Parallel Implementation of Nearest Neighbour Finding Algorithm

Luke Ye
School of Physics, University of Bristol.
(Dated: February 27, 2024)

The investigation compares different implementations of the nearest neighbour algorithm, significant speed up was achieved using parallelisation through MPI and OpenMP in large scale systems. The cell list algorithm was found perform much faster in large systems compared to a brute force approach.

INTRODUCTION

Near neighbour finding algorithms are important in many applications such as particles simulation. The main problem with the algorithm is that it can be computationally expensive especially at higher sample sizes, therefore there is a need for a parallel implementation in order to fully leverage distributed computing systems such as multi-core processors and compute clusters.

The aim of this investigation is to measure and compare the performance of different implementations at different sample sizes, and calculate the speed up at different number of parallel cores.

METHOD

The near neighbour finding algorithm works by looping through each individual "particle" position, and calculating the distance between it and all other particles, the other particle is counted as a neighbour if this distance is within a defined critical radius R_c, eventually this method returns an array of the total number of neighbours for all particles in the system.

Variations on a brute force method was initially implemented, with two versions of the algorithm was implemented in serial, a original version and a second optimised version- the optimised algorithm should improve the improve the efficiency by first removing the square root operation for each loop run, and more significantly removing the double counting of neighbours which should half the number of operations required. Both versions of the serial algorithm was then parallelised using MPI, the final results were reduced and reported by the root node. Due to the nature of the optimised algorithm, the workload decreases over a single run, to address this, another parallel MPI implementation was developed to distributed the workload more evenly across the nodes. Additinally, an OpenMP implementation were created with static scheduling, dynamic scheduling was also tested however it was much slower potentially due to overheads.

Additionally, variations of the cell list approaches were implemented in serial, MPI and OpenMP, which divides the simulation volume into smaller chunks or "cells", with the width of the cell being equal or greater to R_c. This means that only the atoms in the immediate neighbouring cells needs to be considered in the calculation, which should drastically decrease the number of calculations needed especially at larger scale systems.

All implementations of the algorithms were tested in systems with 120, 10549 and 147023 atoms, the validity of the different implementations were checked by outputting the number of neighbours for each atom, which should be consistent across all implementations. The time taken for each implementation was compared, and additionally for the parallel implementations, the performance was tested across different number of nodes/threads.

RESULTS AND DISCUSSION

Model Outputs

TABLE I. Model output metrics for 3 datasets with varying sample size for a cut off radius of $9A$

N	L	Mean	Median	Max	Min
120	$18A$	32.0	30	63	10
10549	$100A$	28.1	30	43	4
147023	$200A$	52.3	54	71	12

All 3 systems have comparable densities of atoms, with values $0.02A^{-3}$, $0.011A^{-3}$, $0.018A^{-3}$ respectively, all within the same order of magnitude, therefore it is reasonable to expect their mean and median number of neighbours to not differ by a large amount as shown in TABLE 1.

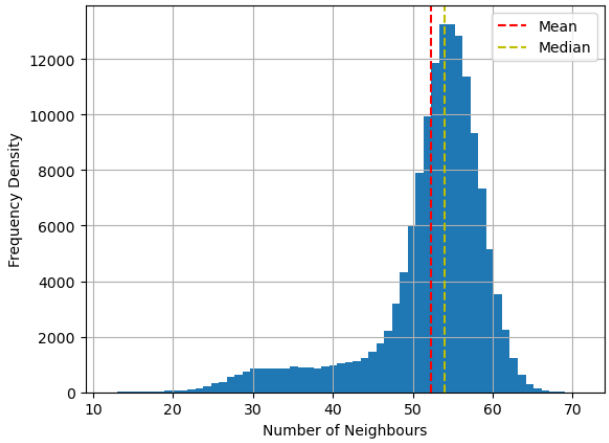

FIG. 1. Histogram of the number of neighbours for the system with 147023 atoms, with mean and median labeled.

FIG.1 shows the distribution of the number of neighbours for the largest system, this system was chosen to represent large scale system as the same distribution was also observed in the 10549 atoms case. The curve is approximately split into two sections, the initial rise between 20 to 45 number of neighbours and the Gaussian peak which follows. It is likely that the first section accounts for the atoms on the edge of the simulation volume, and the Gaussian peak represents the atoms in the center. It is also interesting to note that the mean and median values do not agree due to the asymmetry. In this case, the median could be a better representation of the average number of neighbours in the center of the simulation volume.

