CS426 Project 2 Report

Rüzgar Ayan
21801984

July 13, 2021

1 Implementation Details

For implementing this experiment with MapReduce, I would normally use an array of pairs of floats to represent the 2D coordinates. But since we were not able to change the parameters of these functions, I tried to hold 2D coordinates in a single float value. Converting between 1D and 2D coordinates was easy for integers and I have done that a lot before. But it was trickier for floats, I have used the two functions *pack_float* and *unpack_float* from the code snippet in the discussion https://stackoverflow.com/questions/20752344/storing-2-single-floats-in-one. With these two functions I was able to process 2D coordinates with the MapReduce functions I've written. This packing loses some precision of the original float values, but since we are doing experiments with already random numbers, I thought this wouldn't be much of a change.

2 Questions

2.1 What kind of parallelism, does your strategy apply for approximating $\sqrt{2}$. Discuss in terms of data and task parallelism.

Since the same operations are done in all the proccessors at the same time but just with different data (randomly generated points), we can say that data parallelism strategy was used. The same strategy would probably work for all kinds of Monte Carlo experiments as the individual experiments do not depend on each other. It would be possible to use task parallelism if we were not restricted to use the map, fold and filter functions. We could generate the random points as one task, check if they lie on the rectangle in another task.

2.2 What was your communication scheme, which MPI communication functions did you use and why?

Since the number of experiments might not be divisible by the number of processors, some processors might need to handle more experiments. For example, doing 10 experiments with 4 processors would need the partition of 2+2+3+3 experiments. With that in mind, I started to implement the map, fold and filter functions using the *MPI_Scatterv* and *MPI_Gatherv* functions. These variations of scatter and gather funcions were letting me divide the work in unequal sizes. But then, I decided that these were making the code harder and unreadable.
I converted these to the classic versions *MPI_Scatter* and *MPI_Gather* and handled the excess data (the data that wasn't shared with the Scatter function) in the process with id 0 separately. This will not cause an observable unbalance because the number of excess experiments will be at most equal to the number of processes.

2.3 How MPI address space is used between processes? Are they threads, or are they different OS level processes? What would be in a distributed environment where you had many different computers connected together? Would function pointers, pointers to part of the code segment of virtual memory space, still work if we had a truly distributed environment?

I am guessing that different OS level processes are used in MPI rather than threads. Because the data was not shared between the MPI processes that I have been running in the projects so far.
If we were to send a function pointer from one process to another process via a message, I believe the function pointers would not work in a distributed environment. Because the address spaces of these two processes would not match. But in the code for this project, I am not sending any function pointers. All processes get the function pointers themselves and since all the processes have the same code, there wouldn't be any problem in my case.

3 Results

The timing results are given in separate plots for each one of the inputs 1000, 10000, 100000 and 1000000.

