulation?" IBM. 1 March 2022. https://www.ibm.com/cloud/learn/monte-carlo-simulation.

[4] "MPICH." 1 March 2022. https://www.mpich.org/documentation/.

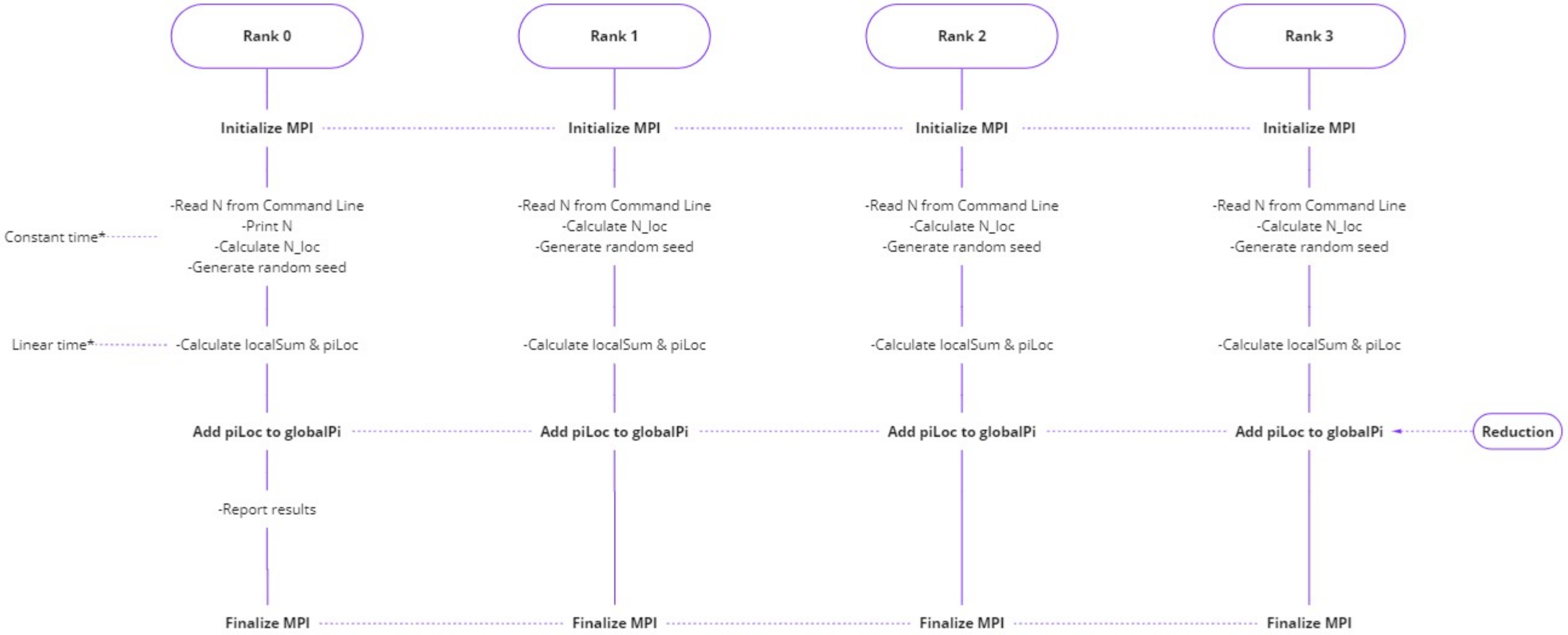
To test the logical validity of my Monte Carlo algorithm, I first implemented it serially to prove that as trials increase, percentage of error from the actual value of Pi decreases.

```
%Read serial trials from csv
sData=readmatrix('serialData.csv');
col1=sData(:,1);
col2=sData(:,2);

%Plot data with a line on the X-axis to visualize convergence
plot(col1,col2,':o','Color',1/255*[50 158 168],'LineWidth',2);
yline(0,'LineWidth',1);
set(gca,'color',1/255*[240 240 240])

%Details
title('% Error With Respect to N (Serial)','FontSize',18);
xlabel('N (number of trials)','FontSize',14);
ylabel('% Error','FontSize',14);
xlim([-120000 1320000])
ylim([-0.2 2.1])
```



It can be observed that as the number of trials increases exponentially, the error flattens at arounf 0 after a value of 10,000 N is read. It is safe to assume that it takes 100,000 trials to consistently generate Pi with accuracy through the C rand() function using a consistent seed. This is a principal aspect of the Monte Carlo method: as the number of trials increases, the higher likelihood the outcome will converge to a value (in this case, Pi).

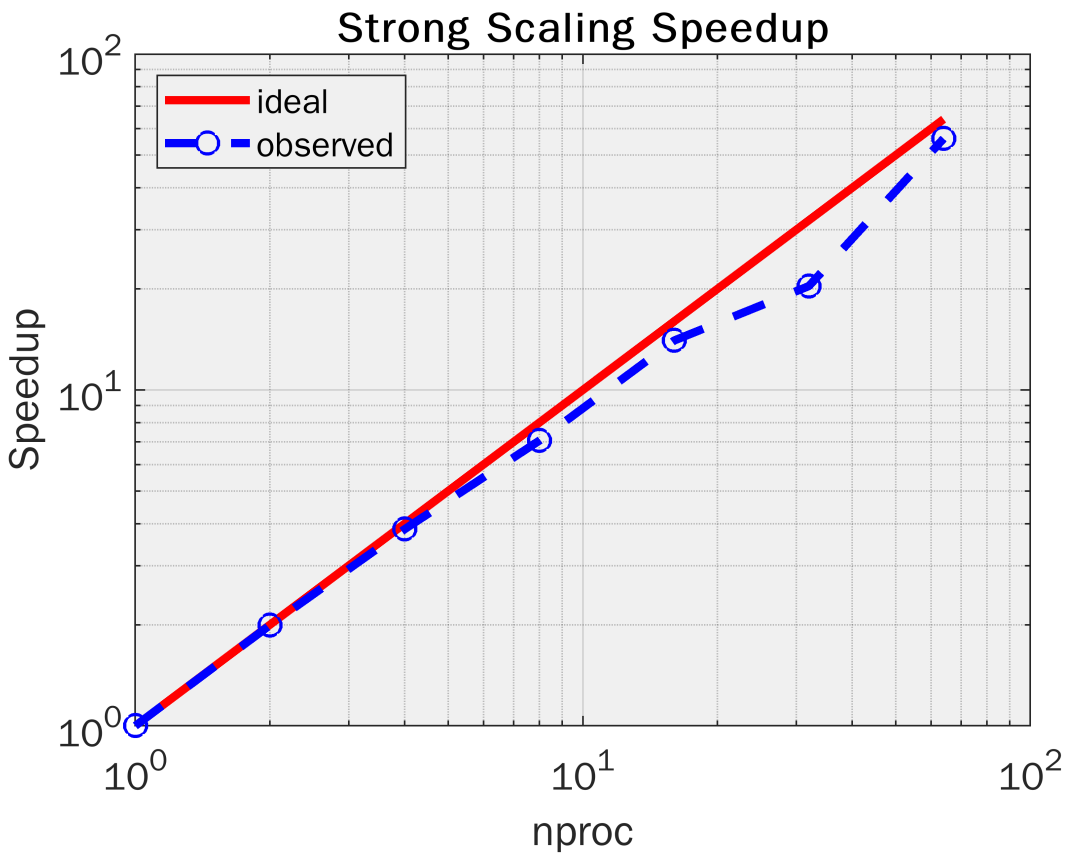
Parallel MC Simulation



Strong Scaling Speedup/Efficiency

Using an exponentially increasing number of processors from 1 to 64 and a problem size of 2^28, a total of 7 tests were run that will be analzyed here. Below the plots for speedup and efficiency are generated.

