HOMEWORK_2

Team_09 : Jahnavi Valisetty
Abhishek Vanamala

Linear Execution

Simulation Design
Each Particle in the first model makes n*2 calculations. For every particle it determines the separation to one another particle, if the particle was inside the cut off range it would likewise compute and apply the suitable power to the particle. The issue with this calculation is that it checks the separation to every other particle. The arrangement we utilized was to partition the plane into a grid of squares, we at that point mapped every particle to a cell(bin) in the framework. By ensuring that the sides of the Bins is not exactly in the cut off range, we can be certain that every single pertinent particle to a given molecule are inside the Moore-neighbourhood of the bin.
Checking only neighbouring "bins" reduce the checks from O(n) for each particle to O(9d) where d is the average number of particles. In order to know that we have achieved the desired result of linear execution we had to plot the running time against the size of the input particles.

Grid side length = [sqrt(0.0005 * n)/0.01] +1
Total grid size = (Grid side length)^2 ~ O(n)

The result for our serial program can be seen in the figure below which has a logarithmic scale, and as we can see it increases linearly.

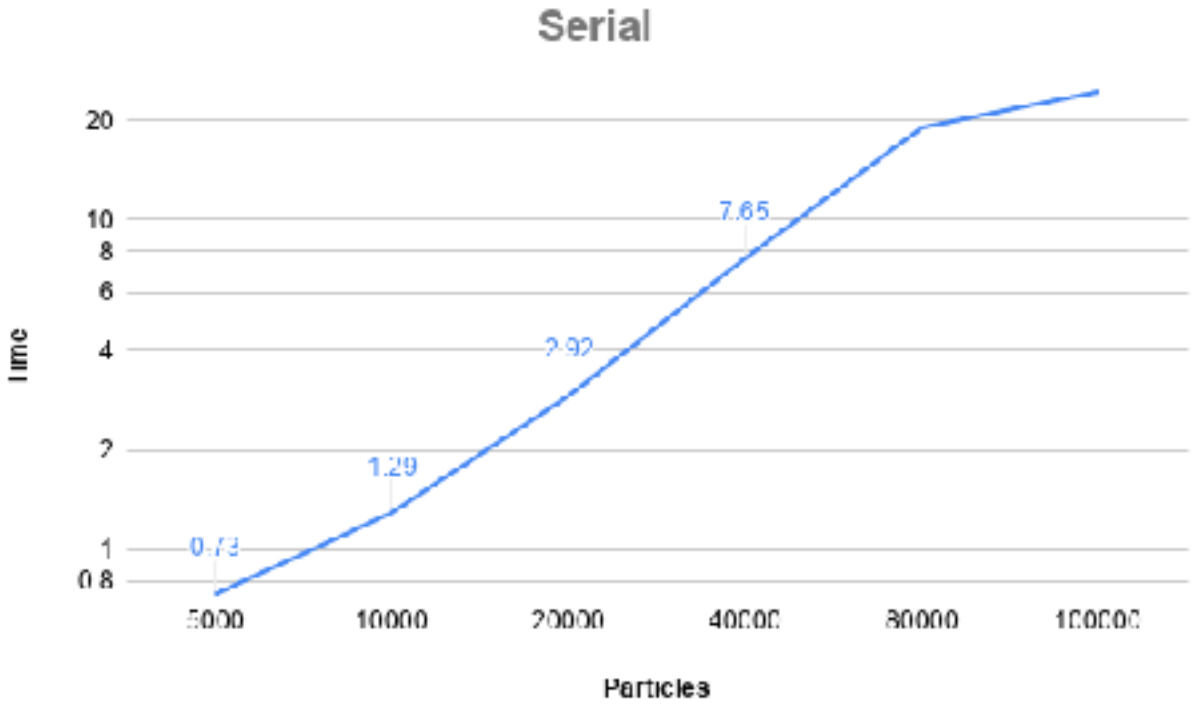

Fig 1. Log-Log Plot of number of particles vs. simulation time for serial (linear execution).

```
(//anaconda3) Jahnavis-MacBook-Air:proj2 jahnavivalisetty$ rm serial.txt
(//anaconda3) Jahnavis-MacBook-Air:proj2 jahnavivalisetty$  ./serial -n 5000 -no -s serial.txt
n = 5000, simulation time = 0.734026 seconds
(//anaconda3) Jahnavis-MacBook-Air:proj2 jahnavivalisetty$  ./serial -n 10000 -no -s serial.txt
n = 10000, simulation time = 1.29891 seconds
(//anaconda3) Jahnavis-MacBook-Air:proj2 jahnavivalisetty$  ./serial -n 20000-no -s serial.txt
