Particle Simulation on Hybrid Clusters

Mark Randles
CS 490 Independent Project
Dr. Hassan Rajaei
2004-08-08

Introduction.

The goal of this project was to extend the research done during the Spring 2004 CS417 Introduction to Parallel Programming class. During that class, the author's topic of study was hybrid clusters, namely using both MPI and OpenMP to facilitate parallel algorithms using multi-level parallelization. This independent project was to extend what knowledge the author had gained in that study and to develop new algorithms to use and exploit the features of multi-level design.

The problem used as a test case was thus: to simulate the motion of a set number of massive particles using motion rules defined by simple Newtonian dynamics. During 417 it became apparent that the original workload created by the problem as implemented, was too small, and therefore it was difficult to get meaningful numbers unless a large number of particles was used. For this project I planed to extend the physics model used previously, by implementing more complicated dynamics that would increase the total computation workload for each particle, and therefore allow for more of a noticeable speedup based on algorithmic changes.

There were a number of specific phases that this project went though. However despite careful planning, the project did not complete as expected, but did provide a source of good thought and valuable information. The phases were as follows:

- Phase 1: Planning and project layout
- Phase 2: Initial coding
- Phase 3: Algorithmic development and implementation
- Phase 4: Final results and conclusion

For the most part the phases flowed into one another, and at no one time was it entirely possible to say that the project was in one specific phase. However for purposes of this paper, we will assume that each phase occurred in order and built upon the previous, unless otherwise noted.

PHASE 1 – Planning and Project layout

A full description of the problem can be found in Appendix 1.

This phase was probably the most important, and longest phase of the project. In may ways, the planning that occurred was the biggest success, as the ideas generated will provide fodder for future continuations of this project and possible for spawning into side projects.

One of the main goals of this phase was to define the problem to be used as the test case for the project. Since I was going to extend the previous problem, there was an initial research base for the formulas needed in the computations. However, after some further research, it became readily clear that my meager physics skills were inadequate to the grand problem conceived. The biggest problem came from trying to figure out how to implement the use of a force vector to define the movement of the object over discrete time steps. The solution was the concept of work. However this raised the issue of possibly needing to integrate complex force equation over some given dt.

For the most part, this was a moot point in my thinking, as I was more concerned about other aspects of the project, such as the algorithms needed to partition the dataset, and the development of the problem fell by the wayside. In hindsight this was a mistake, as in the beginning this should have been the first defined, and a test case worked out, by hand, as an example for eventual implementation. Lesson learned.

The most interesting aspect of the planning phase, were the spurious ideas generated. The original plan, as conceived, was to implement the breaking of spheres from a time t_0 where the sphere was whole and solid, till a time t such that all the particles had collided with the ground and come to a standstill in our frame of reference. This would have generated data that would represent a very nice approximation of the mechanical explosion of the spheres. Now, it's important to note that this would be a mechanical explosion, not an actual physical explosion, which would involve computational fluid dynamics and other such devices to render the physics involved in a real explosion.

However this plan would require the program to go through a number of steps to actual break apart the sphere. A simple static break was needed to form each particle of the sphere that the program would track. However the desired result of the problem was to create the most dynamic simulation of the explosion as possible, and to further that goal a dynamic breaking of the sphere was desired. After some thought, it was decided that this aspect of the problem wasn't inside the scope of the project, and therefore superfluous. However before that decision, some thought was put into the problem, and a few ideas were generated towards breaking the spheres dynamically.

The first and most likely best idea was to use recursion to break apart the sphere into roughly triangular pieces using random vertices picked during each recursion step. This would be a generally efficient and for the most part an effective way to facilitate the random breaking of a

sphere. However, since we would approximate each edge of a piece with straight lines, the actual assembled sphere wouldn't be very sphere like. This wouldn't be important unless one was to use this algorithm to actually render the sphere.

