CPS3236 Assignment:
n-body Simulator

Full name: Miguel Dingli
I.D: 49997M
Course: B.Sc. (Hons) in Computing Science

Table of Contents

1 – Overview 2
1.1 – The Three Source Files 2
1.2 – General Changes 2
1.2.1 – Arguments to the Programs 2
1.2.2 – Timing 2
1.2.3 – File Input 3
1.2.4 – Output when Output Files Disabled 3
2 – Parallel Decomposition of the Problem 3
2.1 – MPI 3
2.1.1 – Main Iterations Loop 4
2.1.2 – MPI_PARTICLE and MPI_REDUCEDP 5
2.2 – OpenMP 6
2.3 – Hybrid (MPI and OpenMP) 7
3 – Design of Performance Measurement Experiments 7
4 – Plots and Analysis of Results 7
4.1 – Distributed Memory 7
4.2 – Naïve Shared Memory 8
4.3 – Hybrid 9
5 – Shortcomings of the Solutions 10

1 – Overview

As an introduction, a general view of the solutions and the general changes applied to the provided `nbody.c` source code file will be discussed.

1.1 – The Three Source Files

To start with, note that the solution was split into three separate source code files as follows:

- **`1_nbody_mpi.cpp`**: a parallel implementation using MPI only, for distributed memory.
- **`2_nbody_omp.cpp`**: a parallel implementation using OpenMP only, for naïve shared memory.
- **`3_nbody_hybrid.cpp`**: a parallel implementation using both MPI and OpenMP.

1.2 – General Changes

The general changes applied to the main function of the `nbody.c` source file will now be discussed.

1.2.1 – Arguments to the Programs

For easier control over the program, the solutions all accept six arguments, in the order listed below:

- **Output folder**: the folder in which the `nbody_x.txt` output files will be placed (e.g. `Output`)
- **Input file**: the file containing the masses and initial positions (e.g. `NBodyInput/input_64.txt`)
- **Max iterations**: the number of iterations to be performed (e.g. 1000)
- **Δt**: the constant delta (e.g. 0.01)
- **G**: the gravitational constant (e.g. 20.0).
- **Output files**: whether to output the `nbody_x.txt` files ("yes") or not ("no").

```
const int maxIteration = std::stoi(argv[3]);            // example: 1000
const float deltaT = std::stof(argv[4]);                // example: 0.01
const float gTerm = std::stof(argv[5]);                 // example: 20.0
const bool outputFiles = strcmp(argv[6], "yes") == 0;   // example: yes
...
std::ifstream fileInput(argv[2]);
...
fileOutput << argv[1] << "/nbody_" << iteration << ".txt";
```

Listing 1 – Program Arguments

1.2.2 – Timing

Additionally, in all three of the solutions, timing was added to the main method. In the case of `1_nbody_mpi` and `3_nbody_hybrid`, `MPI_Wtime()` was used, whereas `omp_get_wtime()` was used in `2_nbody_omp`. Additionally, in the case of `1_nbody_mpi` and `3_nbody_hybrid`, a reduction is performed to find the overall maximum execution time, which is then printed out by the rank-0 MPI task.

```
const double start_time = MPI_Wtime();
...
double end_time = MPI_Wtime() - start_time;
double max_end_time;
MPI_Reduce(&end_time, &max_end_time, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);
if (rank == 0) {
    printf("Execution time: %.2f ms\n", max_end_time * 1e3);
}
```

Listing 2 – Timing in MPI and Hybrid Solutions

```
const double start_time = omp_get_wtime();
...
printf("Execution time: %.2f ms\n", (omp_get_wtime() - start_time) * 1e3);
```

Listing 3 – Timing in OpenMP Solution

1.2.3 – File Input

To read the input from the specified `input_x.txt` file, the solutions start with a blank vector of particles and fill it up with the values read from the input file. In the case of 1_nbody_mpi and 3_nbody_hybrid, this process is only done by the thread with rank 0.

```
// Setting particle count and bodies vector
float mass, posX, posY;
char comma1, comma2;
while (fileInput >> mass >> comma1 >> posX >> comma2 >> posY)
{
    Particle toAdd = Particle();
    toAdd.Mass = mass;
    toAdd.Position = Vector2(posX, posY);
    bodies.push_back(toAdd);
}
fileInput.close();
```

1.2.4 – Output when Output Files Disabled

Finally, note that when the output files are disabled, i.e. by setting the last program argument to "no", the iteration number is printed in each iteration instead of the "Writing to file: …" output and the actual output files. In any case, the timing is always printed at the end of the execution.

