A Distributed and Shared Memory Approach to Parallelising the N-Body Problem

MARCO MARASCO (WORD COUNT: 2123)

This paper introduces a parallelised version of a sequential algorithm to solve the cardinal N-Body problem. The N-Body problem, published by Sir Isaac Newton in his Pricipia, considers the interaction and movements of bodies within a given space, particularly, looking at how they each influence each other's behaviour. Solutions for $N \leq 2$ can be solved analytically, but for $N > 2$, the computational complexity becomes infeasible, and simulation methods are required. The N-Body problem has applications in a plethora of research domains, including astrophysics, structural biology, and even machine learning. This paper utilises a HPC platform to investigate the different methods to parallelise the algorithm, further using the OpenMPI and OpenMP APIs for distributed/shared memory parallelisations. The final algorithm produced by this paper achieved a 20× speed up using 12 nodes with 2 cores each, compared to the sequential algorithm, indicating a near linear speed up with respect to the number of cores.

Additional Key Words and Phrases: N-Body problem, Astrophysics, Parallel Computing, OpenMPI, OpenMP, Distributed Memory Computing, Shared Memory Computing

1 INTRODUCTION

Tracking the motion of bodies that interact with each other with a given system has been an long-standing computational task in physics research [9] [10]. This fundamental "N-Body" problem is concerned with resolving the forces each body has acted upon it, and tracing its movement over time, and has applications in many fields such as astrophysics, fluid mechanics, and molecular dynamics.

1.1 Formal Definition

Consider a set of N bodies in 3-dimensional space, with each body $i \in 1, ..., N$ having position $p_i^t = \langle x_i^t, y_i^t, z_i^t \rangle$ and velocity $v_i^t = \langle x_i^t, y_i^t, z_i^t \rangle$, and time t where $t \geq 0$.

Consider bodies that move under the force of gravity in the x-direction:

$$F_x = \frac{G \cdot m_i \cdot m_j}{d^2} \cdot \left(\frac{x_i - x_j}{d}\right) \tag{1}$$

where F_x is the gravitational force in the x direction, G is the gravitation constant, m_i and m_j are the masses of the two bodies, and d is the eucildean distance between them. The acceleration, change in velocity, and change in position are all updated independently in each direction.

A body i accelerates according to the Netwon's second law of motion, by $a_i = \frac{F_i}{m_i}$, where F_i is the force acting on body i. The net force F_i^{net} on a given body is the sum of all forces (in all dimensions) being acting on the body by all other bodies in the system.

Over a given time interval, δ, a body i accelerates under a constant force, and its velocity changes to:

$$v_i' = v_i + \frac{F_i^{net} \cdot \delta}{m_i} \tag{2}$$

and the position updates to:

$$p_i' = p_i + v_i \cdot \delta \tag{3}$$

1.2 Related Parallelisation Work

The sequential algorithm (elaborated in section 2) provides an exact calculation of body positions/velocities at teach time step, however, with an $O(N^2)$ time complexity, many researchers have gone on to produce algorithms with smaller complexities. Notable examples include the Barnes-Hut Algorithm [1], P^3M method [6] and Fast Multipole Method [5]. Whilst these approaches are admirable, they all are approximation algorithms, whereas the sequential algorithm this paper is concerned with is an exact algorithm.

As $O(N^2)$ time complexity can be far too large in practice, this paper will seek to parallelise this algorithm to increase its utility in practice. Due to the longevity and pervasiveness of the N-body problem, there has been substantial previous work into assessing paralleisation techniques to speed up the computation time, with specific work on the exact algorithm.

Speck et al. [7] produced new work demonstrating a highly scalable and parallelisable approach to the problem. Their method differs from previous work by introducing a "future-based request buffer", which is used between different nodes for communication. Coupled with their HPX platform, their implementation allowed nodes to begin computation whilst still receiving all the data it required. Their work demonstrated a 128x speed up for execution on 128 nodes, compared to execution on a single node.

Gangavarapu et al. [3] experimented with OpenMP and Nvidia CUDA implementations to parallelsise force/velocity computations, and distribution of body data (e.g. mass). This study produced competitive results compared to the sequential implementation of the algorithm, with a maximum speed up of 250x using a single TESLA C-2050 GPU.

Totoo and Loidl [12] published a paper moving away from traditional imperative language approaches, focusing on how parallelisation in functional languages could be used to speed up runtime for the N-body problem. Their work produced a Haskell implementation that exploited the ability to update velocities in parallel in the sequential algorithm, achieving a 7.2x speed up using an 8 core machine over the sequential implementation.