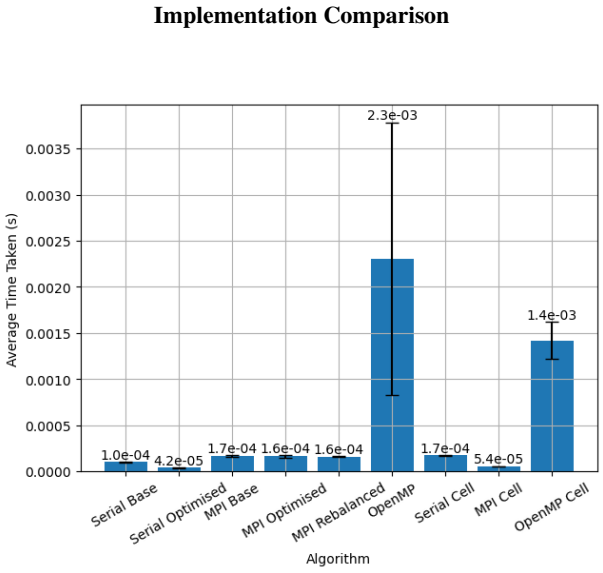

FIG. 2. Compute time for each implementation of the algorithm for 120 atoms

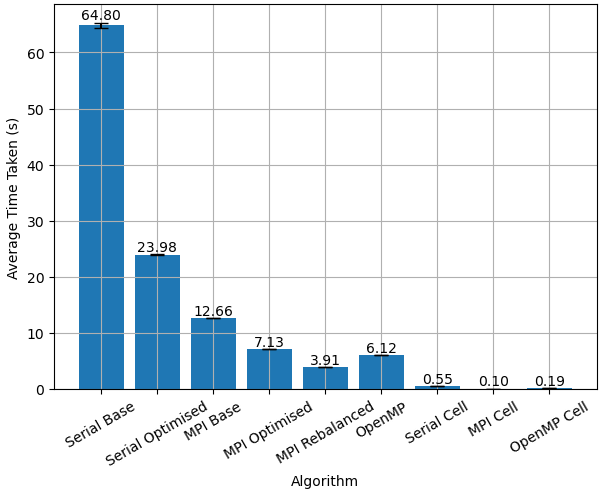

FIG. 3. Compute time for each implementation of the algorithm for 147023 atoms

FIG.2 and FIG.3 shows the time taken to compute for increasingly large systems, 4 nodes/threads were chosen arbitrarily for the parallel implementations to demonstrate any speed ups. The system with 10549 atoms was not shown due to similarity to FIG.3.

In the small system with 120 atoms shown in FIG.2, all implementations achieve similar performance, the optimised serial and MPI cell list algorithms perform the best, however the differences between serial and parallel implementations, and between the brute force and the cell list methods are not immediately apparent. The only clear outliers are both of the OpenMP algorithms, which are much slower than the rest, this indicates overhead present in the OpenMP implementations, likely caused by the time spent on communication between threads, the effect should become negligible at larger scales.

In the large system with 147023 atoms shown in FIG.3, it is clear that the parallel implementations achieved significant speed up compared to the serial methods, the OpenMP performance is now similar to the MPI implementation. More significantly, all of the cell list implementations performed much better compared to the brute force methods, the serial cell list method is 7 times faster than the fastest brute force MPI load balanced implementation, with the parallelised cell list approaches providing further speed up.

The discrepancy in the performance between two methods can be attributed to the fact that the brute force algorithm has a time complexity of $O(n^2)$, this means that the compute time required increases quadratically, whereas in the cell list algorithm, the time complexity is $O(n)$, meaning that the compute time increases linearly. This shows that the cell list algorithm is the much more efficient for large scale systems, in fact, since its performance is comparable to the brute force method also at small scales, it can be argued to be the better implementation overall.

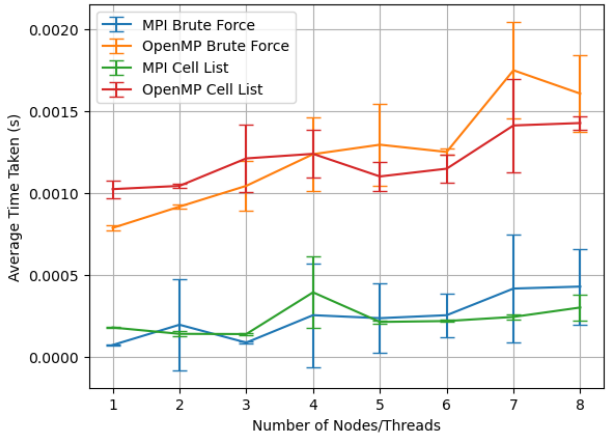

FIG. 4. Compute time against number of nodes/threads for a small system of 120 atoms.

