

Multithreaded Collision Detection in Java

Mark C. Lewis and Berna L. Massingill
Department of Computer Science
Trinity University
San Antonio, TX 78212-7200

Abstract

This paper examines the implementation of a multithreaded algorithm for doing collision detection and processing in Java. It examines details of an efficient implementation in Java for single threading, then describes the methods used to implement multithreading. The method described takes advantage of the spatial locality of collisional dynamics while efficiently dealing with the requirements of temporal ordering of collisions. We find that the multithreaded implementation in Java scales well with additional processors and is competitive with a C++ implementation using MPI for overall speed of execution. As such, the multithreaded framework will be advantageous for a number of different problems and analyses that are problematic in a distributed environment.

Keywords – Parallel Simulation, Multithreading, Collisional Dynamics, Collision Detection

1. Introduction

Collisional dynamics are important in many systems. The work presented here is largely motivated by work on the simulation of planetary rings, in particular those of the Saturnian system [1]. Currently these large scale simulations are done on clusters of machines using distributed processing and MPI. The methodology for these simulations was discussed in earlier papers [2,3]. Significant other work has been done to parallelize various types of other N-body simulations for not only collisional, but gravitational and molecular dynamic simulations as well [4,5,6,7,8,9,10,11]. Collisional systems are more akin to the molecular dynamics simulations because the forces between particles are short ranged. Unlike molecular dynamics simulations, collisions modeled as hard sphere collisions are temporally sensitive and are often modeled as discrete events [12,13,14,15,16,17]. Collisions can also be parallelized using special purpose hardware [18], for projects with a budget for such hardware.

Recent changes in the CPU market are making parallelization critical even on single machines without special hardware. The move toward multicore processors is making it essential for software of all types to be multithreaded. By 2009 it is likely that high end mainstream processors will have eight cores,

making some form of parallelization essential in order to utilize the full potential of the machine. This can be done with message passing, but there are certain operations that are more efficient or are significantly easier to code in a shared memory model. These include any analysis that requires having the full data set to process an output as well as simulations where large timesteps are optimal because of forcings other than collisions. For these reasons, we are working to develop a multithreaded version of a collisional simulation framework.

2. Simulation System

This paper focuses on simulations done with frameworks built in both C++ and Java. The frameworks were designed to handle general N-body systems with a particular emphasis on collisional systems. The C++ version of the code has been used for a number of years to do work on the dynamics of the Saturnian ring system [1]. For the purposes of this paper a simpler system than planetary rings is used. The code is modular so that different types of systems can easily be simulated without alteration to the primary framework.

The framework is built around the idea of a system which keeps track of a population of bodies, boundary conditions, a list of forces, and a list of output methods. In the C++ version of the code these are all specified as template arguments so that compilers can do static linking on all of the method invocations. The Java version takes a similar approach using generics, although with the current Java specification this only helps by allowing additional type checking at compile time, and has no effect on the compiled code.

The primary loop for the simulation does nothing more than tell the system to advance itself repeatedly until the end of the simulation is reached. Each call to advance has the following structure.

```
void advance() {  
    for(Force f:forces) f.apply(pop);  
    pop.endStep();  
    boundaryConditions.apply(pop);  
    for(Output o:outputs) o.doOutput(pop);  
}
```

The exact details vary slightly by language. For example, in C++ the code does not actually keep a list of forces and outputs. Instead a template metastructure tree is used to keep track of those. The conceptual idea is still the same.

The population provides methods for querying and altering the state of the various bodies in it. Each force makes alterations to the bodies in the population. After the forces have been applied the `endStep` method cleans up the particles so they are all in a format appropriate for applying the boundary conditions and doing an output. Typically this advances each of the bodies to the proper state for the end of the time step.

The ability to specify different boundary conditions for the system is essential for the simulation of planetary rings, as the boundary conditions can vary drastically depending on the exact nature of the simulation. For the simulations in this paper we will only use a boundary condition of a simple flattened cube with sheared periodic behavior. So when a particle passes out of bounds on one face it is simply wrapped to the other side. If it passes beyond the bounds in the y direction an x -velocity component is added or subtracted. This imposed shear adds energy to the system to counter the dissipative collisions and helps drive the system to an equilibrium condition. This type of system is commonly found in granular flows, as it has practical applications in real fluids which have zero slip at boundaries with solid surfaces.