2 PARALLEL ALGORITHM

This paper intends to investigate and produce a parallelised version of the sequential $O(N^2)$ algorithm, by investigating different exploitations of the algorithm and parallelisation frameworks. The below pseudocode outlines the steps of the sequential algorithm, and will be referenced throughout discussion as **Algorithm 1**.

Algorithm 1 Sequential Algorithm

```
for t ← 0 to t_max do
    for i ← 0 to N − 1 do                          ▷ Compute forces acting on N bodies
        for j ← 0 to N − 1 do
            if i ≠ j then
                Calculate force of body j on body i (Eq: 5)
                Update net force on body i
            end if
        end for
    end for

    for i ← 0 to N − 1 do                          ▷ Update positions and velocities
        Update position and velocity of body i (Eq: 2, 3)
    end for
end for
```

2.1 Parallelisation Platforms

To perform the experimentation for this research, the University of Melbourne's HPC platform, Spartan [8], was chosen as the appropriate infrastructural platform to conduct testing, as it provided access to non-trivial amounts of computational resources. For the implementations, written in C++ [11], two different APIs are investigated for their applicability in parallelisation. The first is OpenMPI [4] and the second is OpenMP [2]. They provide abstractions for distributed/shared memory paradigms respectively.

2.2 Distributed Memory

In order to adapt the sequential algorithm to work for a distributed memory approach, data dependencies must be identified as distributed memory approaches (likely) do not have access to all data in the program.

2.2.1 Message Passing. A first approach to distribute work amongst p nodes would be for each node to compute the forces/position/velocities for N/p bodies, rather than all N bodies. This introduces a data dependency - at the beginning of each time step each node must have the current position/velocity of each body. To solve this, all nodes must have the initial data for all bodies before simulation. This can be conveniently solved by OpenMPI's **Broadcast** function, which allows a single designated node to send (input) data to all nodes in the MPI communicator.

The next issue to solve is how nodes will communicate updated data for their N/p bodies. Initial thoughts were to reuse OpenMPI's **Broadcast** for each node, but this would require p separate broadcasts. A more appropriate solution would be to employ OpenMPI's **AllGather** function. This allows all nodes to each send/receive data with a single call, essentially reducing all updated data into a receive buffer. This algorithm can be found in **Algorithm 2, Appendix A**.

Note: This study only considers when N is divisible by p, this is extendable via OpenMPI's **AllGatherv** function, but was not implemented as it was not necessary to the efficacy of the parallelisation.

2.2.2 Experimentation. Due to the high message complexity of this approach, experimentation is required to assess the best use of OpenMPI's capabilities for transmitting data. Two tests will be conducted to optimise the runtime for transmission. First will be comparing the type of data structure passed to OpenMPI, particularly an **MPI_Type_struct**, versus an **MPI_Type_contiguous** array (using **Algorithm 2, Appendix A**. Second will be assessing whether message packet size impacts performance, particularly a trade-off between a single **AllGather** for the position/velocity updates, or two smaller **AllGather** calls for position/velocity respectively. This algorithm can be found in **Algorithm 3, Appendix A**.

2.3 Shared Memory

OpenMP allows for exploiting parallelism with a shared-memory model. This requires identifying computations that could be performed in parallel i.e. a group of computations have no inter-dependencies. In the for-loop in **Algorithm 1** line 2, the computing forces can be easily parallelised, as for a given body i, computing the forces from bodies $j \in N \setminus \{i\}$ are completely independent. This same exploitation can be applied to the update for-loop in **Algorithm 1** line 11. These can be both implemented with OpenMP's **parallel for** loop call (see **Algorithm 4, Appendix A**).

A second approach to parallelism is parallelising doubly nested for loops using OpenMP's *collapse* clause, which converts a collection of nested loops into 1. The first use-case for this approach is computing forces of bodies $j \in N \setminus \{i\}$ on body i, as all bodies in the inner and outer loops of **Algorithm 1** lines 2,3 are independent of other bodies within their loop.

This collapse technique can be used in the loop of **Algorithm 1** line 11. Updating positions/velocities in each axis is independent, so the update function can be modelled as a doubly nested for loop, which can be collapsed into 1 (see **Algorithm 5, Appendix A**).