n = 20000, simulation time = 2.92266 seconds, absmin = 0.754640, absavg = 0.957044
(//anaconda3) Jahnavis-MacBook-Air:proj2 jahnavivalisetty$  ./serial -n 40000 -no -s serial.txt
n = 40000, simulation time = 7.6512 seconds
(//anaconda3) Jahnavis-MacBook-Air:proj2 jahnavivalisetty$  ./serial -n 80000-no -s serial.txt
n = 80000, simulation time = 19.0767 seconds, absmin = 0.754649, absavg = 0.957026
(//anaconda3) Jahnavis-MacBook-Air:proj2 jahnavivalisetty$  ./serial -n 100000-no -s serial.txt
n = 100000, simulation time = 24.5177 seconds, absmin = 0.736342, absavg = 0.957021
(//anaconda3) Jahnavis-MacBook-Air:proj2 jahnavivalisetty$ ./autograder -v serial -s serial.txt

Serial code is O(N^slope)
Slope estimates are : 0.823398 1.169981 1.388404 1.318054 1.124512
Slope estimate for line fit is: 1.209904
```

Fig 2. Screenshot of Serial.cpp execution

Open Multi-Processing(open MP)

The second part of the problem was to parallelize the code so that the running time would be O(n)/T , where n is the number of particles and T is the number of threads. We used Open Multi-Processing in order to achieve this.

Synchronisation Used:

Barrier - A **barrier** for a group of threads or processes in the source code means any thread/process must stop at this point and cannot proceed until all other threads/processes reach this **barrier**.

Shared Memory Solution Design:

Making the algorithm parallel with the shared memory Programming API's are truly basic. We give each processor a distinct set of particles to monitor. With such an arrangement we can run the original solution practically unmodified with the exception that we must identify the critical sections and run them in mutual exclusion.

The parts which have critical sections are the ones writing to the grid structure and in connection to the whole code it's not that terrible.

