

Extending the Educational Scope of a Particle-Based Simulation Framework through Parallelization

T. Francis Chen and Gladimir V. G. Baranoski

Natural Phenomena Simulation Group,

School of Computer Science, University of Waterloo, Canada

t4chen@cs.uwaterloo.ca, gvgsbaran@curumin.math.uwaterloo.ca

ABSTRACT

Particle systems have been incorporated into a wide variety of applications in both academia and industry. They can be employed to investigate complex natural phenomena, illustrate scientific concepts and generate special effects for entertainment purposes. Recently, we implemented an educational simulation framework based on particle systems that can be used to perform interactive virtual experiments involving complex physical laws. The positive feedback received from a pilot deployment of this framework motivated us to look for strategies to increase its scope. However, more complex and engaging simulations require the use of a larger number of geometric primitives (particles), which results in higher computational costs. To mitigate these costs, we resorted to the implementation of parallel techniques through the use of the Message Passing Interface (MPI) standard. In this paper, we describe these techniques and discuss the performance gains resulting from their application to the simulation algorithm that forms the core of our framework. These results were obtained through practical test cases which are also described in detail in this paper.

KEYWORDS: Education, particle system, simulation, application, parallel processing.

1. INTRODUCTION

A particle system involves the generation and movement of numerous elemental primitives [1]. The rules governing their motion can be customized for a wide spectrum of applications in different areas. In fact, particle systems are

extensively employed not only in education [2, 3] and research [4, 5], but also in the production of visual effects for movies and games [1, 6].

Numerous parallel implementations of particle systems have been presented in the literature [7, 8, 9, 10]. Additionally, several parallel efficient algorithms have been specifically developed to solve more general N -body problems. Notable examples include the Barnes-Hut algorithm [11], the fast multipole algorithm [12] and the parallel multipole tree algorithm [13].

The work presented in this paper is aimed at educational applications involving three dimensional physics simulations. Such applications have been found to enhance learning by providing high levels of immersion [3]. They usually employ visualization tools to make the presentation and discussion of physical phenomena more interesting and engaging. Recently, we have developed a simulation framework that uses OpenGL [14] graphics features to teach children different scientific concepts. More specifically, the effect of different environments on motion and the theory behind anaglyphs and binocular vision [15] are addressed. This framework was presented as a short interactive demo at a pilot educational event at the University of Waterloo (Canada). Figure 1 presents a sample screenshot of the running simulation meant to be viewed with red-blue anaglyph glasses. The arcing lines represent the trajectories of the spheres.

We plan to increase the educational scope and the effectiveness of our framework in future events. These goals will be achieved by depicting a wider variety of physical phenomena and employing a larger number of particles in our simulations. Since the corresponding computational costs depend on the number of particles, it is necessary to apply a parallel strategy to avoid performance degradation

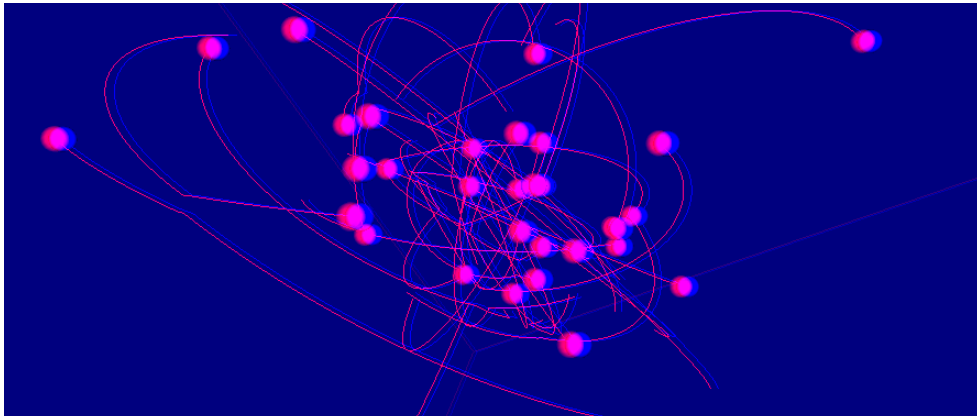


Figure 1. A Screenshot of the Simulation Framework

and keep a high level of effectiveness. In this paper, we present different schemes that can be used to achieve these goals. These schemes, which were implemented using the Message Passing Interface (MPI) standard [16], have been applied to the simulation of different physical phenomena, and their effects on the parallel run time, speedup and efficiency of the simulations are quantitatively analyzed in this work.

The remainder of the paper is organized as follows. Section 2 presents key aspects of the simulation algorithm and outlines the physical phenomena depicted in our educational framework. Section 3 describes the different parallelization schemes investigated in this work. The results obtained from each of these schemes are discussed in Section 4. Finally, Section 5 closes this paper with a summary of our findings.

2. THE SIMULATION ALGORITHM

The simulation algorithm used in our educational framework updates the state of every particle at evenly distributed timesteps. The particles are represented by unit spheres of equal mass, and the state of each particle is defined by its position and velocity. Updates are based on Newtonian equations. Specifically, the position is updated using

$$\mathbf{x}_{i+1} = \mathbf{x}_i + \mathbf{v}_i \Delta t, \quad (1)$$

and the velocity is updated using

$$\mathbf{v}_{i+1} = \mathbf{v}_i + \mathbf{a} \Delta t, \quad (2)$$

where \mathbf{x}_t and \mathbf{v}_t denote the position and velocity at timestep t respectively, \mathbf{a} is the acceleration imposed by the simulation environment, which is outlined in Subsection 2.2, and Δt is the timestep. The actual positions of the

spheres are restricted within a bounding volume. The simulations are initiated with n spheres placed randomly and uniformly throughout the bounding volume. The initial velocity of all spheres is zero.

