# N-Body Simulation using MPI and MPI+OpenMP

**Jared Frazier**
Vrije Universiteit
Student ID: 2795544
`jared.frazier@student.uva.nl`

## Abstract

The $N$-body problem is a classic problem in which $N$ discrete bodies mutually interact in a dynamical system consisting of positions, forces, and velocities [1]. In this report, a parallel algorithm for simulating the evolution of this dynamical system was implemented using C and MPI. Simulations were conducted for 100 timesteps, the number of discrete bodies was $N \in \{512, 1024, 4096, 10000\}$, and the number of MPI processes was between 2 and 128. All experiments were conducted using the Vrije Universiteit's DAS-5 cluster's ASUS nodes with Intel E5-2630v3 CPUs. As a bonus optimization, OpenMP was added to the MPI implementation. For 4 processes with $N = 10000$, the speedup and efficiency for the pure MPI program were 2.31 and 57.64%, respectively. For MPI+OpenMP (aka Hybrid MPI) on the same problem the speedup and efficiency were 7.80 and 194.95%, respectively.

## 1 Introduction

**Algorithm 1** DIRECT $N$-BODY SIMULATION($B$, $\Delta t$, $t_{end}$)
The general scheme loops over small time intervals, first computing the forces on each body for that interval, then updating each body's position and velocity. $B$ is the list of bodies and $t_{end}$ and $\Delta t$ are simulation parameters for the total simulation time and time-step, respectively.

```
1:  for $t$ from 0 to $t_{end}$ by $\Delta t$ do
2:      for each body $b$ in $B$ do                                  ▷ Clear Forces
3:          Set the forces acting on $b$ to 0
4:      $S \leftarrow \{\}$                                           ▷ Ensures $\frac{N(N-1)}{2}$ Interactions
5:      for each body $p$ in $B$ do                                  ▷ Force Calculation
6:          $S \leftarrow S \cap \{p\}$
7:          for each body $q$ in $B \setminus S$ do                  ▷ Uses Negative for Optimization
8:              Update force contribution of q acting on p with the negative force contribution of p acting on q
9:      for each body $b$ in $B$ do
10:         Update position of $b$
11:     for each body $b$ in $B$ do
12:         Update velocity of $b$
```

$N$-body problems are a large class of problems in which a set of $N$ bodies mutually interact in a dynamical system [1]. As the number of bodies $N$ in a system increases, the number of mutually increasing interactions grows as $N^2$. In the case of planetary motion in $\mathbb{R}^2$, the state variables of such a dynamical system are positions , forces , and velocities. Since such a system is governed by classical mechanics, the equations of motion are used to model the evolution of the system over time [1, 2]. The simplest scheme for simulating the $N$-body problem considers $\frac{N(N-1)}{2}$ interaction terms, where the number of interaction terms is halved due to force symmetries. Algorithm 1 depicts this simple scheme and is adapted from [1] with changes based on the details of the serial

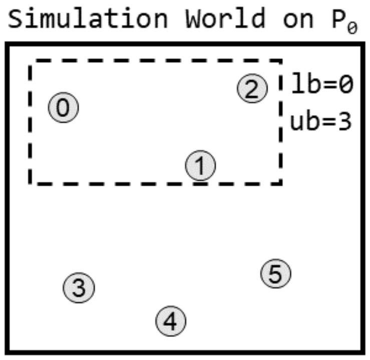

Figure 1: Simulation world with 6 bodies on processor 0 ($P_0$). The dash-lined box represents the bodies for which $P_0$ will perform computations. The lower bound ($lb$) and upper bound ($ub$) indices for these bodies are 0 and 3, respectively.

implementation provided by the course instructor. The remainder of this report is organized as follows: Algorithm 1 is analyzed to determine the regions for which parallelism can be exploited, a description of the experimental setup and tools is provided, and experimental results are discussed and compared with theoretical performance analysis. Note that in this report, pure MPI or MPI refers to the baseline parallel algorithm implemented while MPI+OpenMP or Hybrid MPI refers to the extra feature implemented.

## 2 Method

In this section, the serial algorithm is analyzed to determine the parallel regions, the performance metrics are listed, and the experimental design is elucidated.

### 2.1 Parallelism in the serial algorithm

It is clear from Algorithm 1 that the computational complexity is $\mathcal{O}(N^2)$. That is, the force calculation is the computational bottleneck of the algorithm and the remaining steps (position updates, velocity updates, and clearing forces) are trivially parallel. Thus, a clear source of parallelism for Algorithm 1 is with data parallelism facilitated by block data decomposition [3].

