Parallel Code Analysis of N-Body Simulation

Krishna Wadhwani
Aerospace Engineering
160010031

Naman Aggarwal
Aerospace Engineering
160010058

Arjun Rahar
Aerospace Engineering
160010029

Abhay Jindal
Aerospace Engineering
160010060

***Abstract*—This is the ME766 (High Performance Scientific Computing) course project report. Our objective was to make a simulation for the generic n-body system under gravitational forces using appropriate masses and initial conditions, analysing the time efficiency of the code with different parallel programming methods**

I. N-Body Problem

In Newtonian mechanics, N-body problem is the problem of predicting the motion of astral bodies as particles with gravity. Study of the n-body problem involves ideas of classical mechanics mainly Newtons Universal Law of Gravitation and inertia. Kepler's laws are approximations derived from Newton's Universal Law of Gravitation. In this project, we formulate the gravitational forces between the bodies as given by the Universal Law of Gravitation.

II. Project Introduction

The objective of our project is to model the N-body problem in C++, calculating the coordinates of the particles at each time step and then parallelising the program with openMP, MPI and openACC to get a speedup on the performance of the program. We have analyzed the performance of the parallel programming methods and the speedup they provide at different timesteps. We have used two different algorithms for the analysis - one is the brute force method which runs in $O(n^2)$ time while the other is the barnes-hut algorithm which runs in $O(nlogn)$ time. We have used a dataset of 128 bodies, 1024 bodies and 8096 bodies for the analysis.

Further we have used pygame for a dynamic visualization of the interaction and trajectory of bodies in python.

III. Hardware and Compiler Specifications

Machine: Macbook Pro (15 inch, 2016)
Software: macOS High Sierra, Version 10.13.2
Processor Name: Intel Core i7
Processor Speed: 2.7 GHz
Number of Processors: 1
Total Number of Cores: 4
L2 Cache (per Core): 256 KB
L3 Cache: 8 MB
Memory: 16 GB

Compilers used:
g++ - version 4.9.4 (Homebrew GCC 4.9.4_1)
pgc++ - version 18.4
mpic++ - MPICH version 3.2.1

IV. Algorithm

A. $O(n^2)$ algorithm

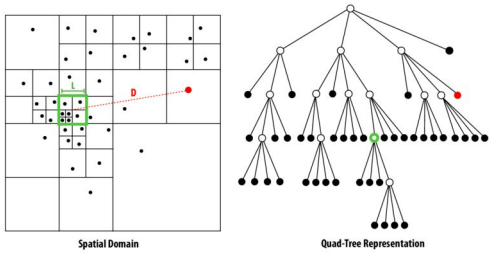
The brute force algorithm has been implemented as follows:

```
for each timestep:
  for bodies 1 to n:
    update position vector of each body:
      r(t+1)=r(t) + v(t)*t + 0.5*a(t)*t*t
    update acceleration vector:
      for every body j where j!=i:
        a(t+1)+=G*m[j]/(r(t+1)*r(t+1))
    update velocity vector of each body:
      v(t+1) = v(t)+0.5*(a(t)+a(t+1))
```

B. O(nlogn) algorithm

The crucial idea in speeding up the brute force algorithm using the barnes hut algorithms is to group nearby bodies and approximate them as a single body. If the group is sufficiently far away, we can approximate its gravitational effects by using its center of mass.

It recursively divides the set of bodies into groups by storing them in a quad-tree. A quad-tree is similar to a binary tree, except that each node has 4 children (some of which may be empty). Each node represents a region of the two dimensional space. The topmost node represents the whole space, and its four children represent the four quadrants of the space. As shown in the diagram, the space is recursively subdivided into quadrants until each subdivision contains 0 or 1 bodies (some regions do not have bodies in all of their quadrants. Hence, some internal nodes have less than 4 non-empty children).



To calculate the net force on a particular body, traverse the nodes of the tree, starting from the root. If the center-of-mass of an internal node is sufficiently far from the body, approximate the bodies contained in that part of the tree as a single body, whose position is the group's center of mass and whose mass is the group's total mass. The algorithm is

fast because we don't need to individually examine any of the bodies in the group. If the internal node is not sufficiently far from the body, recursively traverse each of its subtrees.

V. Serial code

We performed the time analysis of the serial code using brute force algorithm for our 3 datasets, for a duration of 10s by varying the step size at which coordinates of each body is evaluated. The correctness of the output was compared against the two repositories which can be found here and here.

The coordinates were manually compared against some online visualizations and was observed to get closer to the desired output on reducing the timestep.

