Analysis of the N-Body Problem with OpenMPI Implementation.

John Sabo
Parallel and Concurrent Programming
Rowan University

May 3 2024

Abstract

This paper examines the implementation and optimization of an n-body simulation problem using both sequential and parallel computing approaches, with the latter leveraging OpenMPI. The project developed two versions of the n-body calculation script: a straightforward sequential version and an optimized parallel version designed to improve performance by distributing computation across multiple processes. Key functionalities included initializing body attributes, computing forces, and updating positions, with the parallel version incorporating MPI functionalities for data distribution and synchronization. Comparative analysis based on output accuracy and performance metrics demonstrated that the parallel implementation achieves significant speedups up to a certain number of processes, although scalability is limited by system architecture. This study highlights the potential and challenges of using parallel computing for complex physical simulations.

I. Program Design

For this project, two implementations of an n-body calculation script were created. One implementation sequentially and one parallelized with OpenMPI. Both scripts contain two main functions. The first of the two functions being 'initializeBodies' which takes the array of bodies and the total number of bodies as arguments. This function will populate each body's position and velocity with randomly generated data. The only difference between the sequential version and the parallel version for this function would be that the parallel version executes this function on the root process or process zero. The second shared function is 'computeForcesAndMove' which will compute each body's next velocity and position based on the positions and velocities of all other bodies. The difference between the sequential and parallel version is that the parallel version's outer for loop only loops through its local bodies. In other words the parallel version is only updating its local bodies and the sequential version updates all bodies. The sequential version contains the function 'gettime' which does what it seems. It gets the time current time in seconds. The parallel version uses 'MPI-Wtime' in its place.

The main functions in either script vary greatly as this is where all of the OpenMPI was implemented. Within the parallel version, we must first allocate a local array of bodies that will be updated and an array of all bodies ensuring that each process has access to all other bodies current positions and velocities. After allocating memory for the array(s), we must calculate the displacement and count of each processes' local bodies locations within the all bodies array. After this, we initialize the all bodies array from the root process or process zero. After initializing the all bodies array from the root process, we then use OpenMPI's broadcast feature. This will allow us to distribute the entirety of the all bodies array to each process. After broadcasting, we must then populate each process's local bodies array with the correct bodies from the all bodies array. We utilize OpenMPI's scatter-v function

to achieve this. By providing the sending buffer, displacement, data count, and receiving buffer, as well as a few boiler plate items, we can populate each process's local bodies array with the correct bodies from the all bodies array. Now that each process has the correct data within it, we can begin the simulation. Each step of the simulation the 'computeForcesAndMove' will be called in each process followed by one of OpenMPI's all-gather-v functions. This function will ensure that the all bodies array is updated correctly in every process. By providing the sending and receiving buffers, the data count, and the displacement, as well as a few boiler plate items, we can succeed in ensuring the all bodies array is correct and identical across all processes. The sequential version is much simpler as it only needs to allocate memory for the all bodies array and then call 'computeForcesAndMove' for the number of steps specified in the command line. After completing the simulation the program will print out the total time taken. For the parallel version this includes from the broadcast function to the end of the simulation loop. For the sequential version, it only contains the simulation loop.

II. Test Process and Planning

By utilizing a script that would print the output of both scripts to separate text files and using the diff command, we can determine whether the parallel and sequential versions of the program produce identical output or not. As the times would rarely ever be identical, I opted to instead print out the final positions and velocities of all of the bodies in both scripts. By doing this, I was able to verify that the scripts produced identical output for ten bodies and ten steps, one hundred bodies and one hundred steps, and one thousand bodies and one thousand steps. With this enlightenment I felt confident that my scripts were both working as intended.

In order to test the performance of the scripts, I would use the same body count and step count in both simulations. For the parallel version, I tested 1,2,4,8, and 16 processes. I chose not to test higher process counts as sixteen processes was already producing a speed-up of less than one, which is effectively a speed-down. I tested each process count for five iterations to get an average execution time.

By printing out the average values and their associated commands to a text file, I was able to easily read in the values into python and create graphs using matplotlib.

