Vrije University of Amsterdam

A Parallel Version for N-Body Problem

MPI Assignment Report

Ehsan Sharifi Esfahani
Student Number: 2603181

August 2019

1.Introduction

N-body problem is the simulation problem over time for individual bodies interacting with each other gravitationally. This prediction can assist to comprehend better the motion of Sun, Earth and other starts. Along with astrophysics applications, N-body problem has a wide-range of applicability in other high-performance computing domains such as, molecular dynamics, fluid dynamics, boundary value problems, numerical complex analysis and computer graphics. To elaborate this with one example, in plasma physics, the bodies are ions or, and the governing law is electrostatic [1].

In this assignment, a distributed algorithm for solving the N-body problem was implemented using MPI and OpenMP libraries. Then, a performance analysis is conducted to benchmark the given solution.

2.Experimental Setup

In this section, we explain what is the experimental setup and how the experiments are carried out. We start by describing the underling hardware we used. We performed the experiments on VU cluster on Distributed ASCI Supercomputer 4 (DAS-4), a distributed supercomputer with 6 clusters for the academic purpose in the Netherlands. A standard node in DAS-4 is equipped with dual-quad-core compute nodes (primarily SuperMicro 2U-twins with Intel E5620 CPUs) CPU and either 24GB or 48 GB memory with an operating system based on CentOS Linux 7. The machines in each cluster communicate through 1 Gbit/s Ethernet interfaces.

Our program is written in C and compiled by GCC 6.03.0. We also exploit the OpenMP library to improve performance using multi-threading technology in each node. We used 1-16 nodes to run our program in parallel. To make measurements accurate, we run every problem size, for each number of node, for three times and present numbers corresponding to the average value.

3.Parallel algorithm

In this section, we describe how we design our algorithm and then explain the applied optimizations. First off, we implement a simple version by distributing the number of bodies between the different nodes. Every node works on a portion of bodies based on its rank number. For example, if we have 13 bodies and 3 nodes, then the node number 0, 1 and 2 work on the bodies number of 0-4, 5-9, 10-12, respectively. This means that the first node works on 5 bodies, the second one on 5 bodies and the last node works on 2 bodies. As a result, every node can find out which part of the world is dedicated on which node to work. To implement this, we use a ceil() function as below:

```
position=ceil(world->bodyCt/(double)size)
```

This means that we assign an equal part of each body to every node, except the last node which it may get a smaller part. Moreover, every node need a global view of the world to be able to compute the forces. Each body should send and receive the other parts of the world before the force computations. To do so, we use an asynchronous broadcast MPI routine, MPI_Ibcast().

At the end of code, the MPI_Waitall() routine is called to become sure that we receive all the different part of worlds. There is a *if* statement in this code since the last part of the world might be smaller than the other parts. These operations is encapsulated in a function named Send_Receive_World(). Moreover, to be able to send a body to each node, we must define a MPI type and commit it. The function Build_Derived_Data() in our code, implements this ability.

```
1.  static void Build_Derived_Data(struct world *world) {
2.    int lengths[8]={2,2,1,1,1,1,1,1};
3.    const MPI_Aint dis[8] = {
4.          0,
5.          2 * sizeof(double),
6.          4 * sizeof(double),
7.          5 * sizeof(double),
8.          6 * sizeof(double),
9.          7 * sizeof(double),
10.         8 * sizeof(double),
11.         9 * sizeof(double)};
12.
13.
14.   MPI_Datatype types[8] = { MPI_DOUBLE, MPI_DOUBLE, MPI_DOUBLE, MPI_DOUBLE,
15.                             MPI_DOUBLE, MPI_DOUBLE, MPI_DOUBLE, MPI_DOUBLE};
16.   MPI_Type_create_struct(8, lengths, dis, types, &mpi_bodytype);
17.   MPI_Type_commit(&mpi_bodytype);
18. }
```

As shown in listing above, the body structure consists of 8 blocks of data, everyone with different length, which we list them in lengths variable. We use the MPI_Aint to place the displacements of these blocks of data in *dis* array. Clearly, the *types* array indicated the type of every block. Furthermore, we revise every function in the main *while* loop in order to every node only do the computation on the bodies that are assigned to it. This is done by passing the boundaries as a parameter to each function.