**Algorithm 2** PARTITION $N$-BODIES($N$, $P$)
Loop over remaining bodies and assign them as evenly as possible to the remaining processes. $N$ is the number of bodies and $P$ is the number of processes.

```
1: $R \leftarrow [N/P$ for $i$ in 0:P-1]    ▷ Stores the number of bodies that the $i^{th}$ process will be responsible for updating
2: $l \leftarrow N\%P$
3: while $l > 0$ do
4:     $R[l] \leftarrow R[l] + 1$
5:     $l \leftarrow l - 1$
```

For $N$ bodies and $P$ processes, Each process could perform computations on $N/P$ bodies. For $N\%P \neq 0$, naturally some processes will have more bodies on which to perform computations than others. To load balance this decomposition scheme, a simple strategy was employed: after allocating $N/P$ bodies to each process, add a single additional body for a given process to handle and continue to do this while simultaneously subtracting one from a state variable $N\%P$ until the state variable is zero. Algorithm 2 depicts this scheme.

While a process will need the position information about all other bodies stored on all other processes in order to calculate forces, a given process will initially perform force updates (lines 4 to 8 in Algorithm 1) on the subset of bodies assigned to it using the information in Algorithm 2. For the bodies assigned to a particular process, the force updates for these bodies can leverage force symmetries. In order to encapsulate the concept of bodies assigned to a process, lower and upper bound indices corresponding to the bodies that need to be handled on a particular process are computed. Thus, the parallel scheme is depicted in Algorithm 3–implemented using C and the Message Passing Interface (MPI). The corresponding domain decomposition with reference to the lower and upper bound indices is illustrated in Figure 1. Before Algorithm 3, the root process loads the $N$-body simulation data and scatters it to all other processes. Since the `ComputeForces` function requires the position information of other bodies that are updated on different processes, the `MPI_Allgatherv` call collects the updated bodies and then scatters it back to processors. This operation is done inplace to avoid allocating memory for potentially large arrays that would only store position information and then might be copied into the data members of a body `struct`.

**Algorithm 3** PURE MPI DIRECT $N$-BODY SIMULATION($B$, $lb$, $ub$, $\Delta t$, $t_{end}$)
A parallel version of Algorithm 1 that uses lower and upper bound indices to determine which bodies a given process will update. $B$ is the list of bodies, $lb$ is the index in $B$ of the first body for which a process is responsible, $ub$ is the index in $B$ of the last body for which a process is responsible, and $\Delta t$ and $t_{end}$ are simulation parameters.

```
1: for $t$ from 0 to $t_{end}$ by $\Delta t$ do
2:     ClearForces(B, lb, ub)
3:     ComputeForces(B, lb, ub)          ▷ Uses Negative Force Optimization on $B_i \; \forall \; i \in [lb..ub)$
4:     ComputePositions(B, lb, ub)
5:     ComputeVelocities(B, lb, ub)
6:     MPI_Allgatherv(MPI_IN_PLACE, ..., B)          ▷ Gather All Updated Bodies
```

## 2.2 Extra feature: Hybrid MPI

As an additional effort to investigate the performance effects of MPI combined with thread-level parallelism, Open Multi-Processing (OpenMP) was added to the `for` loops of the functions in lines 2 through 5 of Algorithm 3. Loop work sharing using the `#pragma omp parallel for` directive was directly added to the `for` loops of functions 3 through 5 in Algorithm 3. The `ComputeForces` function in line 3 is implemented using two `for` loops. The first `for` loop uses the negative force optimization and computes the forces simultaneously on pairs of bodies for which a process is responsible (see the dashed box in Figure 1). The second `for` loop computes the forces of the bodies for which the processor is *not* responsible on the bodies for which the processor *is* responsible. Using Figure 1 as an example, the first `for` loop uses the negative force optimization on bodies 0-2, while the second `for` loop updates the forces on 0-2 acted on by bodies 3-5. The `#pragma omp parallel for` directive cannot be used on the first `for` loop of `ComputeForces` because there is a loop-carried dependency, which is error-prone in OpenMP [4], due to the force optimization.