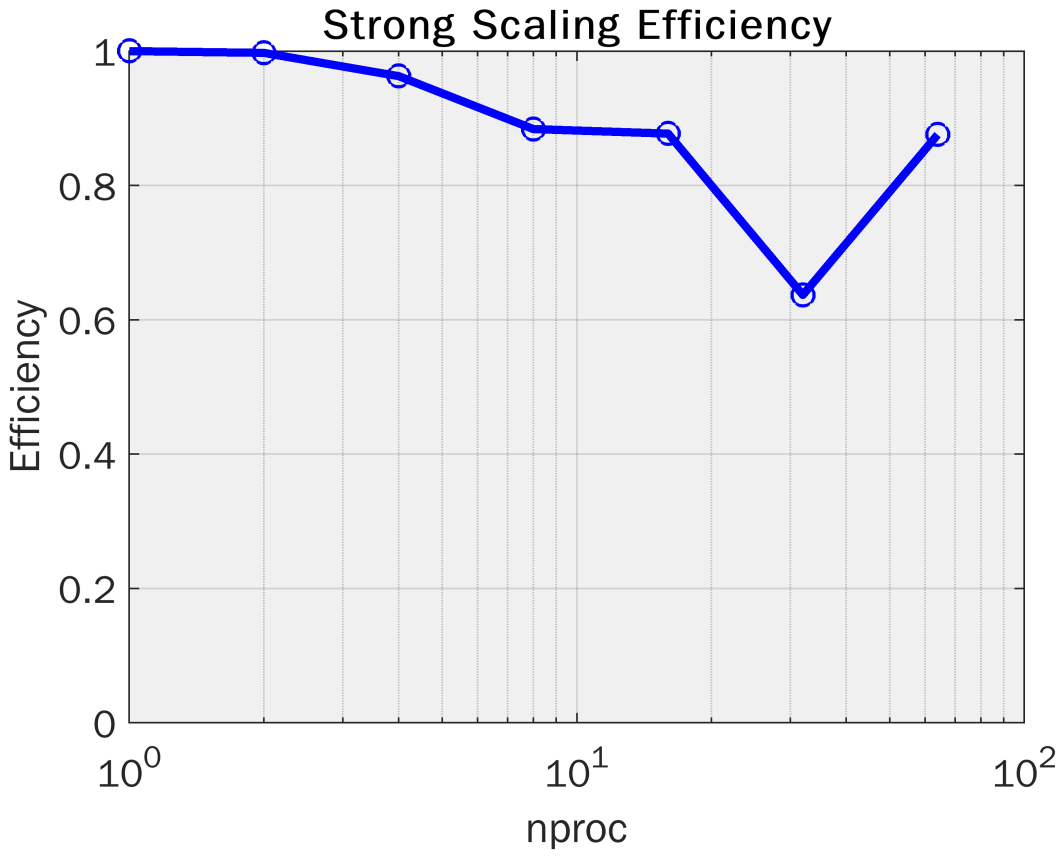
```
%Reading data
speedData=readmatrix('strongScaling1.csv');
nproc=speedData(:,1);
time=speedData(:,2);

speedup = time(1)./time;
idealSpeedup = nproc;

%Speedup
loglog(nproc,idealSpeedup,'r-',nproc,speedup,'bo--','LineWidth',3,'MarkerSize',8);
grid on;
xlabel('nproc','FontSize',14);
ylabel('Speedup','FontSize',14);
set(gca,'FontSize',14);
set(gca,'color',1/255*[240 240 240])
title('Strong Scaling Speedup');
legend('ideal','observed','Location','northwest');
```



The red solid line is the ideal linear speedup line. The dotted blue line is the observed speedup. The correlation is extremely strong up to when the experiment hit 32 processors. The R2 compute cluster has 28 processors per node. Communication between processors slows down a parallel program, ensuring no program can truly be 100% linear in speedup. However, communication between *nodes* drastically impacts the speedup of a

program, especially if all of the processors on a secondary node are not being used. At 32 processors there are by default 4 on a different node than the original 28. The drop in speedup can be entirely attributed to that. The swift regain in correlation can be seen as the program is using 64 processors across 3 nodes. 2 full nodes are in full effect with another using 8 processors out of it's 28. This means there is far less wasteful communication as in the 32 processor test. In conclusion, it would be more effective to use 28 processors instead of 32, and thus 56 instead of 64.

```
%Efficiency
E = speedup./nproc;
semilogx(nproc,E,'bo-','LineWidth',3,'MarkerSize',8);
grid on;
xlabel('nproc','FontSize',14);
ylabel('Efficiency','FontSize',14);
ylim([0 1]);
set(gca,'FontSize',14);
set(gca,'color',1/255*[240 240 240])
title('Strong Scaling Efficiency');
```

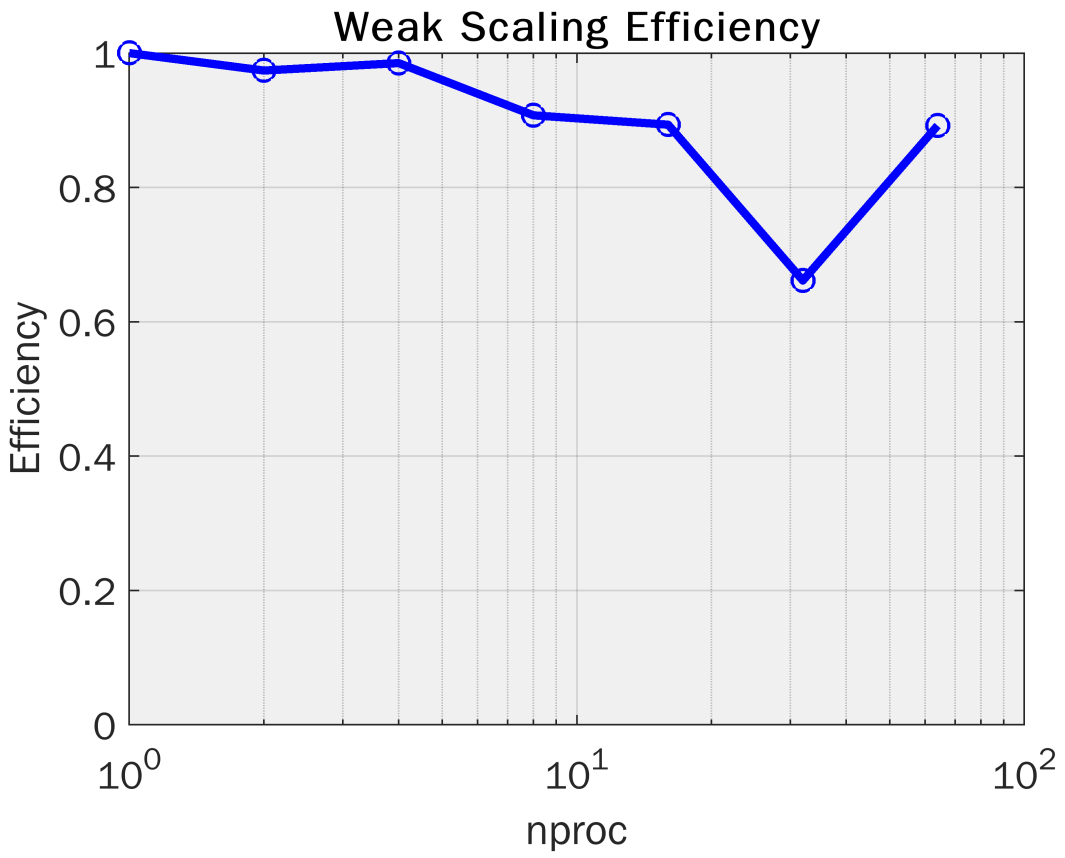


Perfect efficiency is impossible, but the goal is to come as close to 100% as possible. The program was highly efficient in the first 5 tests until, as seen in the speedup plot, there is a major dropoff at 32 processors. As previously noted, communication between nodes severely impacts a programs speedup and efficiency. The same bounce-back can be observed with the 64 processor test here, as more processors are being put to work with less communication between nodes required.