The population we use for this work has spherical particles that travel along straight lines between collisions. The particle collisions are dissipative with a velocity dependent coefficient of restitution appropriate for chunks of ice [19]. This is an artifact of the planetary rings origin of the code.

We do not include any output methods in the simulations, as they are inherently nonparallel and would only add overhead to the methods that we investigate.

The only force that we include in the simulations is a collisional force, and while the entire code was parallelized for this work, the parallelization of the collisional forcing is the primary consideration. First of all, the code spends more than 90% of its runtime in collision processing, so it is the most critical aspect to parallelize. It is also the most interesting piece to parallelize because collisions have an inherent serial nature. Collisions need to happen in a particular order; that is, they are temporally sensitive. Other forces, like gravity, have long range impacts, but timing is generally not a significant issue. With collisions, a collision at an earlier time can alter the path of a particle and prevent it from entering into another collision later on. In general, one cannot even tell what collisions a collection of particles will undergo during a period of time without actually processing them out. To see why this is,

assume that you find that given current trajectories, a particle A would collide with particles B, C, and D during a period of time. It is tempting to say that if the collision with B would happen first then it is the only one that will happen. However, it is possible that B will be struck by another particle before it gets to A, allowing A to strike C. The same could be said for C being struck, allowing the last of the originally identified collisions to be the one that actually transpires.

The collision forcing in the code deals with this in the following manner. First, a spatial data structure is set up to speed the process of finding nearby particles. For this work we use a regular grid with the cell size chosen so that particles cannot cross more than one grid cell during a time step. Second, we perform a search for all particles using this spatial structure to find all the collisions they would be involved in during the time step were they to maintain their current trajectory. These potential collision pairs are placed on a queue by the time at which they would occur. Third, we pull pairs off the queue in order. Each time a pair is pulled off, we update the particles for the collision, remove all other potential collisions involving either particle from the queue, and find the collisions these particles could undergo given their new trajectories and add those onto the queue. A more complete description of this process can be found in Lewis and Stewart [20,21,22].

3. Conversion to Java

The first step in this work was to convert the simulation framework from C++ to Java. The reason for making this change was primarily to simplify the task of multithreading, though an added benefit would be easier integration with the Java based analysis tool used in this work [23]. Java has always had simpler mechanisms for dealing with multiple threads than the POSIX threads libraries and it works across multiple platforms. With the addition of the `java.util.concurrent` package, support for the types of tasks that are needed for this project is improved even further.

An alternative to Java threading would have been to use OpenMP with C/C++. One of the goals with the simulation frameworks and analysis tools that we develop is to keep all of the components free. There are efforts currently under way to add OpenMP support to the GNU compilers, but as of the time the work described here was performed they were not part of the stable release. Therefore, using OpenMP would mean that anyone wanting to use the framework would need to buy a compiler that supports OpenMP. Java, in contrast, has the full JDK with all the threading tools available as a free download.

Some might argue that Java does not compete well in terms of performance with C++ so large scale

simulation codes should not be ported to Java. Testing that hypothesis was another reason for the work described here. We had previously written a small N-body simulator for C++ and Java using a basic T+V integrator and Newtonian gravity to do speed comparisons. Tests on both AMD Opteron and Intel Pentium 4 machines showed Java 1.4.2 and Java 5.0 to be effectively tied with C++ in performance on this small code. That code has now been integrated as part of the Great Programming Language Shootout [24] under the title n-body. On the Intel platform, Java is the top language for that benchmark. This simple test gave us reason to believe that the newer JVMs could perform just as well as C++ code for numerically intensive tasks. The performance of Java virtual machines has also improved with every release to date, so it is likely that even without modifying the code the benchmarks shown here could improve relative to C++ in the future as newer versions of Java are released.

