A Performance Comparison of Various Methods of Calculating the Atom Neighbour Problem.

January 2023

1 Code Overview

The .cpp file includes the methods to perform the Number of Nearest Atom Neighbours using various strategies such as brute force and cell list. In total there are 8 different methods as shown below.

Num	Method
1	noMPICellList
2	MPICellList
3	noMPIBruteForce
4	noMPIBruteForceOptimised
5	MPIBruteForce
6	MPIBruteForceOptimised
7	MPIBruteForceWorkBalanced
8	MPIBruteForceWorkerMaster

Figure 1: Table Showing the Method Variations Used.

For each of these methods there will be an overview of how the code works. Some methods share large parts of their code, therefore I will just note the differences between the two methods where required.

Note: For all methods preceded by noMPI although MPI was included, the method does not take advantage of multiple processes as MPI was only used for the MPI-Wtime() function.

1. noMPICellList: The input cellWidth controls the width of each cell and is responsible for the total number of cells in the cube that the data gets spilt into. A 1D vector "neighbours" was created to store the integer amount of neighbours of an atom at that atoms index. A 4D vector cellList was created to store the atom indexes i in their respective cells. The indexing used for the cells was [ix][iy][iz] where ix corresponds to the cell which i's x value sits in, etc. The indexes are looped over and sorted in their cells. Then the cells are looped through and each cell checks for neighbouring cells. If there is a neighbour, the code loops through the atom indexes of the current cell i and atom indexes of the neighbouring cell j. If i and j are closer than a radius r together, it adds 1 to the index i in the vector "neighbours" otherwise it carries on. The method finishes with the "neighbours" vector containing the neighbour counts for all the atoms.

2.MPICellList: Identical to the noMPICellList however the loop which checks over all the cells is split up and assigned to different processes. Each process has their own "neighbours" vector, which are reduced into a "totalNeighbours" vector in the main node (id = 0) once all the processes have finished.

3.noMPIBruteForce: The simplest method: for each atom index i, it goes through all the other indexes j. If i and j are withing r then 1 is added to the "neighbours" vector at index i, else it continues through the loop.

4.noMPIBruteForceOptimsed: Similar to noMPIBruteForce however using a very simple and effective optimisation. Since calculating the distance between i and j is the same as calculating the distance between j and i, lots of the indexes in the for loop can be skipped. This is done by adding 1 to both the "neighbours" vector at index i and j at the same time. This allows the amount of work to be cut in half and works by setting the j loop to start at j=i+1.

5.MPIBruteForce: Using the noMPIBruteForce with MPI to split the for loop looping over index i for different is to be run by different processes. The "localNeighbours" vector in each process is reduced into a "totalNeighbours" vector in the main process.

6.MPIBruteForceOptimised: This is MPIBruteForce but using the j=0 -¿ j=i+1 optimisation.

7.MPIBruteForceWorkBalanced: This method balances the load between processes. If process 1 is getting significantly more work then process 5, then process 5 may be idle for most of the run: therefore it is efficient to load balance. It works by calculating the total work that the system needs to run and the divides that total by the amount of processes. This gives an estimate for how much work each process should be getting. A vector is created to store the

index boundaries of roughly equivalent amounts of work. The processes then access this vector during the brute force for loop to see which indexes they calculate. The "localNeighbours" vector is then reduced into the "totalNeighbours".

8.MPIBruteForceWorkerMaster: This method is an alternate method to load balance. It works by assigning one node to be the master. The master node then deals out work to workers, who then perform the work and ask for more once they're done. This balances the load as the processes only get more work if they are idle. The code has the master node send out atom indexes for which each process performs their own brute force. Each process adds their finished work to their local "localNeighbours" vector and then sends a signal to the master node to receive more work. While there is work left to do, the master node will keep sending i's and the workers will keep receiving. When there is no more work the master will send a signal to the workers to tell them to reduce to the "totalNeighbours" vector. The master node then has a vector with all the data.

2 Results

The average, maximum and minimum number of neighbours were calculated for the three .xyz files provided. The results are shown below.

File.xyz	Avg	Max	Min
120	24.2333	44	7
10549	19.7205	31	3
147023	36.6397	51	4

Figure 2: Average, Maximum and Minimum neighbours for the data files.

3 Method Performance

Below are tables showing the timings for the various methods. Showing times for (1) process and the quickest time (x) processes.

File = argon120.xyz	
Method (tasks)	Times (s)
noMPIBruteForce (1)	0.00365
noMPIBruteForceOptimised (1)	0.00214
MPIBruteForce (1)	0.00438
MPIBruteForce (10)	0.00101
MPIBruteForceOptimised (1)	0.00438
MPIBruteForceOptimised(5)	0.00102
MPIBruteForceWorkerMaster (1)	0.02031
MPIBruteForceWorkerMaster (10)	0.00236
MPIBruteForceWorkloadManaged (1)	0.00233
MPIBruteForceWorkloadManaged (5)	0.00132
noMPICellList(w=9) (1)	0.00589
MPICellList(w=9)(1)	0.006451
MPICellList(w=9)(5)	0.00352

Figure 3: Timing Data for argon120.xyz

File = argon10549.xyz	
Method (tasks)	Times (s)
noMPIBruteForce (1)	13.7269
noMPIBruteForceOptimised (1)	6.97252
MPIBruteForce (1)	13.5726
MPIBruteForce (20)	0.99361
MPIBruteForceOptimised (1)	7.33788
MPIBruteForceOptimised(20)	0.463155
MPIBruteForceWorkerMaster (1)	16.3264
MPIBruteForceWorkerMaster (20)	1.17682
MPIBruteForceWorkloadManaged (1)	6.97338
MPIBruteForceWorkloadManaged (10)	0.849575
noMPICellList(w=10) (1)	22.2171
MPICellList(w=10)(1)	21.7835
MPICellList(w=10)(20)	2.6461

Figure 4: Timing Data for argon10549.xyz

File = argon147023.xyz	
Method (tasks)	Times (s)
noMPIBruteForce (1)	2630.65
noMPIBruteForceOptimised (1)	1314.71
MPIBruteForce (1)	2661.14
MPIBruteForce (20)	143.08
MPIBruteForceOptimised (1)	1325.9
MPIBruteForceOptimised(20)	82.392
MPIBruteForceWorkerMaster (1)	3216.02
MPIBruteForceWorkerMaster (20)	181.15
MPIBruteForceWorkloadManaged (1)	1328.29
MPIBruteForceWorkloadManaged (20)	80.616
noMPICellList(w=10) (1)	336.987
MPICellList(w=10)(1)	2031.68
MPICellList(w=10)(20)	146.666

Figure 5: Timing Data for argon147023.xyz