Software Engineering 4F03
Distributed Systems
Final Project

Eric Xiao
Weiyuan Bao
Yi Wang

April 24, 2018

1) Provide timing, speedup, and efficiency plots. For total particle numbers of 2,000 4,000, 8,000, 16,000, 32,000 particles using 1,2,4,8,16,32 nodes.

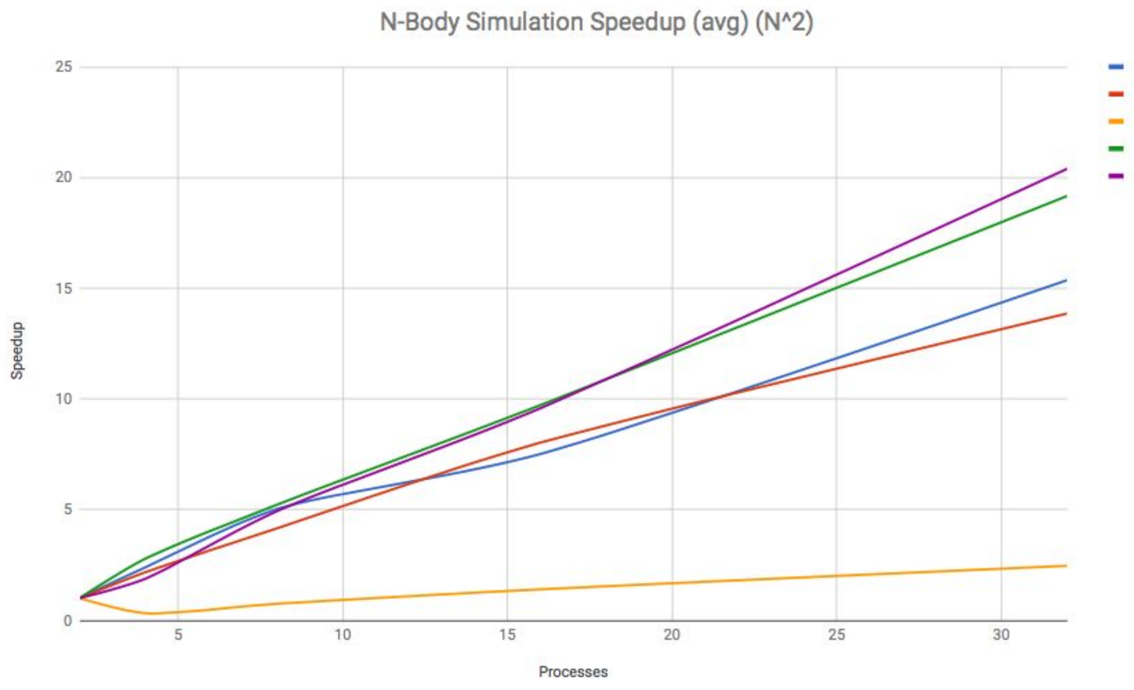
particles	processors	substeps (min time)	substeps (max time)	substeps (avg time)	speedup (min time)	speedup (max time)	speedup (avg time)	efficiency (min time)	efficiency (max time)	efficiency (avg time)
2000	32	0.091891	0.155907	0.116656	18.84	12.32	15.36	0.59	0.39	0.48
2000	16	0.195336	0.300651	0.238694	8.86	6.39	7.51	0.55	0.40	0.47
2000	8	0.313696	0.419760	0.356657	5.52	4.58	5.02	0.69	0.57	0.63
2000	4	0.657240	0.851346	0.750019	2.63	2.26	2.39	0.66	0.56	0.60
2000	2	1.730914	1.921235	1.792143	1.00	1.00	1.00	0.50	0.50	0.50
4000	32	0.413455	0.645299	0.509634	16.75	11.49	13.86	0.52	0.36	0.43
4000	16	0.818889	1.016468	0.879794	8.46	7.29	8.03	0.53	0.46	0.50
4000	8	1.561462	1.887797	1.700770	4.44	3.93	4.15	0.55	0.49	0.52
4000	4	2.979798	3.567539	3.256848	2.32	2.08	2.17	0.58	0.52	0.54
4000	2	6.926240	7.411260	7.061113	1.00	1.00	1.00	0.50	0.50	0.50
8000	32	2.287760	2.702325	2.496728	2.12	2.66	2.46	0.07	0.08	0.08
8000	16	3.342856	3.891870	3.641477	1.23	1.42	1.40	0.08	0.09	0.09
8000	8	5.730229	7.285057	6.411699	0.64	0.75	0.74	0.08	0.09	0.09
8000	4	10.969935	13.832064	12.051192	0.25	0.35	0.31	0.06	0.09	0.08
8000	2	28.036672	29.967777	28.508373	1.00	1.00	1.00	0.50	0.50	0.50
16000	32	7.070516	10.342076	8.946600	20.15	20.26	19.17	0.63	0.63	0.60
16000	16	16.189402	19.257329	17.645743	8.80	10.88	9.72	0.55	0.68	0.61
16000	8	28.879204	38.461536	32.851177	4.93	5.45	5.22	0.62	0.68	0.65
16000	4	54.446020	71.606383	61.587900	2.62	2.93	2.78	0.65	0.73	0.70
16000	2	142.475689	209.568207	171.475105	1.00	1.00	1.00	0.50	0.50	0.50
32000	32	31.235006	38.756226	34.615959	20.09	21.44	20.39	0.63	0.67	0.64
32000	16	67.468127	80.91067	73.723882	9.30	10.27	9.58	0.58	0.64	0.60
32000	8	110.447457	171.030842	143.166855	5.68	4.86	4.93	0.71	0.61	0.62
32000	4	341.025382	395.380800	376.581319	1.84	2.10	1.87	0.46	0.53	0.47
32000	2	627.472482	830.968529	705.991586	1.00	1.00	1.00	0.50	0.50	0.50

