Massively Parallel Marker and Cell with Division Visualization

Tianning Han and Nicholas Westrich

May 2, 2013

Abstract

The goal of this paper is to propose a massively parallel solution for the Navier-Stokes fluid equations utilizing the Marker and Cell simulation technique proposed by Foster and Metaxas. We also aim to show several new visualizations that we have added to the fluid renderer.

1. Introduction

Fluid simulation is a complex computational task. One of the earliest simulation techniques involves the creation of a uniform grid of fluid volumes, known as cells. These cells are then used to compute small segments of fluid velocities.

Once the velocities in these cells are calculated, divergence calculations are performed to ensure that the inflow and outflow of each cell is consistent. This helps to prevent compressibility, causing changes in the volume of simulated fluid.

Because the cell velocities only simulate the underlying fluid currents and behavior, the output visualization does not create a human-interpretable fluid. Because of this, “marker” particles are added to aid in the fluid visualization.

These marker particles follow the cell velocities, and are updated on an interval known as a timestep. Area-based velocity interpolation is performed on a per-marker basis, giving us human-interpretable fluid simulation.

After the markers are correctly interpolated, marching cube/tetrahedron computations can be performed on the marker particles to form a polygonal surface. While we do not cover it in this paper, from this polygonal surface, a good fluid representation can be created, and refraction approximations can be computed and displayed.

2. Related Work

A massively parallel marker and cell solver was described by Averbuch et al in their paper *Highly Scalable Two- and Three-Dimensional Navier-Stokes Parallel Solvers on MIMD Multiprocessors*. This implementation of a parallel Navier-Stokes solver details a cell splitting algorithm that divides along a single axis, giving “slabs” that are distributed among processors.

It also details a method for Poisson pressure equalization, utilizing global communication between processors to retrieve necessary cell data. In addition to Poisson types, they also solve for Helmholtz types to form a time discretization procedure resulting in monotonic problems.

Finally, they detail a method for Local Fourier Basis technique for the overlapping of neighboring subdomains. Local solutions in their paper are then matched using weighted interface Green’s functions.

The Foster and Metaxas Paper, Realistic Animation of Liquids, describes a serial implantation of the original Marker and Cell implementation. In their paper, they provide algorithms for velocity calculations, divergence equalizations, and velocity interpolation.

In addition, their paper also describes the use of marker cells to properly illustrate the motion of a simulated fluid.

3. Parallel Marker and Cell

3.1. Cell Division

We demonstrate two cell division algorithms. The first cell division operation works on scenes that have more processors than the number of cells in the z direction.

```
//dimensions: [0]=x, [1]=y, [2]=z
for num in divisions:
  while true:
    if (dimension[] % 3 != 1):
      dimension[num % 3] /= 2;
      num++; break;
```

Figure 1: Cubic Division Method Pseudocode

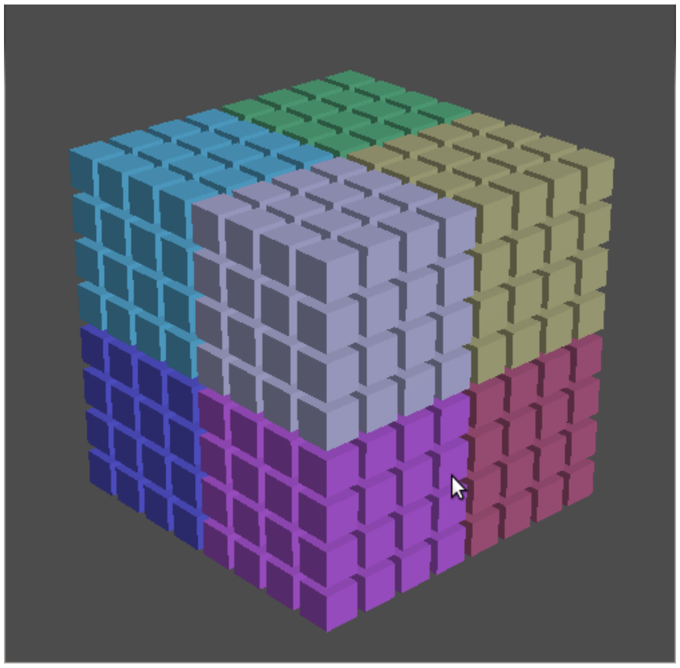
Figure 2: Cubic Cell Division with 8 Nodes