Weak Scaling Efficiency

Starting with a problem size (N) of 2^22 and 1 processor, the execution time was recorded and both the problem size and number of processors were increased by a factor of 2x. In an ideal situation, the execution time should be identical on each repetition as the processors should be handling the same amount of information each time. Of course, this is not possible because communication must occur, and this is especially true as the number of processors increases. This can be seen below.

```
%Reading data
speedData=readmatrix('weakScaling1.csv');
nproc=speedData(:,1);
time=speedData(:,2);

%Plotting
E=time(1)./time;
semilogx(nproc,speedup,'bo-','LineWidth',3,'MarkerSize',8);
grid on;
xlabel('nproc','FontSize',14);
ylabel('Efficiency','FontSize',14);
ylim([0 1]);
set(gca,'FontSize',14);
set(gca,'color',1/255*[240 240 240])
title('Weak Scaling Efficiency');
```



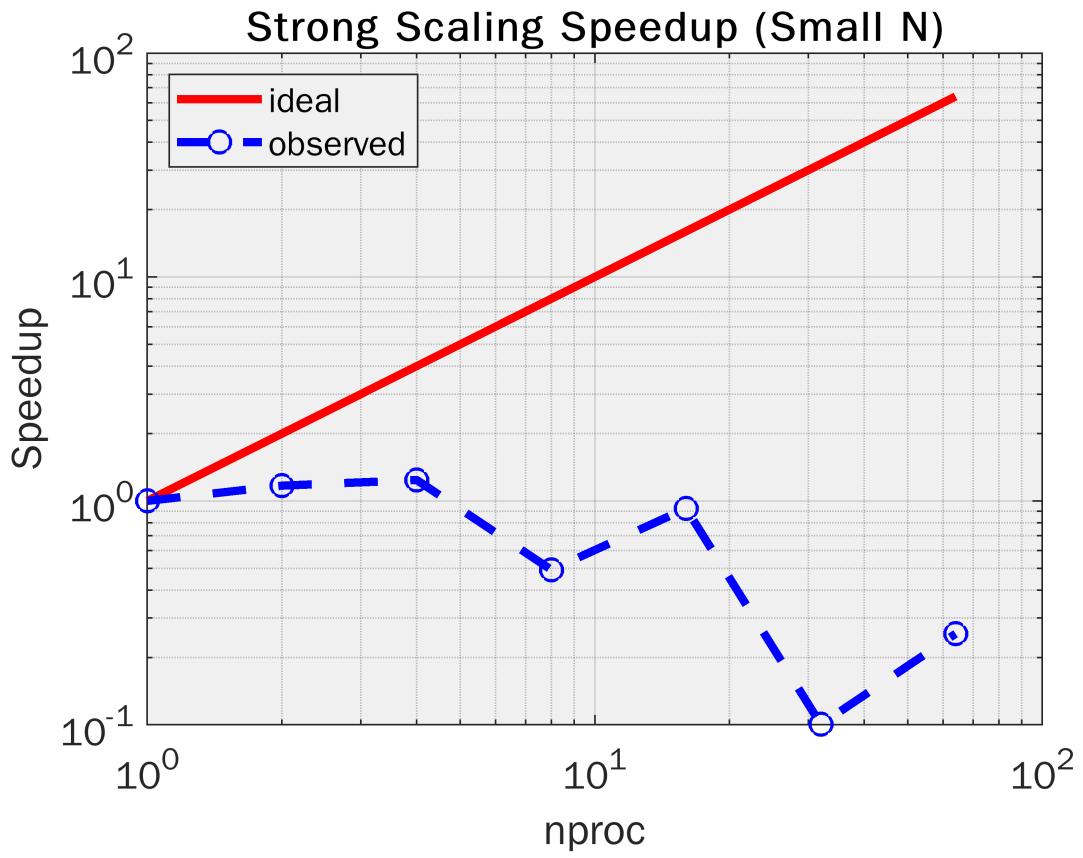
If processors could talk to each other and instantaneously perform calculations with no time in between, every program would reach near maximum efficiency (aside from hardware limitations). However, computers can only

do 1 thing at a time. This communication process is not an issue until the node limit is reached. As mentioned in the strong scaling analysis, Boise State's R2 supercomputer has 28 processors per node. This means that once more than 28 processors are requested for a job, more than 1 node has to be used. This communication between nodes slows execution time tremendously as we see the massive dip in execution time between 16 and 32 processors. Execution time dropped from it's consistent .1 second range to over .14 seconds! When 64 processors are used, the entire computational power of 2 nodes is used with some left over on a 3rd node. This means there is a lot less wasteful cross-node communication. It is important to be mindful of the specifications of your machine when running parallel programs to maximize efficiency.

Strong and Weak Scaling with a Small Problem Size

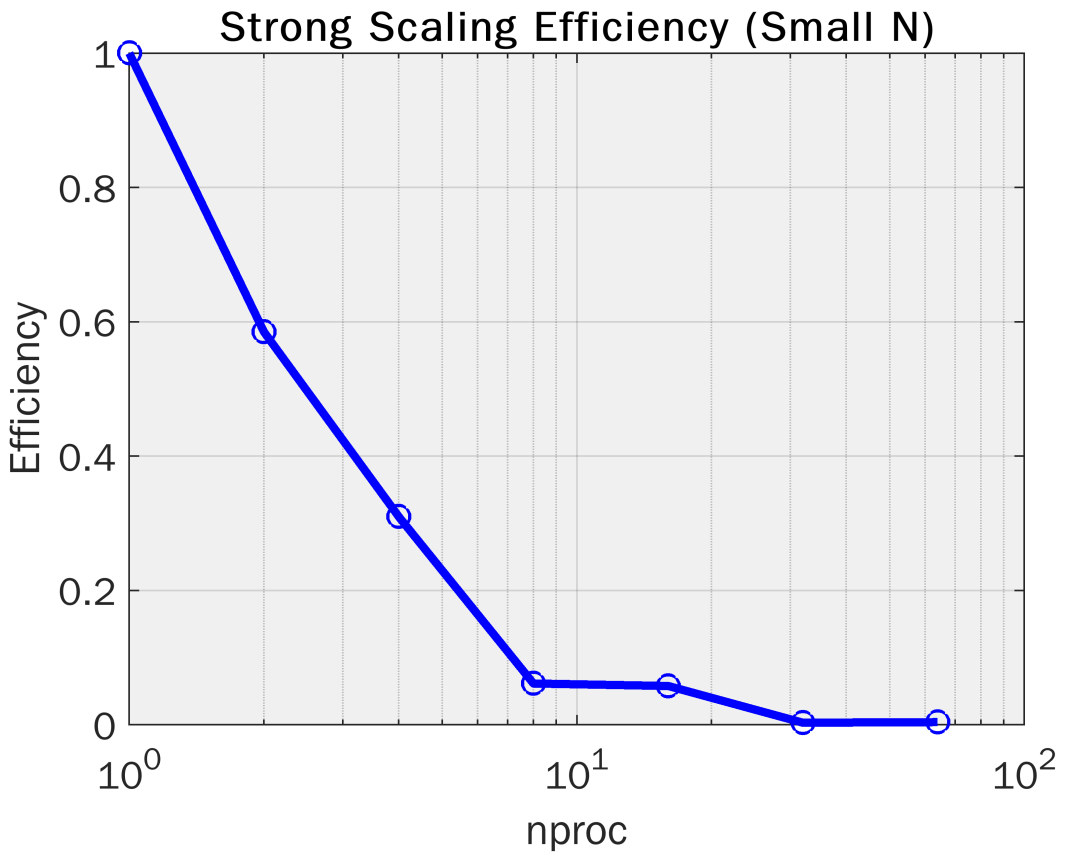
For both scaling experiments the problem size began at N=100. For weak scaling, this number doubled each time the processors doubled for a final N value of 6400. This way, each processor would always have exactly 100 iterations to perform. The plots of speedup, efficiency are followed with analysis below.

```
%Reading data
speedData=readmatrix('strongSmall.csv');
nproc=speedData(:,1);
time=speedData(:,2);
speedup = time(1)./time;
idealSpeedup = nproc;

%Speedup
loglog(nproc,idealSpeedup,'r-',nproc,speedup,'bo--','LineWidth',3,'MarkerSize',8);
grid on;
xlabel('nproc','FontSize',14);
ylabel('Speedup','FontSize',14);
set(gca,'FontSize',14);
set(gca,'color',1/255*[240 240 240])
title('Strong Scaling Speedup (Small N)');
legend('ideal','observed','Location','northwest');
```