2 – Parallel Decomposition of the Problem

In this section, the techniques used for parallel decomposition of the problem in each of the three separate source code files will be outlined, starting with the MPI-only solution.

2.1 – MPI

In the MPI solution, the parallel decomposition of the problem was done by joining the available MPI tasks in a unidirectional ring arrangement where the nodes pass around subsets of the vector of particles in order to compute the forces between all of the particles.

For this to work, each node was equipped with three vectors of particles; **subBodies**, **recvBuffer**, and **sendBuffer**. The idea is that, near the start of the program, the vector of particles (**bodies**) formed by parsing the input file is scattered amongst the MPI tasks, such that each task will receive a subset (**subBodies**) of the particles, which they will keep and update throughout the program execution.

```
// Vectors of particles
std::vector<Particle> bodies;       // main bodies
std::vector<Particle> subBodies;    // main bodies subset
std::vector<Particle> recvBuffer;   // buffer for subset to receive
std::vector<Particle> sendBuffer;   // buffer for subset to send
```

Listing 4 – Vectors of Particles

For the particles to be distributed as evenly as possible, the rank-0 MPI task broadcasts the particle count to all the MPI tasks, so that these now know the total number of bodies. Using this value, each task now computes two arrays **`mainSize`** and **`displs`** to determine the number of bodies that each node will be assigned. These arrays will be used in an `MPI_Scatterv` scatter in which the particles are distributed among the MPI tasks as evenly as possible. The **mainSize** array is also used to determine how many particles an MPI task will receive and how many it will send to/from its neighbours in the ring arrangement.

Kindly refer to the next page for a code snippet.

```
MPI_Bcast(&particleCount, 1, MPI_INT, 0, MPI_COMM_WORLD);

// Setting mainSize and displs arrays
int mainSize[numtasks]; // sizes of main subsets held by each node (a.k.a send counts)
int displs[numtasks];   // displacements
const int baseAmount = particleCount / numtasks; // minimum particles per MPI task
const int remainder = particleCount % numtasks; // remaining particles to be distributed
int displacement = 0;
for (int i = 0; i < numtasks; i++) {
    mainSize[i] = baseAmount + (i < remainder ? 1 : 0);
    displs[i] = displacement;
    displacement += mainSize[i];
}

// Distribute the subsets of bodies
subBodies.resize(mainSize[rank]);
MPI_Scatterv(&bodies[0], mainSize, displs, MPI_PARTICLE, &subBodies[0], mainSize[rank],
MPI_PARTICLE, 0, MPI_COMM_WORLD);
```

Listing 5 – particleCount, mainSize, displs, and Scatterv

2.1.1 – Main Iterations Loop

In the main iterations loop, the tasks will pass their subset around the ring so that each task eventually gets to see all existing subsets. Whilst a task is receiving and sending a subset (using **sendBuffer** and **recvBuffer**), it computes the forces (**ComputeForces(…)**) between its permanent subset and the latest subset it had received. Note that, for this, the **ComputeForces(…)** function was extended to take an additional vector of particles.

For better performance, non-blocking sends and receives were used. At the start of a send-receive cycle, the latest received particles are moved to **sendBuffer** where, whilst the task is sending and receiving, the task will use the particles being sent (i.e. the ones in **sendBuffer**) to compute the forces. After an MPI task computes all forces between its subset and the subset in **sendBuffer**, `MPI_Wait` is called to make sure that the sends and receives have all finished before moving on to the next send-receive cycle.