The next division paradigm cuts only the along the z axis, which improves the surface area to volume ratio of the scene, but cannot operate unless the scene has more cells in the x direction than number of processors operating on the problem.

```
    //[0] = x dimension
    dimension[0] =
      x cells / number of nodes;
```

Figure 3: Z-Cut Division Method

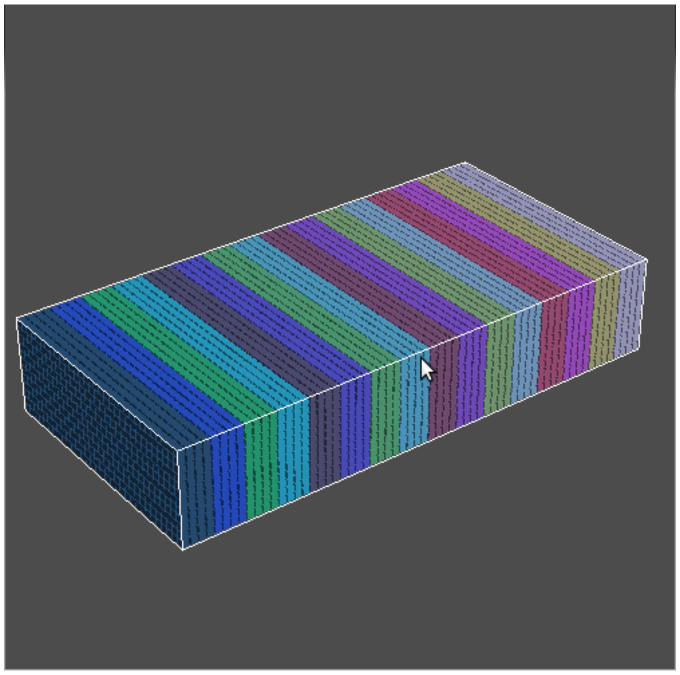
Figure 4: Z-Cut Cell Division with 16 Nodes

3.2. Boundary Cell Computation

After computation of the cell affinities, each node creates a list of its boundary cells so that it can synchronize intersecting boundary data during each timestep. To do this, each node simply builds a list of every cell that is either one less than, or one greater than each of its dimensions.

After this list has been built, it is sorted, and duplicates are removed. Because the boundary cell list is computed only once during startup, the performance impact is negligible. However, utilizing these boundary cells allows us to achieve direct neighbor-to-neighbor communications instead of gather-scatter operations, drastically improving the performance of the simulation.

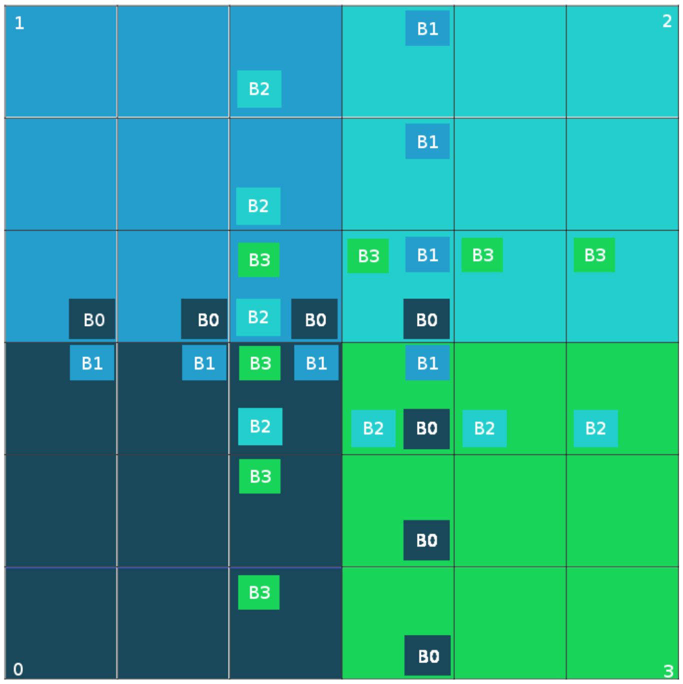

Figure 5: Boundary Cell Computation

3.3. Boundary Synchronization

Boundary synchronization occurs any time computations are performed that require data from the previously computed boundary cells. In our method, each node retrieves the boundary data using neighbor-to-neighbor communications utilizing one way data transfer.