## 2.3 Experimental design

In order to assess the performance of both the MPI and Hybrid MPI algorithms, the wall time (`MPI_Wtime`) of process 0 was recorded before the beginning of the simulation and then immediately after. In this way, the runtime of the program was computed. Speedup and efficiency were computed and were used to describe algorithm performance. A variety of $N$-body problem sizes were chosen. The number of bodies $N$ was chosen to be in $\{512, 1024, 4096, 10000\}$ and the number of iterations $t_{end}$ was 100. The simulation parameter $t_{end}$ was 100 because the serial program runtime for $N = 10000$ and $t_{end} = 100$ was roughly 500 seconds, which was less than the maximum allowed runtime on the cluster used for experimentation. Experiments were conducted with all combinations of computes nodes in $\{2, 4, 6, 8\}$ and number of CPUs per node in $\{1, 4, 16\}$. The Vrije Universiteit's DAS-5 cluster's ASUS nodes with Intel E5-2630v3 CPUs were used for all experiments. All experiments were repeated three times, and the average of the resulting runtimes was used to compute performance metrics. Each set of experiments and corresponding metrics was used to evaluate the algorithms's strong and weak scaling as discussed in Section 3.

Another set of experiments was conducted to see what percentage of the runtime was dedicated to the `MPI_Allgatherv` call. In this way, the communication overhead can be directly evaluated for MPI, and this provides insight into observed speedups for both the pure and Hybrid MPI implementations. Note that the runtime reported in figures is from data collected during the experiments in which the communication overhead was *not* measured. In this way, the runtime reported in the figures was unaffected by potential slowdowns imposed from communication overhead measurements.

## 3 Results and Discussion

In this section, the pure MPI and Hybrid MPI results are discussed. The theoretical estimation of performance is also included.

### 3.1 Pure MPI

Figure 2 shows the pure MPI speedup while Table 1 displays the specific values in the figure. Note that while the y-axis of the figure displays speedup in base 10, the x-axis shows the number of processes as base 2. The ideal speedup is plotted, and it adopts a curved shape due to the scaling of the axes, but with respect to interpretation, analysis of the speedup results can still be classified in the routine ways such as superlinear and sublinear. In this case, for all problem sizes and all number of processes, the speedup is sublinear. The maximum speedup attained was 45.56 (see Table 1) for $N = 10000$. As the number of processes $P$ increased from 32 to 128, the speedups for different problem sizes began to diverge from one another. Problem sizes $N = 512$ and $N = 1024$ followed a similar trend with respect to their speedup in that there appears to be no significant increase until $P = 96$, however, with more processes (i.e., $P = 128$), the speedup decreases. The speedup trend for $N = 4096$ and $N = 10000$ contrast this by asymptotically increasing. However, while the speedup continues to increase for $N = 10000$ at $P = 128$, the speedup decreases for $N = 4096$ and $P = 128$. Moreover, the MPI algorithm demonstrates reasonable strong and weak scalability. It is clear from Figure 2 that as the problem size remained fixed, a sublinear speedup was observed. In this particular problem the strong scalability is of arguably greater importance, that is the number of processors growing proportionally with the problem size, because in practice $N$-body simulations can be conducted at cosmological scales of $10^{11}$ bodies. Thus, while parallelizing the direct $N$-body algorithm $\mathcal{O}(N^2)$ is not conducive to such scales, for the largest problem size tested, the Figure 2 demonstrates the reasonable application of the Algorithm 3 to systems with on the scale of $10^4$ bodies.

$$\begin{aligned} SpeedUp(P) &= \frac{1}{S_f + \frac{P_f}{P}} && \text{Amdahl's Law} \\ &= P - S_f(P-1) && \text{Gustsafson-Barsis's Law} \end{aligned} \tag{1}$$

In order to estimate the theoretical speedup, a consideration of the parallel $P_f$ and serial fraction $S_f$ of Algorithm 3 was performed and Amdahl's and Gustafson-Barsis's laws were invoked. While these laws do not take into account communication overhead, they serve as sufficient estimators of potential parallel performance despite the laws suggesting different limits [4]. If, based on Algorithm 3, the functions in lines 2 through 5 representing the parallel portion of the program for which a final synchronization is needed in line 6, which effectively reverts the variables at the beginning of the loop to the values they *would* have if the program were serial, then the serial fraction $S_f$ of the code is approximately 20% and the remaining fraction of the code for which parallelization is possible is $80\%$. Under Amdahl's law, the maximum $SpeedUp(P)$ that could be attained with $P = 128$ processors, $S_f = 0.20$, and $P_f =$ would be approximately 4.8 while under Gustafson-Barsis's law, the maximum speedup predicted under the same parameters is 102. The theoretical estimation for speedup under Amdahl's law suggests two things: (1) Amdahl's law is generally not sufficient to estimate the theoretical speedup for this particular case, and the experimental data indicates far better performance, and (2) that more liberal estimates of the parallelizable portion of the program might be required. With respect to comment (2), the scheme to estimate the parallelizable portion of the program is fairly simplified, and a more complicated performance model might be needed to estimate theoretical speedup. With respect to the Gustafson-Barsis estimate, it is clear that with $P = 128$ and $N = 10000$, that the observed speedup of $45.56$ is below the estimate by a factor of 2. In either case, since the computational overhead is $\mathcal{O}(\frac{N^2}{P})$ and the communication overhead sends