Once all MPI tasks have seen all of the subsets in circulation, each task moves the particles (**MoveBodies(…)**) of its permanent subset so that in the next iteration, the subset that it will circulate will be an updated one. Note that for all of n tasks to see all n subsets, $n - 1$ send/receive cycles are enough given that a task already has its own subset in possession at the start of an iteration. This avoids an extra send/receive cycle.

```
// Main iterations loop
for (int iteration = 0; iteration < maxIteration; ++iteration)
{
    // Initial subset and its index
    recvBuffer.resize(mainSize[rank]);  // resize to fit subBodies
    recvBuffer = subBodies;             // first subset held will be subBodies
    currIndex = rank;                   // index of subset currently held

    // Send-receive-compute-persist loop
    int i;
    for (i = 0; i < numtasks - 1; i++)
    {
        const int tempIndex = (currIndex + 1) % numtasks; // compute next subset's index
        sendBuffer.resize(mainSize[currIndex]); // must be able to hold current subset
        sendBuffer = recvBuffer;                // current subset copied to sendBuffer
        recvBuffer.resize(mainSize[tempIndex]); // must be able to hold next subset

        // Non-blocking exchange of bodies subsets
        MPI_Isend(&sendBuffer[0], mainSize[currIndex], MPI_REDUCEDP, dst, 0,
             MPI_COMM_WORLD, &sendReq);
        MPI_Irecv(&recvBuffer[0], mainSize[tempIndex], MPI_REDUCEDP, src, 0,
             MPI_COMM_WORLD, &recvReq);
        // Compute and persist
        ComputeForces(subBodies, gTerm, deltaT, sendBuffer);
        if (rank == 0 && iteration != 0 && outputFiles) {
            PersistPositions(fileOutput.str(), sendBuffer, i != 0);
        }

        // Finalize
        currIndex = tempIndex;
        MPI_Wait(&sendReq, MPI_STATUS_IGNORE);
        MPI_Wait(&recvReq, MPI_STATUS_IGNORE);
    }

    // Compute-persist (no need to send/receive last subset)
    ComputeForces(subBodies, gTerm, deltaT, recvBuffer);
    if (rank == 0 && iteration != 0 && outputFiles) {
        PersistPositions(fileOutput.str(), recvBuffer, i != 0);
    }

    // Move bodies in main subset
    MoveBodies(subBodies, deltaT);

    // Set file output stream for next iteration (or just print iteration)
    if (rank == 0) {
        if (outputFiles) {
            fileOutput.str(std::string());
            fileOutput << argv[1] << "/nbody_" << iteration << ".txt";
        } else {
            printf("%d\n", iteration);
        }
    }
}
```

Listing 6 – Main Iterations Loop and Inner Send-Receive Loop

The persistence of positions (**PersistPositions(…)**) occurs from the second iteration onwards, where the rank-0 task will append each newly-updated subset it receives to the output file for the previous iteration. This starts from the second iteration since it is only by then that the updated subsets will start circulating. After the main loop finishes, the final persist for the last iteration occurs by performing an `MPI_Gatherv` operation.

```
// Gather bodies for final persist
if (outputFiles) {
    MPI_Gatherv(&subBodies[0], mainSize[rank], MPI_PARTICLE, &bodies[0], mainSize,
       displs, MPI_PARTICLE, 0, MPI_COMM_WORLD);
    if (rank == 0) {
        PersistPositions(fileOutput.str(), bodies, false);
    }
}
```

Listing 7 – Final Persist

Note that for the rank-0 MPI task to receive the subsets in sorted order, especially so that the particles are output to the output files in sorted order, the subsets are passed around backwards rather than forwards. Also note that since the positions are persisted in subsets rather than the entire vector at once, the **Persist-Positions(…)** function was extended to include a boolean which indicates whether to write or append.

2.1.2 – MPI_PARTICLE and MPI_REDUCEDP

For sending and receiving particles, a custom data type **MPI_PARTICLE** was defined. However, one can appreciate that in the subsets that are passed around, the Velocity vector is not necessary in computing the forces between the particles in these subsets and the permanent subsets held by the MPI tasks since only the velocity vectors in the permanent subsets are updated. For this reason, another custom data type **MPI_REDUCEDP** was defined to be able to send less data by leaving a gap where the Velocity vector would otherwise be.

```
// MPI_PARTICLE structured type and commit
MPI_Datatype oldTypes1[1] = {MPI_FLOAT};
int blockCounts1[1] = {5};
MPI_Aint offsets1[1] = { offsetof(Particle, Position) };
MPI_Type_create_struct(1, blockCounts1, offsets1, oldTypes1, &MPI_PARTICLE);
MPI_Type_commit(&MPI_PARTICLE);

// MPI_REDUCEDP structured type and commit
MPI_Datatype oldTypes2[2] = {MPI_FLOAT, MPI_FLOAT};
int blockCounts2[2] = {2, 1};
MPI_Aint offsets2[2] = { offsetof(Particle, Position), offsetof(Particle, Mass) };
MPI_Type_create_struct(2, blockCounts2, offsets2, oldTypes2, &MPI_REDUCEDP);
MPI_Type_commit(&MPI_REDUCEDP);
```

Listing 8 – MPI_PARTICLE and MPI_REDUCEDP

2.2 – OpenMP

The OpenMP solution is much more straightforward than the MPI solution and consists of some changes to the ComputeForces(…) function and a change to the MoveBodies(…) function. In both cases, a simple "#pragma omp parallel for default(shared)" was added to the loops (the outer loop in the case of ComputeForces(…)) so that the loop iterations are distributed between the available threads.