Figure 5: Boundary Cell Computation shows a two dimensional example of the boundary cells that each node needs to receive to perform its necessary computations. Because nodes communicate directly, this synchronization has a low performance impact when compared to a master/slave gather and scatter type synchronization

3.4. Marker Division/Generation

Marker division also occurs on a per node basis. Each node is responsible for only its subset of the total particle array, which means that each node also must generate (total number of cells / number of nodes) particles within its own boundaries.

Because particle generate can be uniform or random, the generation algorithm simply generates the appropriate number of particles only within the node's cell boundaries. Additionally, particles that are generated within any obstacle in the scene are discarded.

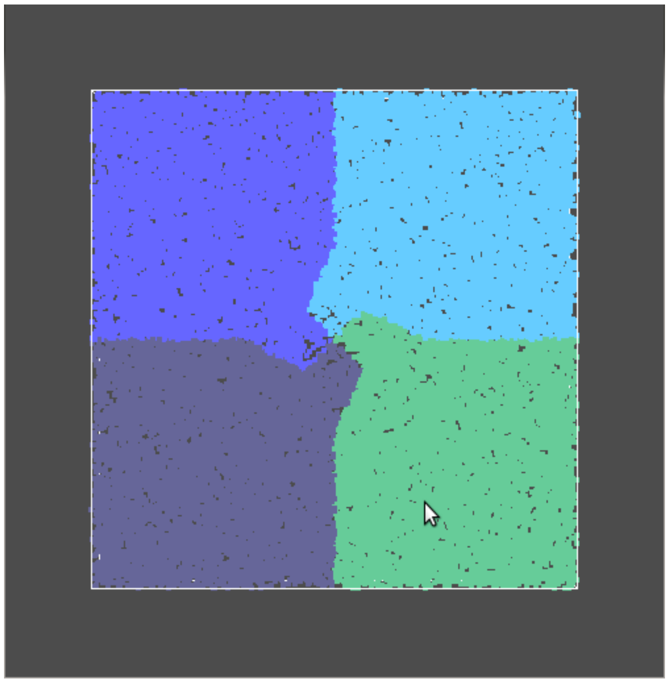

Figure 6: Parallel Particle Generation

3.5. Process Splitting

Because the Blue Gene/Q that we are working with does not have any OpenGL capable display, we found it necessary to completely separate the simulation and rendering code.

To accommodate this design decision, the simulation program outputs particle and cell data directly to stdout, which we either pipe directly into the renderer and/or a file.

The rendering is then capable of reading data from a file or data stream, and reconstructing the frame that the simulation has recorded. Additionally, the rendering implementation also computes marching cubes locally to form the final surface mesh.

Evidently, because the rendering and simulation are separated, the first frame of data in the rendering is incomplete. This explains why Figure 6: Parallel Particle Generation has already undergone an iteration of velocity computations and particle movement.

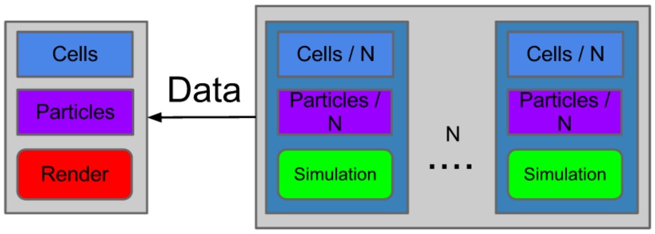

Figure 7: Process Splitting Diagram

3.6. Cell Velocity Computations

Because cell velocity computation requires the boundary cells, each node must first synchronize the previous iterations results in the boundary interfaces. However, once the boundaries are synchronized, the Navier-Stokes equations are solved locally within the node boundaries as described by Foster and Metaxas.

During the next portion of computations, we compute divergence and equalize inflow and outflow among cells to preserve fluid volume. This is accomplished by performing local divergence calculations and averaging data across node neighboring node boundaries.