2.1. Collision Detection

A direct approach is used to detect collisions. At every timestep, every sphere, s_i , has its position checked against every other sphere, s_j , $j \neq i$. The state of s_i is adjusted if the distance between s_i and s_j is smaller than the sum of their radii. This results in $O(n^2)$ operations, which correspond to the main bottleneck of the simulation.

Note that adjustments cannot be applied as collisions are detected. For example, let three spheres be denoted as s_1 , s_2 and s_3 . Now, assume s_1 collides with s_2 and s_1 is adjusted. However, if s_3 is also involved with a collision with s_1 , then the original velocity from s_1 is needed to adjust s_3 . In other words, new velocities cannot be used for adjustments at the current timestep. Instead, adjustments are applied to a *copy* of the current timestep's spheres. The originals are discarded and replaced by the copies only when all possible collisions have been checked.

2.2. Simulation Environment

Besides the collisions affecting the motion of the spheres, the simulation environment dictates their overall behaviour. Specifically, we implemented a bounding volume restricting the position of the spheres, and one of three environmental contexts (two ways to model gravity and Brownian motion) steer their movement.

The primary difference between the two gravity implementations is the direction in which they point: downwards or towards the center of the simulation space. When gravity is

directed downwards, the simulation mimics balls bouncing within a box. On the other hand, when gravity is directed towards a point, the spheres may orbit around the center, similar to a planet’s motion in a solar system.

When small particles are suspended in fluids, they present undiminishing “jiggling,” which can be attributed to Brownian Motion [17]. This phenomenon is most noticeable when the particles are a few micrometers in diameter. In our framework, this phenomenon is simulated by randomly generating a small distance and direction for each sphere at every timestep. The position of each sphere is then modified accordingly.

Figure 2 presents sample screenshots of the simulation framework under the three implemented environmental contexts outlined above. The lines near each sphere are trajectories traced by the corresponding sphere.

When all the procedures mentioned in this section are in place, Algorithm 1, presented as “Update,” is executed at every timestep. This algorithm performs the following operations: copying of spheres (lines 1 and 8), application of particle system’s rules (lines 2 and 7) and collision detection (lines 3 to 6). While the former two are $O(n)$, the latter is $O(n^2)$, which highlights the primary bottleneck of this algorithm.

Algorithm 1 Update(Spheres $S[1..n]$)

```

1:  $Temp = S$ 
2: ApplyNewton( $Temp$ )
3: for  $i = 1 \rightarrow n$  do
4:   for  $j = 1 \rightarrow n$  do
5:     if  $i \neq j$  AND Collide( $Temp[i], S[j]$ ) then
6:       Adjust( $Temp[i]$ )
7: ApplyEnv( $Temp$ )
8:  $S = Temp$ 

```

3. PARALLELIZATION SCHEMES

In this section, three primary types of parallelization schemes are presented to distribute work across P processors. The first type divides the algorithm by spheres, and the second divides by space. In the former each processor is responsible for a subset of all the spheres, while in the latter, each processor is responsible for spheres within a division of the total volume. The third type corresponds to variations on the space division scheme. In these variations, the divisions are dynamically resized throughout the course of the simulation.

3.1. Division by Spheres

Under this division scheme, processor p is responsible for $n_p \approx n/P$ spheres. However, when using Algorithm 1 to resolve collisions, each sphere needs to be checked against all other spheres for collisions. Therefore, instead of looping for i from 1 to n in line 3 of Algorithm 1, i will loop from $1 + \alpha_p$ to $\alpha_p + n_p$. In other words, processor p is responsible for updating spheres with index $1 + \alpha_p$ to $\alpha_p + n_p$ inclusive. This reduces the amount of work per processor to $O(n^2/P)$. Now, after every update, an all-to-all broadcast is required to update all processors with the updated spheres. The complete process is executed by Algorithm 2, which is presented as “DivisionBySpheres.”

Algorithm 2 DivisionBySpheres(Spheres $S[1..n], \alpha_p, n_p$)

```

1:  $Temp = S$ 
2: ApplyNewton( $Temp$ )
3: for  $i = 1 + \alpha_p \rightarrow \alpha_p + n_p$  do
4:   for  $j = 1 \rightarrow n$  do
5:     if  $i \neq j$  AND Collide( $Temp[i], S[j]$ ) then
6:       Adjust( $Temp[i]$ )
7: ApplyEnv( $Temp$ )
8:  $S = Temp$ 
9: AllToAllUpdate( $S$ )

```

Figure 3 illustrates the update flow resulting from this division scheme with three processors. In the figure, each column represents data of one processor, and each 1×3 box represents the states of all the spheres. Initially, all processors have the same states for all spheres, denoted by the top row of empty boxes. To update all spheres, each processor will account for its own set of spheres: horizontal pattern for the processor 1, vertical pattern for processor 2 and diagonal pattern for processor 3. Each processor then broadcasts its own updates to all processors in order to maintain consistency. A second update and broadcast repeat this process, which is illustrated by the two lowest rows of boxes in the figure.