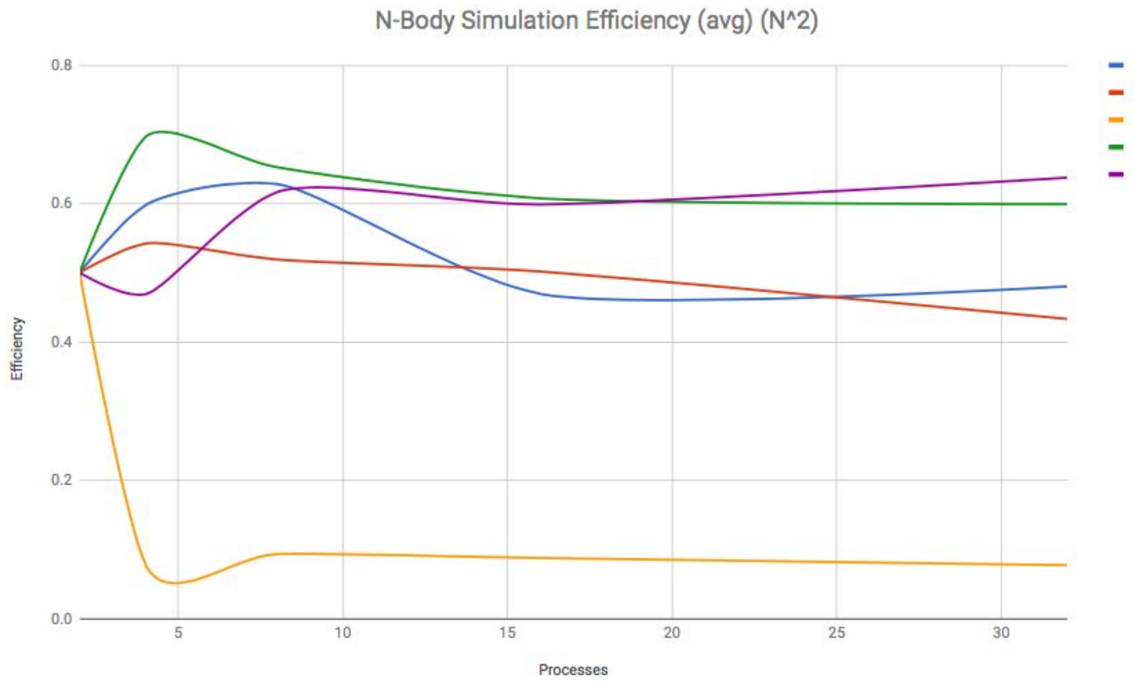
Note:

The particle distribution was done in a fashion that it was even distributed between the 3 types of particles; in which the remaining particles go light then heavy particles.

The values for when the number of processors (p) equals to 1 and 2 will be the same, because there is always 1 processor designated as the master processor.

Data collection and analysis were done using the second method implemented and described below. The plotted data is based on the avg times, speedups, etc.





Legend:

Blue: n = 2,000	Red: n = 4,000	Yellow: n = 8,000
Green: n = 16,000	Purple: n = 32,000	

Definitions:

Superlinear Speedup:

- The data of a serial program may not fit into cache, but could fit in a parallel version

Locality:

- Access to one location is followed by an access to a nearby location

Spatial locality:

- If a memory location is accessed, then a nearby location is likely to be accessed in the near future

Temporal locality:

- Data are referenced repeatedly in a small time window Special case of spatial locality: accessing the same data

Strongly Scalable

- problem size is fixed
- we increase number of processes/threads
- the efficiency is about the same

Weakly Scalable

- we increase problem size and number of processes/threads at the same rate
- the efficiency is about the same

Taken from lecture slides.

Implementation:

Two implementations of the N-body Simulation was coded, our first attempt is shown in *main.c* and our second and more efficient attempt is shown in *faster_main.c*. The second version was referenced from code found online https://github.com/isislab-unisa/nbody-parallel.

First there were some assumptions made in both implementations:

1) As two particles approach each other, there is a possibility that both of them will overlap. The decision was to not check for this case explicitly (i.e. check if the distance between the two particles was less than epsilon), in this scenario, the force should be infinity.
2) When a particle is out of frame of the picture, the decision was made to let it continue its course. This is for the fact that the picture is only a “snapshot” of the universe and it would not be realistic for a particle to bounce of the frame of the picture.

The first implementation that was tested was a ring pass communication method. This method was implemented using the send and receive method in MPI to pass the particles around to the respective processor; a cyclic method was used to distribute the particles.

The second method used was to keep a duplication of the particle data on each of the processors and each processor calculates a portion of the particles new forces. The values were then gathered on master and then updated for the image.

2) Discuss weather or not the algorithm is strongly scalable and/or weakly scalable. Why/why not?

The algorithm is **strongly scalable**. It is very apparent from the graphs that when the number of particles (n) is fixed and the number of processors (p) increases, the efficiency is held strong after the initial “boost” in efficiency. The algorithm is also **weakly scalable**. Throughout the whole simulation efficiency stays at about the 0.50 - 0.60 range, and averaging a higher efficiency at higher values of n. Refer to the *Anomalies* section for outliers.

Understanding of the algorithm used will explain the reasons for it being both strongly and weakly scalable:

```
for $q = 0, 1, \ldots, n-1$
    $\mathbf{F}_{q,i} = 0$
    for $k = 0, 1, \ldots, n-1$
        if $k \neq q$
            compute $\mathbf{f}_{qk,i}$
            $\mathbf{F}_{q,i} = \mathbf{F}_{q,i} + \mathbf{f}_{qk,i}$
```

The number of calculations done in serial:

$$n * (n - 1) = O(n^2).$$

The number of calculations done in parallel:

$$\frac{n}{p} * (n - 1) = O(n^2)$$

Although the serial and parallel versions have the same big O values, the divide by p is very crucial in HPC. There is an at most reduction in 32 times the amount of calculations done by each processor. In both the methods (ring pass and n^2) the initial and final number of communication calls are the same, i.e. the calls to send the initial data to each processor and the collection of the final force calculations.