III. Data

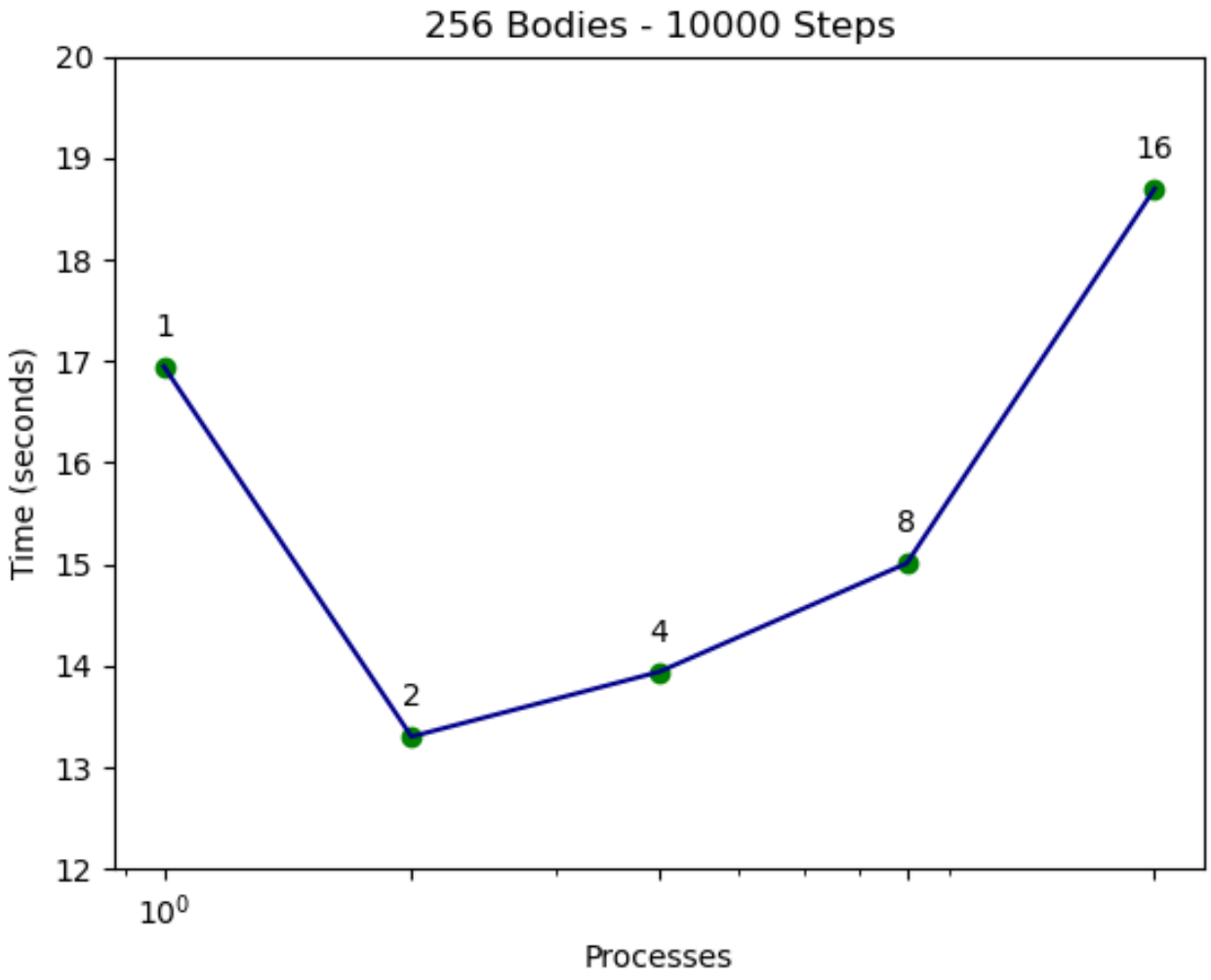

Figure 1: *This is the efficiency graph for 256 bodies ran for 10,000 steps with OpenMPI. Each process count was ran for five iterations to determine its average value.*