Before working on the multithreaded version of the simulation code a single threaded version of the code was created and its performance was compared to that of the original C++ code. Table 1 shows comparative run times for simulations varying the filling factor and particle sizes¹. The benchmarks that we show here were performed on two different platforms: a dual Opteron 246 (2 GHz) running 64-bit SuSE Linux and a quad Xeon 3.16 GHz running 32-bit Fedora Core 3. Using these two platforms allows us to compare the scaling of the methods that we are using in a broader way so that it is less likely that we are seeing performance gains based on special attributes of our hardware. Part of the motivation for this work is the widespread availability of processors with multiple cores. Unfortunately, we do not yet have access to any of these machines to perform benchmarks on.

The C++ code was converted to Java with language appropriate modifications. The two codes were run on identical input sets to ensure that they produced the same results, and various bugs found by doing this were fixed. Once the two codes agreed in their output, we found that the C++ version had a speed advantage on the order of a factor of 2 or more. The single function that is called most frequently in the code and where most of the time was being spent is the function in the population that takes two particles and determines if and when they would collide. This method was written in a standard object-oriented manner in both languages where structures are looked up in arrays and values are retrieved from those structures. In C++ there is no need to work too hard to optimize this routine because the use

¹ Filling factor is a measure of what fraction of the total simulation space is filled by particles. It is calculated as the sum of the volumes of the particles divided by the volume of the simulation space.

of templates allows the compiler to do significant static analysis, inline many expressions, and identify common subexpressions. That process is much less efficient in Java because all the bindings are dynamic. To help the compiler optimize, we simply introduced our own temporary variables and used those so that it is clear that method calls do not have side effects altering their values. Once this change was made the Java code was able to perform at speeds within 10% of the C++ code. This was felt to be sufficient for continuing on with the process of parallelizing the code through multithreading.

Table 1 shows the benchmarking results for the serial versions of the two codes on our platforms with variations in the filling factors and particle sizes. These results show that the Java version of the code competes nicely with the C++ version of the code, so switching languages in order to get easier threading capabilities does not introduce a significant handicap. This is especially true when the particles fill less of the space, as is common in planetary ring simulations.

<i>Filling Factor</i>	<i>Particle Radius</i>	<i>Platform</i>	<i>C++</i>	<i>Java</i>
0.01	0.003	Quad Xeon	166	176
0.01	0.003	Dual Opteron	89	99
0.1	0.006	Quad Xeon	262	314
0.1	0.006	Dual Opteron	165	197

Table 1: This table shows timing results for the serial version of the code using two different simulation systems on two different hardware platforms with both the C++ and Java platforms. The times are given in seconds and have an error of 5% over multiple runs.

It is also interesting to note the difference between the Opteron and Xeon platforms. The fact that the Opteron performs significantly better despite operating at only 66% the clock speed of the Xeon machine should not be all that surprising to people who have followed the CPU market. What makes the result a bit more interesting is that the Opteron based machine is roughly two years older than the Xeon based machine, which was purchased in Fall 2005.

4. Parallelization Scheme

After getting the Java version of the code to produce the same results as the C++ version and to perform similarly, the primary task became parallelizing the code with multithreading. Another paper by the authors deals with this topic in more detail, but it warrants some discussion here as well [25].

Before discussing the methods that were applied in various parts of the code, it is worth noting the scheme

that was used to facilitate the parallelization and to provide flexibility in how different aspects of the code were performed. A singleton class called `ThreadHandler` was created along with two subclasses. The `ThreadHandler` class includes abstract methods for basic operations such as adding a new task to be processed and waiting for all current tasks to finish. The two subclasses implement these methods in different ways. One uses normal Java threads and creates new threads for each new task. The other uses a `java.util.concurrent.ExecutorService`, specifically a fixed thread pool. Each is told when the program is started how many threads to allow tasks to run in concurrently. The `ThreadHandler` class also includes methods that use the basic methods to implement for loops that distribute the work between threads using three strategies: round robin, consecutive chunks, and dynamic allocation. The first two are fixed schemes that vary simply in how the work is broken up. Consecutive chunks were typically preferred as they minimize the frequency of two threads accessing the same cache line. The dynamic allocation scheme makes a task for each iteration through the loop. This scheme works well for long tasks where load balancing can be an issue. The thread pool based implementation using the `ExecutorService` was the clear performance winner in our tests, undoubtedly because it lacked the overhead of frequently creating new threads. All the results shown in this paper use that implementation.