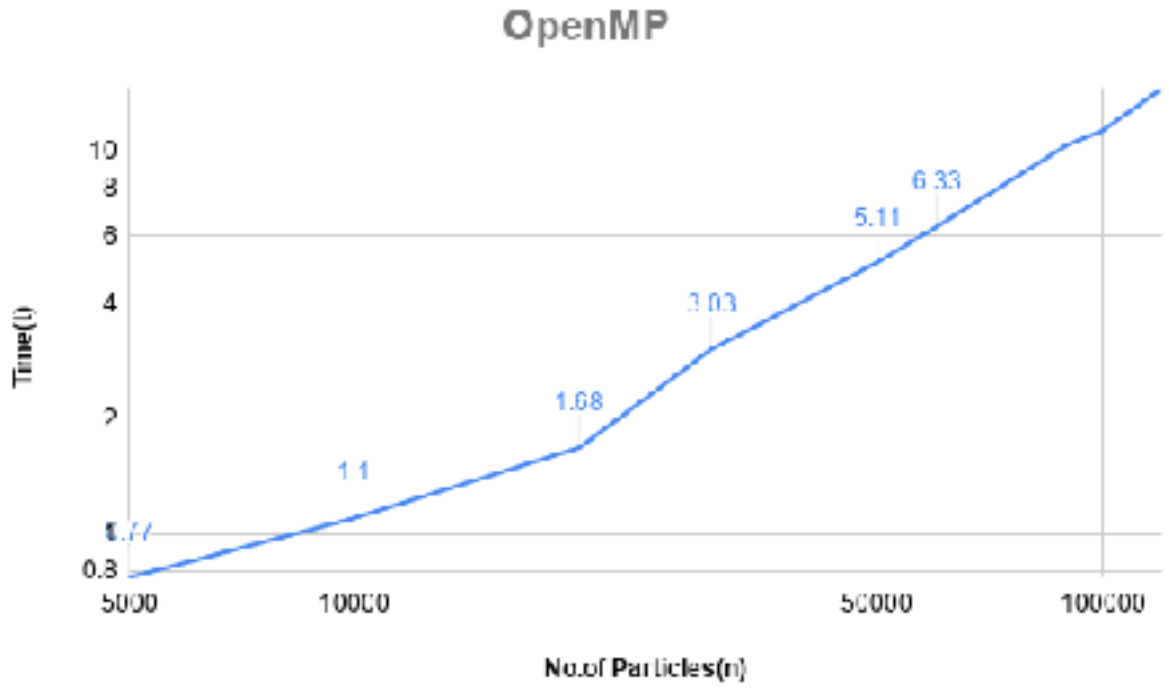

Fig 3. Log-Log Plot of number of particles vs. simulation time for OpenMP

```
Jahnavis-MacBook-Air:proj2 jahnavivalisetty$ export OMP_NUM_THREADS=1
Jahnavis-MacBook-Air:proj2 jahnavivalisetty$ ./openmp -p 1 -n 5000 -no -s openmp.txt
NThreads: 1, n = 5000, simulation time = 0.770426 seconds
Jahnavis-MacBook-Air:proj2 jahnavivalisetty$ export OMP_NUM_THREADS=2
Jahnavis-MacBook-Air:proj2 jahnavivalisetty$ ./openmp -p 2 -n 10000 -no -s openmp.txt
NThreads: 2, n = 10000, simulation time = 1.10561 seconds
Jahnavis-MacBook-Air:proj2 jahnavivalisetty$ export OMP_NUM_THREADS=4
Jahnavis-MacBook-Air:proj2 jahnavivalisetty$ ./openmp -p 4 -n 20000 -no -s openmp.txt
NThreads: 4, n = 20000, simulation time = 1.68696 seconds
Jahnavis-MacBook-Air:proj2 jahnavivalisetty$ export OMP_NUM_THREADS=6
Jahnavis-MacBook-Air:proj2 jahnavivalisetty$ ./openmp -p 6 -n 30000 -no -s openmp.txt
NThreads: 6, n = 30000, simulation time = 3.03439 seconds
Jahnavis-MacBook-Air:proj2 jahnavivalisetty$ export OMP_NUM_THREADS=10
Jahnavis-MacBook-Air:proj2 jahnavivalisetty$ ./openmp -p 10 -n 50000 -no -s openmp.txt
NThreads: 10, n = 50000, simulation time = 5.11679 seconds
Jahnavis-MacBook-Air:proj2 jahnavivalisetty$ export OMP_NUM_THREADS=12
Jahnavis-MacBook-Air:proj2 jahnavivalisetty$ ./openmp -p 12 -n 60000 -no -s openmp.txt
NThreads: 12, n = 60000, simulation time = 6.33722 seconds
Jahnavis-MacBook-Air:proj2 jahnavivalisetty$ export OMP_NUM_THREADS=18
Jahnavis-MacBook-Air:proj2 jahnavivalisetty$ ./openmp -p 18 -n 90000 -no -s openmp.txt
NThreads: 18, n = 90000, simulation time = 10.3983 seconds
Jahnavis-MacBook-Air:proj2 jahnavivalisetty$ export OMP_NUM_THREADS=20
Jahnavis-MacBook-Air:proj2 jahnavivalisetty$ ./openmp -p 20 -n 100000 -no -s openmp.txt
NThreads: 20, n = 100000, simulation time = 11.2705 seconds
Jahnavis-MacBook-Air:proj2 jahnavivalisetty$ export OMP_NUM_THREADS=24
Jahnavis-MacBook-Air:proj2 jahnavivalisetty$ ./openmp -p 24 -n 120000 -no -s openmp.txt
NThreads: 24, n = 120000, simulation time = 14.493 seconds
```

Fig 4. Screenshot of Openmp.cpp execution

The best way to measure the level of parallelization in a program is to examine the speedup factor.