2.4 Hybrid OpenMP OpenMPI

Using distributed/shared memory parallelisation in tandem may provide a powerful parallelised version of **Algorithm 1**. Once the results of the experiments have been analysed, the best of both the distributed/shared memory parallelisations will be combined and evaluated against the sequential algorithm.

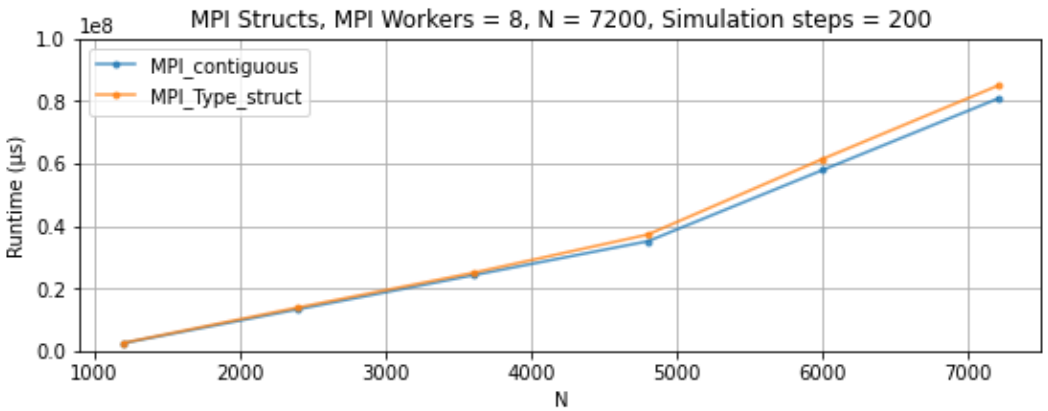
3 EXPERIMENTS

3.1 Data Generation

At runtime each body was initialised by a given process with a psuedo-random non-zero mass, position, and velocity. To minimise outliers/noise, all tests were run over 3 instances, and their average runtime was recorded for plotting. As the details of the current Spartan network infrastructure are not public, it is assumed communication times between nodes are equal w.r.t. to a given message.

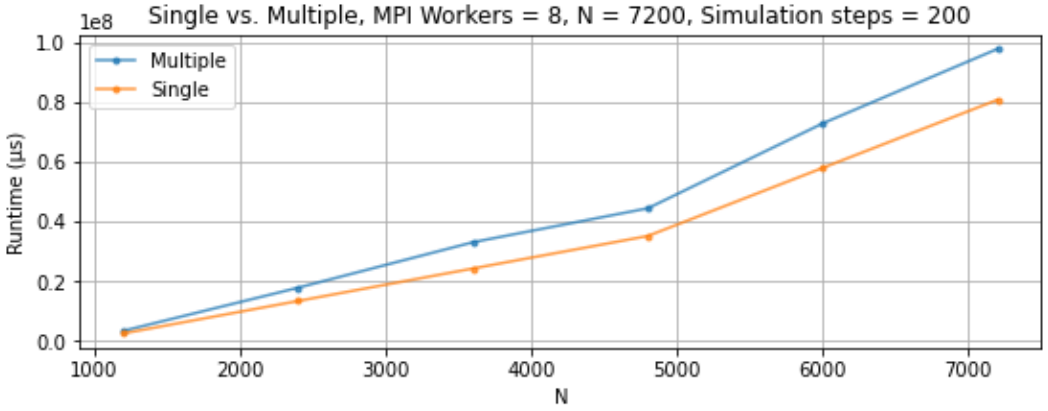
3.2 OpenMPI

3.2.1 Message Struct Types. This experiment was performed to evaluate the different types of OpenMPI data structures, outlined in **section 2.2.2**. An experiment was run comparing the use of *MPI_Type_contiguous* vs. *MPI_Type_struct*. Prior to experimentation at scale, it was unknown which approach would be superior.



The results indicate the *MPI_Type_contiguous* to be slightly superior, on average 5% faster than *MPI_Type_struct*. This is likely due to the *MPI_Type_struct* also having to encode additional data for handling potentially heterogeneous types/memory offsets, thus increasing the size of messages to be sent.

3.2.2 Smaller Packets vs. More Packets. This experiment compared **algorithms 2 & 3** outlined in **section 2.2.2**, they will be referred to as the single/multiple approach respectively. It is suspected that in the lower values for N, the runtime for the single approach will dominate the multiple approach, however, as N increases and the packet sizes increase, their difference between runtimes may decrease due to larger amounts of data being transmitted. Note: the sequential algorithm is omitted due to skewing the graph scale (the single approach had an average of 8.5× speed up).

