Elastic Collisions in Two Dimensions COSC3500 Project Milestone 3

Joel Stuart - 43203714

University of Queensland
joel.stuart@uq.net.au

October 23, 2016

Abstract

This report investigates a parallel implementation of a multi-bodied simulation of elastic collisions in two dimensions. It details the parallelisation and optimisation strategies used in this implementation. It also discusses the performance of the implementation at varying problem sizes and details the verifcation methods.

I. Introduction

To implement a physics simulation such as shooting an object into an asteroid belt or shooting one pool ball at another, the physics of a collision between objects needs to be modelled.

For this report, the scope of the problem has been reduced to the two dimensional plane and to elastic collisions. An elastic collision is a collision between two bodies where the total kinetic energy of the two objects before the encounter is equal to the total kinetic energy after the encounter.

The simulation of object collisions is applicable to many areas of science including (but by no means limited to) describing the collision of atoms (Rutherford backscattering) and various physics problems.

This report wil investigate a parallel implementation of a multi-bodied simulation of elastic collisions in two dimensions.

It will detail the parallelisation and optimisation strategies used. It will discuss the performance of the implementation as well as the verification methods used.

II. Serial Implementation

i. Algorithm

The implementation of this N-bodies simulation abstracts the problem space as a collection of objects with known position and velocity over a collection of time frames or time steps.

The algorithm used to implement this simulation can be broken down into three key areas:

- Generating the object cloud and external object / white ball
- Propagating objects forward in time according to their velocities and positions
- Checking for and handling collisions between objects

ii. Object Generation

A user set number of objects are generated for an initial time reference using C's built-in pseudo-random number generator $rand()$.
Each object is stored as an array of size 4 which contains x and y components for both position and velocity.

iii. Object Propagation

Using the generated set of objects at initial time frame i, all subsequent time frames $i + 1$ through j can be calculated by simply adding the velocity vector of each object to its position vector.

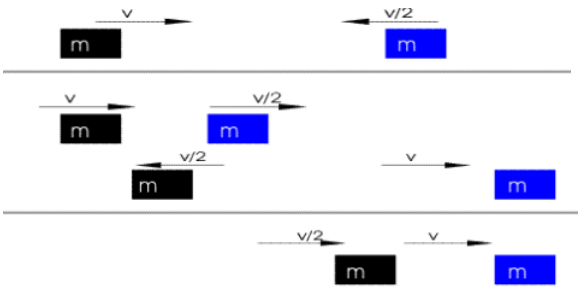
Thus, by simply repeating this process, the position and velocity of every object in every time frame can be calculated.

iv. Collision Checking and Handling

Using the above known positions and velocities, the distance between the centre point of each unique pairing of objects can be compared to the size of each object. Assuming the objects are circular and if the distance between the two bodies is found to be less than their sum of radii, the objects can be said to be overlapping i.e. colliding.

When two objects are known to be colliding and have the same mass, swapping their velocities at moment of impact will satisfy the laws of momentum.

Figure 1: *Elastic collision on same mass objects - [?]*



Profiling reveals this area of the implementation as a computational hotspot as the problem size increases. This is due to a number of expensive operations (multiplication and square root) being called for each unique pairing of objects.

This can be approximated as $!n \times t$ calls for each operation where n is the number of objects and t is the number of time steps.

v. Verification

The following steps have been taken to verify the implementation:

- Check functionality at small object and time step values
- Print all position and velocity values into a file and manually check physics.
- Print all collisions and related object information into file upon occurance

As discussed briefly in the previous section, the collision physics used in the collision handling will not be valid for angled collisions. This will be corrected in future implementations by separating collision logic into head collisions and angled collisions.

In future revisions, object positions may be visualised with an external program to further verify correctness.