Going back to the description of the code in section 2, there are a number of different pieces that need to be parallelized. Some of the methods include loops over all the particles that do certain basic processing. The `endStep` method is an example of this. These loops were easy to parallelize and were simply handed off to the `ThreadHandler` by putting the body of the loop inside an anonymous inner class.

The more interesting methods to parallelize were part of the `applyForce` method for the collision forcing. Without multithreading, profiling showed that the code spent the vast majority of its time in this method and those that it called. The method can be nicely divided into three pieces. First, the spatial data structure is constructed and the code determines which particles rest in each bin. Second, the method `findInitial` runs through the grid and finds all potential collisions between all particles given their initial trajectories. Third, a loop runs through these potential collisions, pulling elements off the queue of collisions, processing them, updating the queue, and searching for new collisions involving the particles that underwent each collision. Each of these pieces was parallelized differently.

The building of the grid contains a few loops that can be fairly easily split up. The only challenging one is the last one which loops over the particles, calculates which grid cell each falls into, and adds that particle to

the list for the cell. The body of the loop is rather short, so creating several grids and merging them together turned out to be extremely inefficient. Instead, the lines that access the grid were synchronized on full rows. The large number of rows in the grid allowed this to happen efficiently, as the odds of two threads trying to access the same row at a given time is small.

The `findInitial` method was also rather straightforward to parallelize. It runs through the grid doing checks on all particles in the grid relative to the adjacent grid cells. Two alterations were needed for this to be multithreaded: First, access to the queue data structure on which the potential collisions are placed was synchronized so that different threads could not update the queue simultaneously and possibly damage it. Second, the outer loop over the grid was made to use the thread pool. In order to produce good load balance in the general case, a dynamically assigned loop was used, even though for the simulations presented here the more standard loop styles would work just as well. Because each iteration through the loop does a significant amount of work, dynamically creating tasks for each iteration does not produce a significant performance penalty and will be helpful for more complex simulations where the particles and collisions are not uniformly distributed through the simulation region.

The most interesting part of the collision handling to parallelize was the last section where the collisions are actually processed. As was mentioned earlier, collisions are temporally sensitive, so one cannot arbitrarily reorder collisions. However, collisions are also spatially localized, so what happens in a collision can only impact other events within a certain region of space in a given period of time. As a result, collisions can be processed in parallel if they are far enough apart. We considered keeping a list of the collisions being processed so that when a new collision was pulled off the queue it could be checked against those currently active. This method was never implemented for fear of

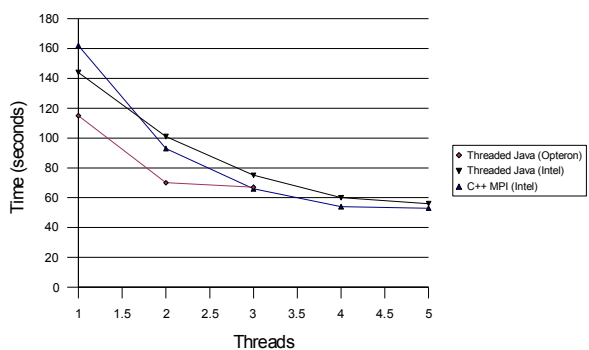


Figure 1: This plot shows the timing results for a simulation with a filling factor of 0.01 and particles of radius 0.003.

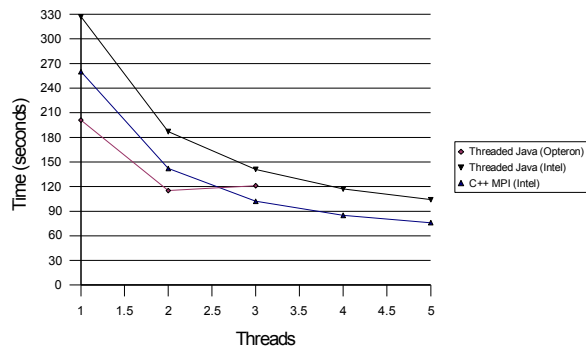


Figure 2: Timing results for a simulation with a filling factor of 0.1 and particle radii of 0.006.

the overhead it would add and the general complexity of the method. Instead, the spatial grid that is already used for collisions was augmented to include boolean values for each cell indicating that a particle in that cell was currently undergoing a collision. The cell size is already chosen to be large enough so that any two collisions can be safely done in parallel as long as they are not in adjacent cells.