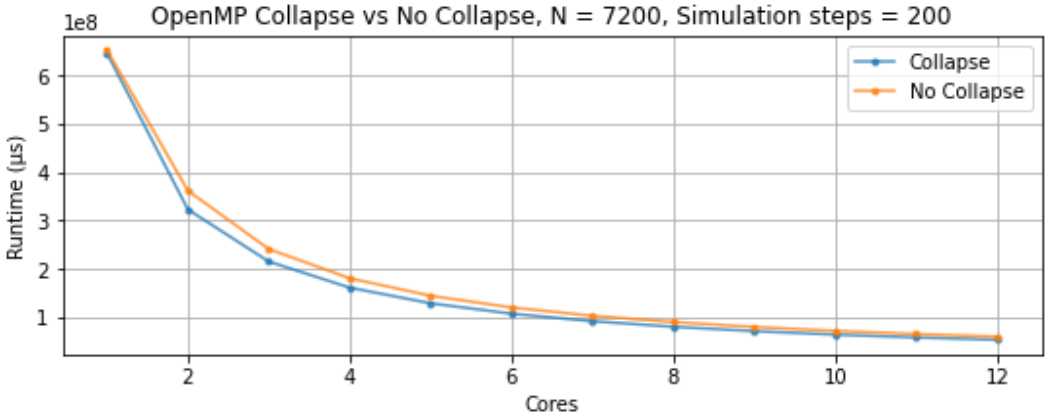


The results from this experiment were as predicted. For $N = 1200$, the multiple approach was approximately 35% slower than the single approach, but this reduced to approximately 21% for $N = 7200$. Whilst this trend indicates the multiple approach may be more scalable for N, within the parameters of N for this study, the single approach will produce better runtime results.

3.3 OpenMP

3.3.1 Collapsed Loops. As mentioned in **section 2.3**, some of the for loops in the sequential can be parallelised, and further, some of the doubly-nested for loops can be collapsed and parallelised as a single loop. It is expected that the collapsed approach will be the superior method, as collapsing and then allocating loops to processors in one step should reduce allows for a more optimal allocation of work to the processes, compared to each process being assigned multiple loops in the non-collapsed method.

The below graph compares the approaches and how they scale with respect to the number of cores they have access to. It is expected as the number of cores grows, the runtime will decrease accordingly, however, at scale it is unknown which approach will perform faster. (Note: only $N = 7200$ was test of respect that Spartan is a shared platform).



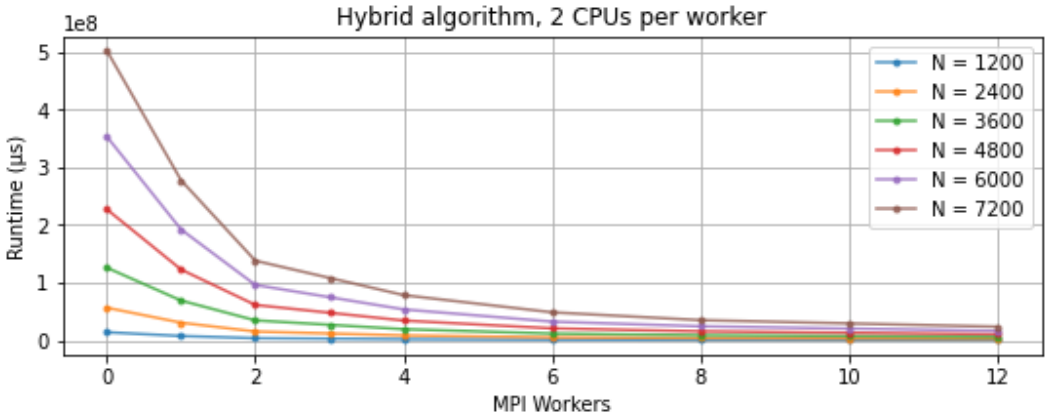
The non-collapsed approach outperformed the collapsed approach, being on average 12% faster than the collapsed approach. The reason for this may be due to the overheads incurred from

assigning each collapsed loop. It is also pleasing to observe by using just OpenMP, there was an approximate linear speed up compared to the sequential algorithm with respect to increasing the number of cores.