But in the ring pass method, the number of calculations done had a trade off with the number of send and received needed. In this example there is no trade off, there is only one send or gather called at the end of the force calculations. **Thus the communication costs do not overtake the computation time.**

3) Explain any anomalies in the plots, what could have caused them?

In the timings collected, there is a strange dip in performance for when n equals 8,000. **A potential cause for this can be attributed to the server**. It was observed in various runs of the data collection, that the simulation would “hang” or “timeout” for a unknown reason. This could potentially be because of the load on the server; as experienced in previous assignments, when multiple people were running their jobs, the simulation times were significantly longer. Even with a small subset of the problem this was still the case. As well, it was observed that when a large job was running, many jobs would “hang” and this could have delayed portions of the code execution.

Further interpolating of the graphs, it can be seen that in the efficiency graph that when n equals 4,000, the behaviour is different than that of 2,000, 16,000 and 32,000. The apparent “boost” or hump in efficiency isn’t as obvious as the others. This can further suggest that the above hypothesis is correct. As a script was used to collect the data the simulations for n equals 8,000 were ran right after n equals 4,000. By looking at the graphs, there are signs of decay in efficiency already taking into effect for 4,000 and then when n equals 8,000, that is the tipping point and efficiency dropped significantly.

4) Explain your modifications/findings when tuning your program for best performance (provide plots for comparison i.e., MPI vs. MPI with OpenMP).

Refer to the *Implementation* section for a detailed explanation of the high level design of the different modifications tested.

From previous assignments and lectures there is always a tradeoff between communication time and memory efficiency. Two implementations were tested:

1) The first one was aimed at to **reduce the memory usage**. This was done by using the ring pass algorithm and the MPI send and recv function as mentioned in the implementation section.

   The idea was to reduce the average number of force calculations done by each processor by letting each processor do a subset of the force calculations. Once all the force calculations were done on the processors, they were then sent back to master to update the particle states, and the cycle starts all over again.

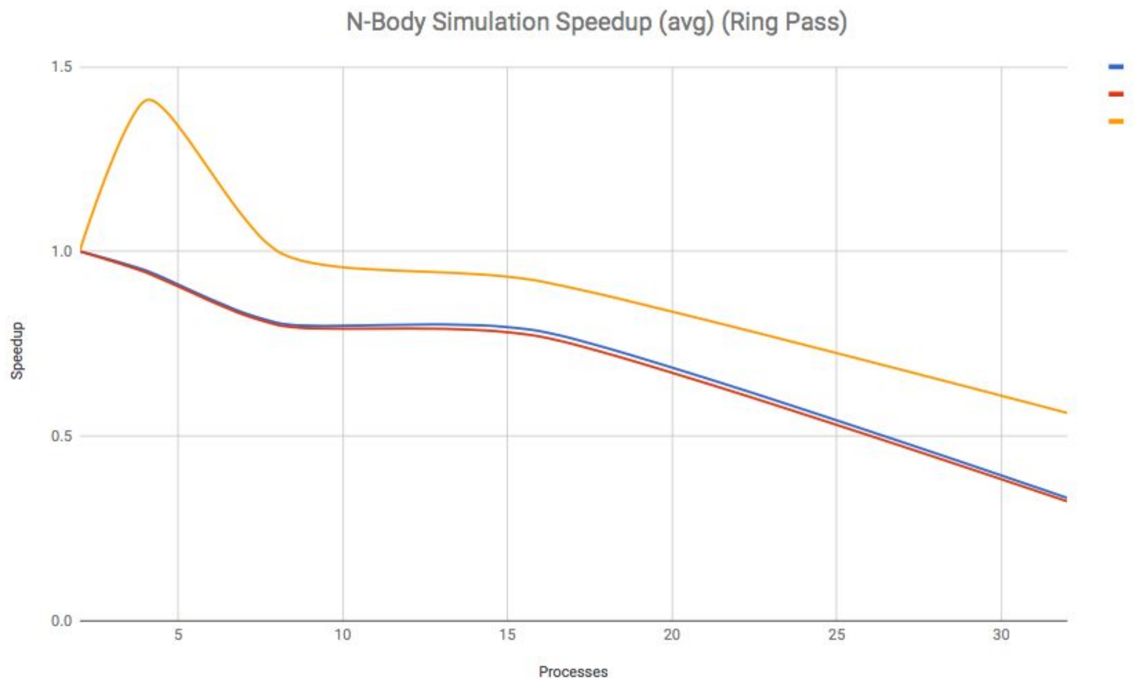
   This results in a much smaller memory footprint, because only a fraction of the total memory had to be allocated on each processor. As well as reducing the computation time (but each processor would have to wait for the longest one to before receiving new data).