3.2. Division by Space

In the second division scheme, each processor is responsible for only spheres whose center is within a specific division of the full bounding volume. These divisions are evenly spaced along each of the principal axes. For example, in a $30 \times 30 \times 30$ volume divided into $3 \times 2 \times 1$ divisions, each division would be $10 \times 15 \times 30$ in size.

A sphere in a division can only collide with another sphere in the same or neighbouring divisions. Neighbour in this context means to touch on a face, edge or corner. Spheres

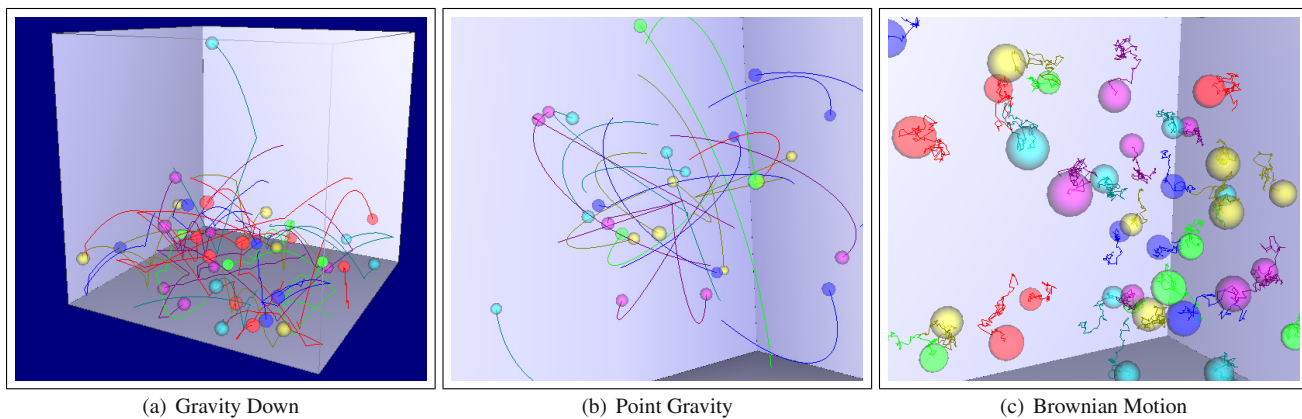


Figure 2. Implemented Environmental Contexts

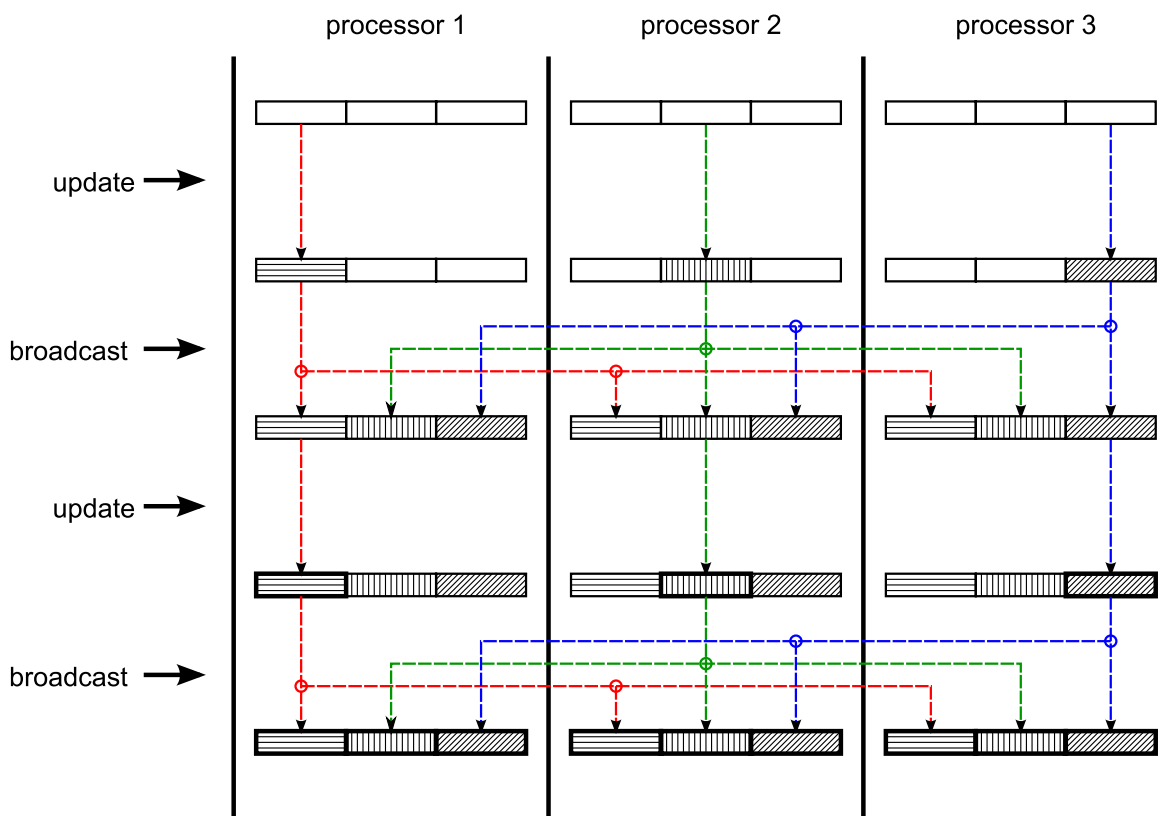


Figure 3. Update Flow in the Sphere Division Scheme

that overlap division boundaries are *shared* by the corresponding processors. That is, each processor obtains a copy of the spheres. The collision adjustments for each processor only need to account for its own spheres plus the relevant shared spheres. Using Algorithm 1, the work performed for collision detection and updates is $O(n_{p,t} \cdot (n_{p,t} + h_{p,t}))$, where $n_{p,t}$ is the number of spheres within the division of processor p at timestep t , and $h_{p,t}$ is the number of spheres obtained by processor p at timestep t from sharing. The number of spheres shared, $h_{p,t}$, is usually numerically negligible compared to $n_{p,t}$. Hence, if the spheres are evenly distributed amongst the processors, then the work reduces to $O(n^2/P^2)$. After collision adjustments are performed, each processor discards the shared spheres originating from other processors. Spheres that have moved outside of a processor’s division will need to be *transferred* to the neighbours. That is, an originating processor will lose its copy of the sphere. Spheres that have moved inside from outside will need to be similarly obtained. This process is executed by Algorithm 3, which is presented as “DivideBySpace.”

Algorithm 3 DivideBySpace(Spheres $S[1..n_{p,t}]$)

- 1: ShareSpheres(S , $Neighbours$)
 - 2: $Temp = S$
 - 3: ApplyNewton($Temp$)
 - 4: **for** $i = 1 \rightarrow n_{p,t}$ **do**
 - 5: **for** $j = 1 \rightarrow n_{p,t} + h_{p,t}$ **do**
 - 6: **if** $i \neq j$ AND Collide($Temp[i]$, $S[j]$) **then**
 - 7: Adjust($Temp[i]$)
 - 8: ApplyEnv($Temp$)
 - 9: $S = Temp$
 - 10: DiscardShared(S)
 - 11: TransferAndObtainSpheres(S , $Neighbours$)
-