3.4 Hybrid

Using the results from sections **3.2** and **3.3**, a final algorithm (see **Algorithm 6, Appendix A**) was created that combined the best methods for parallelising the sequential algorithm, by utilising both OpenMPI and OpenMP. The final hybrid implementation utilised 2 cores per node, and testing was run over various node sizes (Note only 2 cores were used for this section due to: the large number of tests to be executed and; respect that Spartan is a shared platform).

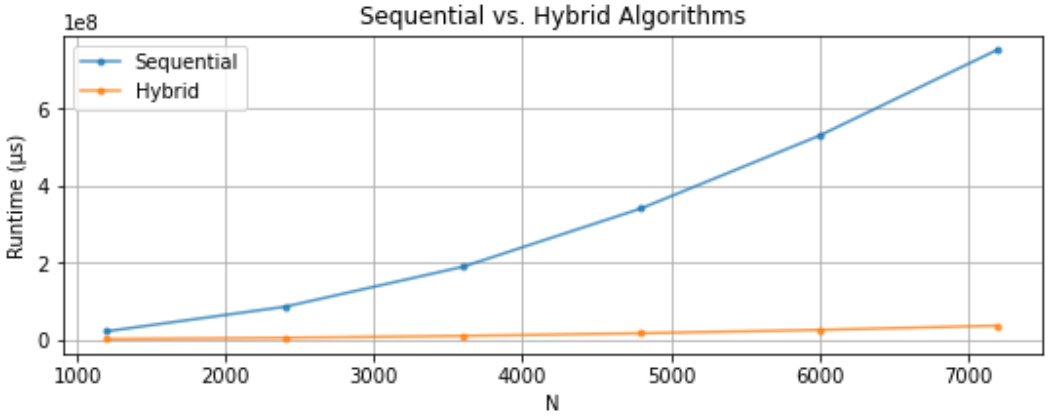
To investigate the final implementation, tests were done to investigate how the runtime sped up with respect to the size of N, but also the number of workers in the OpenMPI network. Note: 0 workers in the below graph represents the sequential algorithm runtime.



This graph affirms one of the main goals of this study, to demonstrate the speedup for a range of different numbers of cores/nodes. Analysis of the runtimes indicated the algorithm sped up proportional to the number of workers, ie running on 12 workers was approximately 24 times faster than 1 worker.

3.5 Sequential vs. Hybrid

Having determined the fastest resource allocation from **Section 3.4** of 12 workers with 2 cores each, this study can now properly examine if all the previous experimentation has paid off for a final parallelised algorithm (see **Algorithm 6, Appendix A**) compared to the sequential algorithm. Below shows how the sequential and hybrid algorithms performed as the value of **N** increased.



Pleasingly, the algorithm achieves an average speed up of 20× for having 24 more cores. This non-linear speed up is likely due to runtime impacts from OpenMPI communication during runtime.

4 CONCLUSION

The aim of this paper, to parallelise the sequential exact approach to the N-body problem has been met. Over the course of this study, the various experimentations with the OpenMPI and OpenMP APIs provided a wealth of knowledge of how best to deploy these tools on the Spartan cluster. The sequenital algorithm was adpated for use in a distributed-memory environment by assigning each process to compute $\frac{N}{p}$ bodies, and the experimentation regarding OpenMPI indicated that despite message size, less communication calls were better for runtime, and that the **MPI_Type_contiguous** array was the best MPI data structure to use. The sequential algorithm was further modified to be parallelised within a shared-memory environment, by computing the forces/updates for the bodies in parallel. The experiments with OpenMP indicated that doubly nested for loops were better to not be collapsed, as this had degredation on runtime. The final parallel algorithm achieved a pleasing 20× speed up for 12 MPI workers with 2 cores, when compared to the sequential algorithm, that is a near linear speed up with respect to the number of cores.

Whilst this study has produced sound results, there is still the capacity to further improve and extend this implementation. Particularly from the work described by Gangavarapu et al. [3] in **section 1.2**, they demonstrated massive speed ups using GPUs. This study did not evaluate the viability of GPUs when implementing the algorithm's shared-memory adaptations, but from Gangavarapu et al.'s work utilising them with OpenMPI may provide more speed up

REFERENCES

[1] Josh Barnes and Piet Hut. 1986. A hierarchical O(N log N) force-calculation algorithm. *Nature* 324, 6096 (01 Dec 1986), 446–449. https://doi.org/10.1038/324446a0

[2] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Comput. Sci. Eng.* 5, 1 (Jan. 1998), 46–55. https://doi.org/10.1109/99.660313

[3] Tushaar Gangavarapu, Himadri Pal, Pratyush Prakash, Suraj Hegde, and Vasantha Geetha. 2019. Parallel OpenMP and CUDA Implementations of the N-Body Problem.