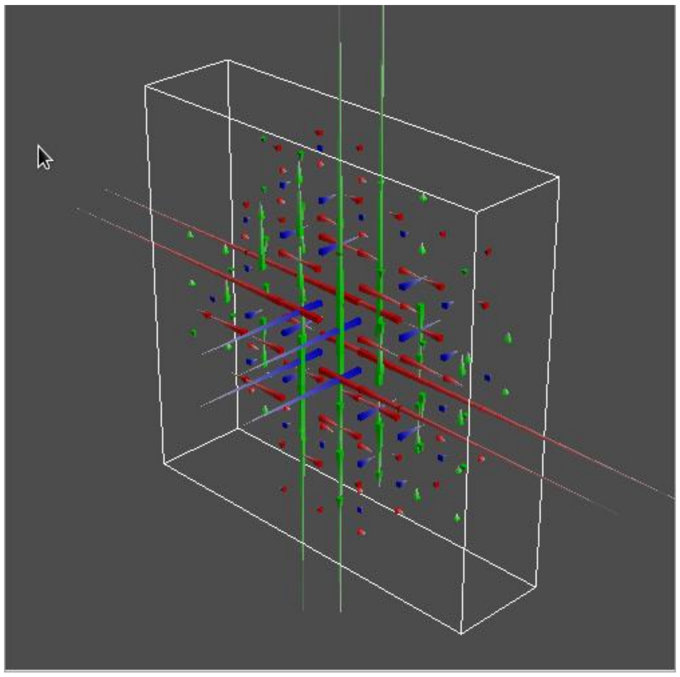

Figure 8: Parallel Velocity Computation

3.7. Marker Position Interpolation

Much like cell velocity computations, once the boundary cells are synchronized, each node computes its share of interpolated marker particle velocities and updates their position on a node-by-node basis.

However, if the particle leaves the boundaries of any node, the particles' node affinity must be updated. Because particles are stored locally on each node, we must use a bidirectional data transfer to properly update its node affinity.

```
//send particles
for p in particles:
  if p is outside node bounds:
    dest = computeDestNode(x,y,z);
    sendParticleToDest(p, dest);
    particles.remove(p);

//receiving particles
while (receiving):
  particles.add(p);
```

Figure 9: Particle Reassignment Pseudocode

3.8. External Force Field Computation

To accommodate more complex scenes, like a scene with a large wave, we had to add external force field capabilities. The addition of these capabilities allows us to apply external forces to add energy to a scene. This also means that we can control the directionality of the energy, and thus, generate interesting phenomena like waves.

```
If (wave_generate == true):
  for n in wave_duration:
    for cell in wave_cells:
      add wave velocity(cell);
```