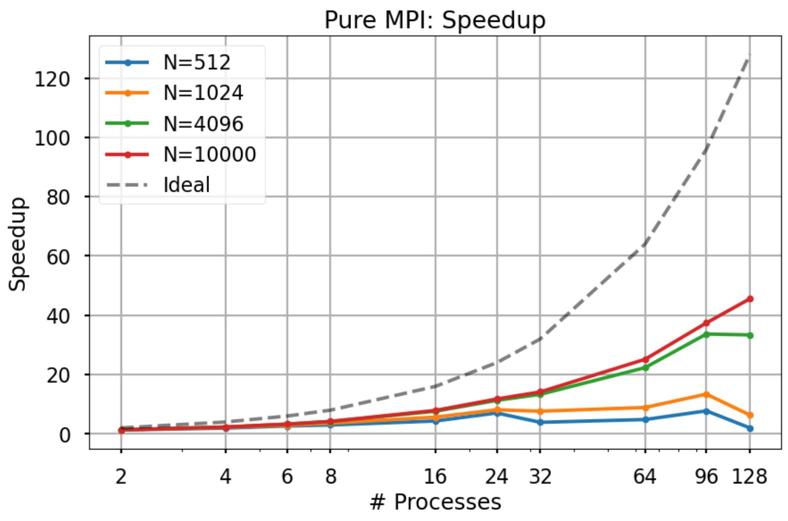

Figure 2: Pure MPI speedup for different problem sizes and number of processes.

Table 1: Pure MPI speedup.

| | **Processes** | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **N** | **2** | **4** | **6** | **8** | **16** | **24** | **32** | **64** | **96** | **128** |
| **512** | 1.3 | 1.93 | 2.65 | 3.0 | 4.33 | 7.04 | 3.89 | 4.83 | 7.75 | 2.0 |
| **1024** | 1.33 | 2.17 | 2.95 | 3.61 | 5.64 | 8.11 | 7.64 | 8.89 | 13.39 | 6.33 |
| **4096** | 1.35 | 2.31 | 3.27 | 4.17 | 7.62 | 11.29 | 13.32 | 22.33 | 33.67 | 33.37 |
| **10000** | 1.34 | 2.31 | 3.29 | 4.2 | 7.9 | 11.75 | 14.14 | 25.14 | 37.42 | 45.56 |

messages of size $\mathcal{O}(N)$, then the ratio of computation to communication $\mathcal{O}(N/P)$ suggests that the algorithm is suitable for $N >> P$.

Figure 3 shows the pure MPI efficiency while Table 2 displays the specific values in the figure. The maximum efficiency for the algorithm was $67.53\%$ for $P = 2$ and $N = 4096$. There is a decreasing trend in efficiency for all problem sizes as the number of processes increases. Notable deviations in this trend occur for $P = 24$ and $P = 96$ wherein there is a slight increase in efficiency followed by a steep decline. This appears to be related to the number of CPUs per node that were assigned as a simulation parameter. For $P = 24$, the number of nodes was $6$ while the number of CPUs per node was $4$. For $P = 96$, the number of nodes was $6$ and the number of CPUs per node was $16$. For these configurations of simulation parameters, and for the steepest increases in efficiency corresponding to $N = 512$, Table 4 shows that a smaller percentage of the total runtime is composed of the MPI communication.

Figure 4 shows the pure MPI runtime for different processes, and notably displays the significant differences between the single process algorithm and the parallel algorithm. Focusing on the max size problem where $N = 10000$, Table 3, whose values correspond to Figure 4, shows that the fastest runtime corresponded to $P = 128$ and was $11.23$ seconds compared with $511.48$ seconds for the serial algorithm. The same trends with respect to efficiency and $P = 24$ and $P = 96$ are also present. There is also a steep increase in the runtime for $P = 128$ and $N = 512$ for Figure 4, which can explained by the fact that for $N > P$ but not $N >> P$, the relative amount of communication as shown in Table 4 is $96.6\%$.