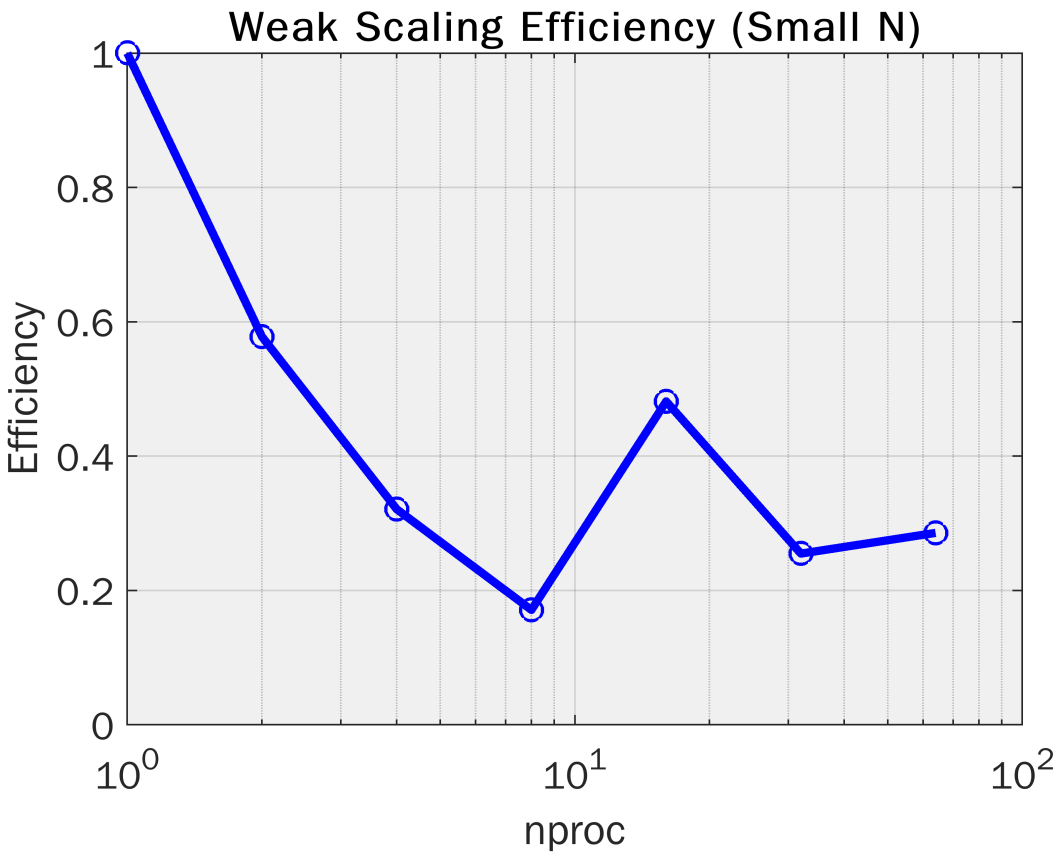



```
%Efficiency (Strong)
E = speedup./nproc;
semilogx(nproc,E,'bo-','LineWidth',3,'MarkerSize',8);
grid on;
```

```
xlabel('nproc','FontSize',14);
ylabel('Efficiency','FontSize',14);
ylim([0 1]);
set(gca,'FontSize',14);
set(gca,'color',1/255*[240 240 240])
title('Strong Scaling Efficiency (Small N)');
```




```
%Reading data
speedData=readmatrix('weakSmall.csv');
nproc=speedData(:,1);
time=speedData(:,2);
%Plotting
E=time(1)./time;
semilogx(nproc,E,'bo-','LineWidth',3,'MarkerSize',8);
grid on;
xlabel('nproc','FontSize',14);
ylabel('Efficiency','FontSize',14);
ylim([0 1]);
set(gca,'FontSize',14);
set(gca,'color',1/255*[240 240 240])
title('Weak Scaling Efficiency (Small N)');
```



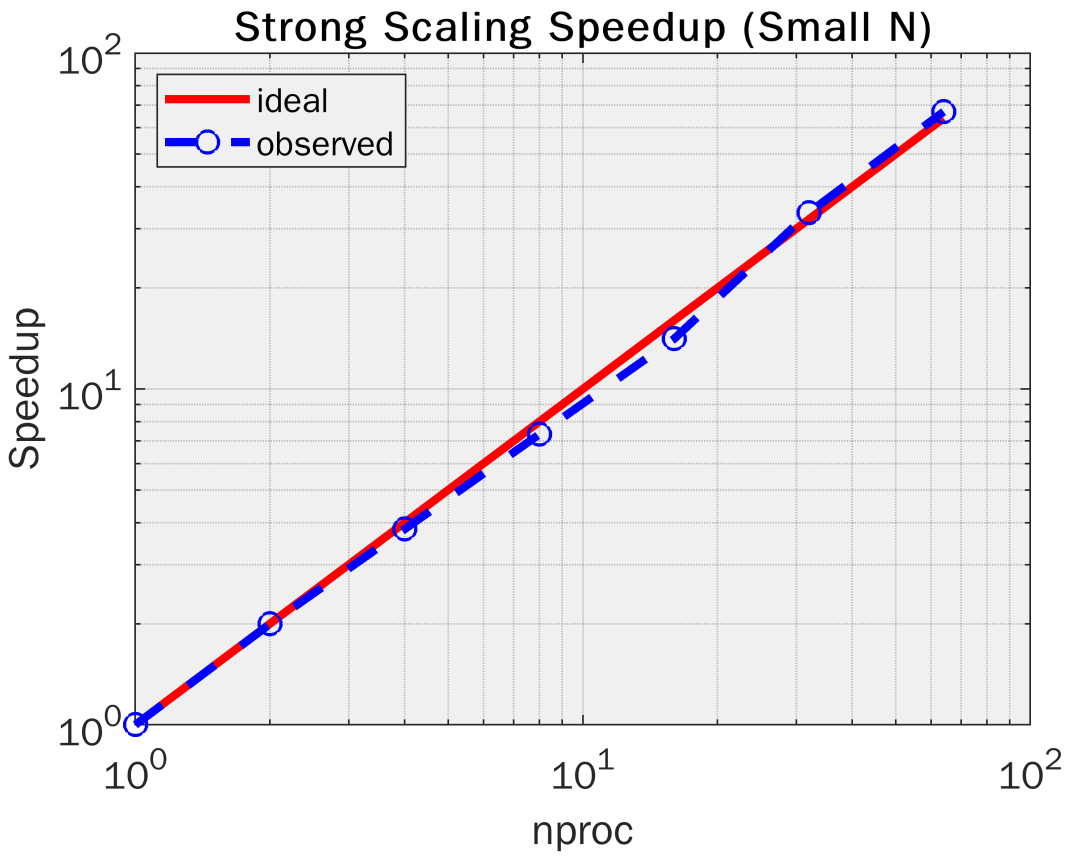
Starting with the weak scaling efficiency plot, it is clear that as the number of processors increases, the efficiency decreases dramatically. There is a small jump right as the processors hit the 16 nproc mark, likely because the communication is similar to that of 8 processors and the calculations are able to be solved twice as quickly. However, once the 2nd and 3rd nodes are introduced, the efficiency quickly drops back below 30%. The weak scaling experiment shows stronger efficiency than the strong scaling efficiency plot because of the simultaneous growing problem size.