In the ComputeForces(…) function, further changes were applied. The force vector used to store the accumulated force for a particular particle was converted into two floats representing the components of the vector. This was done so that a reduction can be applied based on both of these floats. The accumulated force vector's components are then used to compute the acceleration normally.

In both functions, no race conditions should be present since any iteration is only changing values which are not read elsewhere in the loop. For example, in the ComputeForces(…) loop, only p1.Velocity is changed, and it is not accessed anywhere else throughout the function.

```
#pragma omp parallel for default(shared)
for (size_t j = 0; j < p_bodies.size(); ++j)
{
    ...
    float f1 = 0.0f, f2 = 0.0f;

    #pragma omp parallel for reduction(+:f1,f2) default(shared)
    for (size_t k = 0; k < p_bodies.size(); ++k)
    {
        ...
        // Accumulate force
        Vector2 temp = direction / (distance * distance * distance) * p2.Mass;
        f1 += temp.Element[0];
        f2 += temp.Element[1];
    }

    // Compute acceleration for body
    Vector2 acceleration = Vector2(f1, f2) * p_gravitationalTerm;
    ...
}
```

Listing 9 – Summarised ComputeForces Function

```
#pragma omp parallel for
for (size_t j = 0; j < p_bodies.size(); ++j)
{
    p_bodies[j].Position += p_bodies[j].Velocity * p_deltaT;
}
```

Listing 10 – MoveBodies Function

2.3 – Hybrid (MPI and OpenMP)

The hybrid solution consisting of both MPI and OpenMP is exactly the features of the previous two solutions joined together and so it will not be discussed. The general idea is that the subsets of particles will be passed between nodes, each of which will use its multiple cores in ComputeForces(…) and MoveBodies(…) to perform the force computation between its main subset and the received subset and to move the particles much faster.

3 – Design of Performance Measurement Experiments

For the performance measurement experiments, a *job.sh* script and a *submit.sh* script were defined for each type of configuration (MPI, OpenMP, and Hybrid) to make running different types of executions for a particular configuration much simpler. Kindly refer to the README to locate these scripts.

The *job.sh* for a particular configuration is the actual job that will be submitted using $qsub$, and contains all fixed aspects, such as job name, modules added, and the actual running of the program. The *submit.sh*, on the other hand, is in charge of setting variable aspects such as the number of cores and which input file will be used (i.e. number of particles). These are then passed on to the *job.sh* script whilst submitting the job. The *submit.sh* script also creates all required directories so that output files are better-organized.

After an execution for a particular configuration, the execution time is found in an output file in the working directory. The execution time for all configurations were extracted and included in an excel sheet which one can locate by referring to the README.

The following are the different configurations used in the performance measurement experiments for each of the four input files. In each case, four readings were taken.

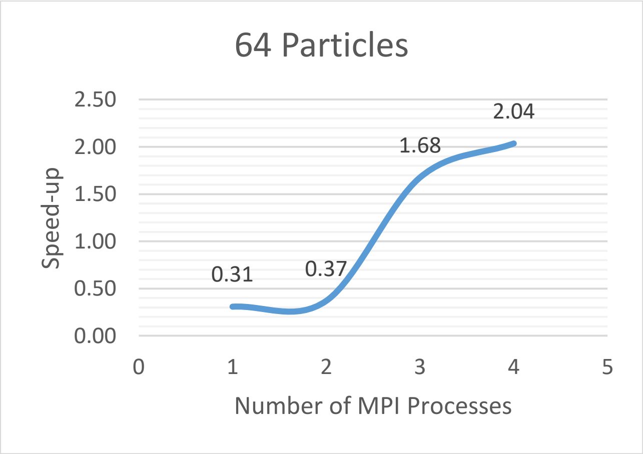
- Distributed memory: **1**, **2**, **3**, and **4** nodes.
- Naïve shared memory: **1**, **2**, **4**, **6**, and **12** cores.
- Hybrid: all combinations of **1**, **2**, **3**, and **4** nodes and **2**, **4**, **6**, and **12** cores.