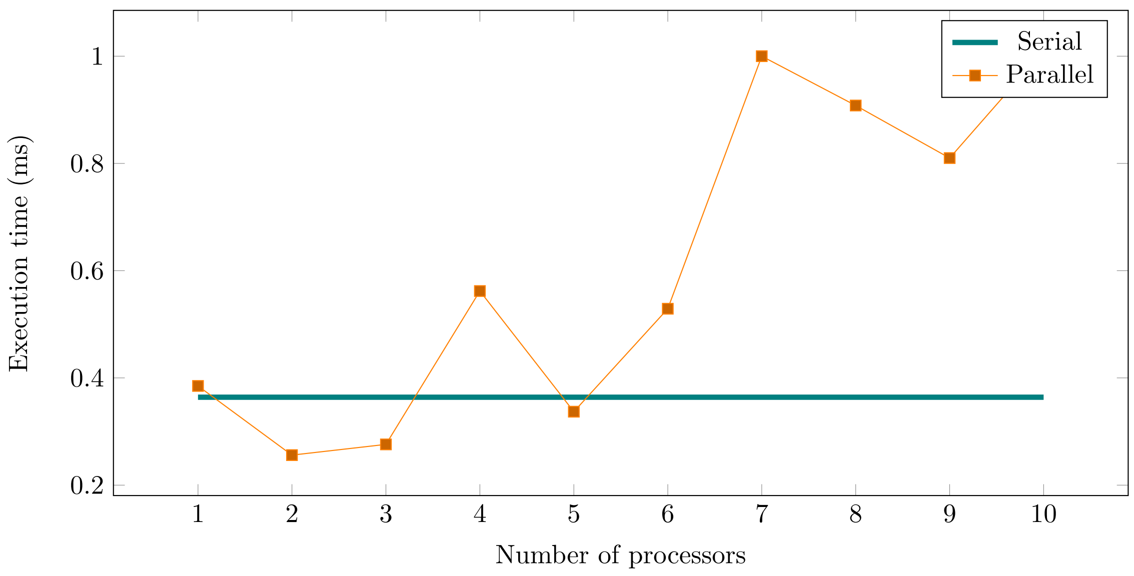

Figure 1: Serial and Parallel Execution Times with 1000 Experiments

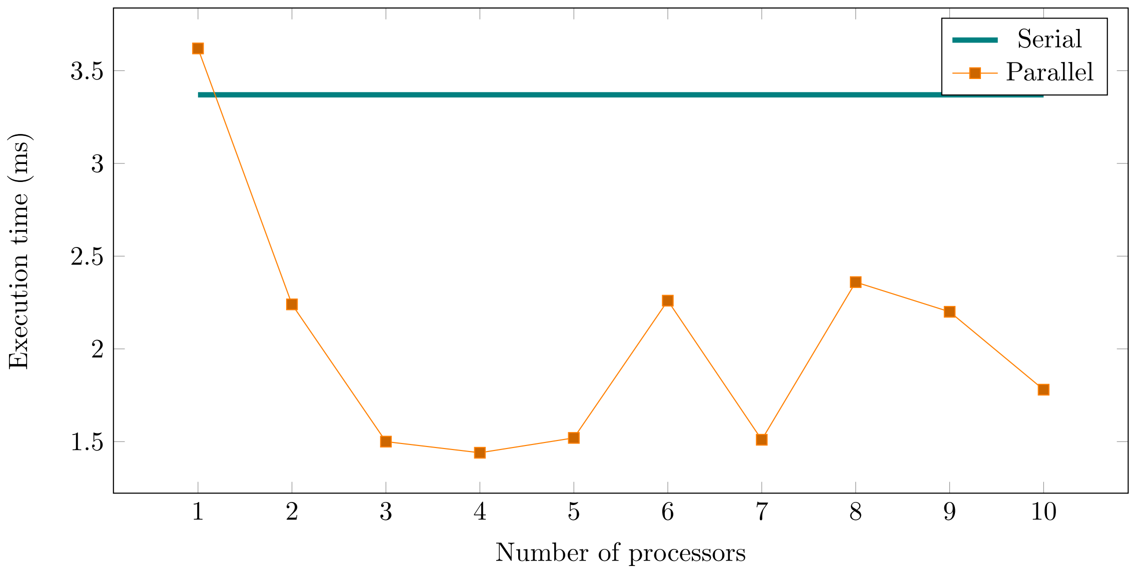

Figure 2: Serial and Parallel Execution Times with 10000 Experiments

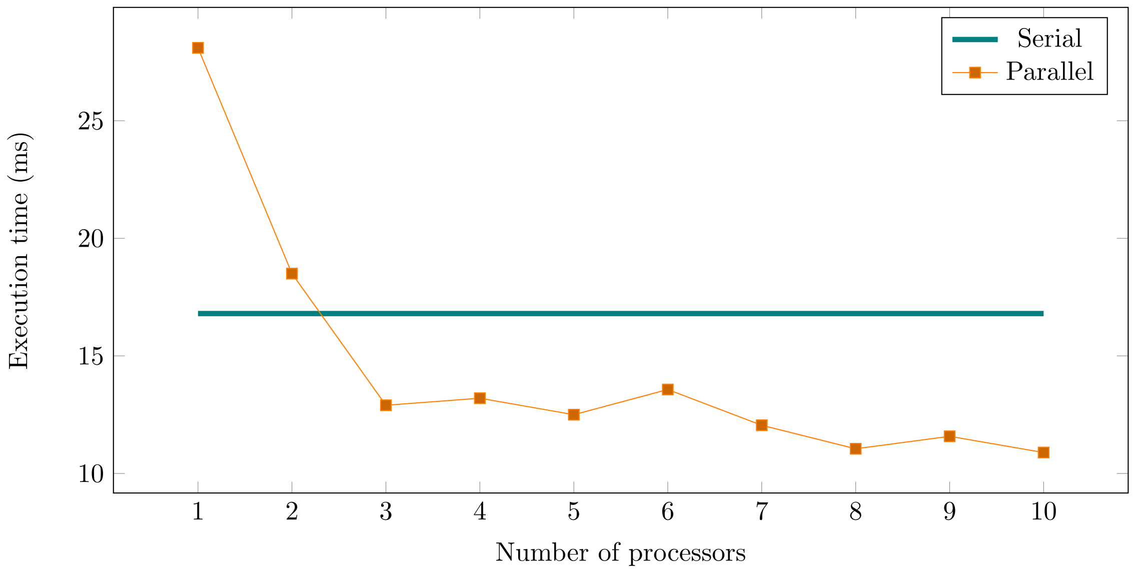

Figure 3: Serial and Parallel Execution Times with 100000 Experiments

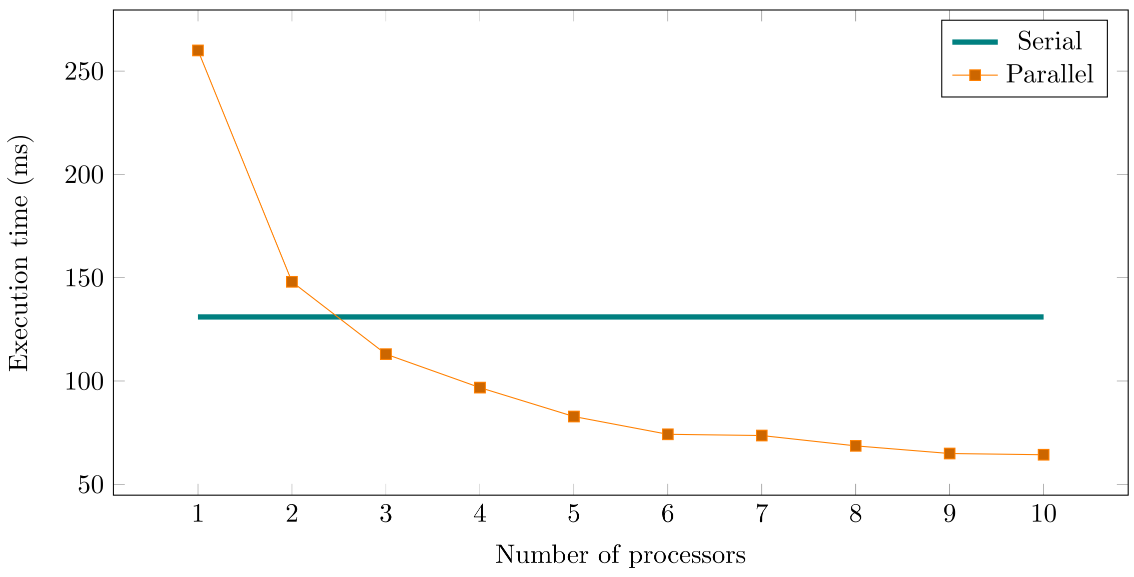

Figure 4: Serial and Parallel Execution Times with 1000000 Experiments

In all the plots, we see that using the parallel code with just 1 processor performs worse than the serial code by adding some overhead. Because in order to use MapReduce and successfully parallelize the code, some extra calculations were added to the parallel code (such as packing and unpacking the floats as explained in the Implementation Details).

As the number of processors increase, there is the expected decrease in the runtimes. But the value we get from each extra proccessor decreases as we add more and more. In the cases of 1000 and 10000 experiments, having more processors even make the runtimes higher. But in the cases of 100000 and 1000000 experiments, we see some slight improvement for every additional processor. Looking at all these results, running the parallel code with about $3 - 4$ processors seems the most ideal in terms of efficiency. With more experiments, this number of ideal processors would probably go up as well.