When it comes to the strong scaling plots, it is quite obvious that small problem sizes are not good for multiple processors. As the number of processors increases, the efficiency steadily decreases along with the speedup. This is because the problem size is so small, one processor can do the calculations just as fast as a million could (probably much faster) because it does not have to spend time on communication. Parallel computing is about finding a perfect balance of processors for the given problem size in order to maximize time spent computing and minimize time spent communicating. In the case of N = 1000, no more than 1 processor is every really needed.

Strong and Weak Scaling with a Large Problem Size

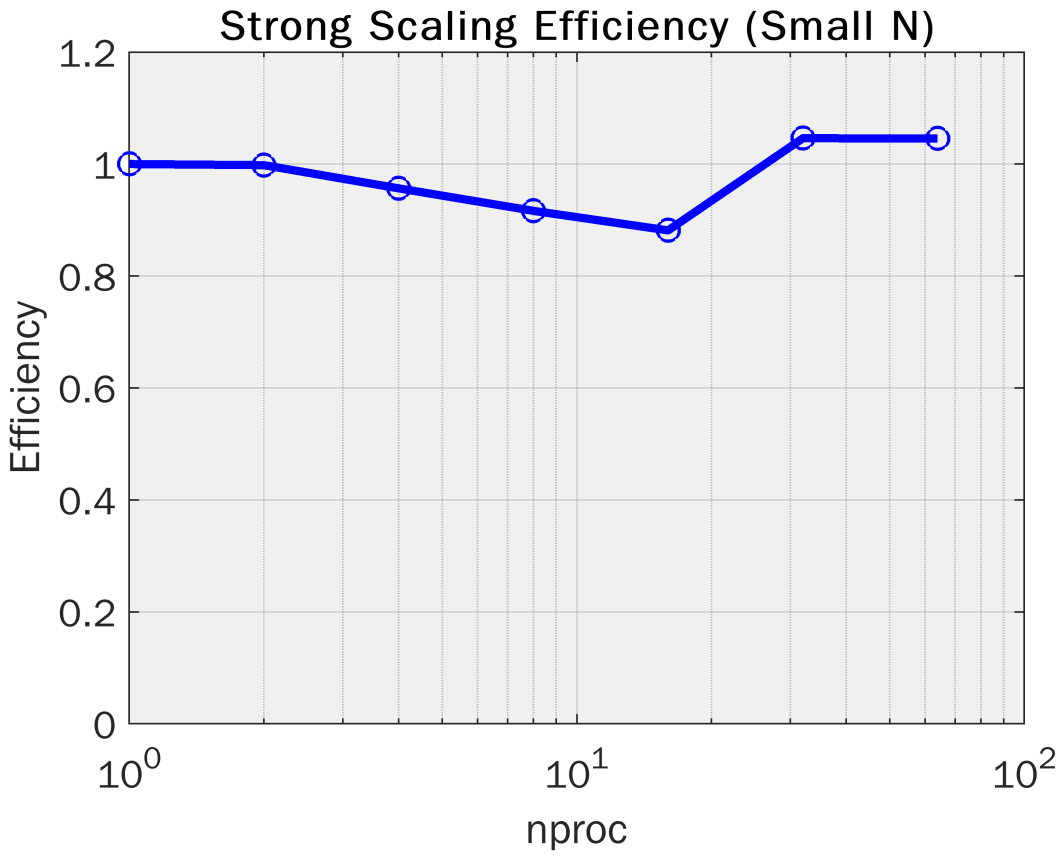
Weak scaling here starts with a problem size of 2^28 and increases exponentially to over 17 billion trials of the Monte Carlo method. On the other hand, the large strong scaling problem set has a problem size of 2^30. Speedup, efficiency, and analysis of these larger datasets can be found below.

```
%Reading data
speedData=readmatrix('strongLarge.csv');
nproc=speedData(:,1);
time=speedData(:,2);
speedup = time(1)./time;
idealSpeedup = nproc;
%Speedup
loglog(nproc,idealSpeedup,'r-',nproc,speedup,'bo--','LineWidth',3,'MarkerSize',8);
grid on;
xlabel('nproc','FontSize',14);
ylabel('Speedup','FontSize',14);
set(gca,'FontSize',14);
set(gca,'color',1/255*[240 240 240])
title('Strong Scaling Speedup (Small N)');
legend('ideal','observed','Location','northwest');
```