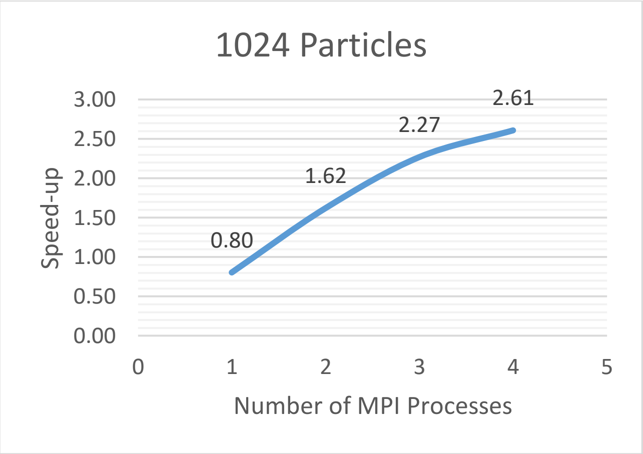
4 – Plots and Analysis of Results

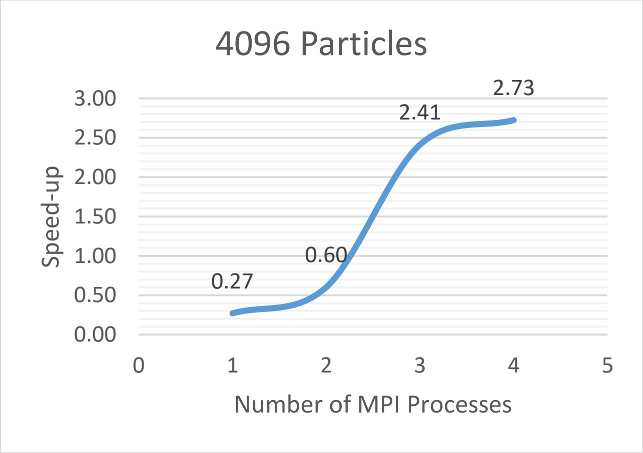
In each plot, the base execution time was assumed to be the execution time for one core using the Naïve shared memory source code, which produces the least extra overhead. The speed-up in each case was then calculated by dividing this execution time by the average of the four readings for a particular configuration.

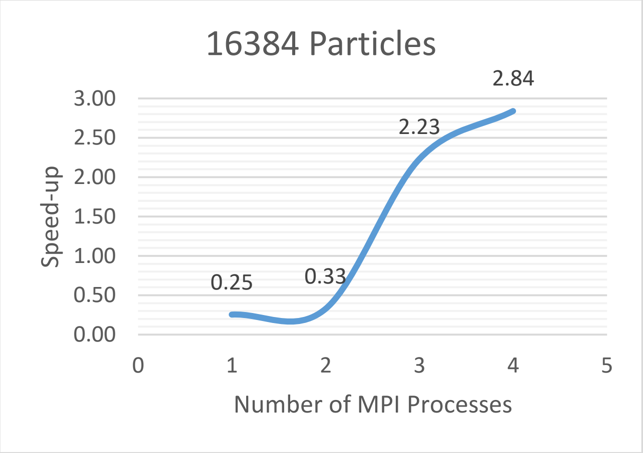
4.1 – Distributed Memory

The following section presents the speed-up results for the distributed memory configuration. From each plot, it is clear that the distributed memory solution performs worse when using only one or two MPI processes. However, a speed-up of 2 or more was observed when the number of MPI processes was increased to 4.