When considering sharing and transferring of spheres, one division may be adjacent (neighbour) to a maximum of 26 other divisions. This is illustrated by the center partition in a $3 \times 3 \times 3$ arrangement. However, instead of communicating with a maximum of 26 processors, each processor only needs to communicate with the six neighbours that are face adjacent. We illustrate this with a two dimensional example. In Figure 4, quadrant 4 has a circle (representing a sphere) that needs to be shared with quadrant 1. However, instead of communicating directly with quadrant 1, quadrant 4 will first share this sphere with quadrant 3 (upper hollow arrow) since quadrant 3 needs the circle as well. Now, since the circle overlaps with the boundary between quadrant 3 and quadrant 1, quadrant 3 will now share it with quadrant 1 (left solid arrow). Thus, communication with a corner-adjacent quadrant can be replaced by two communications with face adjacent quadrant. Additionally, all

communication in one direction can be performed by all quadrants in parallel (2 types of arrows). The parallel horizontal communication step will allow quadrant 2 to obtain the circle as well.

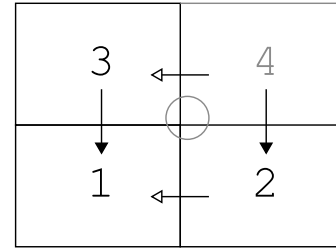


Figure 4. Corner Sharing

Generalizing the two dimensional example to three dimensions, only six parallel communication steps (two directions in each dimension) need to be performed over all processors. For each communication step, two explicit communications are required: one to state the number of spheres to share, and one to actually share the data. This adds up to a total of twelve explicit communications. Transferring of spheres can be performed in a similar fashion.

3.3. Dynamic Division by Space

The space division scheme presented in the previous subsection does not guarantee uniform load balance throughout the course of the simulation. This is because the spheres may clump in arbitrary divisions. Two variations of the space division schemes were implemented to account for this situation. Both are given the same initial divisions: spaced evenly along the axes. At every 100 timesteps, the divisions may be dynamically resized based on

- the extremal position of the spheres, or
- the spatial density of the spheres.

Within the context of Algorithm 3, these dynamic resizing are performed after discarding shared spheres (line 10) and before sphere transferring (line 11).

To perform resizing based on the extremal positions, every processor obtains the maximum and minimum coordinates of all spheres in each dimension. Processors then resize their divisions so that each division occupies an equal volume based on these extremal positions. The most outside divisions may have their sizes increased to include the neglected portions delimited by the simulation’s bounding box.