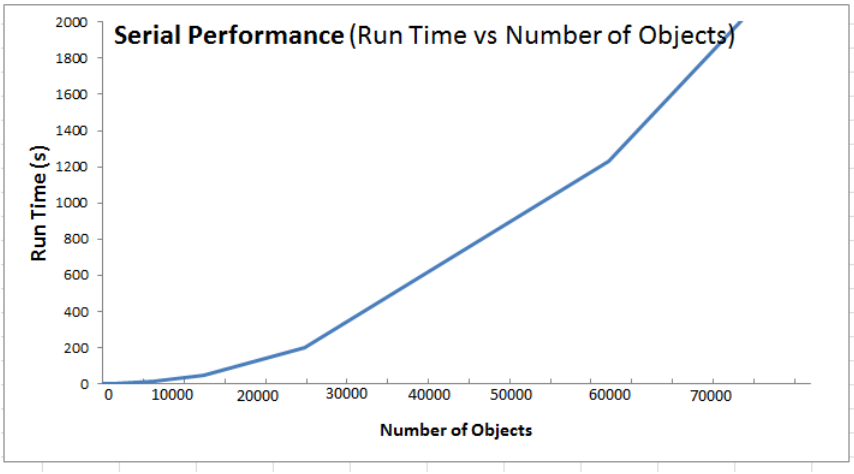
vi. Serial Performance

Results of performance scaling with problem size can be seen in table 1.

Table 1: *Performance (run time) over number of objects*

Problem size		
Num. Objects	Num Timesteps	Run Time (s)
50	20	< 0.000001
500	20	0.05
1000	20	0.28
5000	20	5.50
10000	20	19.80
500	100	0.21
1000	100	0.70
5000	100	13.86
10000	100	45.00
500	500	2.16
1000	500	7.16
5000	500	130
500	1000	4.40
500	5000	20.30

Figure 2: *Graph of Serial Implementation Performance*



III. Parallel Implementation

i. Parallelisation and Optimisation Strategies

As the time portion of this problem is inherently sequential, parallelisation over the time domain is outside the scope of this project.
Thus, the parallelisation methods used in this project can be broken down into the following (concerning the parallelisation of objects within each timestep):

- OpenMP parallelisation for the initialisation of all objects (including random number generation)
- OpenMP parallelisation for the propogation of objects forward in time
- MPI parallelisation to calculate collisions between every unique object pair (through domain decomposition of the list of objects)

The overall runtime speedup of the OpenMP parallelisation optimisations was expected to be minor compared to speedups through parallelisation of the collision detection and simulation routines.
This was suggested by initial profiling of the serial implementation and Big-Oh analysis of the collision detection and simulations routines (due to nested for loops with costly mathematical operations and numerous array operations within).

To support MPI parallisation, several adjustments needed to be implemented:

- Domain decomposition of the object list to facilitate parallel collision detection and simulation
- Maintaining a buffer of colliding object pairs which lie outside of the local domain of each thread to avoid concurrency conflicts.
  This buffer was then used to serially calculate the collisions after the local collisions had been gathered into the global object list.

ii. Parallel Performance

Results of performance scaling with problem size can be seen in Table 1 and Figure 2.

Table 2: *Average performance over problem size*

Problem size		
Num. Objects	Parallelisation	Run Time (s)
200	Milestone 1 Code	0.41s
200	Milestone 2 Code (serial)	0.67s
200	OpenMP (2)	0.61s
200	OpenMP (4)	0.61s
200	OpenMP (8)	0.61s
200	OpenMP (2) MPI (2)	0.59s
200	OpenMP (2) MPI (4)	0.59s
2000	Milestone 1 Code	8.8s
2000	Milestone 2 Code (serial)	8.6s
2000	OpenMP (2)	6.8s
2000	OpenMP (4)	6.8s
2000	OpenMP (8)	6.8s
2000	OpenMP (2) MPI (2)	4.3s
2000	OpenMP (2) MPI (4)	2.6s
5000	Milestone 1 Code	50s
5000	Milestone 2 Code (serial)	37s
5000	OpenMP (2)	37s
5000	OpenMP (2) MPI (2)	24.3s
5000	OpenMP (2) MPI (4)	11.1s