Speedup Factor: The speedup is defined as the ratio of the serial runtime of the best sequential algorithm for solving a problem to the time taken by the parallel algorithm to solve the same problem on p processors. The efficiency is defined as the ratio of speedup to the number of processors.

This is done by running the program with a fixed number of threads for various occasions, extracting the median estimation of that and then divide with the time taken for one thread to run. The plots of the OpenMP execution and the Threads usage can be seen underneath. The speedup-factor is near ideal, however not exactly.
The purpose behind this is various elements. In any case, the main consideration is that all together for our grid enhancement to work we should refresh the situation of every particle inside the framework. This work runs in O(k · n) for each thread and to have optimal speed we should have k to be exceptionally low. Nonetheless, for our situation this linear work appears through effectively after 2 threads. In any case, this basic area of the code is required for our decision
of plan.

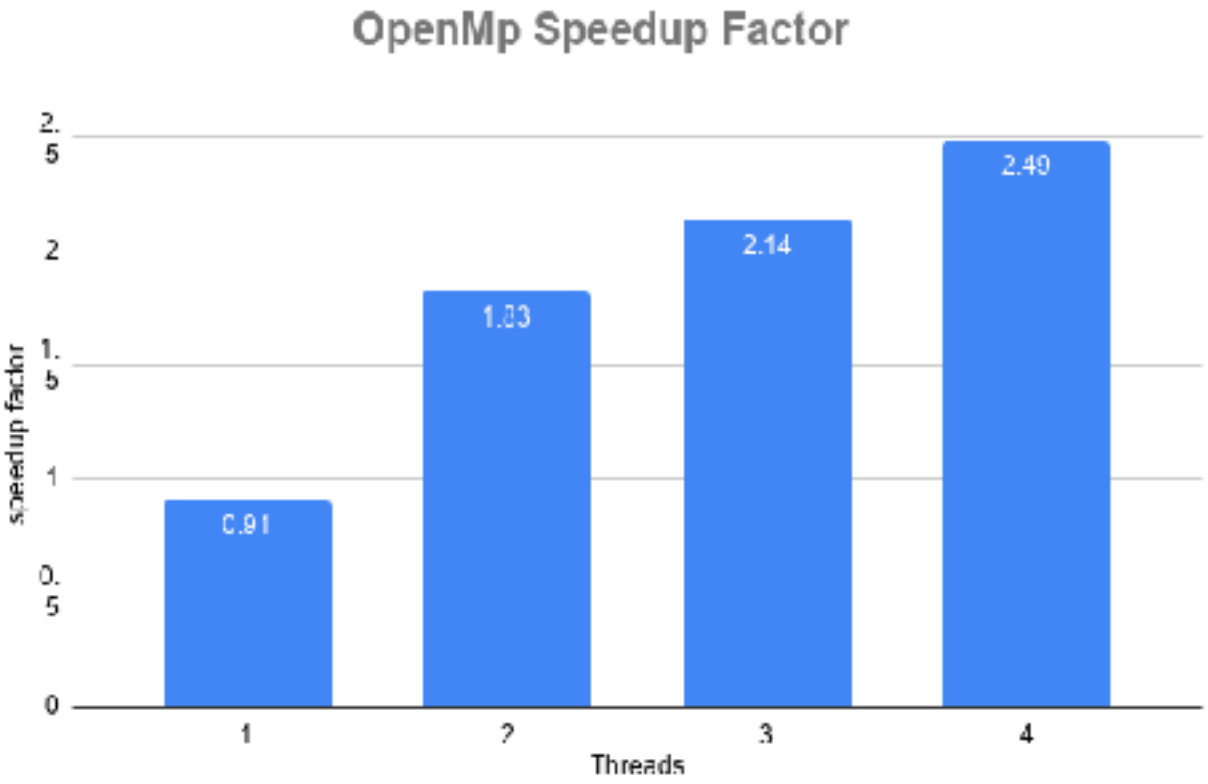

Fig 5. Plot for number of Threads vs. Speedup Factor for OpenMP

```
Jahnavis-MacBook-Air:proj2 jahnavivalisetty$ export OMP_NUM_THREADS=2
Jahnavis-MacBook-Air:proj2 jahnavivalisetty$ ./openmp -p 2 -n 80000 -no -s openmp.txt
NThreads: 2, n = 80000, simulation time = 10.3514 seconds
```

```
Jahnavis-MacBook-Air:proj2 jahnavivalisetty$ export OMP_NUM_THREADS=3
Jahnavis-MacBook-Air:proj2 jahnavivalisetty$ ./openmp -p 3 -n 80000 -no -s openmp.txt