Still not satisfied with the original algorithm, I came up with a second method of doing the random breaking of the sphere. This approach offered a much more computer science approach to the problem, but would be more computationally intensive. First a set of points would be scattered across the surface of the sphere. The points would be abstracted as a graph of nodes. The actual algorithm would transverse the graph and connect a node to it's three closest neighbors, dividing up the surface of the sphere into hopefully, roughly enclosed regions.

Each method has drawbacks and advantages. The first algorithm presented would be less expensive in computation time, because it only requires a single random number generation in each recursion, along with the connecting of 1-3 nodes, whereas the second algorithm requires each node to transverse the graph and look for the three closest points, which is on the order of O(n^3) at least in the worst case. If a self-organizing data structure was used, the complexity of the algorithm could be reduced, but i would venture that high end of the complexity spectrum is around O(n) or O(n log n). Another advantage that the first algorithm presented has is it's much easier to parallelize, as the recursive calls could be handed off to separate threads of execution, if a shared memory system was used. Theoretically, the speedup for this problem would be negligible on a MPI cluster, as the communication overhead would most likely kill any speedup.

As the planning went on it became apparent that the real meat of this project would be the data partitioning algorithim. The slowest part of the algorithm used in the 417 version of this project, was the inter-node communication. The method of partitioning used in the original project was a simple vector division, where each node got X number of particles where X = [total numb particles] / [number of nodes]. This was effective as the particles didn't interact, so each node only needed to crunch the numbers for those particles that were allocated to it and then report back the results after it was done. However for the new problem it would be possible for the particles to interact, and we would be using a test case where such interactions would be guaranteed, so the type of partitioning used in the original project wouldn't be the best choice.

Since the particles in the new problem would interact, each node

would need to know the previous state of the other particles to see if such an interaction would occur. This presents a enormous problem in terms of speedup as now for each time step we require a global sync to maintain the particle position lists on each node. The simplest method for implementing this is to require each process to block after it's computation is done on the part of the list it has, then report the results back to a root or parent node, who in turn, will broadcast the list back to all the children nodes. Your basic scatter and gather with a global sync.

However, this might not be the most efficient way to divide our problem. The scatter-gather approach ignores some of the special features of the actual problem, namely it ignores the fact that we are exploding the spheres along 3 axises and the individual particles may move away from a “zone of interaction”, that would exist between the spheres. If we take into account the actual movement of the particles we can divide our initial list of particles into two separate lists, one of interacting particles and one of non-interacting particles.

However to divide our list we need to pre-compute a approximate landing point for each particle. To do this we do not need to apply the full suite of physics that will happen in the actual computation loop, but rather a simplified set of equations, namely the standard motion equation, dx = ½ * a * t^2 + v0 * t + x0, where dx is the change in position along our axis x, a is the acceleration, v0 is the initial velocity of the particle, x0 is the initial position of the particle, and t is an arbitrary time. This formula for motion applies for each axis of motion so we come up with a set of equations:

```
x = ½ * a * t^2 + vx0 * t + x0
y = ½ * a * t^2 + vy0 * t + y0
z = ½ * a * t^2 + vz0 * t + z0
```

Where x,y,z are the three axises. For this experiment the x-z plane is the ground, and altitude of the particle is measured along the y axis. So by solving the y axis equation for t using a y=0, we can find the total time in flight for the particle, and then by substituting that time into our x and z equations we can solve for the approximate landing coordinates of that particular particle.

After we've calculated the approximate landing position we can then find any line segments that intersect, were a single line segment is defined by the initial position of the sphere (which exists as a point mass) to the approximate final resting place that was just calculated. Any lines that intersect are interacting particles and should be placed on the list accordingly. If, after all the lines have been compared, it

intersects with no lines, then it should be treated as a non interacting particle.

If a particle does not interact then we can process it much like a particle in the original problem, by dividing up the array into chunks based on the number of processors dedicated to the task of processing the non-interacting list. Each node can work independently of each other and results can be received quickly and efficiently. To divide up the interacting particles we need to use a more communication orientated approach. One method of partitioning the interacting dataset would be to assign each node a 2d area of the x-z plane and it tracks and moves what particles are currently contained in it. However his approach requires each node to communicate with it's neighbors to exchange any particles that may cross the boundaries. Also each node would need to guarantee that each particle could be rolled back to a previous state, providing a facility for error correction across the boundaries. However this approach require lots of network traffic, since after each time step each processor needs to sync to the rest of the nodes and communicate with it's neighbors.