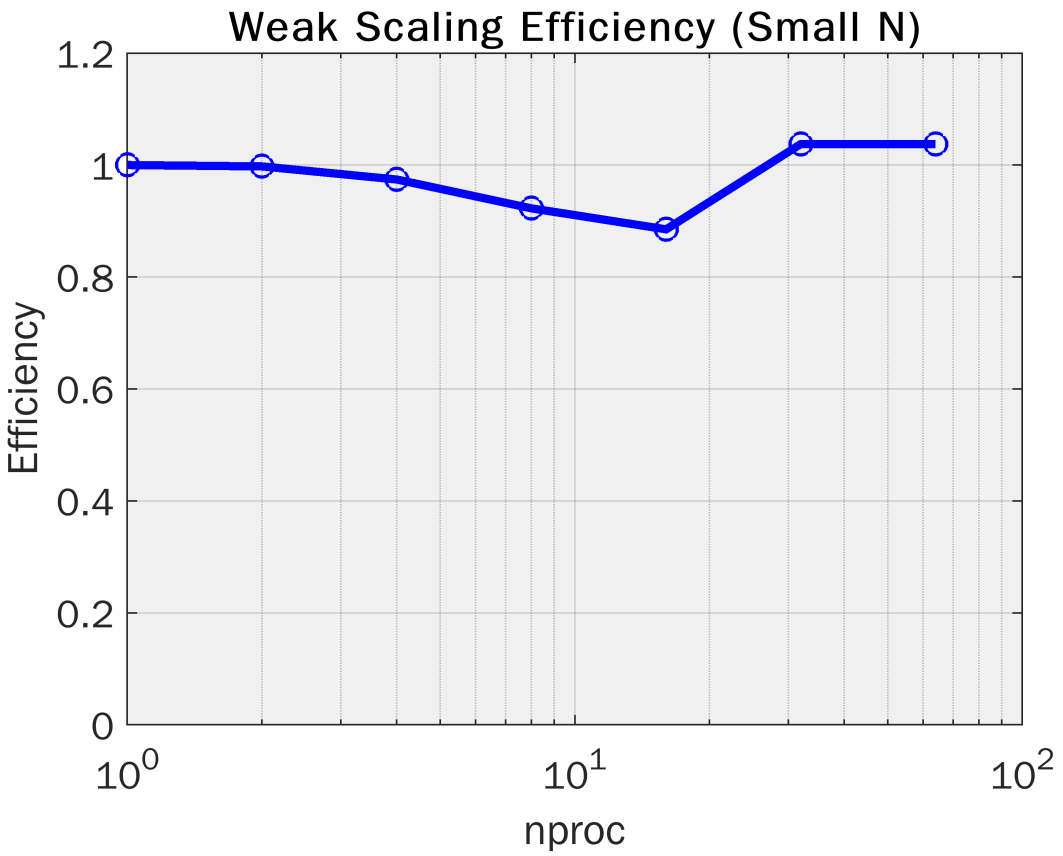



```
%Efficiency (Strong)
E = speedup./nproc;
semilogx(nproc,E,'bo-','LineWidth',3,'MarkerSize',8);
grid on;
xlabel('nproc','FontSize',14);
```

```
ylabel('Efficiency','FontSize',14);
ylim([0 1.2]);
set(gca,'FontSize',14);
set(gca,'color',1/255*[240 240 240])
title('Strong Scaling Efficiency (Small N)');
```




```
%Reading data
speedData=readmatrix('weakLarge.csv');
nproc=speedData(:,1);
time=speedData(:,2);
%Plotting
E=time(1)./time;
semilogx(nproc,E,'bo-','LineWidth',3,'MarkerSize',8);
grid on;
xlabel('nproc','FontSize',14);
ylabel('Efficiency','FontSize',14);
ylim([0 1.2]);
set(gca,'FontSize',14);
set(gca,'color',1/255*[240 240 240])
title('Weak Scaling Efficiency (Small N)');
```

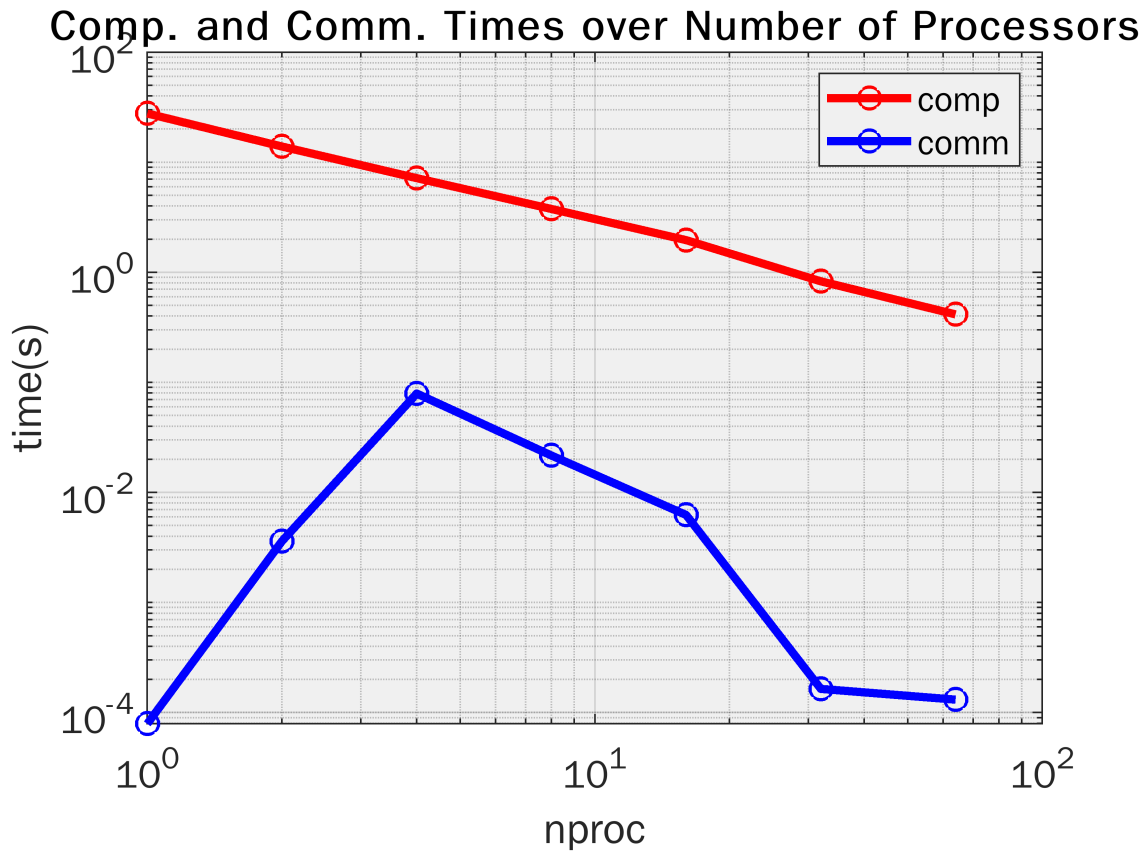


The speedup plot for strong scaling with this problem size shows incredible promise for even larger problems. As the number of processors grows, the speedup actually passes linear speedup which means that this program has extremely high parallel efficiency. This is demonstrated in the efficiency plot that has over 100% efficiency. This is in direct violation of the 2nd law of thermodynamics and is technically physically impossible. It is likely that between runs there were certain nodes that were used or certain processors that have the slightest hardware advantages. For problems of this magnitude, small advantages can show huge increases in efficiency. Again, this is impossible, but it goes to show the computing power multiple processors can have when tasked with a massive problem size. A nearly identical plot is mapped out under weak scaling efficiency, with consistent values close to 100% and for nproc=32,64 efficiency of over 100% again! This is more important in the weak scaling plot than in the strong scaling plot because it shows just how scalable the program is. As the problem size grows, efficiency grows in close correlation with the number of processors! If this chart pattern were to continue, you could solve a problem of infinite size and only be limited by the compute power available. Parallel computing is meant to tackle problems of large magnitude, and the efficiency shown on these plots compared to those from the smaller problem size prove just that.

Analysis of Communication vs Computation Time

Using the large strong scaling dataset, I took the time each processor took to perform calculations and communicate back to rank 0. After averaging those times, the following plots were generated.