NThreads: 3, n = 80000, simulation time = 8.90348 seconds
```

```
Jahnavis-MacBook-Air:proj2 jahnavivalisetty$ export OMP_NUM_THREADS=4
Jahnavis-MacBook-Air:proj2 jahnavivalisetty$ ./openmp -p 4 -n 80000 -no -s openmp.txt
NThreads: 4, n = 80000, simulation time = 7.65426 seconds
```

```
Jahnavis-MacBook-Air:proj2 jahnavivalisetty$ export OMP_NUM_THREADS=1
Jahnavis-MacBook-Air:proj2 jahnavivalisetty$ ./openmp -p 1 -n 80000 -no -s openmp.txt
NThreads: 1, n = 80000, simulation time = 20.7638 seconds
```

Fig 6. Screenshot for determining speedup factor for openmp

Message Passing Interface(MPI)

Analogous to shared memory implementation with messages instead of shared variables. Split particles based on location onto processors in addition to bins.

Communication Used:
Broadcast – A broadcast is one of the standard collective communication techniques. During which one process sends the same data to all the processors in a communicator.

Message Passing Solution Design
Changing over a shared memory result into a message passing result is a big step. Here we need to update little to have minimum memory sharing.
We allocated each thread with a number of rows of the grid matrix instead of partition of the particles array . Along these lines, each string was answerable for a lot of particles which were geologically close. We then executed functionality for transferring the particles between threads and furthermore sending over the border rows since regardless we need the Moore-neighbourhood of every particle.