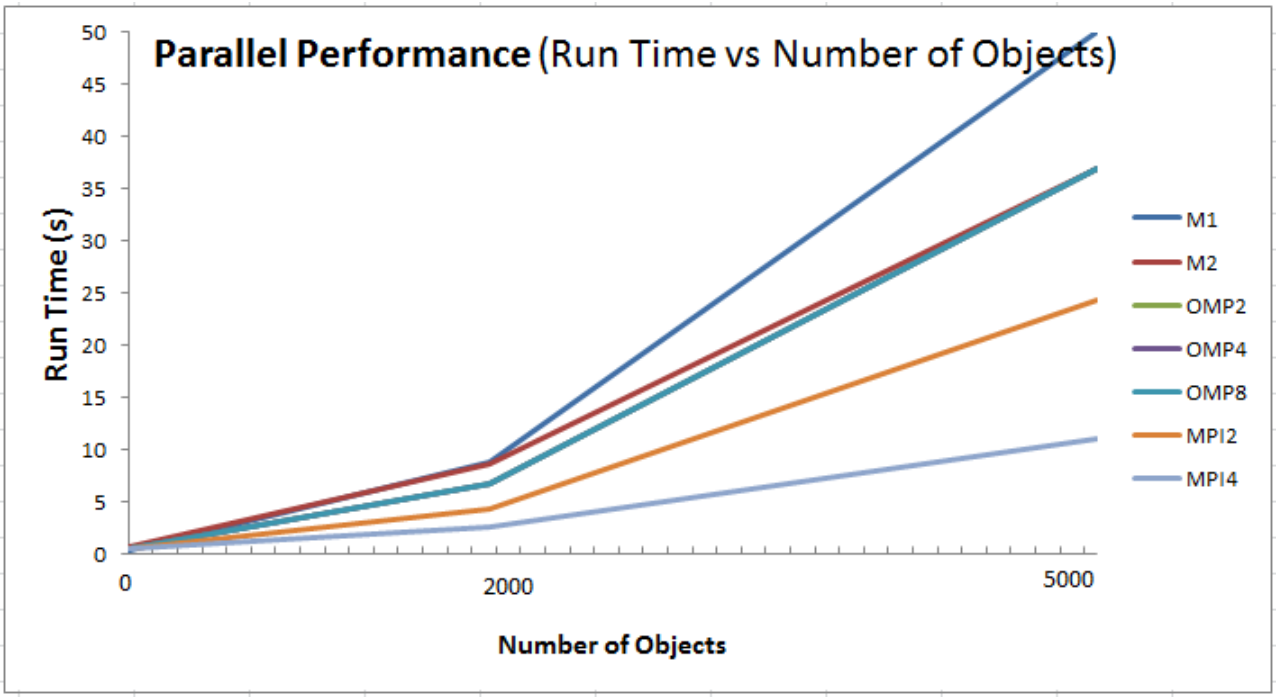
iii. OpenMP Analysis

As suggested earlier in the report, the effectiveness of the OpenMP parallelisation in terms of overall computation speedup was minor due to comparitive cost of the collision detection and simulation. For moderately sized problems, OpenMP offered a slight improvement in run time.

iv. MPI Analysis

The most effective speedups offered were by the MPI parallelisations, which resulted in significant speedups with large numbers of objects. This is a result of the performance of the implementation being bound by the collision

Figure 3: *Graph of Parallel Performance (Average Run Time vs Number of Objects)*



algorithm. This analysis is further expanded on in section 4.

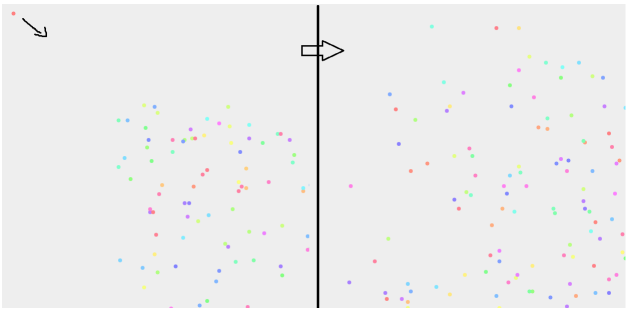
v. Verification

To verify the small scale implementation, a visualiser was built in Java which plots all of the objects within 2d space and animates over the time steps, using the output file of the C program. This visualisation is a means to verify the implementation at smaller problem sizes.

The visualiser is bundled with the report in both .jar file and executable form. This is also located on the repository along with the source code.