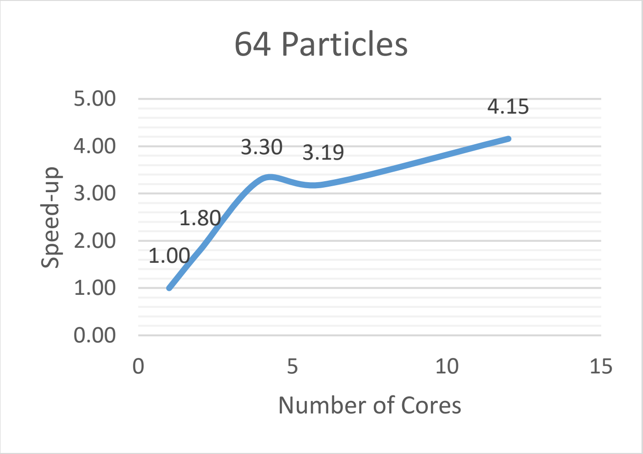


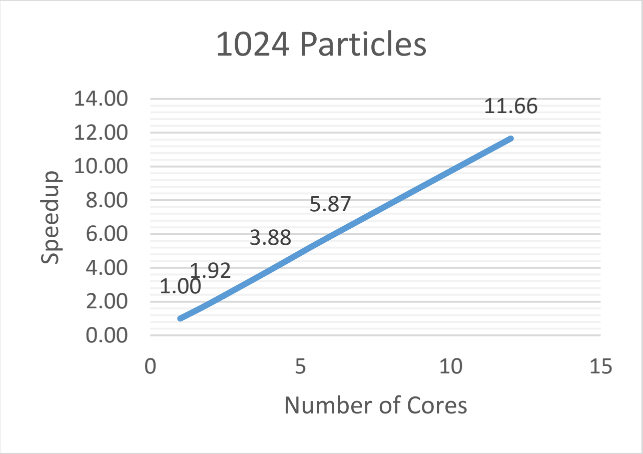


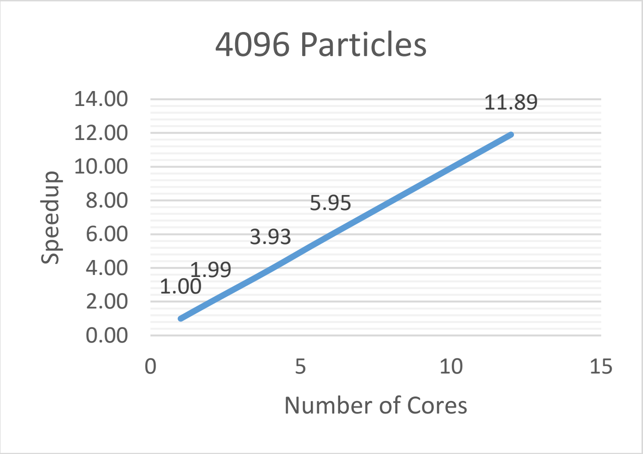


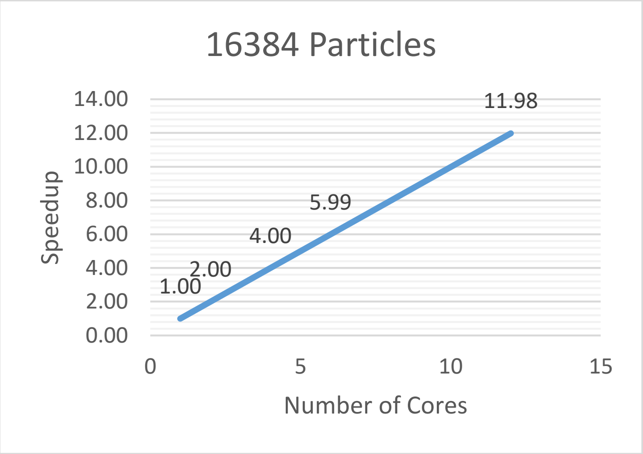
4.2 – Naïve Shared Memory

The following section presents the speed-up results for the naïve shared memory configuration. Surprisingly, the OpenMP solution gives near linear speed-up in the 1024, 4096, and 16384 particles executions. In the case of 64 particles, speed-up was observed, but this was most probably hindered by the fact that a small problem size makes the overhead more significant.