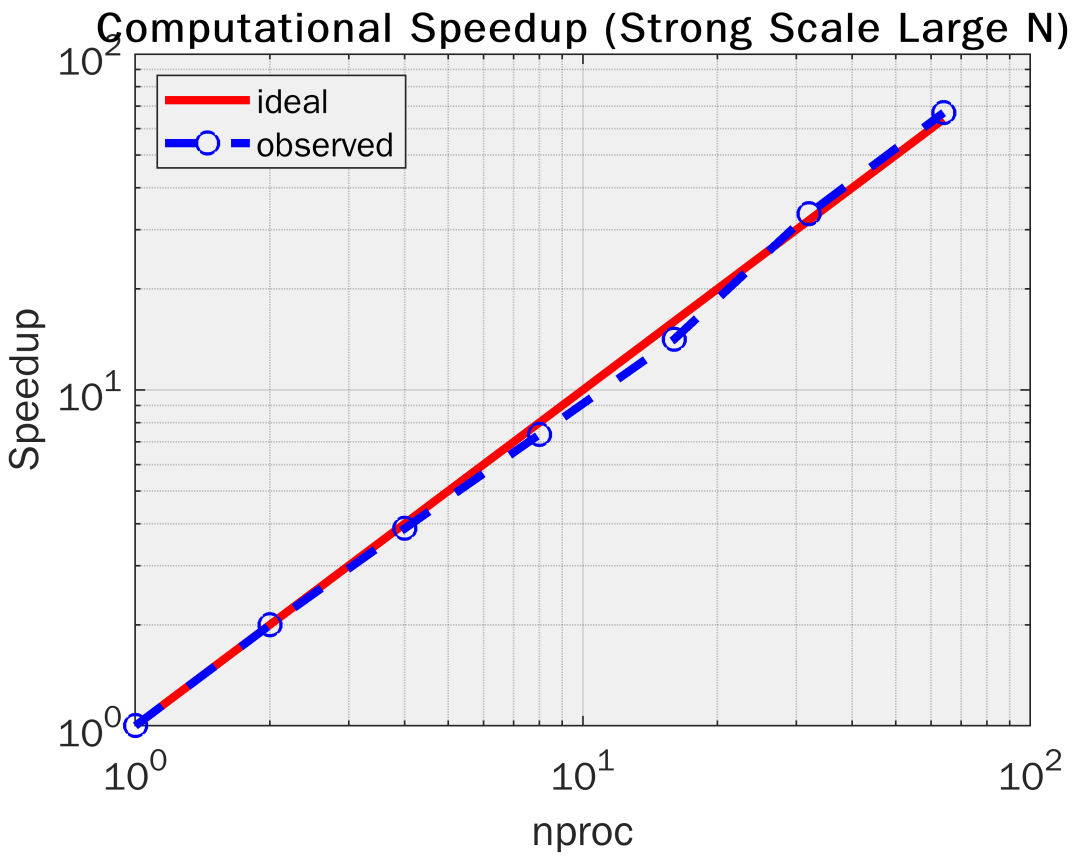
```
%Reading data


%Plot
loglog(nproc,comp,'ro-',nproc,comm,'bo-','LineWidth',3,'MarkerSize',8);
grid on;
xlabel('nproc','FontSize',14);
ylabel('time(s)','FontSize',14);
set(gca,'FontSize',14);
set(gca,'color',1/255*[240 240 240]);
title('Comp. and Comm. Times over Number of Processors');
legend('comp','comm','Location','northeast');
```



For the first three trials, the communication and computation times show an expected direction. However, to my surprise, from 16 processors and on, the communication time decreases as computation time decreases as well. Regardless, the communication time is extremely low and negligible for the most part, as the only MPICH library function used to communicate is Reduce. Because of this, communication times can stay low while computation times steadily decrease as number of processors increases. This can help explain the over 100% efficiency demonstrated in the strong scaling efficiency. Normally, when computation and communication cross, that is when the program is reaching it's computational limits. Based on what this plot is telling us, this algorithm could complete a massive amount of computations with very little time wasted on communication.

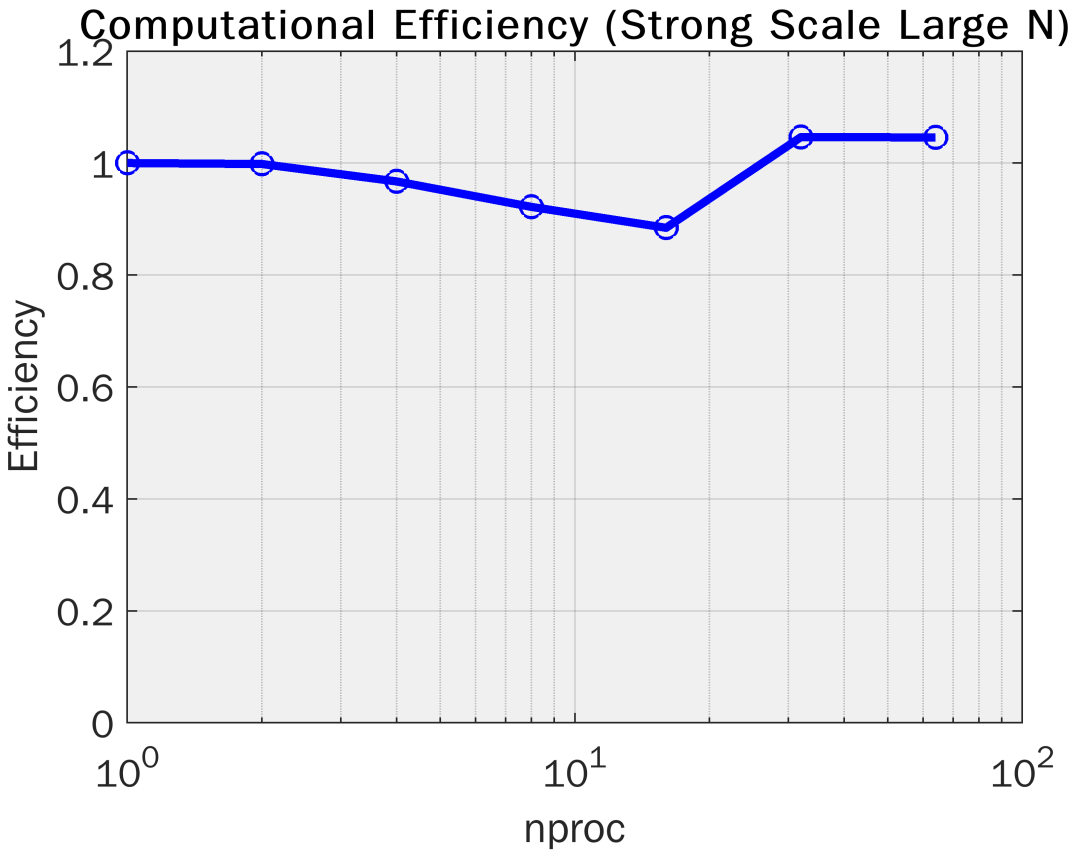
Plotting the speedup and efficiency of both the computation and communication times alone might tell more about what is happening. Here is the speedup and efficiency of the computational execution time of the large strong scaling experiment.

```
%Reading data
speedData=readmatrix('timeData.csv');
nproc=speedData(:,1);
time=speedData(:,2);

speedup = time(1)./time;
idealSpeedup = nproc;
%Speedup
loglog(nproc,idealSpeedup,'r-',nproc,speedup,'bo--','LineWidth',3,'MarkerSize',8);
grid on;
xlabel('nproc','FontSize',14);
ylabel('Speedup','FontSize',14);
set(gca,'FontSize',14);
set(gca,'color',1/255*[240 240 240])
title('Computational Speedup (Strong Scale Large N)');
legend('ideal','observed','Location','northwest');
```




```
%Efficiency (Strong)
E = speedup./nproc;
semilogx(nproc,E,'bo-','LineWidth',3,'MarkerSize',8);
```