Using the boolean markers on the grid and the synchronized queue, it is fairly straightforward to have the method that pulls the next collision from the queue check if that collision is safe (i.e., can be processed at the same time as others currently being processed). If it is, it simply returns it. If not, then the next collisions on the queue are checked sequentially until a safe one is found. If no safe collision is found, the method waits until another collision stops processing, as completion of processing of one collision may change which collisions are safe and also add more collisions to the queue. When the queue is empty and no threads are currently processing, then the method returns null to indicate that no more collisions remain.

At first this scheme was implemented with the processing of each collision being a separate task. The overhead of scheduling those tasks was significant and slowed the simulation down. Instead, it works better to have one task for each thread in the pool, with each task performing a loop that pulls collisions off the queue. This works because access to the queue was completely synchronized earlier.

As was mentioned earlier, the C++ version of the code already included support for distributed parallelization with MPI [2,3]. This parallelization breaks the simulation region up spatially and reduces communication requirements by giving each processor extra particles that overlap onto the regions controlled by the adjacent processors. This method incurs overhead from the communication between machines and from the extra particles that are simulated. In the next section we will see how this compares to the synchronization and thread handling overhead incurred

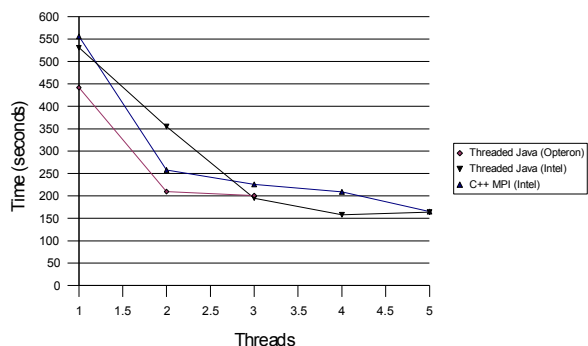


Figure 3: This figure shows timing results for a simulation identical to that in figure 1, but with the time step increased by a factor of 3.

by the multithreaded code.

5. Results

As with the single threaded version, a number of tests were performed to evaluate the performance of the multithreaded code and how well it scaled as the number of threads in the thread pool was increased. For this we looked at two systems. One used a filling factor of 0.01 with smaller particles (0.003 units in radius). The second used particles twice as large with a filling factor of 0.1. The first required roughly 70,700 particles while the second used roughly 88,400 particles. These are both small compared to simulations in practice, but they are large enough and were run long enough (30 time units) to accurately measure how the code scales.

The system with the lower filling factor is more akin to simulations of Saturn's A ring. In this system, the collisions are fairly rare so a lot of time is spent looking for collisions that do not happen and only a small fraction of the time is spent actually processing the collisions. The system with the larger filling factor is closer to the nature of most granular flow simulations; in it a lot more time is spent actually performing the collisions and searching for subsequent collisions.

The timing results for these simulations are shown in Figures 1 and 2. There is quite a bit of information in these plots, as each shows performance values for Java on two platforms and C++ on one using either different numbers of threads, or in the case of the C++/MPI code, different numbers of processes. On both hardware platforms, the number of threads/processes was taken up to one more than the actual number of processors.

What one can see in these figures is that up through the number of processors on the machine, both the threaded version of the code and the MPI version of the code scale reasonably well. Unfortunately, neither is linear in the number of processors for the simulations that we looked at. The system in Figure 2, with the higher filling factor, spends significantly more time

processing collisions than the system in Figure 1. As both contain about the same number of particles, they spend similar amounts of time searching for the initial collisions. The fact that both scale similarly implies that the threaded code is able to distribute the initial finding and the processing work with equal efficiency. That is a significant point to note as the processing of collisions is a significantly more complex operation and has to do some overhead work to prevent collisions that might conflict from happening in parallel.