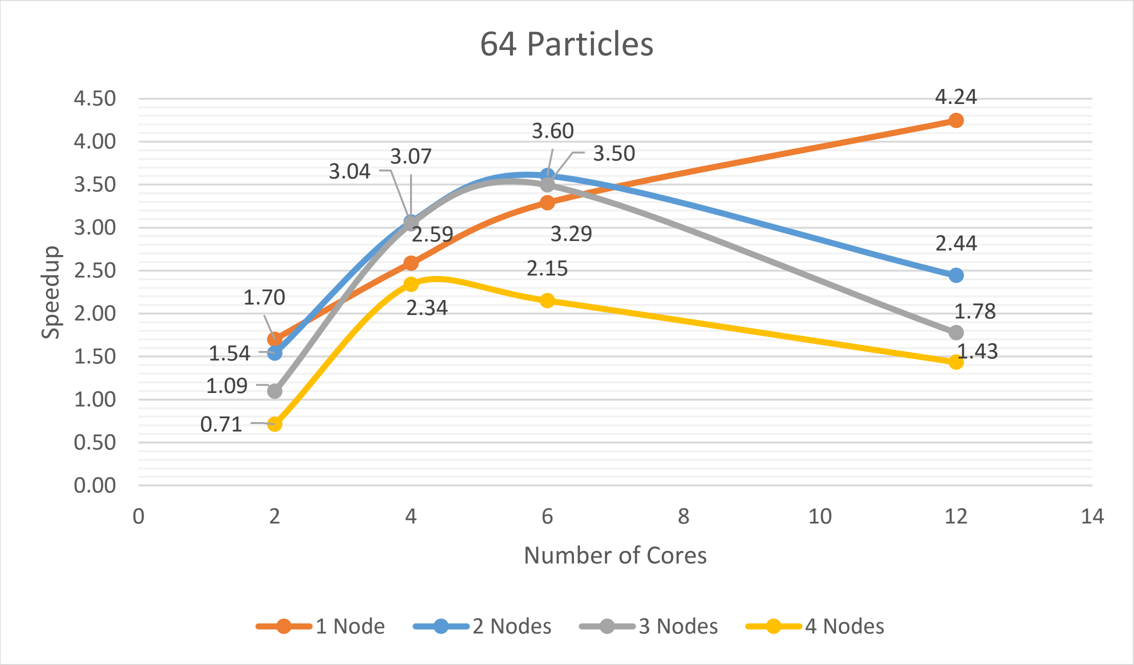


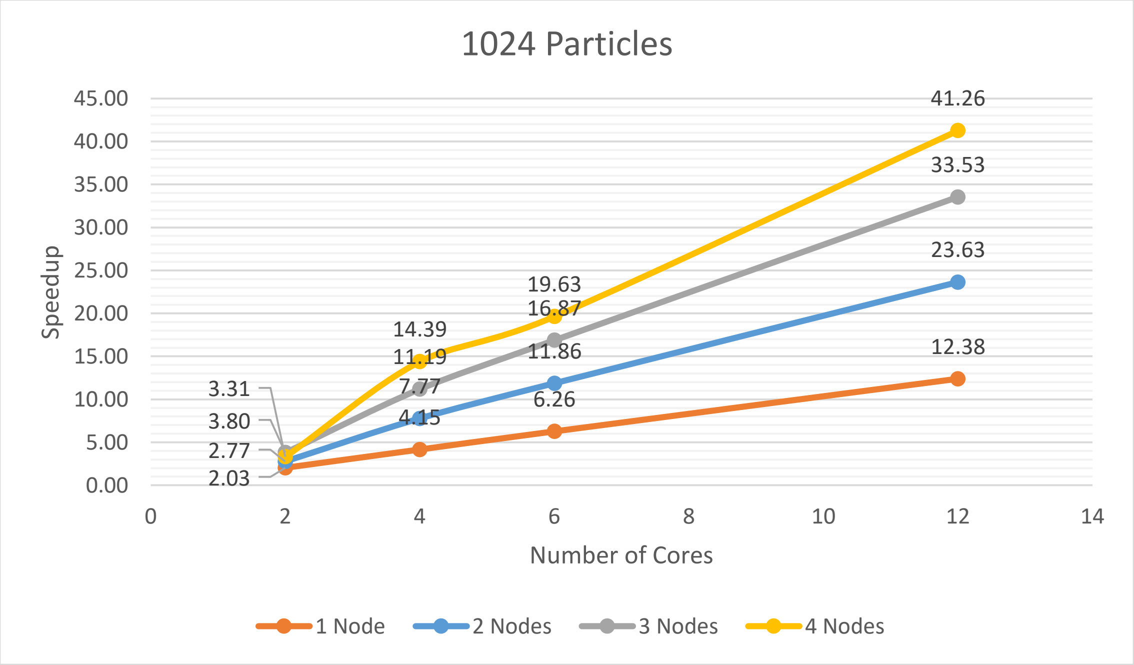
Kindly to the Hybrid plots overleaf.