A second approach to partitioning the interacting list would be to divide the list into only a few large chunks, probably no more then 2-4 pieces. Each piece would contain a set of particles that will interact with one another, and in nearly all probability will not interact with any of the other particles. Each piece would progress as normal, without the sync after each time step, and after it's finished computing each piece is combined with the others. Once the list is reassembled, the intersection algorithm that initially divided the particle set into interacting and non-interacting will process the list based on the final position. Any particles that would interact are rolled back, along with each particle that it influence, and added to a new list. After the list is processed, the new list is subjected to the same process as the original interacting list, till no more interactions occur.

This however could be a very time consuming algorithm. If there are many interactions between particles, then the processing loop might have to be repeated many times. However, if the initial pieces are chosen well, it may be possible that only one iteration of the processing loop would be necessary.

The total time spent in this phase of the project was nearly two-thirds of the whole time spend on the project. This may seem inordinate, but one of the goals of this project was to establish a long-term research track and without proper planning difficulties would happen. Therefore in the interest of longevity a long time was spent planning and thinking of ideas for future research and implementation.

Phase 2 – Initial coding

The PNG wrapper code can be found in Appendix B. The single processor particle class can be found in Appendix C and the MPI version can be found in Appendix D.

Before the core of this project could begin some initial coding needed to be done to develop some tools that would help in the latter stages. About 10 hrs of total time were spent coding two tools, the first was a wrapper class for libpng to facilitate graphical output and the second was a robust particle class.

Libpng is a library for writing PNG graphics files primarily on Linux machines. The advantages of PNG files over JPEG files for graphics is PNG uses a lossless compression so no image data is lost vs. the lossy compression used in JPEG encoding. By using a lossless compression algorithm we don't lose any image data in the compression and can therefore assume that what the image shows is exactly what happened.

The basic code for the wrapper was written during the 417 project. For this project the code was cleaned up and broken into smaller functional chunks with a central structure to hold the data associated with the PNG file. Also functions were added for drawing points and lines in a particular file.

The actual implementation of the features of the wrapper were trivial, some work was put into verifying the code used to translate a world coordinate into the pixel coordinate in the PNG file, but otherwise little actual needed to be done.

The particle class is a suite of functions and structures that define a single particle to be used in our simulation. Two versions were developed for use, one would be suited for a shared memory or single processor application, as it uses function pointers, and a second version that would be of use on a MPI cluster.

The goal of the particle class to provide a simple to use, flexible method of organizing the data associated with a particle. Each particle would have a set of parameters associated with it, and without a good way to organize that data it would be very difficult to manage them in the actual problem. For all intents and purposes each class is a different implementation of the same basic idea and in this document will be treated as such.

The first particle class developed was the single processor version. The primary difference between this version and the MPI version is the use of pointers. The single processor version of the particle structure uses three function pointers to set the movement functions for each axis, along with a vector for storing any sort of floating point data that would be associated with the particle, such as a mass, staring velocities, ending velocities, etc. Functions were provided to both initialize the particle, as well as set the movement vectors and update the position data based on a time. For the most part the functions were trivial in implementation, and no major issues were presented.

After the single processor version was implemented a pointer less version was needed to simplify the use of the class on MPI clusters where the memory space is not shared. To go about this the function pointers were stripped out of the original class and a method was created to allow each MPI node to create its own lookup table for the addresses to the movement functions. This way the flexibility of using a few different movement functions is retained but no actual addresses are passed between MPI nodes, just indexes. For the most part the rest of the implementation was the same as the single processor class.

After these two tools were written it was time to start on the actual algorithms that were planned in the first phase. The total time spent writing these tools was about 10hrs with testing and time spent looking at literature for the PNG wrapper.