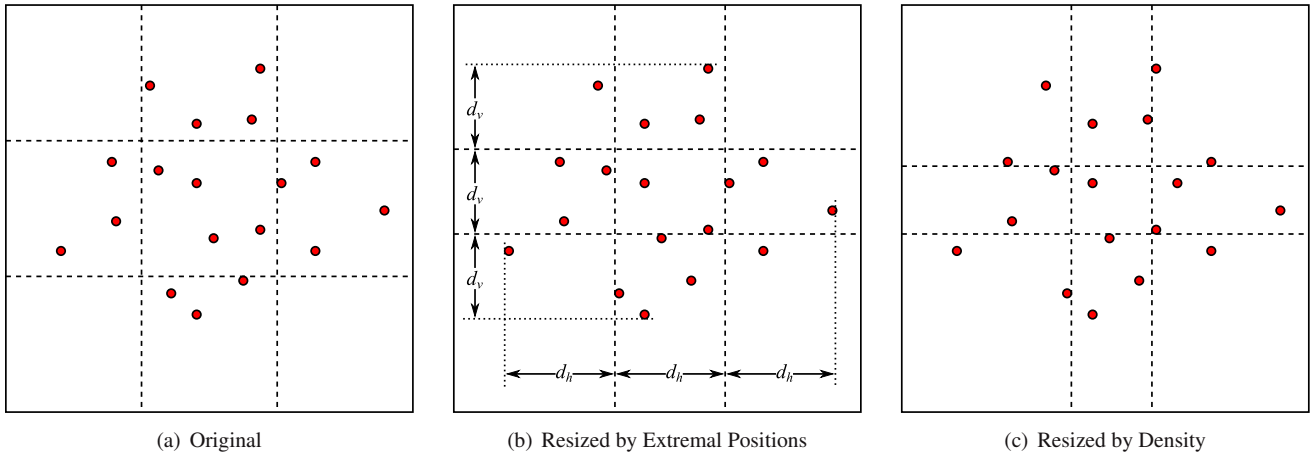


Figure 5. Resizing in Dynamic Space Division Scheme

To perform resizing based on the spatial density, every processor obtains the coordinates of all spheres. The coordinates are decoupled so that they can be sorted separately for each dimension. This sorting is performed by each processor. Based on these sortings, each processor then resizes their own division so that each row and column of divisions will be occupied by a similar number of spheres.

Two dimensional representations of these dynamic division schemes are illustrated in Figure 5. The original setting where the divisions are evenly spaced is presented in Figure 5(a). In Figure 5(b), the divisions are resized by equally distributing the distances between the minimum and maximum coordinates in the horizontal and vertical directions. These distances are represented by d_h and d_v respectively in Figure 5(b). The divisions along the edges are extended to include the full bounding box. In Figure 5(c), the divisions are resized based on density. Note that each row and column of divisions contains exactly 6 spheres.

4. RESULTS

This section presents the performance results across the various division schemes. Performance analysis was performed on a machine with eight Xeon processors at 1.40 GHz each and 3.7 Gb of RAM. The system was running Ubuntu 6.06 on an i686 architecture. Programs were implemented in C++ and compiled using g++ 4.0.3. For all cases, 1000 timesteps were used, and the bounding volume was set to $120 \times 120 \times 120$.

4.1. Division by Spheres

When dividing the work for each processor by spheres, the different environmental contexts do not drastically affect the load balancing. Therefore, the focus of our analysis

is on the difference in performance for different problem sizes. For all problem sizes, there is a near linear speedup for two processors. However, as the number of processors increases, the communication costs take over.

Figure 6 presents plots of the performance results. The total number of spheres is n . The run times for problem sizes 2000 and 3000 are divided by 4 and 9 respectively to normalize against the problem size of 1000. In the speedup plot, Figure 6(b), the linear curve is included as a reference.

For three processors, the speedup is drastically reduced due to the cost in communicating with the additional processor. This cost is large enough to offset the gains by using two processors for the smallest problem size. For the largest problem size, this communication cost is still comparable to the computation cost, allowing for the highest efficiency (speedup divided by number of processors), as seen in Figure 6(b).

The results also suggest that for 3000 spheres significant communication costs occur from 5 processors onwards. In the case of 2000 spheres, computation and communication always balance out so that there is no large jump. However, there is a minor peak in run time at five processors as shown in Figure 6(a). We remark that the larger problem sizes have a higher efficiency until communication costs dominate the run time.

4.2. Division by Space

When the work is divided by space, the environmental context has a critical effect on the performance. The grid sizes used in this analysis are presented in Table 1. Figure 7 presents plots of the performance results for a fixed problem size of 1000 without dynamic division schemes. The

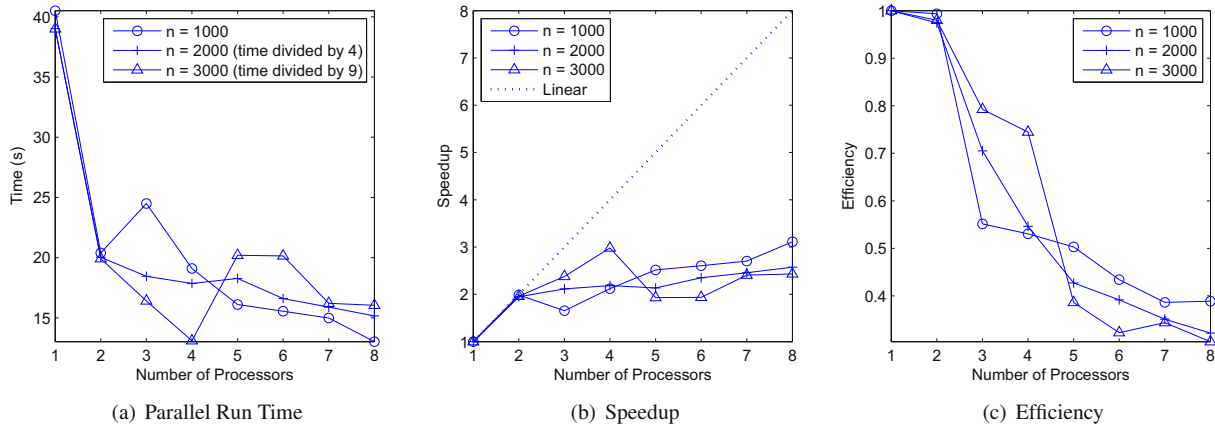
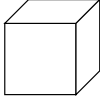
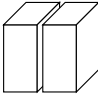
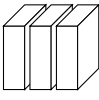
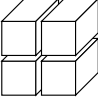
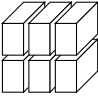
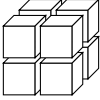