The bundled visualiser is able to visualise objects in the domain and range of [-200, 200] (adjustable in source code). It is worth noting that the frame rate of the visualiser suffers with large problem sizes (greater than 500 objects).

Figure 4: *Java Visualiser Demonstration*



Further verification of the implementation can be achieved by checking individual objects in the output file. The following steps have been taken to verify the implementation:

- Using the visualiser, check object behaviour is as expected.
- Print all position and velocity values into a file and manually check physics, ensuring conservation of energy.

- Print all collisions and related object numbers into file upon occurance.

IV. Large Scale

i. Testing and Experimental Process

The testing process was as follows.

Building upon the previous milestones, the following steps have been taken to verify the implementation:

- Using the visualiser, check object behaviour is as expected.
- Print all position and velocity values into a file and manually check physics, ensuring conservation of energy.
- Print all collisions and related object numbers into file upon occurence.
- Test validity of individual outputs for each thread with the above methods.
- Check for coherency of results between threads by printing and comparing values.

ii. Further Optimisations

Building toward a large scale implementation, several parts of the code were rewritten to minimise memory usage and free up memory.

An optimisation discovered while writing this report, was to adopt the single precision floating point data type (32 bits) as opposed to the double precision data type(64 bits). For the needs of the project, single precision had sufficient accuracy. The performance gain of this optimisation was a speed up of about 1.6x for the large scale tests.

iii. Performance Analysis

Using the results of the small scale parallelisations, and after testing of large scale with various configurations, only three configurations were used for the large scale implementations. These configurations are as follows:

- The serial implementation.
- The parallel implementation with 4 MPI ranks and 2 OpenMP threads.
- The parallel implementation with 8 MPI ranks.

The rationale for these configurations is as follows:
Since the algorithm used to calculate collisions between two objects is roughly $O(N^2)$, and all other algorithms used in the implementation is $O(N)$, the runtime of the implementation will be bound by the performance of the collision implementation.
Thus, the parallelisation method which will theoretically provide the most scalable speedup will be one which achieves the most parallelisation of the collision algorithm.

The above configurations were chosen to maximise the usage of the available resources on the cluster whilst testing the above argument.

Regarding the bounds on the number of objects used; tests of greater than 70,000 objects caused the process on the cluster to crash. This was attributed to running out of local memory due to numerous allocations of large arrays on each MPI rank. A rewritten implementation with a focus on memory management may be able to extend this limit further. This was may be considered in future work.

As can be seen in Figure 5, the implementation scaled well with MPI ranks. The above theory on the optimum parallelisation configuration was supported.

Furthermore, for the numbers of MPI ranks used, i.e. for small numbers of nodes, the implementation appeared to have near Linear Strong Scaling. This can be seen in Figure 6. However, this scaling is not expected to hold as the MPI ranks further increases, with communication costs expected to quickly result in diminishing returns in performance.

Figure 5: *Graph of Large Scale Performance (Average Run Time vs Number of Objects)*