## 3.2 Hybrid MPI

The same data was reported for the Hybrid MPI implementation as for the pure MPI implementation. For Figure 5, the speedup is superlinear, and the number of processes $P$ for which this holds true increases as the problem size increases. Referring still to Figure 5, the Hybrid MPI algorithm is superlinear for $N = 10000$ and $P \leq 24$, while for $N = 512$ and $N = 1024$, the Hybrid MPI

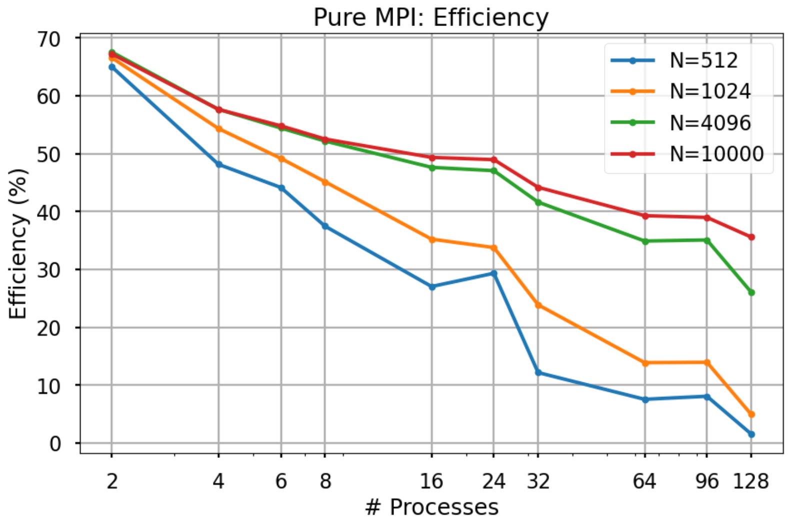

Figure 3: Pure MPI efficiency for different problem sizes and number of processes.

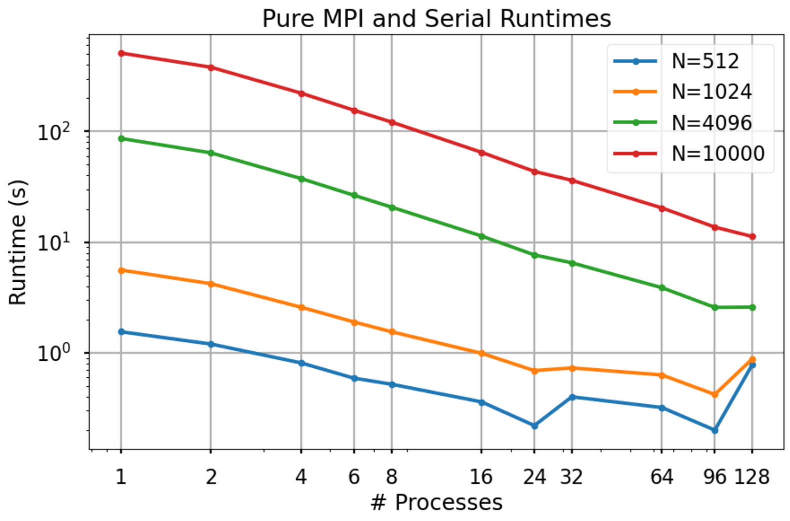

Figure 4: Pure MPI (# Processes $>$ 1) runtime and serial runtime (# Processes = 1) comparison for different problem sizes.

Table 2: Pure MPI efficiency (%).

| | **Processes** | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **N** | **2** | **4** | **6** | **8** | **16** | **24** | **32** | **64** | **96** | **128** |
| **512** | 64.97 | 48.14 | 44.15 | 37.46 | 27.04 | 29.33 | 12.17 | 7.55 | 8.08 | 1.56 |
| **1024** | 66.55 | 54.32 | 49.18 | 45.14 | 35.22 | 33.77 | 23.88 | 13.89 | 13.95 | 4.95 |
| **4096** | 67.53 | 57.65 | 54.42 | 52.17 | 47.63 | 47.05 | 41.62 | 34.9 | 35.08 | 26.07 |
| **10000** | 67.2 | 57.64 | 54.8 | 52.52 | 49.35 | 48.96 | 44.18 | 39.28 | 38.98 | 35.59 |