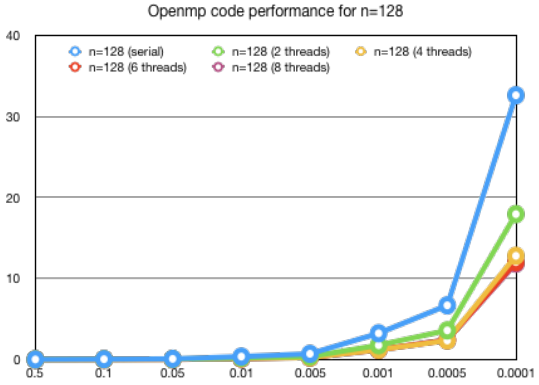
VI. Code Profiling

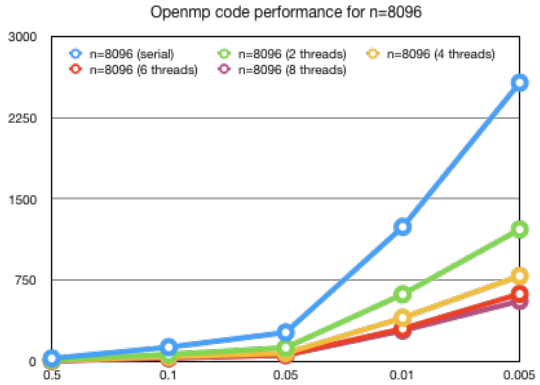
We used gprof to profile our serial code. From the flat profile of the serial code, it was observed that the maximum amount of time is spent in acceleration_update() method due to the large number of times this function is called. So to improve the running time of our code, our parallel program must reduce the timespent in acceleration_update(). Parallelising the acceleration_update() function itself is of no use as the time spent in 1 call of the function is quite small. So to efficiently parallelise the code, we created different number of threads or processes and divided the number of bodies among each such thread/process. Therefore, each thread/process calcuated the entire trajectory of n/num_and as all the threads executed simulataneously, we obtained a significant speed up on the serial code.

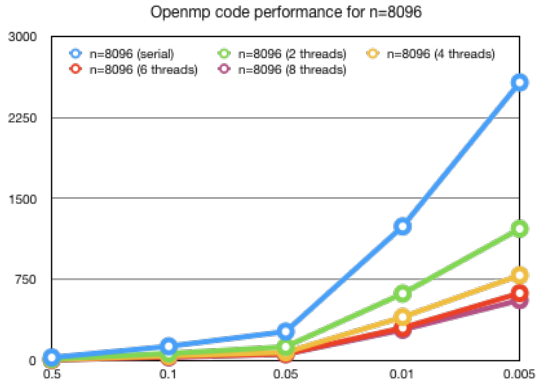
VII. Parallelising with openmp

Parallelisation with opemp involved creating different number of threads and each thread computed the complete trajectory of a certain number of bodies. For example: for n=128 bodies and 4 threads, each thread computed the brute-force algorithm of roughly 32 bodies. The results obtained in this manner will be slightly different from that obtained by the serial code as the order in which the trajectory is calculated is different. The answers obtained converge to the serial code answers on increasing the number of iterations or reducing the timestep.

Analysis of code performance with different number of openmp threads was performed and graphically plotted.





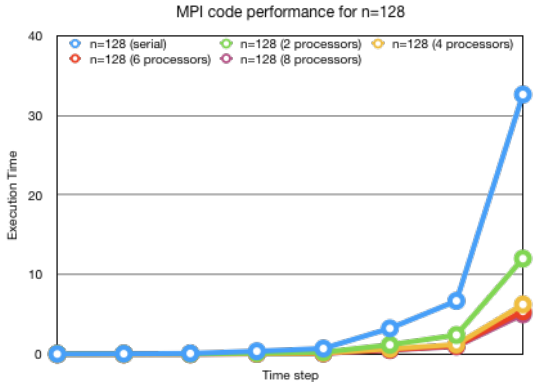


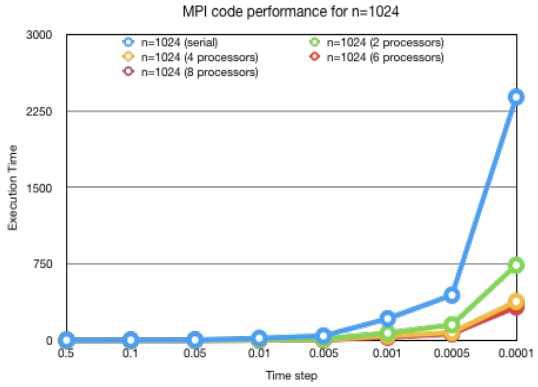
Observation: We got a better speed-up on increasing the number of threads as evident by the graphs. The performance of program with different number of threads and serial code is similar till the time step of 0.05 seconds which is 2000 iterations of coordinates evaluation for each body for 10s duration. For a higher number of iterations in the same duration, openmp parallel code significantly outperforms the serial code. In fact, for higher number of iterations, code with 8 threads performs substantially better than the code with 2 threads