Figure 6. Performance Results: Division by Spheres

Table 1. Mapping between Number of Processors and Grid Sizes

Number of Processors	1	2	3	4	6	8
Visual Representation						
Grid Size ($x \times y \times z$)	$1 \times 1 \times 1$	$2 \times 1 \times 1$	$3 \times 1 \times 1$	$2 \times 2 \times 1$	$3 \times 2 \times 1$	$2 \times 2 \times 2$

superlinear speedup is a result of not using an optimal sequential algorithm as a comparison. If the load is balanced, the amount of total work performed is inversely proportional to the *square* of number of processors. Additionally, there is no need to perform all-to-all broadcasts in contrast to previous scheme. This yields a drastically reduced parallel run time. To account for this, the speedup plot presented in Figure 7(b) includes a quadratic curve, and the efficiency is measured by dividing the speedup by the square of the number of processors. A linear curve is also included in the speedup plot as a reference.

Brownian motion can be regarded as always optimally load balanced since the spheres are initially evenly distributed and there is no tendency for the spheres to cluster. This results in the performance being improved consistently as the number of processors is increased.

For downwards directed gravity, the spheres tend to settle to the bottom of the bounding volume. Therefore, when the volume is divided into only vertical columns, such as the case of two and three processors, the efficiency is high. However, when using four, six and eight processors, the vertical division creates an uneven load balance by having more spheres for the “bottom” processors, which results in the large decrease in performance depicted in Figure 7. Note that Brownian Motion has relatively low efficiency at

three processors. This may be attributed to a higher communication overhead since there may be spheres that jump back and forth at division boundaries.

For center directed gravity, the spheres tend to clump near the center of the bounding volume. When using three processors, most spheres will be reallocated to one processor responsible for the center column. For six processors, a similar behaviour occurs, *i.e.*, all spheres relocate to the two processors in the center column. Thus, the performance is comparable between one and three, and two and six processors. For one, two, four and eight processors, all divisions share the center of the volume equally, resulting in performance comparable to that of the optimally balanced Brownian motion.

4.3. Dynamic Division by Space

The performance results from using dynamic division with downwards directed gravity and center directed gravity are presented in Figure 8. The results for Brownian motion under non-dynamic space division, labeled as “Static BM,” are included for comparison as an optimally load balanced baseline. Efficiency is evaluated by dividing the speedup by the square of the number of processors. In the speedup plots, Figures 8(b) and 8(e), the linear and quadratic curves are included as references.

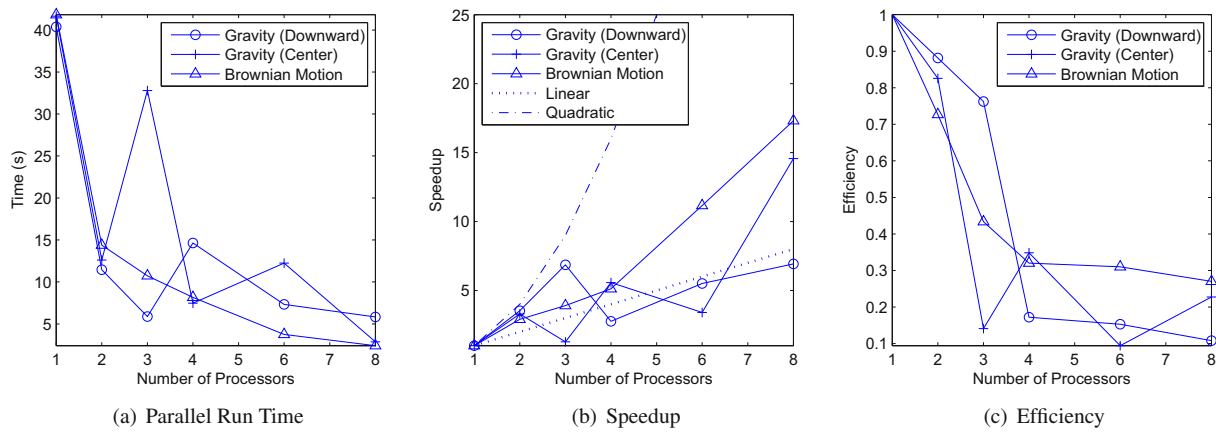


Figure 7. Performance Results: Division by Space

The performance results show that dynamic division by extremal positions does not yield considerable difference from using static divisions in terms of performance. This is due to existence of spheres that are far removed from the main clump of spheres. Therefore, this dynamic division scheme does not distribute the work evenly. However, when dividing by density, the performance drastically improves and becomes comparable to the Brownian motion baseline.

5. CONCLUSION

In this paper, we presented results from applying different parallelization schemes to reduce the computational bottleneck on a three dimensional simulation framework. To increase its scope of applications, a larger number of elements were needed, and we applied parallel techniques to improve performance accordingly.

Significant gains were obtained by incorporation of parallelization techniques via MPI. Three main schemes were investigated, namely, the distribution of work load by spheres and by space (statically and dynamically). The results of using the sphere division scheme show how the broadcast cost can become a major contributing factor when too much data is distributed. It also illustrates the importance of limiting communication overhead to maintain high efficiency in these applications. The largest gains were achieved by using the dynamic space division scheme based on sphere density. This scheme was able to consistently improve performance for all variations of the simulation environments employed in this study.

