Lab 2: Close Range Particle Interactions; MPI and OpenMP
Team ONE

Daniel Putt (ID:1654588)
Vignesh Jeganathan (ID:1552354)

October 15, 2018

Serial

Methodology of Simulation:

- A 3d vector is used to store the particles as per the bins they belong. In the 3d vector: 1st dimension is row of bins, the 2nd dimension is column of bins and the 3rd dimension is a variable list of all particles along with their details (position, velocity and acceleration) belonging to the [row, column].
- After binning, each particle is forced by checking only the neighbor bins, thereby reducing the order of computation
- Once the forcing is done, a check is carried out to find if the particles have changed bins. If changed, the information of those particles are moved from old bin to the bin where they belong.
- This process is repeated for the given time steps.

Serial

Methodology of Simulation:

- Domain is decomposed in x direction. It is split, equal to the number of processors.
- Each processor gets its own set of particles initially and stored in a local vector.
- In every time step, each processor gets ghost particles that are within cut- off distance, from nearby processors.

- Each processor creates a 3d address vector of all the particles belonging to a specific processor and the ghost particles. Vector is binned with bin length equal to cut- off distance. The addresses are stored as per the location of bins in the domain. In 3d vector- 1st dimension is row of bins, the 2nd dimension is column of bins and the 3rd dimension is a variable list of all addresses of particles belonging to the [row, column]
- Force is applied to particles that belong to the processor but not the ghost particles. This is accomplished by using 3d vector only as a reference to look up for particles that are closer to the particle in question, than using them to force particles. Forcing is done using the local particle vector created before.
- Once the particles are forced, their positions are updated
- A check is done to find if all the particles belong to the respective processors after forcing. The particles that dont belong to that specific processor is moved to the processor where they belong and their information is deleted from the processor.
- This process is repeated for the given number of time steps.

Elements used in Simulation:

- A MPI datatype called PARTICLE is created to store the acceleration, velocity and position of particles
- MPI_Bcast, MPI_Send, MPI_Recv, MPI_Reduce are used for communication among the nodes.
- Vector datatype is used to store particles within processors. Vector commands such as size(), erase(), resize(), begin(), end() were used to manipulate particles in vectors.
- To communicate between processors, arrays are used.

Results and Discussion:

Following are the list of simulations completed:

Observations and Inference:

- It can be observed that the strong scaling efficiency is the lowest (0.43) for 500 particles than all other efficiencies. It is because of the overhead in data communication in the MPI. It takes more time to communicate among the processors than performing the calculation itself. However, one can see that the strong scaling efficiency for 5000 particles are way higher (0.84). This again proves the point that the MPI is effective

Strong Scaling:				
S.no	**Number of Particles**	**Number of Processors**	**Speed up**	**Efficiency**
1	500	Serial	1	1
2	500	1	1.19	1.19
3	500	2	1.77	0.88
4	500	4	0.79	0.2
5	500	6	0.88	0.15
6	500	8	0.82	0.10
7	500	16	0.74	0.05
Average Strong Scaling Efficiency- 500 particles				0.43
S.no	**Number of Particles**	**Number of Processors**	**Speed up**	**Efficiency**
1	5000	Serial	1	1
2	5000	1	1.28	1.28
3	5000	2	2.11	1.06
4	5000	4	3.62	0.91
5	5000	6	4.81	0.80
6	5000	8	5.31	0.66
7	5000	16	5.79	0.36
Average Strong Scaling Efficiency- 5000 particles				0.84
Weak Scaling:				
S.no	**Number of Particles**	**Number of Processors**		**Efficiency**
1	500	Serial		1
2	500	1		1.19
3	1000	2		0.93
4	2000	4		0.71
5	3000	6		0.62
6	6000	8		0.37
7	9000	16		0.38
Average Weak Scaling Efficiency				0.70
S.no	**Number of Particles**	**Number of Processors**		**Efficiency**
1	5000	Serial		1
2	5000	1		1.28
3	10000	2		1.07
4	20000	4		0.92
5	30000	6		0.82
6	60000	8		0.47
7	90000	16		0.39
Average Weak Scaling Efficiency				0.83

for large simulations, where number of calculations surpasses message communication overhead.