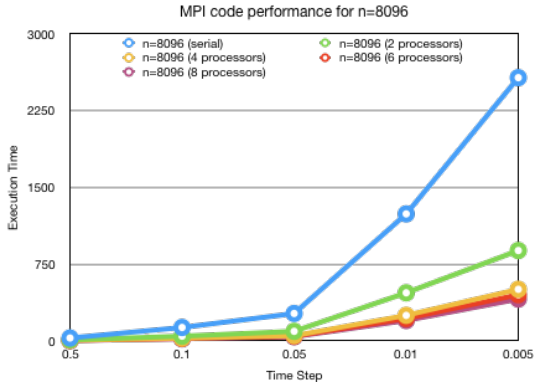
For n=128 bodies and timestep of 0.0001s, we get a speedup of 1.82, 2.65, 2.71 and 2.75 for number of threads = 2,5,6 and 8 respectively.

VIII. Parallelising with MPI

Serial code was parallelised in a manner in which the task of computing accelerations for the different bodies was divided among different processes. There is no data transfer between the processes and all the processes are completely independent. Some errors are induced in the sense that, while computing the accelerations for the bodies, latest position is used for some bodies while the position at previous time-step is used for the others. These errors become negligible for sufficiently small time-steps, and the n-body solution is seen to converge as the time-step becomes sufficiently small with respect to the required precision. Code performance is analysed for 2, 4, 6 and 8 processes for a simulation of duration 10 seconds. Below are the graphs attached for 128 and 1024 bodies respectively.







Observation: For large time steps, serial code and the MPI parallelised code give almost the same performance. Performance of the MPI code improves drastically with smaller and smaller time-steps. Note that, for 1024 bodies, using 8 processes gives performance no better than 6 processes. Using 8 processes is expected to give a significant boost in performance as compared to 6 processes for a large number of bodies and smaller and smaller time-steps. An interesting observation can be made that the speedup provided by MPI drastically increases with the size of the problem as can be seen by the substantial gap in the performance graph of the MPI parallelised code for n=8096 bodies. Numerically, MPI code with 8 processors provides a massive speedup of 6.31 times the serial code for 8096 bodies with a time step of 0.05.

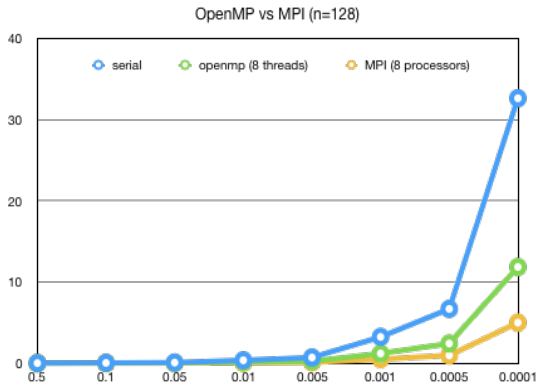
IX. OPENMP v/s MPI

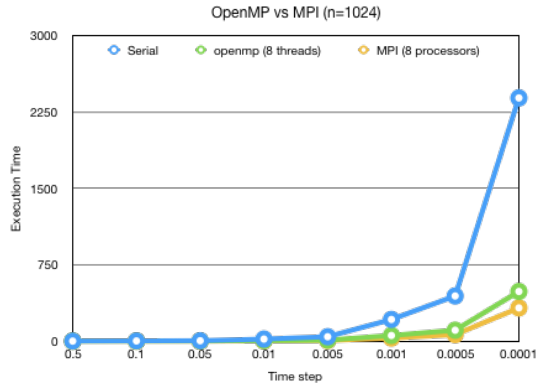
Best performance (corresponding to 8 threads and 8 processes respectively) of openMP and MPI parallelised codes is compared for 128 and 1024 bodies respectively for a duration of 10 seconds.

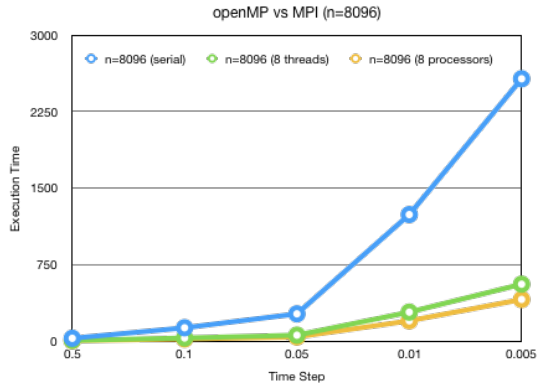
For 128 bodies, for a time-step of 0.0001 seconds, openMP implementation with 8 threads takes time 11.8456 seconds whereas MPI implementation with 8 processes running in parallel takes time 4.996 seconds. Serial code runs in approximately 32.625 seconds.