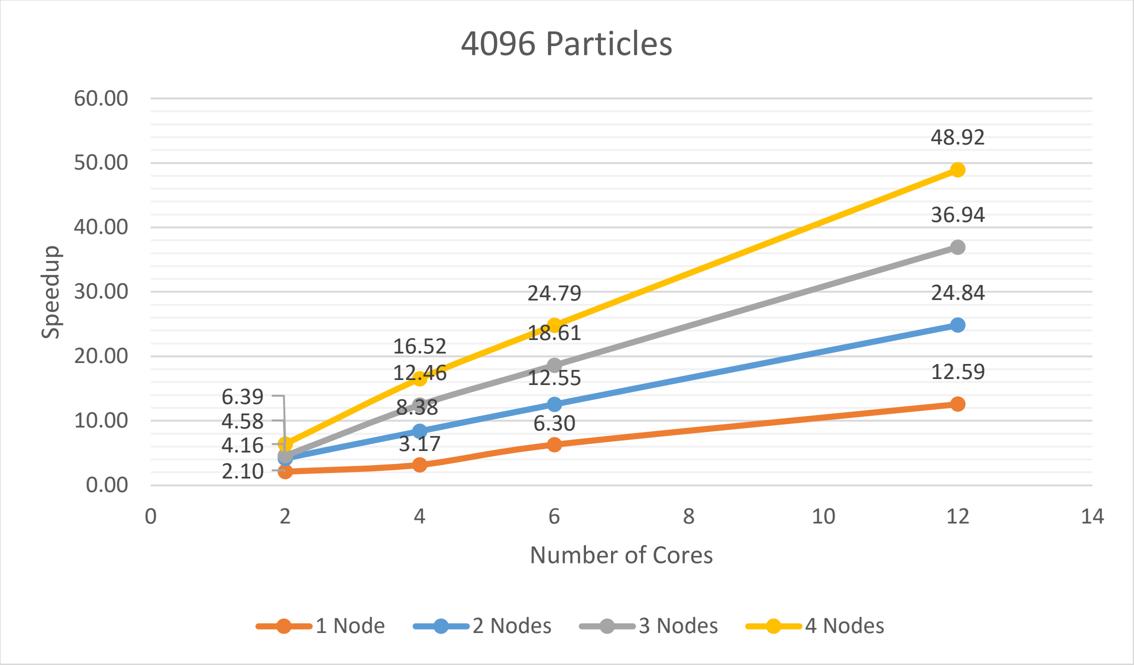
4.3 – Hybrid

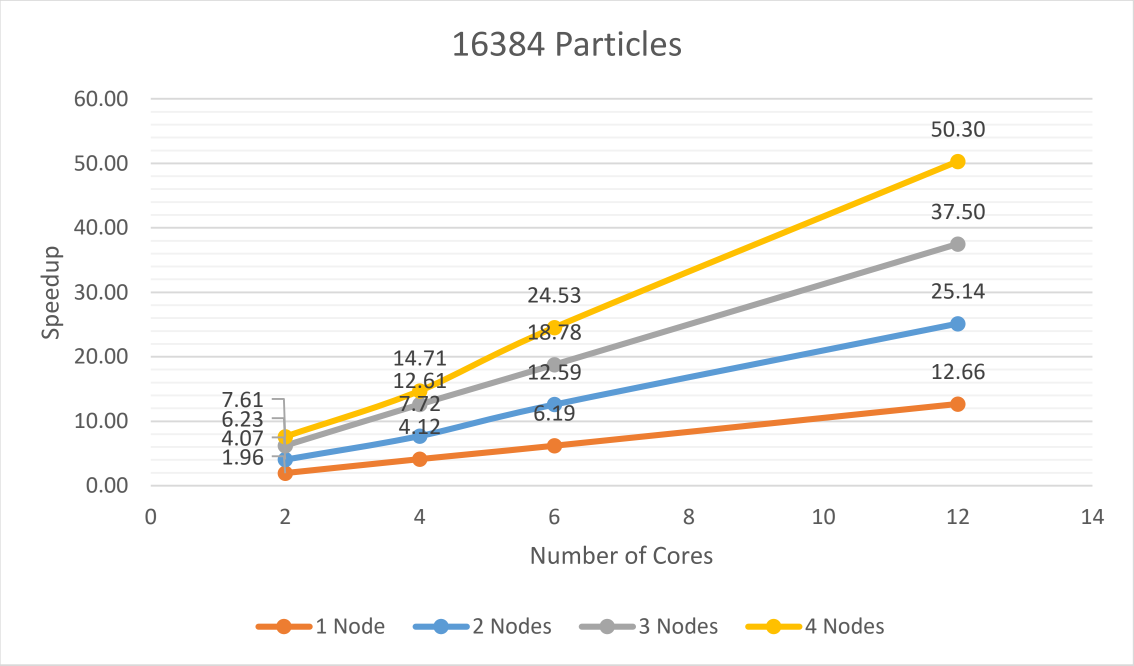
The following section presents the speed-up results for the hybrid configuration. All the plots for a particular number of input particles were grouped together, resulting in four plots. Similar to the case of the naïve shared memory solution, the results for 64 particles, although there is clearly speed-up, this is hindered by overhead.

In the remaining three solutions, the speed-up obtained is more than expected. Considering the four curves in each plot, each one corresponding to a different number of MPI processes, it is clear that both the number of cores and the number of MPI processes (i.e. number of nodes) contribute significantly to the speed-up.









5 – Shortcomings of the Solutions

The following is a list of shortcomings that were identified:

1. None of the three solutions take advantage of the symmetry of the force between two particles. Instead, they calculate all of the forces. Applying this symmetry would have greatly improved the performance of the solutions, but may be less straightforward then the presented solutions.
2. In the MPI solution, even less data could have been sent by keeping a masses array, since these do not change throughout the execution. The *currIndex* index could have also been used for this purpose.
3. Between the solutions, there is a small difference in the results that grows steadily with the number of iterations. Since the difference is not random and is consistent between runs of the same solution, this phenomenon was assumed to be due to floating-point precision, which is affected when operations are performed in a different order, rather than because of the solution itself.