Phase 3: Algorithmic development and implementation

Code for this section is avaliable in Appendix E.

This phase is still not complete but progress has been made in some key areas of implementation of ideas generated in Phase 1. The main focus during this phase was to implement the guess-ahead algorithm discussed in previous sections.

The guess-ahead algorithm has two main parts: the pre-computation and guess section and the computation section. Each part had the possibility to be a separate program and/or a totally independent section of code, and was treated as such.

The first part of the guess ahead algorithm that needed written was the guess-ahead part. For this essentially two separate loops were used. The first loop calculated the point of impact if no interaction would occur and the second loop would find any intersections on those lines. The logic was layed out like so:

```
FOR i = 0 to MAX_NUMBER_PARTICLES
     coord = FIND_INTERSECT_COORDS(particle[i]);
     STORE_INTERSECT_COORDS(coord):
END FOR i

FOR i = 0 to MAX_NUMBER_PARTICLES
     FOR j = 0 to MAX_NUMBER_PARTICLES
          int = INTERSECT_LINES(particle[i],particle[j]);
          IF int = TRUE THEN
               STORE_PARTICLE_INTERACT(particle[i]);
               BREAK;
          END IF
     END FOR j
     IF j != MAX_NUMBER_PARTICLES
          STORE_PARTICLE_NON_INTERACT(particle[i]);
     END IF
END FOR i
```

As you can see this is not the most efficient algorithm, as the second loop has a complexity of O(n^2). The combined complexity of both loops is on the order of O(n^3). However, there is very little we could do to speed up this part of the algorithm. One possible optimization would be to find the normal between two different explosions and then compare that normal vector to each of the line segments (treated as vectors) and any line that is nearly 180 degrees in direction from the normal would get automatically put on the non-interacting particles list.

The methods needed to implement the first part of the loop were discussed in Phase 1, and are trivial in implementation. However for the second loop there are some tricks that need to be covered. Since we are dealing with massive particles, each particle must have a certain volume of space in our virtual world. This become important when we are dealing with acute angle collisions. When we approach acute angle collisions, it could be possible that our particles do not collide as their “ideal” paths as calculated in the first loop do not intersect. However since each particle has a specific volume, it would be possible that the volumes would intersect even though their ideal paths would not. Therefore it's necessary to track the maximum dimension of the particles and see if those paths intersect.

The second part of the guess-ahead algorithm has not been implemented. However it's implementation is highly depended upon the way that the rest of the nodes available can be divided up. As discussed in Phase 1 there are a few different scenarios that could be used in computing the second part of the total algorithm. This author

was most likely going to use the more communication intensive approach with the global sync after each time step, first, then try the less communication intensive approach, as to compare the two. However at the time of writing neither had been implemented.

Phase 4: Final results and conclusion

Since no code was actually completed no conclusive results were obtained. It is the authors conjecture that this approach to solving the given problem will result in a speedup over a more conventional solution. It is the authors hypothesis that any potential speedup will come from the reduction of communication overhead between nodes, and the division of data as presented is an attempt to achieve that.

The final goal of this project as it stands it to lay a base for future exploration by this author and others into a variety of topics such as the sphere breaking and dynamic data partitioning. The code that was implementation will facilitate future exploration of the ideas presented. The author believe that this project to be a success, as it established a future track of research for this author and others.

Appendix A – Problem Description

Two mechanical explosions happen in a virtual world space. Each particle thrown off by each explosion is to have it's motion tracked in three dimensions The particles are massive and have a particular volume. Each piece will represent a chunk of a regular sphere that has a uniform thickness which is cleanly broken through along the edge of the piece.

Each particle will experience a suite of forces that include air drag, collision with other particles, and rotational dynamics. The physical properties for each of these forces will be tracked and applied as to create the most realistic simulation of the mechanical explosion possible.

The simulation will progress in discrete time steps of a small time *dt* which will be less then a half second in duration. The position of the particle will be recorded at the end of each time step.

Appendix B

Appendix C

Appendix D

Appendix E