[4] Richard L. Graham, Timothy S. Woodall, and Jeffrey M. Squyres. 2006. Open MPI: A Flexible High Performance MPI. In *Parallel Processing and Applied Mathematics*, Roman Wyrzykowski, Jack Dongarra, Norbert Meyer, and Jerzy Waśniewski (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 228–239.

[5] L Greengard and V Rokhlin. 1987. A fast algorithm for particle simulations. *J. Comput. Phys.* 73, 2 (1987), 325 – 348. https://doi.org/10.1016/0021-9991(87)90140-9

[6] R. Hockney and J. Eastwood. 1966. Computer Simulation Using Particles.

[7] Z. Khatami, H. Kaiser, P. Grubel, A. Serio, and J. Ramanujam. 2016. A Massively Parallel Distributed N-body Application Implemented with HPX. In *2016 7th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA)*. 57–64.

[8] Bernard Meade, Lev Lafayette, Greg Sauter, and Daniel Tosello. 2017. Spartan HPC-Cloud Hybrid: Delivering Performance and Flexibility. https://doi.org/10.4225/49/58ead90dceaaa

[9] 1642-1727 Newton, Isaac. 1846. *Newton's Principia : the mathematical principles of natural philosophy.* First American edition, carefully revised and corrected / with a life of the author, by N. W. Chittenden. New-York : Daniel Adee, 1846. https://search.library.wisc.edu/catalog/999810640302121 Translation of Philosophiae naturalis principia.;Also available in PDF and TIFF formats from the NOAA Central Library.

[10] Wang Qiu-Dong. 1990. The global solution of the N-body problem. *Celestial Mechanics and Dynamical Astronomy* 50, 1 (01 Mar 1990), 73–88. https://doi.org/10.1007/BF00048987

[11] Bjarne Stroustrup. 1995. *The C++ programming language.* Second edition, reprinted with corrections August, 1995. Reading, Mass. : Addison-Wesley, 1995. ©1991. https://search.library.wisc.edu/catalog/999786409402121 Includes bibliographical references (pages 11-12) and index.

[12] Prabhat Totoo and Hans-Wolfgang Loidl. 2014. Parallel Haskell implementations of the N-body problem. *Concurrency and Computation: Practice and Experience* 26 (03 2014). https://doi.org/10.1002/cpe.3087

APPENDIX A

Algorithm 2 MPI Allgather Algorithm

$size \leftarrow MPI\_Size$
$rank \leftarrow MPI\_Rank$
$start \leftarrow rank \cdot \frac{N}{size}$
MPI_Broadcast initial body masses/positions/velocities

for $t \leftarrow 0$ to t_{max} **do**
    for $i \leftarrow start$ to $start + \frac{N}{size} - 1$ **do** ▷ Compute forces acting on $\frac{N}{size}$ bodies
        for $j \leftarrow 0$ to $N - 1$ **do**
            if $i \neq j$ **then**
                Calculate force of body j on body i (Eq: 5)
                Update net force on body i
            end if
        end for
    end for

    for $i \leftarrow 0$ to $N - 1$ **do** ▷ Update positions and velocities
        Update position and velocity of body i (Eq: 2, 3)
    end for
    MPI_Allgather updated body data ▷ All nodes collect updated data from other nodes
end for

Algorithm 3 MPI Allgather Algorithm V2

```
size ← MPI_Size
rank ← MPI_Rank
start ← rank · N/size
MPI_Broadcast initial body masses/positions
MPI_Broadcast initial body velocities

for t ← 0 to t_max do
    for i ← start to start + N/size − 1 do          ▷ Compute forces acting on N/size bodies
        for j ← 0 to N − 1 do
            if i ≠ j then
                Calculate force of body j on body i (Eq: 5)
                Update net force on body i
            end if
        end for
    end for

    for i ← 0 to N − 1 do                           ▷ Update positions and velocities
        Update position and velocity of body i (Eq: 2, 3)
    end for
    MPI_Allgather updated body positions ▷ All nodes collect updated data from other nodes
    MPI_Allgather updated body velocities
end for
```