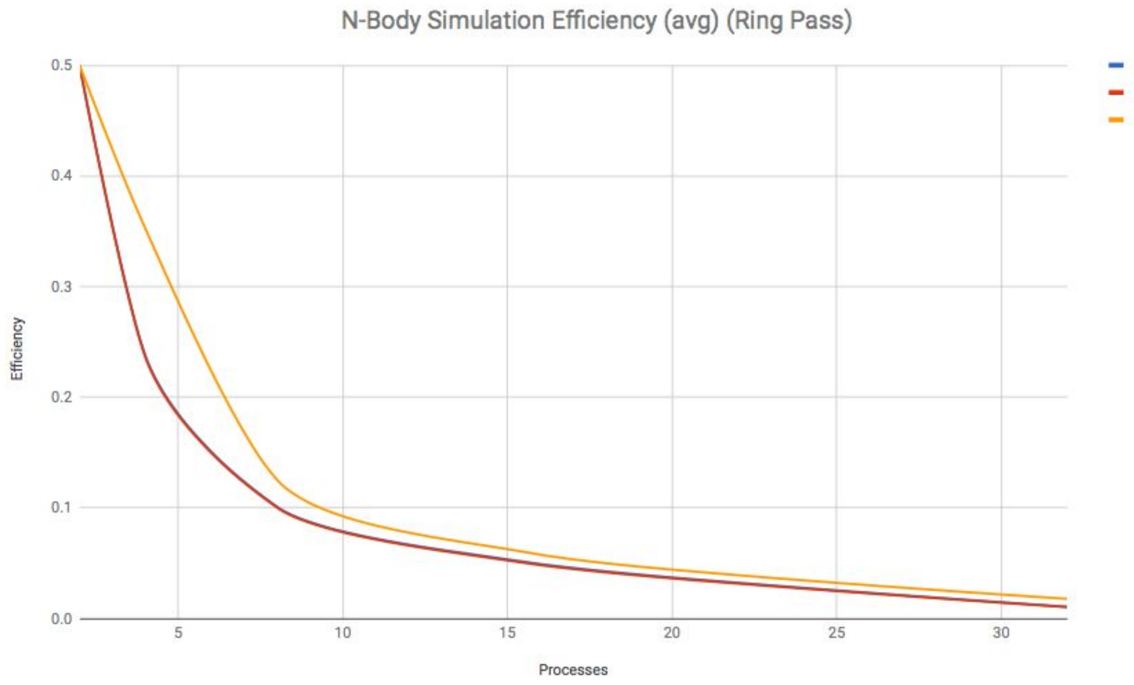
   The **downside of this method is the communication overhead**. For each force calculation multiple send and receives were done to accomplish this. It was observed that this was the main bottleneck of this implementation. Simulations with over 8,000 particles became infeasible.