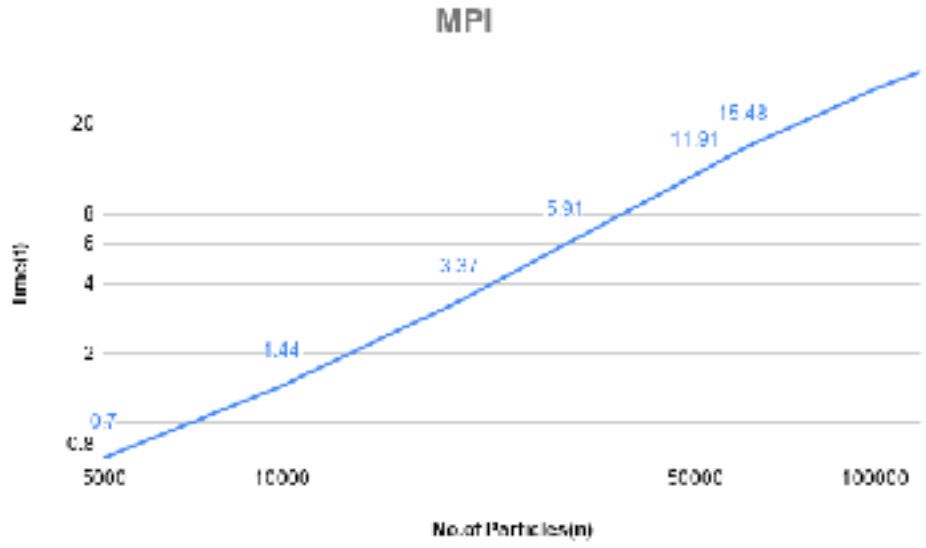

Fig 7. Log-Log Plot of number of particles vs. simulation time for MPI

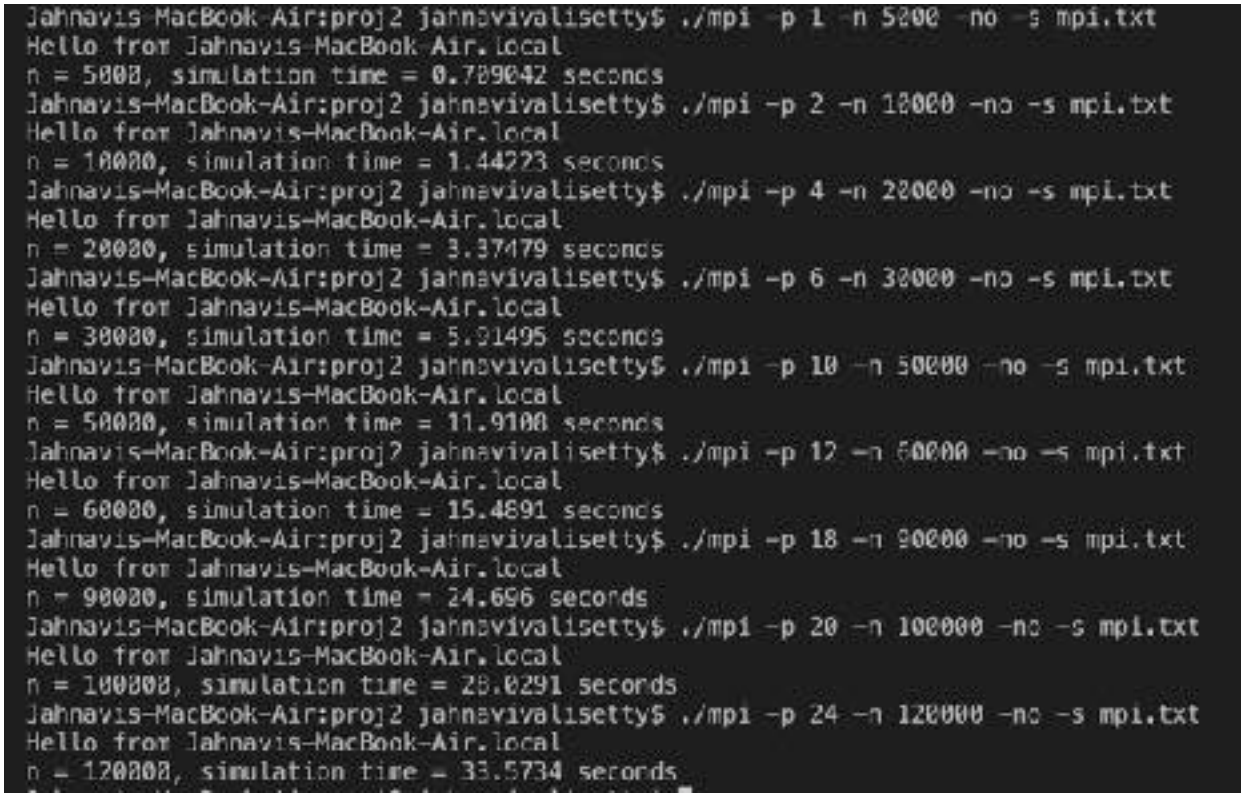

Figure 8. Screenshot of mpi.cpp execution

Time and Overheads:

The MPI execution was the hardest to get productive speedup , unfortunately we were not able to get it as effective as we thought. Message passing is generally more slower than shared memory. While threads utilizing shared memory consistently has a similar data as every other thread, message passing has no shared memory and should hence send messages so as to share data, this obviously takes additional time.

The breakdown of our MPI usage shows that the calculation time of the development of the particles (move and force) pursues the close to ideal speedup similarly just as different implementations. Be that as it may, when we run multiple processes in MPI the message passing takes about 0.4 seconds, this is a huge division of the entire execution time. When running with four procedures the message passing represents 37% of the calculation time. Sadly we couldn't figure out how to expel this overhead. The message passing will take more and more critical division of the computation time in light of the fact that the actual computation will take less and less time as more processes are involved.