A simple test to see if the threaded version truly does scale well in situations where the MPI version does not can be performed by simply increasing the time step used in the simulation. Figures 1 and 2 showed simulations where a time step of 0.1 was used. The simulations in Figure 1 were repeated with a time step of 0.3 to produce the results shown in Figure 3. We see that with the larger time step the MPI version scales well to 2 processors, then flattens out significantly. The threaded versions scale better all the way through 4 processors allowing the Java code to outperform the C++ code when 3 or 4 processors are used.

The ring simulations that the framework was originally designed to perform use a different overall system than what was tested here, with added complications due to the fact that particles are bound on orbits. However, planetary rings are extremely thin (only a few particle diameters thick) and the systems we simulate typically have a rather low optical depth. That makes them more similar to the systems shown in Figures 1 and 3 than to Figure 2. Those systems are also the ones where the Java code performs most similarly to the C++ code. Therefore, this multithreaded framework is likely to see useful applications in the next few years for certain classes of problems.

6. Conclusions and Future Work

This work has presented a general simulation framework that has been used for planetary ring simulations and discussed its conversion to Java and the addition of multithreading. A significant conclusion for all simulation practitioners is that the Java language is suitable for high performance applications, even those beyond small scale number crunching. However, some care must be taken to minimize the impact of dynamic binding in Java which makes interprocedural analysis and optimization for the removal of common subexpressions very difficult.

With regard to parallelization, it is also shown that multithreading in Java can provide good performance that scales reasonably well with the number of processors. Indeed, for some systems the lack of duplicated work allows the multithreaded Java version to outperform a C++ version using MPI. This will become more useful as the number of cores on chips

continues to increase or when output methods require the particles be grouped in sorted order or something else that requires having all the particles in a shared memory. For example, output aimed at examining the streamlines of ring particles needs to have the particles sorted by radial distance from the planet.

There is another area that is not examined in this paper, but which could benefit from the threaded implementation. Currently, the MPI version of the code does not implement load balancing, and even when that is added there will be a delay in how quickly it can adapt to the distribution of particles. The threaded version of the code is completely dynamic in how the load is distributed for both finding collisions initially and processing them across a time step. Future investigation will look at how well the threaded version performs when the particle distributions are significantly non-uniform.

Related to load balancing is the use of more dynamic data structures for determining nearby particles that might potentially be involved in collisions. Work has been done to explore the efficiency of various trees for this purpose [26,27]. Tree data structures are asymptotically slower than grids for the simplest of operations, but when the nature of the particle distribution moves away from being uniform, their dynamic nature can lead to overall performance benefits. The multithreading methods discussed in this paper are specifically tailored for grid data structures, so further work will be required to determine efficient multithreading techniques when other data structures are used. A simple example of when a tree can be beneficial is a simulation of an embedded moonlet in a ring. To be safe, the grid must always use a conservative cell size given by twice the maximum particle radius plus a multiple of the distance a particle's random velocity will carry it over a time step. A single moonlet can force the grid size to be significantly larger than is required by the velocity dispersion and size of most of the particles. A tree method can keep track of maximum particle size and velocity dispersion below any given node to provide much more selective searching in this case, and that results in a faster simulation despite the $O(\log n)$ access time for any single particle in the tree.

Given the current plans of chip manufacturers, all efforts to add more efficient thread support to simulations and other high performance applications are likely to reap significant benefits in the coming years. This is only a small step to help prepare for the computing systems that are inevitably coming down the pipeline.