   A number of reasons can attribute to this. One of which could be how the data is stored, if a more sophisticated method was used as opposed to storing the particles in a sequential order, block distribution could have potentially been used. Thus allowing minimizing the total number of sends by increases the total data sent per send and reducing the overhead in communication calls. Because of the communication overhead, the setup of the server and hardware used in the server are very critical to the runtime of this implementation, which is a black box to us.

particles	processors	substeps (min time)	substeps (max time)	substeps (avg time)	speedup (min time)	speedup (max time)	speedup (avg time)	efficiency (min time)	efficiency (max time)	efficiency (avg time)
2000	32	5.420268	7.405747	6.080367	0.36	0.30	0.33	0.01	0.01	0.01
2000	16	2.347120	3.518747	2.572474	0.83	0.62	0.78	0.05	0.04	0.05
2000	8	2.276158	3.410435	2.498479	0.85	0.64	0.81	0.11	0.08	0.10
2000	4	1.929853	2.909393	2.123874	1.00	0.75	0.95	0.25	0.19	0.24
2000	2	1.936711	2.193946	2.014270	1.00	1.00	1.00	0.50	0.50	0.50
4000	32	22.494743	30.628418	24.143493	0.34	0.26	0.32	0.01	0.01	0.01
4000	16	9.338468	13.269096	10.115679	0.82	0.60	0.77	0.05	0.04	0.05
4000	8	8.529026	12.691684	9.711651	0.90	0.63	0.80	0.11	0.08	0.10
4000	4	7.533408	11.084644	8.241063	1.01	0.72	0.94	0.25	0.18	0.24
4000	2	7.644948	7.938195	7.774083	1.00	1.00	1.00	0.50	0.50	0.50
8000	32	100.399679	154.042107	126.381479	0.68	0.47	0.56	0.02	0.01	0.02
8000	16	44.130447	122.060309	77.281923	1.54	0.60	0.92	0.10	0.04	0.06
8000	8	56.593956	95.026317	70.922577	1.20	0.77	1.00	0.15	0.10	0.13
8000	4	46.598235	57.248741	50.393676	1.46	1.27	1.41	0.36	0.32	0.35
8000	2	67.899381	72.821263	70.988472	1.00	1.00	1.00	0.50	0.50	0.50
16000	32									
16000	16									
16000	8									
16000	4									
16000	2									
32000	32									
32000	16									
32000	8									
32000	4									
32000	2									

These were the values collected for the first method. As shown, as p decreased, the run time decreased as well. This was inverted from the theoretical values that were expected; as the number of processor increase, *usually* the run time decreases as well. But understanding what has happened, it is intuitive that as the p decreases, each processor has more force calculations to do and communication happens between a much smaller subset of processors, although the number of communication calls remains the same.





Legend:

Blue: n = 2,000 Red: n = 4,000 Yellow: n = 8,000

2) The second method was aimed to **reduce the communication overhead**. This was done by storing a local version of the particle data on each processor and then each processor would compute their portion of the total particles.

   It was observed from assignment 2, that calculations of n ^ 3 where n equals 8,000 were feasible in a reasonable amount of time. Here the total amount of calculations performed was n ^ 2 where n equals 32,000. So inspirations of using the slower method but in parallel led to this implementation.

   This resulted in storing data about 32,000 particles on each processor. I.e., if n equals 32,000 and p equals 16, the average memory footprint would be:

Array	Data type	Data size	Size	Memory
type	int	2 bytes	32,000	64 Kb
masses	double	8 bytes	32,000	256 Kb
positions	2 * double	16 bytes	32,000	512 Kb
my_forces	2 * double	16 bytes	2,000	32 Kb
my_velocities	2 * double	16 bytes	2,000	32 Kb

   This is approximately the amount of memory that is required on each processor, about ¾ of a megabyte of data is required on each processor for the respective n and p value.

   Plots for this method can be found in the *Timing, Speedup and Efficiency Plots* section.

   This showed significant gains compared to the first implementation. Which can be directly related to the reduced number of communication calls. As well with a less complicated algorithm, more sophisticated MPI functions were used, such as broadcasts, gathers, etc. . Thus reducing the communication overhead much more.

In conclusion we found that the communication costs were the major bottleneck of our implementations.