Fig 9. Plot for number of Threads vs. Speedup Factor for MPI

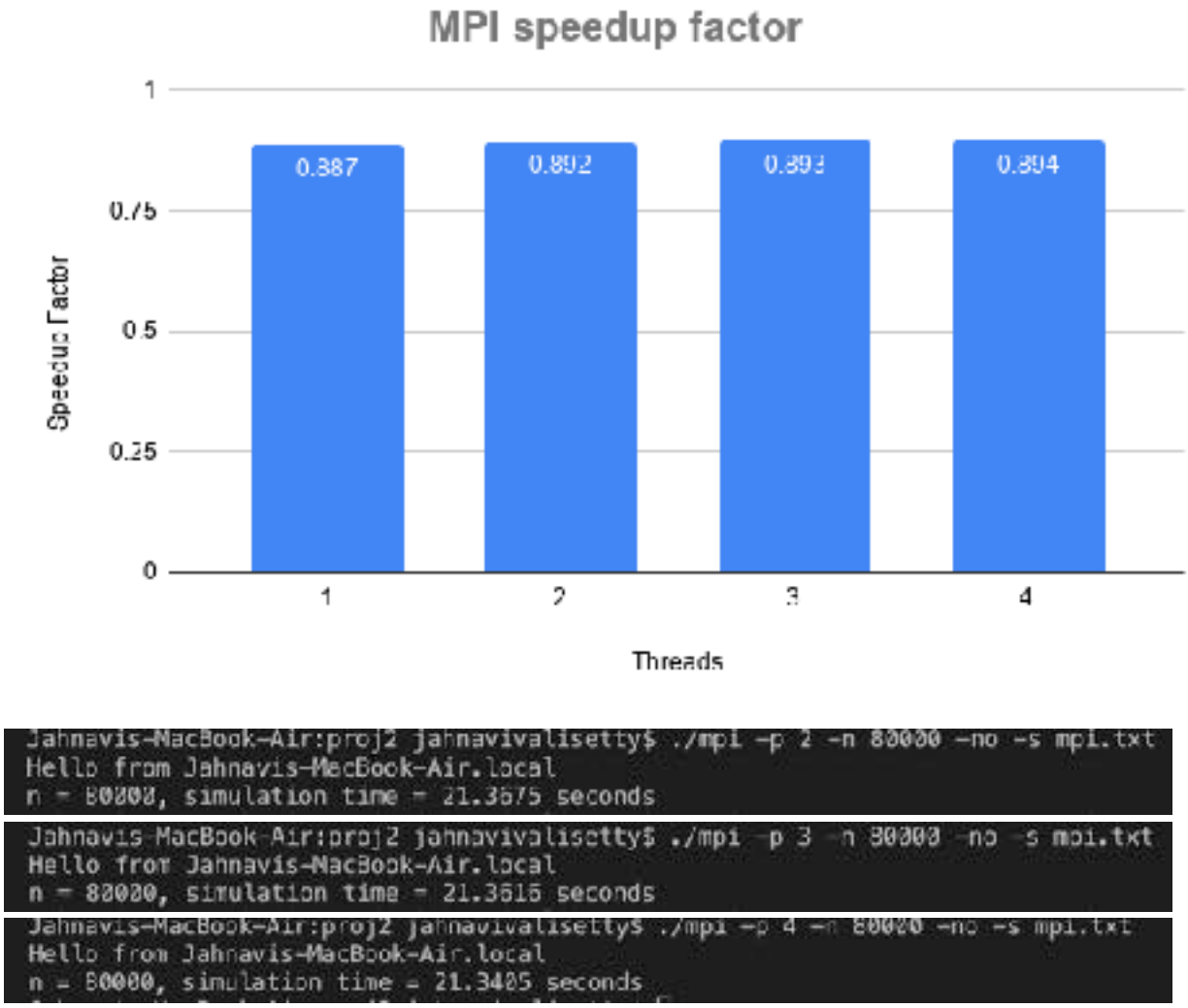

Fig 10. Screenshot for determining speedup factor for mpi