Algorithm 4 OpenMP No Collapse Algorithm

```
for t ← 0 to t_max do
    #pragma omp parallel for
    for i ← 0 to N − 1 do                           ▷ Compute forces acting on N bodies
        for j ← 0 to N − 1 do
            if i ≠ j then
                Calculate force of body j on body i (Eq: 5)
                Update net force on body i
            end if
        end for
    end for

    #pragma omp parallel for
    for i ← 0 to N − 1 do                           ▷ Update positions and velocities
        Update position and velocity of body i (Eq: 2, 3)
    end for
end for
```

Algorithm 5 OpenMP Collapse Algorithm

for $t \leftarrow 0$ to t_{max} **do**
    #pragma omp parallel for collapse(2)
    for $i \leftarrow 0$ to $N - 1$ **do** ▷ Compute forces acting on N bodies
        for $j \leftarrow 0$ to $N - 1$ **do**
            if $i \neq j$ **then**
                Calculate force of body j on body i (Eq: 5)
                Update net force on body i
            end if
        end for
    end for

    #pragma omp parallel for collapse(2)
    for $i \leftarrow 0$ to $N - 1$ **do**
        for $d \in \{x, y, z\}$ **do**
            Update position and velocity of body i in d-direction (Eq: 2, 3)
        end for
    end for
end for

Algorithm 6 Hybrid Algorithm

$size \leftarrow MPI\_Size$
$rank \leftarrow MPI\_Rank$
$start \leftarrow rank \cdot \frac{N}{size}$
MPI_Broadcast initial body masses/positions/velocities

for $t \leftarrow 0$ to t_{max} **do**
    #pragma omp parallel for
    for $i \leftarrow start$ to $start + \frac{N}{size} - 1$ **do** ▷ Compute forces acting on $\frac{N}{size}$ bodies
        for $j \leftarrow 0$ to $N - 1$ **do**
            if $i \neq j$ **then**
                Calculate force of body j on body i (Eq: 5)
                Update net force on body i
            end if
        end for
    end for

    #pragma omp parallel for
    for $i \leftarrow 0$ to $N - 1$ **do** ▷ Update positions and velocities
        Update position and velocity of body i (Eq: 2, 3)
    end for
    MPI_Allgather updated body data ▷ All nodes collect updated data from other nodes
end for

APPENDIX B

N	Runtime (μs)
1200	21240494
2400	84885562
3600	188752161
4800	340175425
6000	528873108
7200	752018919

Table 1. Sequential Algorithm Runtime

N	Contiguous (μs)	Struct (μs)
1200	2440228	2549881
2400	13322600	13887409
3600	24204972	24962547
4800	35087345	37235785
6000	57907119	61453242
7200	80726893	84862628

Table 2. Data for Section 3.2.1

N	Single (μs)	Multiple (μs)
1200	2440228	3302677
2400	13322600	17765107
3600	24204972	33002424
4800	35087345	44370314
6000	57907119	72815654
7200	80726893	97778931

Table 3. Data for Section 3.2.2

CPUs	Collapse (μs)	No Collapse (μs)
1	646206730	652668797
2	323103365	361875768
3	215402243	241250512
4	161551682	180937884
5	129241346	144750307
6	107701121	120625256
7	92315247	103393076
8	80775841	90468942
9	71800747	80416837
10	64620673	72375153
11	58746066	65795594
12	53850560	60312628

Table 4. Data for Section 3.3.1

Workers	N=1200 (μs)	N=2400 (μs)	N=3600 (μs)	N=4800 (μs)	N=6000 (μs)	N=7200 (μs)
1	11485136	45997397	103494143	183963659	287911004	415485062
2	5784784	23103893	51983760	92153311	144022041	206949954
3	4529687	18031803	40571556	71864062	112265181	162279652
4	3274590	12959712	29159353	51574814	80508322	117609350
6	2019493	7887622	17747150	31285565	48751462	72939048
8	1537796	5997811	13495075	23508084	36635604	52640798
10	1298550	5049035	11360328	19788102	30707351	44154252
12	1059305	4100258	9225581	16068120	24779097	35667706

Table 5. Data for Section 3.4

N	Sequential (μs)	Hybrid Parallel (μs)
1200	21240494	1059305
2400	84885562	4100258
3600	188752161	9225581
4800	340175425	16068120
6000	528873108	24779097
7200	752018919	35667706

Table 6. Data for Section 3.5