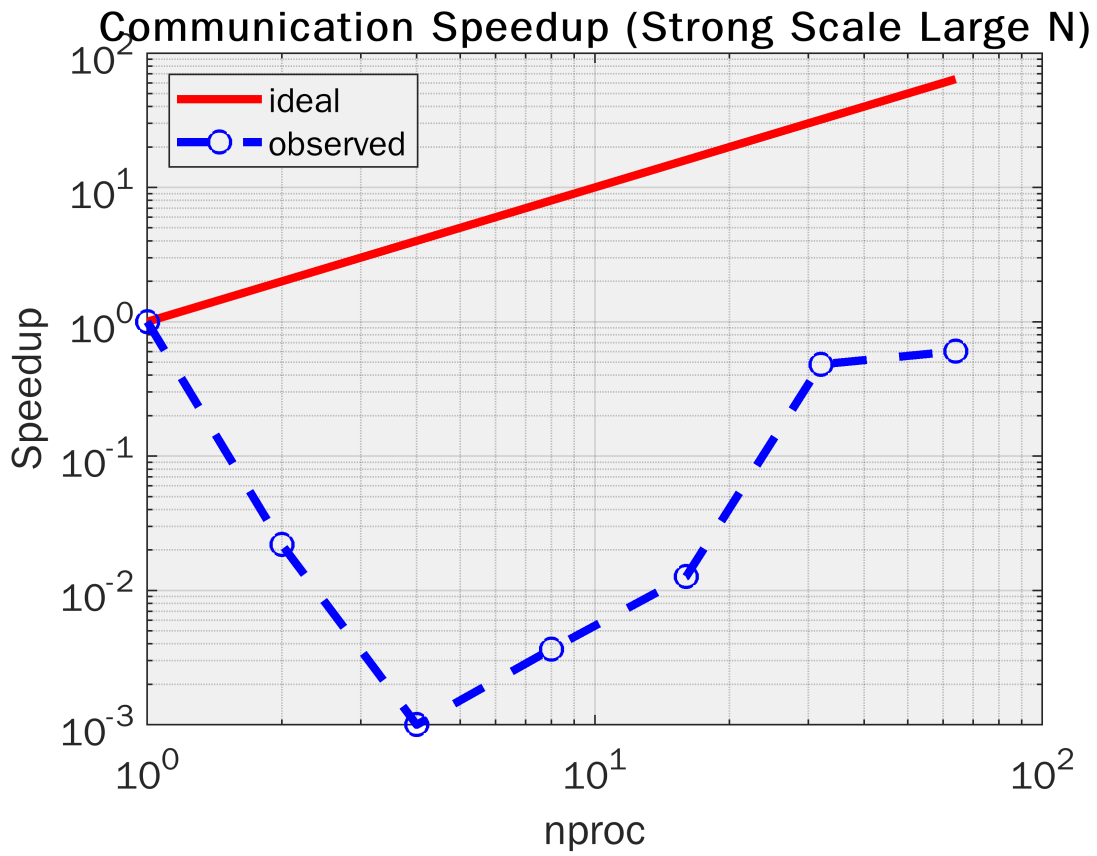
```
grid on;
xlabel('nproc','FontSize',14);
ylabel('Efficiency','FontSize',14);
ylim([0 1.2]);
set(gca,'FontSize',14);
set(gca,'color',1/255*[240 240 240]);
title('Computational Efficiency (Strong Scale Large N)');
```



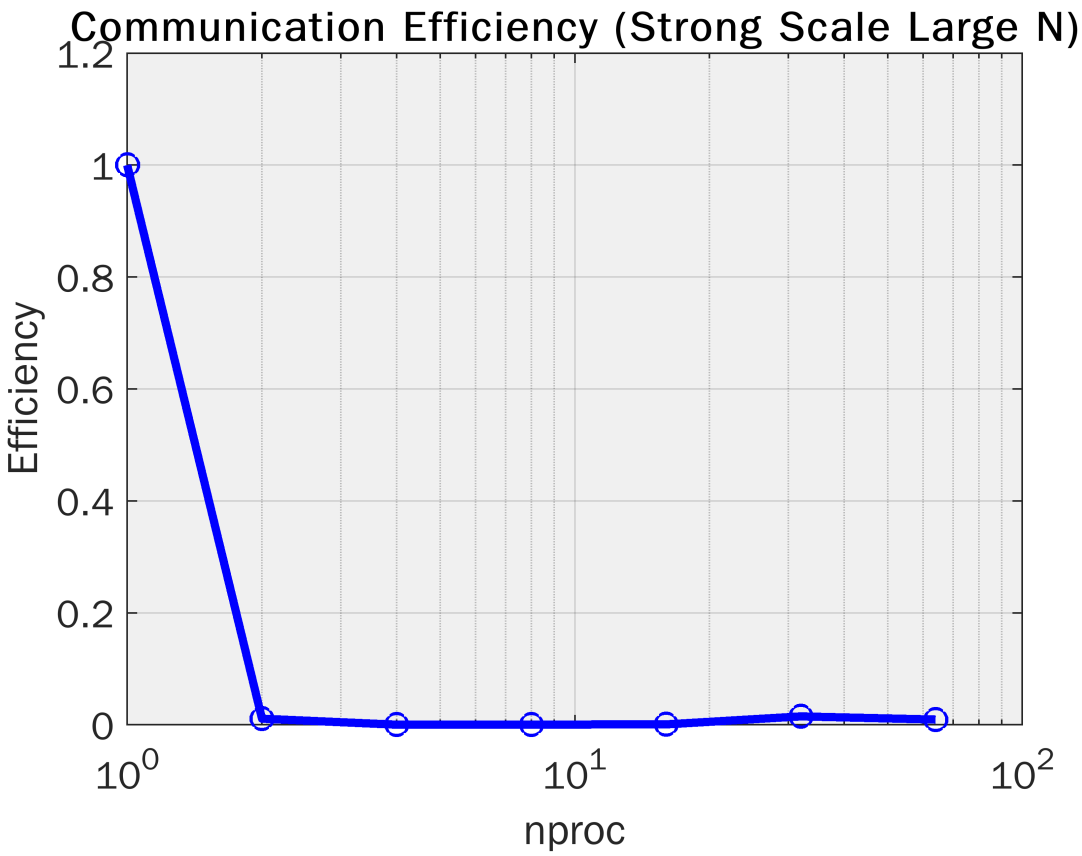
As mentioned previously, the communication times in this experiment proved to be so small that they were practically negligible. This is why in this situation we see a direct correlation between the speedup and efficiency plots of the large strong scaling experiment and its computational time. Mathematically, if computation takes up most of the time, it will have the most effect on the plots themselves. Now it's time to look at the communication plots.

```
%Reading data
speedData=readmatrix('timeData.csv');
nproc=speedData(:,1);
time=speedData(:,3);

speedup = time(1)./time;
idealSpeedup = nproc;
%Speedup
loglog(nproc,idealSpeedup,'r-',nproc,speedup,'bo--','LineWidth',3,'MarkerSize',8);
grid on;
xlabel('nproc','FontSize',14);
ylabel('Speedup','FontSize',14);
set(gca,'FontSize',14);
```

```
set(gca,'color',1/255*[240 240 240])
title('Communication Speedup (Strong Scale Large N)');
legend('ideal','observed','Location','northwest');
```




```
%Efficiency (Strong)
E = speedup./nproc;
semilogx(nproc,E,'bo-','LineWidth',3,'MarkerSize',8);
grid on;
xlabel('nproc','FontSize',14);
ylabel('Efficiency','FontSize',14);
ylim([0 1.2]);
set(gca,'FontSize',14);
set(gca,'color',1/255*[240 240 240]);
title('Communication Efficiency (Strong Scale Large N)');
```



As expected, communication has a terrible speedup and efficiency. This part of the Monte Carlo method does not contribute to finding the solution, and thus is going to be extremely inefficient. Fortunately in this case, there is not much required communication to solve the problem as each processor is able to generate its own data. In cases where more communication is present, this can cause major issues in the overall speedup and certainly weigh down the overall speedup when strong scaling an algorithm.

In conclusion, there must be a balance in parallel computing. The computational power of the machine, the size of the problem, and the amount of communication needed are all important factors when deciding the most optimal way to parallelize a program.