Future work will primarily involve the investigation of the efficacy of these division schemes when applied to additional simulation environments. These will include mul-

tiply scattered points of attraction/repulsion and different bounding volume shapes. The performance gains from using more processors or different parallel architectures will also be investigated.

REFERENCES

- [1] W. T. Reeves, "Particle systems—a technique for modeling a class of fuzzy objects," *ACM Trans. Graph.*, vol. 2, no. 2, pp. 91–108, 1983.
- [2] T. F. Chen and G. V. G. Baranoski, "BSim: A system for three-dimensional visualization of Brownian motion," University of Waterloo, Tech. Rep. CS-2006-41, 2006.
- [3] C. E. Wieman, W. K. Adams, and K. K. Perkins, "PhET: Simulations that enhance learning," *Science*, vol. 322, no. 5902, pp. 682–683, October 2008.
- [4] P. Jetley, F. Gioachin, C. L. Mendes, L. V. Kalé, and T. Quinn, "Massively parallel cosmological simulations with ChaNGa." in *IPDPS*. IEEE, 2008, pp. 1–12.
- [5] K. McClements, M. Dieckmann, A. Ynnerman, S. Chapman, and R. Dendy, "Surfatron and stochastic acceleration of electrons at supernova remnant shocks," *Physics Review Letters*, vol. 87, no. 25, pp. 255 002(1)–255 002(4), December 2001.
- [6] J. van der Burg, "Building an advanced particle system," *Game Developer*, pp. 44–50, March 2000.
- [7] S. Y. Belyaev and M. Plotnikov, "Object-oriented high-performance particle systems," A. I. Melker, Ed., vol. 5127, no. 1. SPIE, 2003, pp. 272–278.
- [8] F. Fleissner, P. Eberhard, C. Bischof, M. Bucker, P. Gibbon, G. R. Joubert, B. Mohr, F. P. (eds, F. Fleissner, and P. Eberhard, "Load balanced parallel simulation of particle-fluid dem-sph systems with moving boundaries," in *Proceedings of Parallel Computing: Architectures, Algorithms and Applications*, pp. 37–44.
- [9] I. F. Sbalzarini, J. H. Walther, M. Bergdorf, S. E. Hieber, E. M. Kotsalis, and P. Koumoutsakos, "PPM - a highly efficient parallel particle-mesh library for the simulation of continuum systems," *Journal of Computational Physics*, vol. 215, no. 2, pp. 566–588, July 2006.

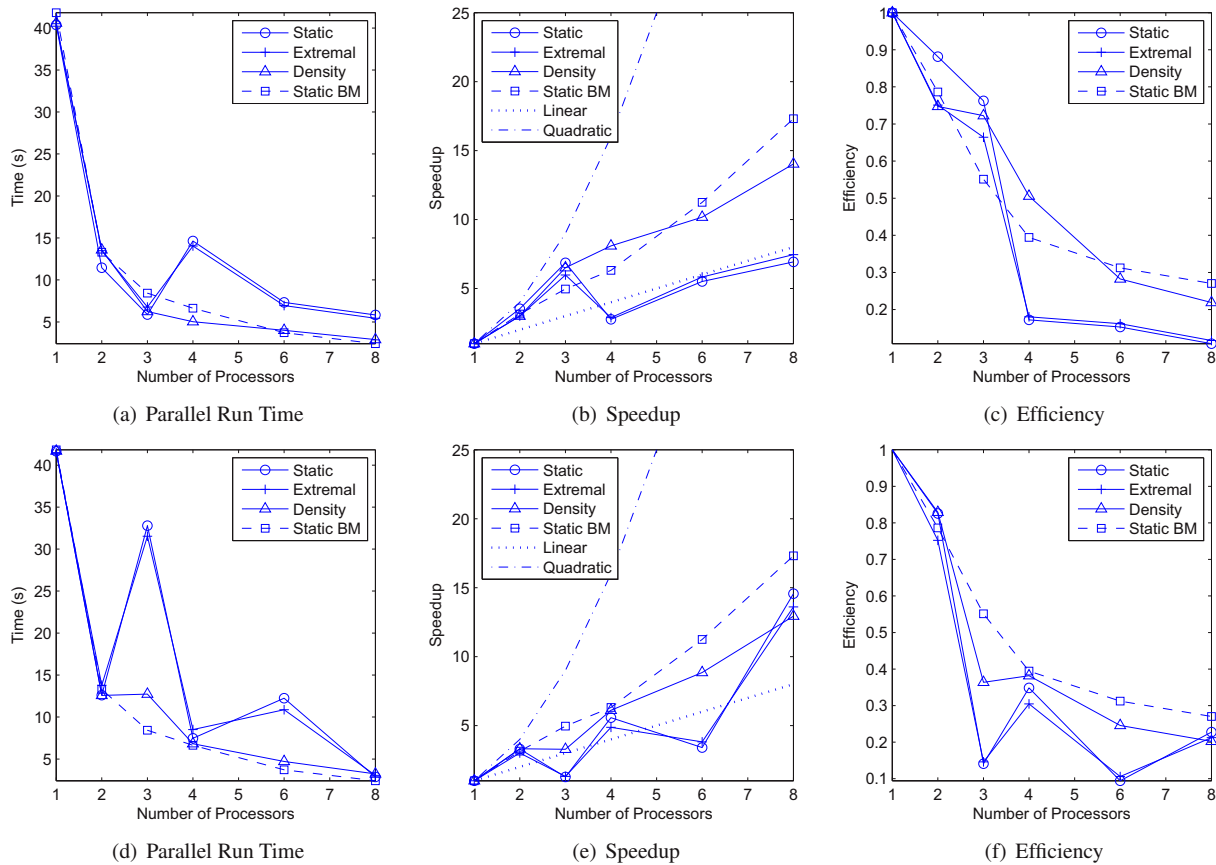


Figure 8. Performance Results: Dynamic Division by Space with Gravity Down (top row) and Gravity Center (bottom row)