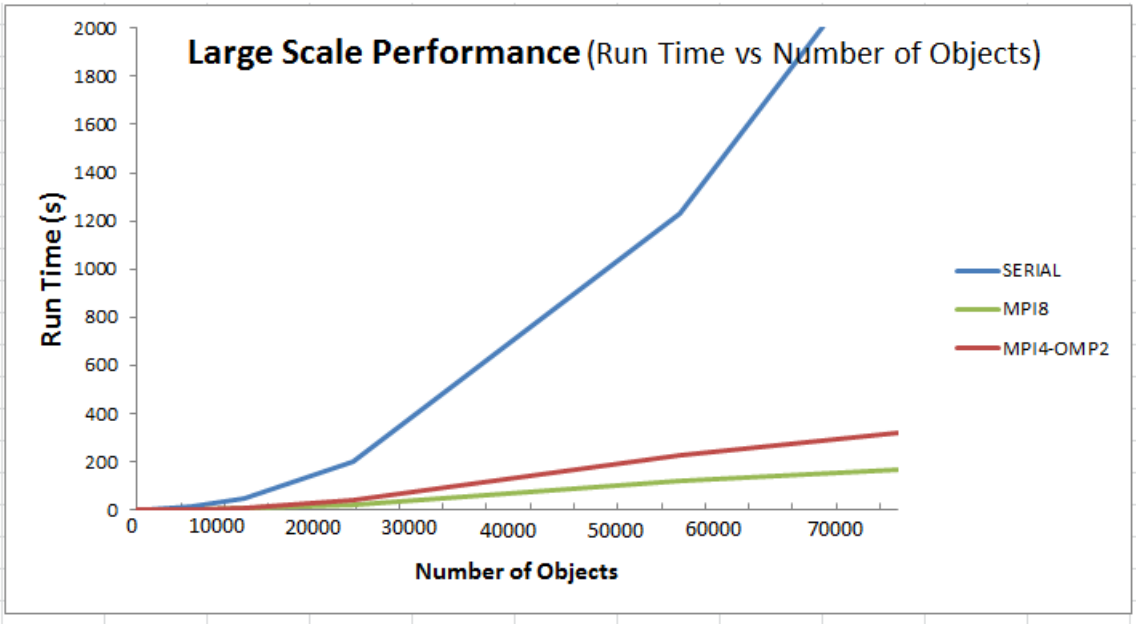
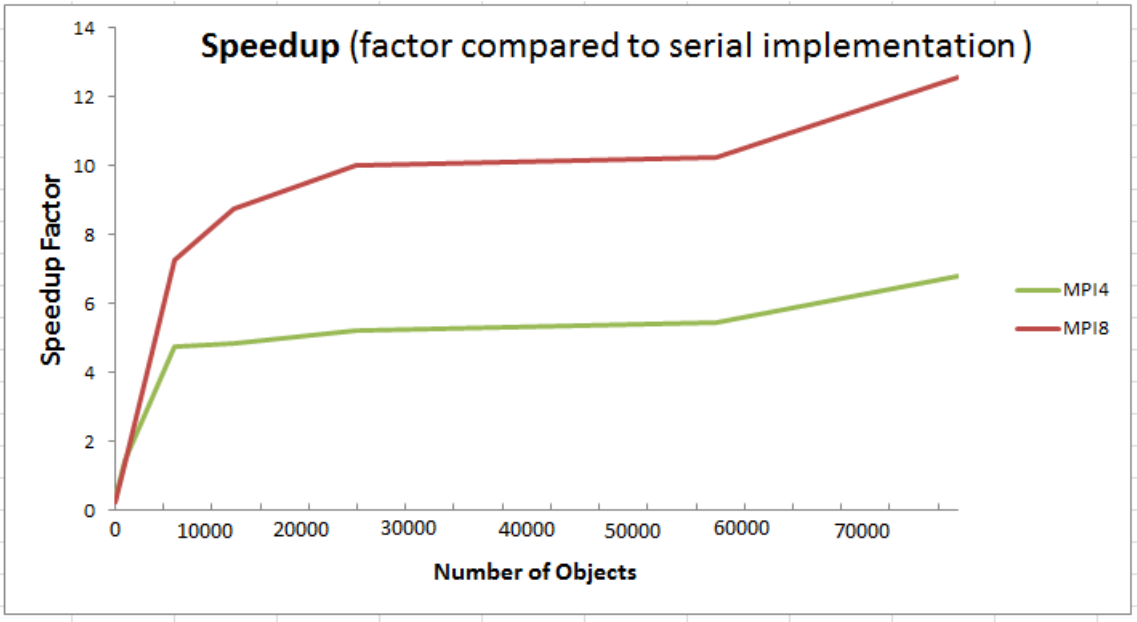


Figure 6: *Graph of Speed Up Factor (compared to Serial Implementation) vs Number of Objects*



Due to the intrinsically deterministic nature of time propagation making time parallelisation impossible as well as concurrency issues and memory usage issues plagueing the object parallelisations, there was no possible Weak Scaling.

In terms of asymptotic performance, the parallel implementation runs well below the serial implementation's $O(N^2)$, estimated as bounded close to (but above) $O((N/p)^2)$, where p is the number of parallel MPI processes. This estimation assumes Linear Strong Scaling, which as discussed above, is not expected to hold indefinitely as p increases.