Bibliography

[1] Lewis, M. C. and Stewart, G. R., Expectations for

- Cassini observations of ring material with nearby moons. *Icarus*, 2005, 178, 124-143.
- [2] Lewis, M. C. and Wing, N. 2002. A distributed methodology for hard sphere collisional simulations, Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, NV, USA, 404-409.
- [3] Lewis, M. C. and Wing, N. 2003. Analysis of a distributed methodology for hard sphere collisional simulations, Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, NV, USA, 1208-1214.
- [4] L. Verlet. "Computer experiments on classical fluids: I. Thermodynamical properties of Lennard-Jones molecules". *Phys. Rev.*, 159, 98-103, 1967.
- [5] S. Plimpton. "Fast Parallel Algorithms for Short-Range Molecular Dynamics". *Journal of Computational Physics*, 117, 1-19, 1995.
- [6] Fujimoto, R. M. "Parallel discrete event simulation", *Communications of the ACM*, 33, 10, 30-53, 1990.
- [7] Jacobs, P. H., Lang, N. A., and Verbraeck, A. "D-SOL: A distributed Java based discrete event simulation architecture", Proceedings of the 34th conference on Winter Simulation, San Diego, CA, vol 1, 793-800, 2002.
- [8] Mascarenhas, E., Knop, F., Rego, V. "ParaSol: A multithreaded systems for parallel simulation based on mobile threads", Proceedings of the 1995 Winter Simulation Conference, 690-697, 1995.
- [9] Wisdom, J. and Tremaine, S., "Local Simulations of Planetary Rings". *Astronomical Journal*, 1988, 95, 925-940.
- [10] R.W. Hockney, S.P. Goel, and J.W. Eastwood, "Quiet High Resolution Computer Models of a Plasma", *Journal of Computational Physics*, 14, 148, (1974).
- [11] G. A. Kohring. "Dynamical Simulations of Granular Flows on Multi-Processor Computers". In *Proceedings of the European Community on Computational Methods in Applied Science (ECCOMAS '96)*, Paris, France, September 9-13, 1996.
- [12] B. J. Alder and T. E. Wainwright. "Studies in molecular dynamics. I. General method". *The Journal of Chemical Physics*, 31, 2, 459-466, 1959.
- [13] J. J. Erpenbeck and W. W. Wood. "Molecular dynamics techniques for hard-core systems". In J. B. Berne, editor, *Statistical mechanics. Part B: Time-dependent processes*, pages 1-40. Plenum, 1977.
- [14] A. T. Krantz. "Analysis of an Efficient Algorithm for the Hard-Sphere Problem". *Transactions in Modeling and Computer Simulation*, 1996.
- [15] B. D. Lubachevsky. "Simulating billiards: Serially and in parallel". *International Journal on Computer Simulations*, pages 373-411, 1992.
- [16] M. Marin. "Billiards and Related Systems on the Bulk-Synchronous Parallel Model". In *Proceedings of 11th Workshop on Parallel and Distributed Simulation (PADS'97)*, pages 164-171, Lockenhaus, Austria, June 10-13, 1997.
- [17] P. McKenzie and C. Tropper. "Parallel simulation of billiard balls using shared variables", In *10th Workshop on Parallel and Distributed Simulation (PADS '96)*, Philadelphia, PA, May 1996.
- [18] Makino, J., Taiji, M., Ebisuzaki, T., and Sugimoto, D. "GRAPE-4: A massively parallel special-purpose computer for collisional N-body simulations", *The Astrophysical Journal*, 480: 432-446, 1997.
- [19] Bridges, F., Hatzes, A., & Lin, D., *Nature*, 1984, 309,333.
- [20] Lewis, M. C. and Stewart, G. R., A new methodology for granular flow simulations of planetary rings – coordinates and boundary conditions, *Proceedings of the IASTED International Conference, Modeling and Simulation*, Marina Del Rey, CA, USA, 2002, 292-297.
- [21] Lewis, M. C. and Stewart, G. R., A new methodology for granular flow simulations of planetary rings – collision handling, *Proceedings of the IASTED International Conference, Modeling and Simulation*, Palm Springs, CA, USA, 2003, 292-297.
- [22] Lewis, M. C. and Stewart, G. R., Modifications to a methodology for simulations of perturbed planetary rings. *Proceedings of the IASTED International Conference, Modeling and Simulation*, Marina Del Rey, CA, USA, 2004, 184-190.
- [23] Lewis, M. C., SwiftVis download and documentation, <http://www.cs.trinity.edu/~mlewis/SwiftVis/>
- [24] Gouy, I., Great Programming Language Shootout, <http://shootout.alieth.debian.org/>
- [25] Massingill, B. L. and Lewis, M. C., Parallelizing a Collisional Simulation Framework with PLPP (Pattern Language for Parallel Programming), accepted to PDPTA 2006.
- [26] Lara, D. "Spatial Data Structures for Efficient Collision Detection", Honors Thesis, Trinity University, 2005.
- [27] Lewis, M. C. Efficient collision detection optimized for long timesteps. *Proceedings of the IASTED International Conference, Modeling and Simulation*, Cancun, Mexico, 2005.