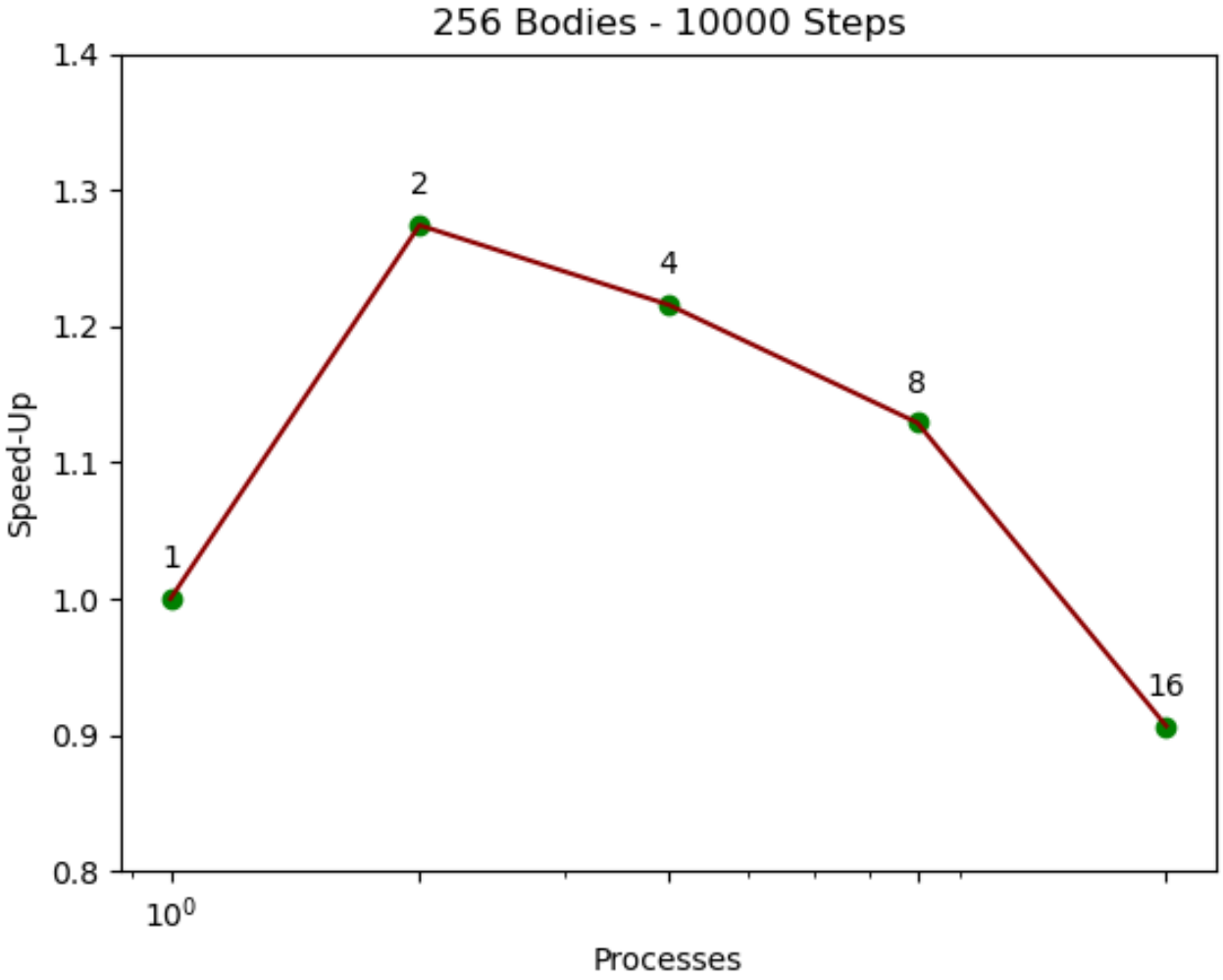

Figure 2: *This is the speed up graph for 256 bodies ran for 10,000 steps with OpenMPI. To determine each process count's speed up, you divide the sequential time of the program by the time taken for that process count. This is why one process has one speed up.*

IV. Analysis

In figure one, we see the time taken for the script versus the process count. For two processes, the time decreases showing that the OpenMPI implementation, being verified, was doing the same thing the sequential script was doing, in less time. As we move to four and more processes we see the time taken increase but still stay under one process's time taken. I believe this to be the result of two possibilities. One possibility being that transferring excessive data between more and more processes is resulting in decreased efficiency when we get to sixteen process. The second possibility being that only have two cores on my machine does not allow MPI to work to its fullest potential. I would be curious to get this working on a machine with more cores in its processor.

In figure two, we see the speed up for the script versus the process count. The speed up graph is an inverse of the efficiency graph. We see that the speedup is at its highest for two process. This leads me to attribute the cause to one of the previously mentioned possibilities.

V. Conclusion

My implementation of an optimization of the n-body problem with OpenMPI was a success. It showed a positive speedup with increased processes to a certain point and was able to produce identical output as the sequential version of the script.

I found great enjoyment in this project as I have a love for physics and space. I did make an attempt to utilize javascript's web workers in order to improve my original solar system simulation. Web Workers are not the most optimal solution for optimization in the browser as javascript is a single threaded language. After a wholehearted attempt, I gave up on that for now and decided to instead use OpenMPI.

Overall, I would say I was successful in completing my original objective: optimizing the calculation for the n-body problem.