- In weak scaling of 500 particle, one can observe that the speedup goes down tremendously when the number of processors are increased. This goes against the norm of using multiple processors to gain speedup. This is again because of the communication overhead associated with using MPI. If one sees the 5000 particle simulation, one can observe that the speedup improves in increasing the number of processors.
- The strong scaling efficiencies saturate after some time in 5000 particles. This is the consequence of Amdahls law: The speedup can be only to the extent of availability of parallel portion in the code. The reduction in efficiencies can also be traced back to this fact.

OpenMp

Version- Neighbor List

This was my first attempt at the problem. I created a struct of cells, where each cell held pointers to the particles in the cell, as well as pointers to the neighbor cells.
The advantage here is increased clarity, and after the overhead of calculating and storing the neighbors, they are known and the work to calculate neighbors is reduced.
Disadvantages are the excessive memory usage. The storage of pointers to neighbors isn't worth the memory usage.
Since I am storing pointers, and not the particles themselves, locality is almost certainly not good, but moving particles between cells doesn't have a high cost.
At every timestep I empty the bins and sort the particles into their new bins.

Version- Z-ordered Neighbor List

I attempted to increase locality by reordering my cells so that the distance between neighbor cells in the actual array was minimized. Since this version still only stores pointers in the cells, the benefit wasn't large, but it did help some.
I also attempted a sorting on the actual particles, but since the cost of sorting completely offset the temporary benefit of increased locality, I did not pursue sorting further.

Version- Particles in cells

In this version, instead of storing pointers to the particles, I loaded the particles into their respective bins. I then erased the initial particle array. Every subsequent calculation is performed on the particles stored in the cell struct.
The benefit here is increased locality. My array was sorted column major, and so at least vertical neighbors, and therefore their particles, would be close in memory.
However, the cost of moving a particle from cell to cell wasn't easy to manage, and my attempts did not achieve better results than my neighbor list codes.

Neighbor Lists			
Number of Particles	**Number of Processors**	**Time**	**Efficiency**
500	Serial	0.029769	1
500	1	0.013737	2.17
500	2	0.008716	1.71
500	4	0.07803	0.95
500	6	0.00789	0.63
500	12	0.010945	0.23
500	18	0.017991	0.09
500	20	0.018884	0.07
1000	2	0.0159	1.87
2000	4	0.017267	1.72
3000	6	0.018884	1.58
6000	12	0.02299	1.29
9000	18	0.02914	1.02
12000	20	0.039217	0.76

Discussion of OpenMp Results:

- My neighborlist code, with z-ordering works best out of my versions.
- It is important to note that the parallelization could still use some improvement. It is likely that the efficiency is harmed with larger numbers of threads due to memory bandwidth problems.
- There could also be more efficient use of memory. Not using neighbor lists would be a start, but also, a majority of the cells are empty at any time in the simulation. Making a version where only cells that have particles in them are objects at a given time would dramatically decrease the required allocated memory required to simulate the particles.
- In all of the codes, the algorithm was split into three essential parts: apply forces, move positions, update bins.
- The first two could be parallelized easily, as each loop only updated single particles at a time. The third part, updating the cells, took more care and, in general, required some more single thread calculations.

Neighbor Lists, Z-Ordering			
Number of Particles	**Number of Processors**	**Time**	**Efficiency**
500	Serial	0.029183	1
500	1	0.014176	2.06
500	2	0.009217	1.58
500	4	0.006947	1.05
500	6	0.007407	0.66
500	12	0.010622	0.23
500	18	0.016893	0.10
500	20	0.027408	0.05
1000	2	0.0159	1.78
2000	4	0.017267	1.63
3000	6	0.018884	1.43
6000	12	0.02299	1.24
9000	18	0.02914	1.04
12000	20	0.039217	0.81

Particles in Cells			
Number of Particles	**Number of Processors**	**Time**	**Efficiency**
500	Serial	0.029183	1
500	1	0.01417	0.91
500	2	0.03638	0.69
500	4	0.041739	0.30
500	6	0.028205	0.30
500	12	0.041019	0.10
500	18	0.050469	0.06
500	20	0.065617	0.04
1000	2	0.063279	0.79
2000	4	0.09339	0.54
3000	6	0.088471	0.57
6000	12	0.152151	0.33
9000	18	0.1556	0.32
12000	20	0.409443	0.12