- [10] M. S. Warren and J. K. Salmon, "A portable parallel particle program," *Computer Physics Communications*, vol. 87, pp. 266–290, 1995.
- [11] J. Barnes and P. Hut, "A hierarchical $O(N \log N)$ force-calculation algorithm," *Nature*, vol. 324, no. 6096, pp. 446–449, December 1986.
- [12] L. F. Greengard, *The rapid evaluation of potential fields in particle systems*. The MIT Press, 1987.
- [13] J. Board, J.A., Z. Hakura, W. Elliott, D. Gray, W. Blanke, and J. Leathrum, J.F., "Scalable implementations of multipole-accelerated algorithms for molecular dynamics," may 1994, pp. 87–94.
- [14] OpenGL, D. Shreiner, M. Woo, J. Neider, and T. Davis, *OPENGL(R) PROGRAMMING GUIDE, VERSION 2*, 5th ed. Addison-Wesley Professional, August 2005.
- [15] F. G. Waack, *STEREO PHOTOGRAPHY*. The Stereoscopic Society, 1985.
- [16] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "High-performance, portable implementation of the MPI Message Passing Interface Standard," *Parallel Computing*, vol. 22, no. 6, pp. 789–828, 1996.
- [17] M. J. Nye, *MOLECULAR REALITY*. American Elsevier Publishing Company, Inc., 1972, pp. 126–136.

6. APPENDIX

In this appendix, we present the skeleton of the MPI code used to implement the parallelization schemes discussed in this work.

Division by Spheres

```

int main(int argc, char **argv) {
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &numProc);

    //...initialize simulation data here

    //record responsibilities
    int startIdx = ...;
    int endIdx = ...;

    //set buffers for communication
    double* outbuf = ...;
    double* inbuf = ...;

    //for each frame
    for(...) {
        //do physics
        updateBalls(..., startIdx, endIdx);

        //prep outbound
        for(int i = 0; i < mySize; i++) {

```

```

    outbuf[i] = ...;
}

//communicate spheres
MPI_Allgather(outbuf,MPI_DOUBLE,
    inbuf,MPI_DOUBLE, MPI_COMM_WORLD);

MPI_Barrier(MPI_COMM_WORLD);
}
MPI_Finalize();
return 0;
}

```

Division by Space

```

int main(int argc, char **argv) {
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numProc);

    //set 3d grid based
    dims[0]=...;dims[1]=...;dims[2]=...;

    //get id of neighbours
    MPI_Cart_create
        (MPI_COMM_WORLD, num_dimen, dims, ...);
    MPI_Comm_rank(cartcomm, &rank);
    MPI_Cart_coords(cartcomm, rank, 3, coords);
    MPI_Cart_shift(cartcomm, 0, 1,
        &nbrs[LEFT], &nbrs[RIGHT]);
    MPI_Cart_shift(cartcomm, 1, 1,
        &nbrs[DOWN], &nbrs[UP]);
    MPI_Cart_shift(cartcomm, 2, 1,
        &nbrs[FRONT], &nbrs[BACK]);

    //set local bound box
    Box myBox;
    myBox.set(...);

    //...initialize simulation data here

    //for each frame
    for(...) {
        MPI_Barrier(MPI_COMM_WORLD);

        //share spheres on edges
        shareSpheresEdge(x-axis);
        shareSpheresEdge(y-axis);
        shareSpheresEdge(z-axis);

        //do physics
        updateBalls(...);
        //discard extras
        removeExtras(...);

        //for dynamic scheme
        //resize divisions every 100 frames
        if(frameNum%100 == 0) {
            if(resizeByDensity) {
                //get number of spheres
                // per processor
                MPI_Allgather(&mySize, 1, MPI_INT,
                    allSizes, 1, ...);

                //create and fill buffers with
                // local x, y and z coords
                double* myX = (double*)malloc(...);
                double* myY...

                //get all x, y and z coords
                MPI_Allgatherv(myX, mySize,

```

```

        MPI_DOUBLE, allX, allSizes, ...);
        MPI_Allgatherv(myY, ...

        //sort all coordinates
        mySort(allX);
        mySort(allY...

        //set local bounding volume
        myBox.set(...);
    }

    if(resizeByExtremal) {
        //get global max/min
        MPI_Allreduce(ballMaxLocal,
            ballMaxGlobal, ...);
        MPI_Allreduce(ballMinLocal,
            ballMinGlobal, ...);

        //set local bounding box
        myBox.set(...);
    }

    //transfer outside spheres
    shareSpheresOutside(x-axis);
    shareSpheresOutside(y-axis);
    shareSpheresOutside(z-axis);
}
MPI_Finalize();
return 0;
}

```