4. Applied optimizations

To optimize our naïve code, we used different methods. In this section, we are going to describe them in details.

4.1. Merging the compute_positions() and compute_velocities() functions in one routine

There are four main functions in the main loop, namely clear_forces, compute_forces, compute_velocities and compute_positions. In every function, there is a *for* loop which reduce the performance since managing the loop operations, such as checking the conditions and increasing the iterator in every iteration, in every code can cause a significant overhead. To improve the achievable performance, we combined the two functions related to position computations and velocities computations in one function named compute_velocities_positions (). This can improve performance. Moreover, these main functions have been modified to only produce the data related to their own node. It is because there is no need to compute all the computation for all bodies in each node.

4.2. Using OpenMP

OpenMP is a library to implement the multi-thread capability in shared-memory systems in C, C++ or Fortran. This provide a new opportunity to enhance application performance. We use the OpenMP pragma to parallel the main loop in the function of compute_velocities_positions() and compute_forces. This can create different thread for each iteration of the main loop and we can also use the advantage of paralleling on every node. This is shown in the listing below.

```
1.  #ifdef _OPENMP
2.  #pragma omp parallel for private(b,c) shared (world) reduction(- : tem_yf[:world-
    >bodyCt]) reduction(- :tem_xf[:world->bodyCt])
3.  #endif
4.  for (b = start_myrange; b < end_myrange; ++b) {
5.      for (c = b + 1; c < end_myrange; ++c) {
6.          ....
7.      }
```

5. Performance analysis

Kernel part of each algorithm is the major part of execution time. In this program, the major part is conducted by the *while(steps--)* loop. Hence, we measure the execution time as the difference between the time stamp before and after this loop.

According to Amdahl's law, also known as strong scaling, the theoretical speedup in latency of the execution of a task *at fixed workload* that can be expected of a system can be formulated as below [2]:

$$S_{latency} = \frac{1}{f + \frac{(1-f)}{p}}$$

Where:

- $S_{latency}$ is the theoretical speedup of the execution of the whole task;
- f is serial fraction of code;
- p is the number of processors.

Therefore, Amdahl predicted that speedup will get to a maximum amount ($\frac{1}{f}$) and it gets fixed when the number of processors get infinite. Amdahl`s law is pessimistic because it is assumed that the parallelization is perfect. However, in practice, the situation is worse than predicted by Amdahl's Law due to different reasons such as load unbalancing, scheduling, cost of communications, I/O, too fine-grained job, too coarse-grained jobs. So, this law cannot give us a good prediction in most cases, however, it can give us a reasonable insight of parallelism.