Thus the asymptotic performance, a, of the parallel implementation is bounded by the following:

$$O((N/p)^2) > a > O(N^2) \tag{1}$$

iv. Discussion

Concerning the initially proposed scientific task (i.e. a simulation of physical collisions between 2D objects such as asteroids of pool balls), the project has demonstrated the required large scale simulation capabilities.

Due to the HPC focus of this project and the small room for rigorous investigation within the topic, scientific investigation relating to the topic of 2D Elastic Collisions was minimal.

Within the implementation itself, the most significant problems encountered were concurrency issues and memory usage issues. Concurrency issues were resolved by doing multi-staged passes of collisions in serial after the initial parallel pass.

Memory usage issues were remedied primarily by tighter memory management and changing from double precision floating numbers to single precision. As discussed earlier, this contributed to a significant speedup, but also extended the memory constraints.

Unfortunately, memory usage issues still remained a bottleneck on the number of objects possible in parallel implementation - stemming in part from the solution to concurrency issues. The author remarks that such an issue is probably not an uncommon occurance within the parallel optimisation field.

Possible improvements or areas of further study may include:

- Solutions to concurrency issues.
- Solutions to memory usage.
- Decomposition of objects by location rather than arbitrary number.
- More efficient collision detection algorithms using reductive heuristics.
- Improved visualisation and verification methods.
- Investigation of performance in three dimensions.

V. Conlusion

This report has summarised the performance and scalability of an implementation of Elastic N-Body Collisions. This problem was found to scale well with parallel MPI ranks, with the cost of communications expected to quickly become a bottleneck for larger numbers of MPI ranks.

VI. Appendix A - Collision Detection Algorithm

```
//Check collisions between every object
for(i=0; i < num; i++){
    for(j=localNum+1; j < localTopVal; j++){
        //If not the same object.
        if (i != j){

            double dx = objList[i].xpos - objList[j].xpos;
            double dy = objList[i].ypos - objList[j].ypos;
            double dist = sqrt((dx*dx)+(dy*dy));

            //If collision detected between objects (radius defined in header)
            if (dist < rad){


                //If not in thread domain, add object numbers to buffer for later
                if (i < localNum || i >= localTopVal){
                    buffer[bufferUsed] = i;
                    buffer[bufferUsed+1] = j;
                    bufferUsed += 2;
                //Else do collision
                } else{
                    collisions += 1;
                    collide(i, j, objList, localObjList, localNum);
                }
            }
        }
    }
}
```

VII. Appendix B - Buffer Algorithm

```
//For each thread, recieve data and update into buffer array and global object list
for(t=1; t<size; t++){
    int tag, collisionsT;
    int localNum = localSize*t;
    int *localBuffer = malloc(sizeof(int)*2*NUM_OBJ);
    obj* localObjList = malloc(sizeof(obj)*localSize);
    int bufferUsed, b;
    MPI_Recv(&localObjList[0], localSize, mpi_obj, t, 1, comm, &status[t]);
    MPI_Recv(&bufferUsed, 1, MPI_INT, t, 2, comm, &status[t+1]);
    MPI_Recv(&bufferLocal[0], bufferUsed, MPI_INT, t, 3, comm, &status[t+2]);

    //Insert buffer array size
    bufferUsedA[t] = bufferUsed;

    //Update buffer array with buffer from thread
    updateBufferA(bufferA, bufferLocal, bufferUsed, bufferPos);

    //Increment position within flat 2D big buffer array
    bufferPos += bufferUsed;

    //Update global object list
    updateList(localNum, localSize, objList, localObjList);

    free(localBuffer);
    free(localObjList);

}

//Now all data has been loaded into root,
//Update global object list with buffered collisions!
bufferPos = 0;
//For each thread/rank
for(t=0; t<size; t++){
    int b;
    int bufferUsedLocal = bufferUsedA[t];
    //For each object pair
    for (b = 0; b < bufferUsedLocal; b++){
            collisions += 1;
            bufferCollide(bufferA[bufferPos+b], bufferA[bufferPos+b+1], objList);
            b += 1;
    }
    bufferPos += bufferUsedLocal;
}
```