Table 3: Pure MPI (processes $> 1$) and serial (processes = 1) runtime (s).

| | **Processes** | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **N** | **1** | **2** | **4** | **6** | **8** | **16** | **24** | **32** | **64** | **96** | **128** |
| **512** | 1.55 | 1.2 | 0.81 | 0.59 | 0.52 | 0.36 | 0.22 | 0.4 | 0.32 | 0.2 | 0.78 |
| **1024** | 5.6 | 4.21 | 2.58 | 1.9 | 1.55 | 0.99 | 0.69 | 0.73 | 0.63 | 0.42 | 0.88 |
| **4096** | 86.54 | 64.08 | 37.53 | 26.51 | 20.73 | 11.35 | 7.66 | 6.5 | 3.87 | 2.57 | 2.59 |
| **10000** | 511.48 | 380.58 | 221.84 | 155.56 | 121.74 | 64.78 | 43.53 | 36.18 | 20.35 | 13.67 | 11.23 |

Table 4: Pure MPI communication time as a percentage of total runtime.

| | **Processes** | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **N** | **2** | **4** | **6** | **8** | **16** | **24** | **32** | **64** | **96** | **128** |
| **512** | 0.83 | 5.13 | 5.3 | 15.83 | 38.47 | 37.37 | 65.43 | 85.37 | 85.53 | 96.6 |
| **1024** | 1.17 | 4.23 | 5.87 | 9.77 | 20.87 | 24.57 | 38.77 | 73.23 | 63.2 | 87.33 |
| **4096** | 0.27 | 1.13 | 1.57 | 2.48 | 4.67 | 5.77 | 9.35 | 9.77 | 19.17 | 36.4 |
| **10000** | 0.73 | 0.77 | 0.83 | 2.23 | 2.73 | 2.7 | 3.9 | 4.5 | 6.83 | 11.4 |

demonstrates superlinear speedup for $P \leq 6$. Notably, the maximum speedup attained for the Hybrid MPI algorithm was $38.16$ while for the pure MPI algorithm it was $45.56$ (see Table 5 and Table 1). A significant difference between these two values is that the former required $6$ compute nodes and $4$ CPUs per node, while the latter required $8$ compute nodes and $16$ CPUs per node. In terms of computational resource requirements, the Hybrid MPI algorithm is less costly, but could not achieve the scalability of the pure MPI implementation.

With respect to efficiency, Figure 6 demonstrates significantly higher efficiencies when compared to the pure MPI algorithm. From Table 6, the maximum efficiency achieved by the Hybrid MPI algorithm was $215.22\%$ for $N = 10000$ and $P = 6$. In contrast with the pure MPI algorithm, the extrema for the Hybrid MPI algorithm is a much wider range. The minimum efficiency of $0.08\%$ with $P = 128$ is 20 times less efficient than the pure MPI implmentation for the same problem size of $N = 512$ (see Tables 6 and 2).

The justification for the Hybrid MPI more extreme performance becomes clearer based on the runtime and communication overhead data available in Figure 7, Table 7, and Table 8. The minimum runtime in the Hybrid MPI algorithm $0.15$ seconds compared with $0.20$ seconds in the pure MPI algorithm. However, the communication overhead in Table 8 in the Hybrid MPI algorithm indicates that a greater proportion of the total runtime is dedicated to implicit barriers and thus synchronization that is enforced by the OpenMP directives. This is especially apparent because as the number of processes $P$ increases for fixed size problems $N$, Figure 7 shows that the runtime of the parallel algorithm actually becomes worse than the serial implementation. This suggests that while the current Hybrid MPI algorithm can achieve high performance for fewer computational resources than the pure MPI, not only does it scale poorly, but without prior performance data for the Hybrid MPI, it might be difficult to use for very large problems for which a large number of computing resources are available. However, for limited computing resources, the Hybrid MPI implementation clearly is more efficient. For example, $N = 10000$ and $P = 4$, the Hybrid MPI runtime is $65.59$ seconds compared with $221.84$ seconds for the parallel algorithm (see Tables 7 and 3).