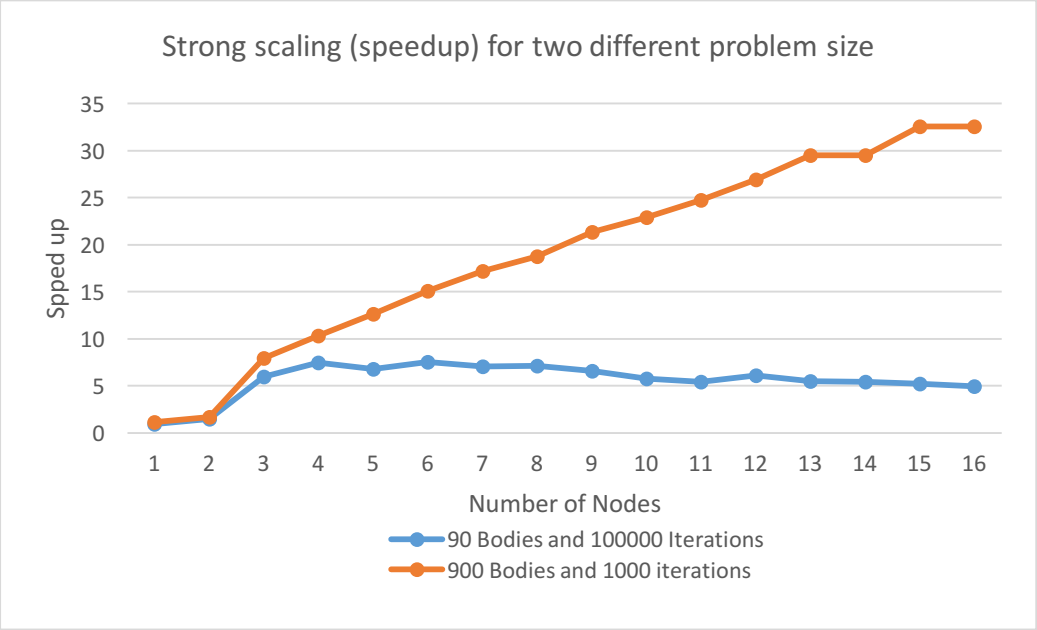

Figure 1. Speedup chart for different problem sizes with multi-threaded version

As it can be seen in figure 1, when the problem size is 90 bodies with 100000 iterations, speed up increases when the number of nodes raises and then it gradually gets constant. Here, the achievable speedup with more than 10 nodes is almost about 5. Moreover, the achievable speedup with one node is slightly more than one because of using the multi-threading technique, OpenMP.

Furthermore, we observe that the speedup for the problem with 900 bodies and 1000 iterations is much more than problem with 90 bodies and 100000 iterations. It can be explained by the fact that the main bottleneck in the problems with large number of iterations is communication, i.e. broadcasting is an expensive operation. However, the bottleneck in the program with large number of bodies is computation, i.e. computing the forces is expensive. So, this algorithm may provide a poor performance for problems with enormous iterations and a lot of machines. As a result, the problem with bigger integration (100000) provides less performance in comparison with the other one with smaller iteration (1000).

Furthermore, the speedup, for the problem with 900 bodies and 1000 iterations, increases when the number of nodes increase. It is because each node gets a smaller part of the world when the number of node goes up.

The fraction of work that can be done in parallel often grows as the problem size increases; Gustafson's law takes this observation into consideration. Gustafson’s law, known as weak scaling, rationalizes that as the size of the problem grows, the serial part will become a smaller percentage of the entire process. It mentions that speedup should be measured by scaling the problem to the number of processors, not by fixing the problem size. This concept is shown in figure 2, the speedup and number of nodes with a fixed problem size per node (100 bodies with 1000 iterations per node) has an almost constant relationship when the number of nodes.

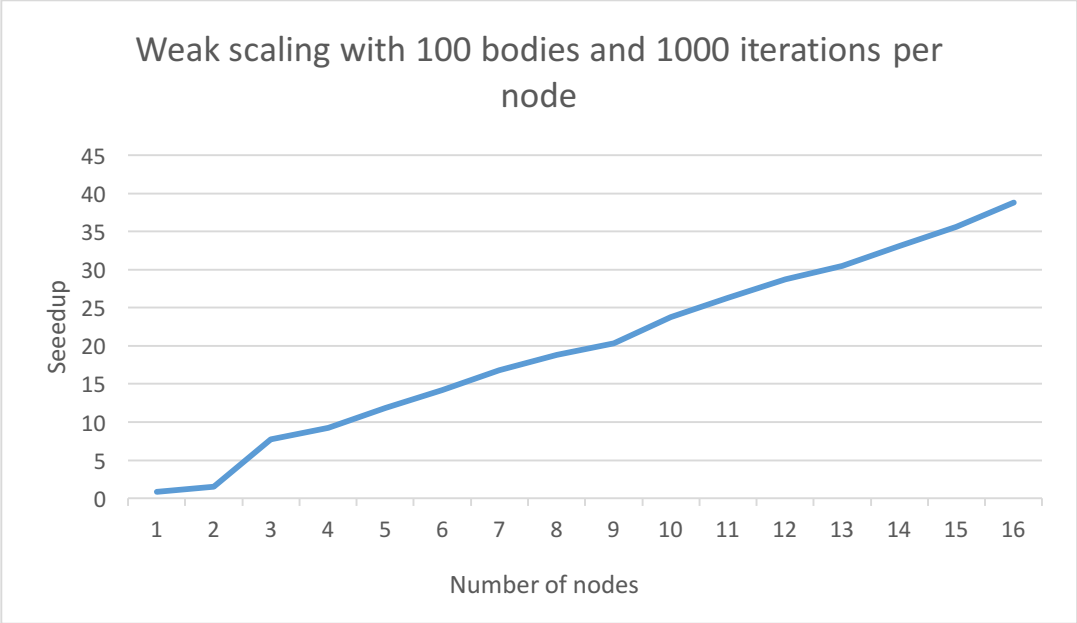

Figure 2. Weak Scaling for the fixed problem size of 100 bodies and 1000 iterations for each node