FIG.4 shows parallelisation in small scale systems do not provide any speed up, and increases the compute time with more nodes/threads, this is consistent with both the brute force and the cell list method, and across both MPI and OpenMP, this is most likely due to the increase in communication time between nodes.

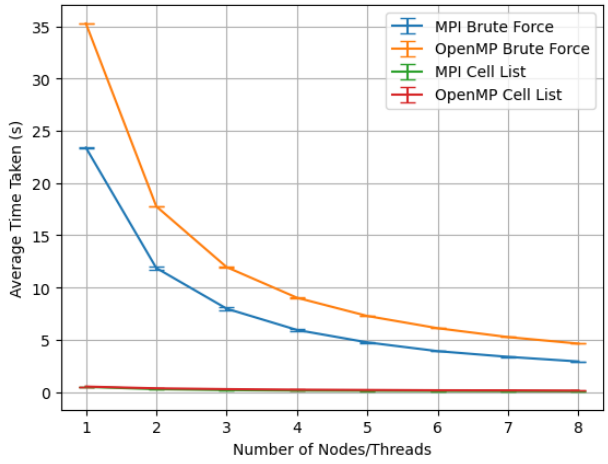

FIG. 5. Compute time against number of nodes/threads for a large system of 147023 atoms.

FIG.5 shows that at a large scale, significant speed up is achieved via parallelisation, with the time decreasing reciprically with increasing number of nodes/threads. Both cell list implementations are shown to be much more efficient compared to the brute force methods. It is interesting to note that, in all cases the MPI implementation performs slightly better than OpenMP.

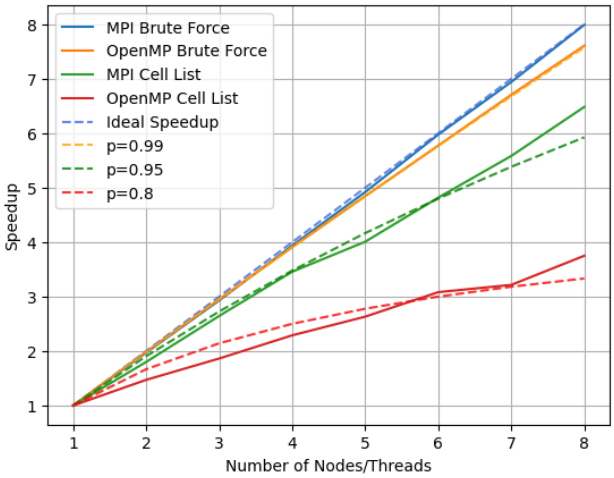

FIG. 6. Speed up against number of nodes/threads for system with 147023 atoms.

FIG.6 shows the parallel speed up from the data in FIG.5. Both implementations using the brute force method achieves the largest speed up, the MPI load balanced implementation achieves close to 100% efficiency and the OpenMP implementation obtains a p value of 0.99 using Amdahl's law, which means that 99% of the process ran in parallel. In comparison, the speed up achieved by the cell list method is substantially less, obtaining p values of 0.95 for the MPI implementation and 0.8 for the OpenMP implementation. This is likely caused by the creation of the cell list, which runs serially. Theoretically this means that the brute force method scales better at higher number of nodes, however from FIG.5 it is clear that at any large scale system and any realistic number of nodes, the cell list method would most likely be a much better implementation.

CONCLUSION

The investigation successfully implemented the nearest neighbour algorithm in parallel using both MPI and OpenMP demonstrating speed up with a high degree of efficiency in large systems. However, no speed up was observed in small systems, and parallelisation can potentially increase the compute time due to the time required for communication across different nodes/threads. A brute force method was also compared against the cell list approach, it was found that the cell list performs equally efficiently at small scales, and drastically faster in larger systems, and therefore is the better algorithm. Further improvements could be made to memory management, currently both algorithms copies the same memory to all nodes/threads. This could lead to memory bottlenecks in very large systems, where it would be useful to split the chunks of memory and distribute between the nodes/threads.