Table 5: Hybrid MPI speedup.

| | **Processes** | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **N** | **2** | **4** | **6** | **8** | **16** | **24** | **32** | **64** | **96** | **128** |
| **512** | 1.59 | 6.07 | 10.4 | 1.05 | 0.54 | 0.55 | 0.18 | 0.11 | 0.11 | 0.11 |
| **1024** | 1.62 | 7.0 | 11.49 | 3.24 | 1.88 | 1.9 | 0.65 | 0.4 | 0.4 | 0.39 |
| **4096** | 1.64 | 5.85 | 9.4 | 13.91 | 15.25 | 18.55 | 7.54 | 5.26 | 5.57 | 5.53 |
| **10000** | 1.64 | 7.8 | 12.91 | 14.14 | 24.16 | 38.16 | 19.35 | 18.14 | 21.65 | 23.67 |

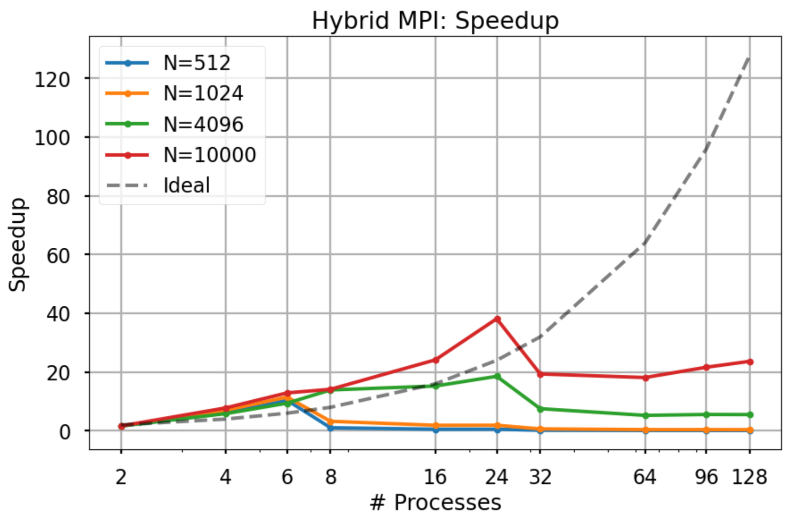

Figure 5: Hybrid MPI speedup for different problem sizes and number of processes.

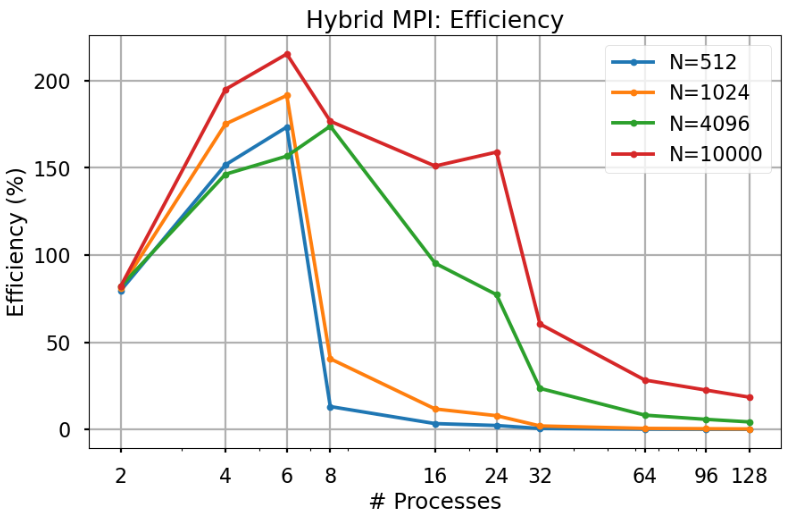

Figure 6: Hybrid MPI efficiency for different problem sizes and number of processes.

Table 6: Hybrid MPI efficiency (%).

| | **Processes** | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **N** | **2** | **4** | **6** | **8** | **16** | **24** | **32** | **64** | **96** | **128** |
| **512** | 79.58 | 151.69 | 173.36 | 13.13 | 3.37 | 2.29 | 0.57 | 0.17 | 0.12 | 0.08 |
| **1024** | 81.14 | 175.09 | 191.46 | 40.54 | 11.76 | 7.91 | 2.03 | 0.62 | 0.42 | 0.3 |
| **4096** | 81.9 | 146.31 | 156.67 | 173.85 | 95.29 | 77.3 | 23.56 | 8.22 | 5.8 | 4.32 |
| **10000** | 81.96 | 194.95 | 215.22 | 176.7 | 151.0 | 158.99 | 60.46 | 28.34 | 22.56 | 18.49 |

Table 7: Hybrid MPI (processes $> 1$) and serial (processes = 1) runtime (s).