Efficiency is defined as the ratio of speedup to the number of processors. Efficiency measures the fraction of time for which a processor is usefully utilized. As illustrated in Figure 3, there is a downward trend in the efficiency, efficiency gradually reduces when the number of processors is increases. However, it is possible to keep the efficiency in many parallel systems fixed by increasing both the size of the problem and the number of processing elements simultaneously, a system with this feature usually called scalable parallel system.

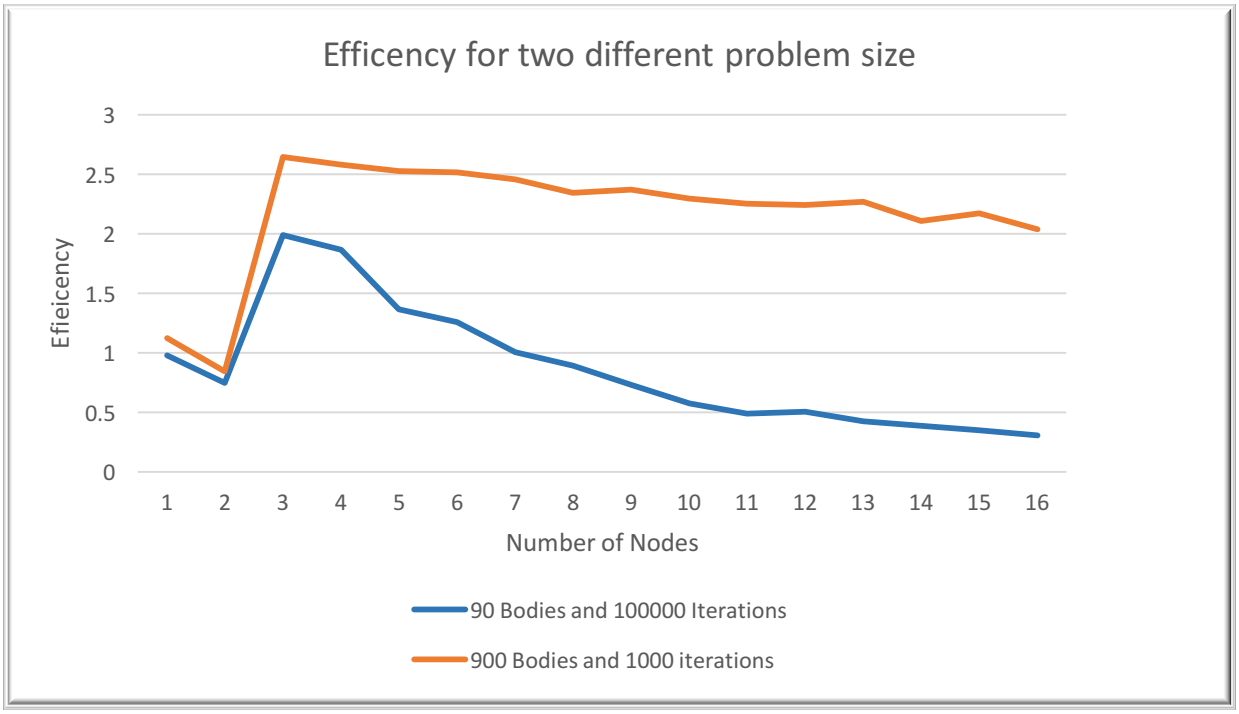

Figure 3. Efficiency of the parallel code with two different problem sizes

References

1- Singh, J. P., Holt, C., Totsuka, T., Gupta, A., & Hennessy, J. (1995). Load balancing and data locality in adaptive hierarchical N-body methods: Barnes-Hut, fast multipole, and radiosity. *Journal of Parallel and Distributed Computing*, *27*(2), 118-141.

2- Wikipedia Encyclopaedia, https://en.wikipedia.org/wiki/Amdahl%27s_law, visited at Nov 2018.