MPI implementation with 8 processes running in parallel gives a speedup of about 6.5X as compared to a speedup of approximately 2.8X given by openMP implementation using 8 threads.

From the time analysis, it is clear that MPI outperforms openmp for same number of processes in both. This is possibly due to the overhead of creating and destroying threads in case of openmp.







For 1024 bodies, for a time-step of 0.0001 seconds, MPI provides a speedup of about 7.3X as compared to a speedup

of about 5X with openMP for 8 processes and 8 threads respectively.

X. Parallelising with openACC

For each iteration, the task of updating positions and velocities of bodies is parallelised by using an openACC directive as shown. For openACC for-loop pragmas, the default variable sharing method is *shared*. The openACC compiler generates a parallel kernel for the loop identified by the programmer.

```
for each timestep:
    #pragma acc parallel loop
    for i = 1 to n:
        body_array[i].r_update();
        body_array[i].v_update();
```

This code takes more time to run than the serial code. This is because of data transfers between device (GPU) and host (CPU) at each iteration. We put an additional directive before the outer loop which copies the body_array[] array from the host to the device before the execution of the loop block and copies body_array[] from the device, back to the host after completion of the loop block.

```
#pragma acc data copy(body_array[0:size]){
    for each timestep:
        #pragma acc parallel loop
        for i = 1 to n:
            body_array[i].r_update();
            body_array[i].v_update();
}
```

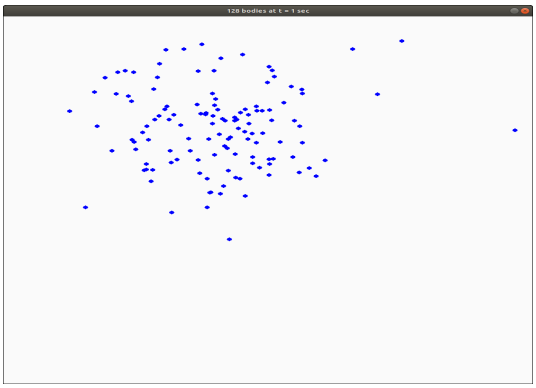
The data transfer overhead is thus accounted for and great speedups are obtained.

The speedup obtained for the openACC code is similar as that due to openmp with 4 threads.

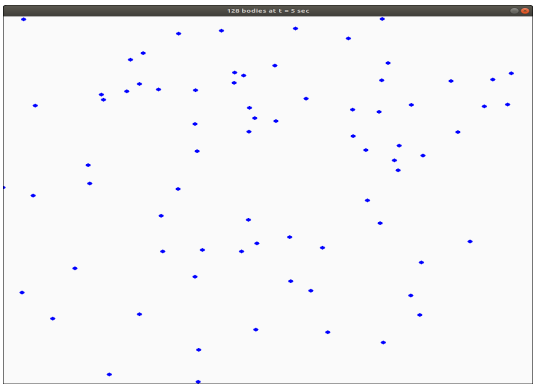
XI. visualization

It is done using the library pygame in python. The positions of the bodies are stored after each timestep iteration which is then used to create the visual simulation.

t = 1 seconds



t = 5 seconds



XII. Conclusion

In conclusion, we parallelised the serial code of n-body problem with the information provided by the profilinf data obtained from gprof and were abke to significantly improve the performance of the serial code with openMP, MPI and openACC.

With openmp, we were able to achieve a speedup of almost 2.8 times for n=128 bodies (0.0001 timestep), 5 times for n=1024 bodies(0.0001 timestep) and 4.5 times for 8096 bodies(0.005 timestep)

With MPI, we were able to achieve a speedup of almost 6.5 times for n=128 bodies (0.0001 timestep), 7.3 times for n=1024 bodies(0.0001 timestep) and 6.3 times for 8096 bodies(0.005 timestep)

Complete code and analysis files can be found at our github repository: here.

Acknowledgment

The authors would like to thank Professor Shivasubramanian Gopalakrishnan, Mechanical Engineering, IIT Bombay for his guidance and teaching.

References

[1] Guy Blelloch and Girija Narlikar, *Parallel Algorithms. Series in Discrete Mathematics and Theoretical Computer Science*, Volume 30, 1997.

[2] Hanno Rein and Daniel Tamayo, *A new paradigm for reproducing and analyzing N-body simulations*, November 2016.

[3] https://en.wikipedia.org/wiki/N-body_problem#Simulation

[4] http://beltoforion.de/article.php?a=barnes-hut-galaxy-simulator&hl=en&s=idTree#idTree

[5] http://arborjs.org/docs/barnes-hut

[6] https://www.prl.res.in/prl-eng/assets/pdf/openacc_gs.pdf

[7] https://intothewave.wordpress.com/2015/02/17/pgi-fortran-compiler-on-linux-mac-os-x/