| | **Processes** | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **N** | **1** | **2** | **4** | **6** | **8** | **16** | **24** | **32** | **64** | **96** | **128** |
| **512** | 1.55 | 0.98 | 0.26 | 0.15 | 1.48 | 2.88 | 2.83 | 8.47 | 13.99 | 13.9 | 14.38 |
| **1024** | 5.6 | 3.45 | 0.8 | 0.49 | 1.73 | 2.97 | 2.95 | 8.64 | 14.04 | 14.0 | 14.51 |
| **4096** | 86.54 | 52.83 | 14.79 | 9.21 | 6.22 | 5.68 | 4.66 | 11.48 | 16.45 | 15.54 | 15.65 |
| **10000** | 511.48 | 312.01 | 65.59 | 39.61 | 36.18 | 21.17 | 13.4 | 26.44 | 28.2 | 23.62 | 21.61 |

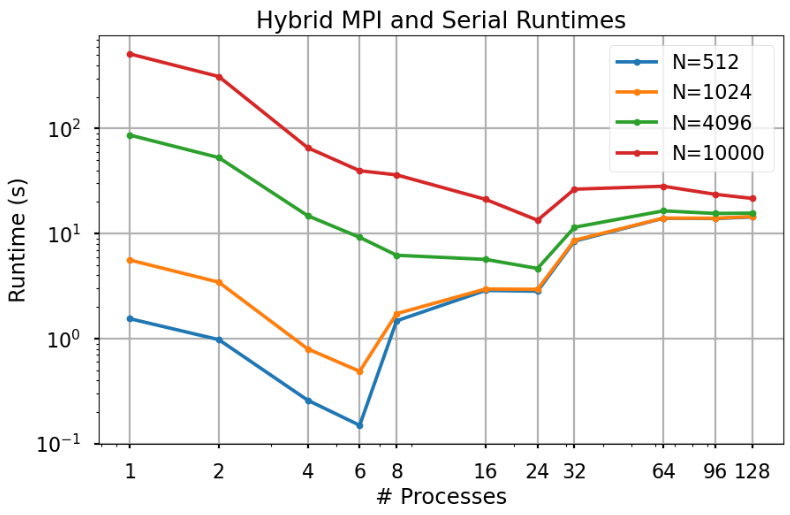

Figure 7: Hybrid MPI (# Processes $> 1$) runtime and serial runtime (# Processes = 1) comparison for different problem sizes.

Table 8: Hybrid MPI communication time as a percentage of total runtime.

| | **Processes** | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **N** | **2** | **4** | **6** | **8** | **16** | **24** | **32** | **64** | **96** | **128** |
| **512** | 9.27 | 11.97 | 15.67 | 31.27 | 26.37 | 26.47 | 25.43 | 20.63 | 22.0 | 23.0 |
| **1024** | 1.07 | 8.2 | 6.97 | 19.58 | 23.43 | 24.5 | 24.95 | 23.13 | 20.5 | 24.1 |
| **4096** | 0.53 | 9.6 | 9.87 | 6.75 | 16.07 | 16.17 | 20.7 | 20.63 | 22.5 | 24.9 |
| **10000** | 0.7 | 4.63 | 8.63 | 8.33 | 9.9 | 7.7 | 9.67 | 14.87 | 18.03 | 19.93 |

## 4 Conclusions

Two MPI implementations for $N$-body simulations with a range of problem sizes and computational resources were investigated. It was found that the pure MPI algorithm scales better with increasing processes; however, for limited computational resources, the Hybrid MPI implementation significantly out performs the pure MPI implementation. Additionally, I had used MPI as part of the Programming Large-Scale Parallel Systems course, so that basic familiarity made this assignment somewhat smoother to implement. Lastly, the use of several seminal MPI resources facilitated learning useful design patterns for parallel programs [3–5].

## References

[1] A. Brandt, “On distributed gravitational n-body simulations,” 3 2022. [Online]. Available: http://arxiv.org/abs/2203.08966

[2] H. Gould, J. Tobochnik, and W. Christian, *An Introduction to Computer Simulation Methods Applications to Physical System*, 3rd ed. Open Source Physics, 2016.

[3] M. J. Quinn, *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Education Group, 2003.

[4] R. Robey and Y. Zamora, *Parallel and High Performance Computing*. Simon and Schuster, 2021.

[5] P. S. Pacheco, *Parallel Programming with MPI*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1996.