Figure 10: Wave Generation Pseudocode

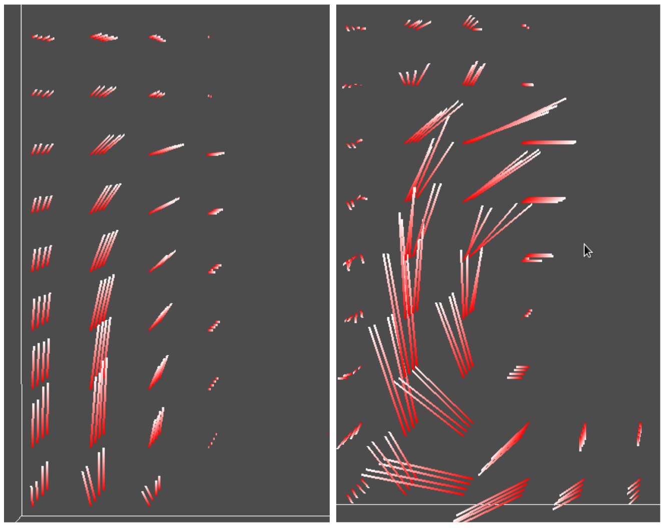

Figure 11: Force Field Velocities

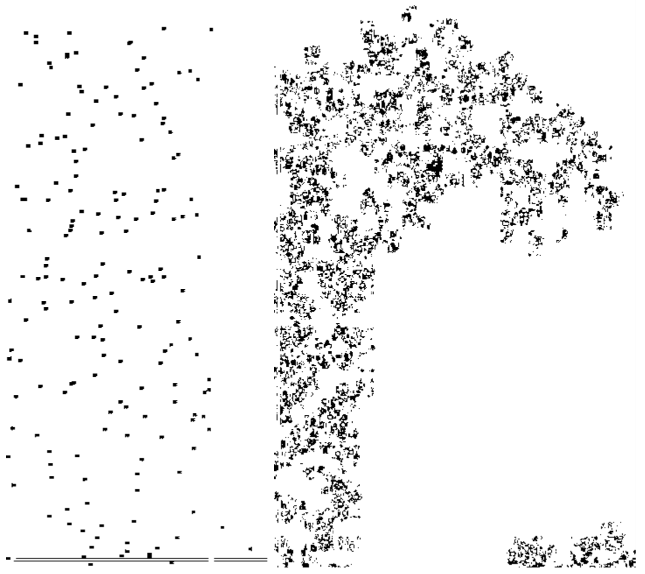

Figure 12: Force Field Particles

3.9. Obstacle Inclusion

Another feature that we've included in our algorithm is obstacle handling. As described in the Foster and Metaxas paper, we have implemented obstacles as a simple extension of the boundary cell calculations. From this, we were able to add several additional scenes. Creating a parallel version of the obstacle code was relatively straightforward, since it only required local cell velocity zeroing.

```
//obstacle list built by parser
For o_cell in obstacle_list:
  setVelocitiesToZero(o_cell);
```

Figure 13: Obstacle Pseudocode

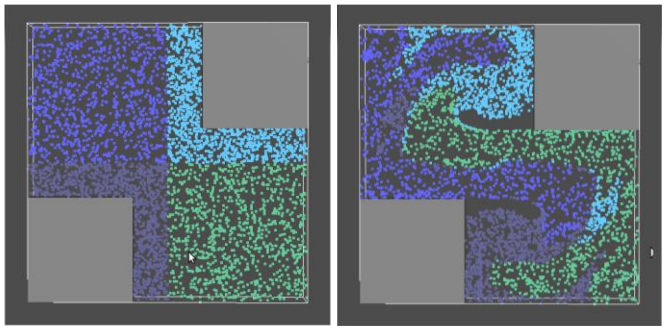

Figure 14: Starting and Ending states - Obstacle Scene

3.10 Rendering Features

In addition to our parallelized simulation, we also included several key additional rendering features. The rendering program for the recorded simulation data includes time-accurate rendering options, as well as restarting, rewinding, and variable speed capabilities. Ultimately, these features helped us to debug our simulation implementation, as well as prepare interesting live demonstrations.

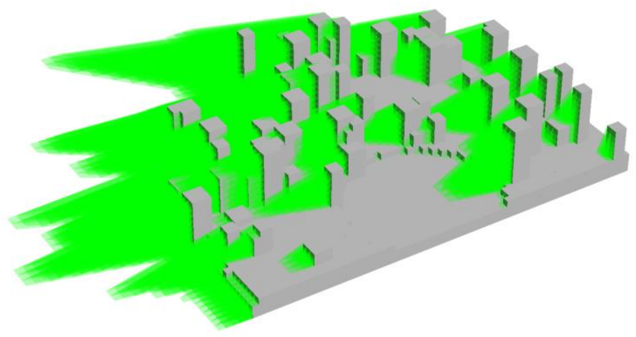
Figure 15: Large City Shadow Volumes

4. Scaling

We were able to run our fluid simulation code on the Blue Gene/Q using several different configurations. We ran three cubic scenes. Each of the scenes has a spiral velocity at the center of the front face. Each of these test scenes was run until the maximum velocity of any cell was less than 10 percent of the scenes' largest starting velocity.

From this threshold value, we were able to record computation, synchronization, and total run times on a variety of node counts, ranging from 4 to 1024.

In the first run of the tests, we enabled the data output on the Blue Gene/Q. However, because we were unable to utilize MPI file IO, we had to gather all of the data onto rank zero to be output to the data file, leading to massive increases in synchronization times.

Because of the increased synchronization times, we notice that on smaller test sizes, the optimal configuration actually falls in between 32 and 128 nodes. As the problem size increases, we see that the scaling continues as the number of nodes increases.

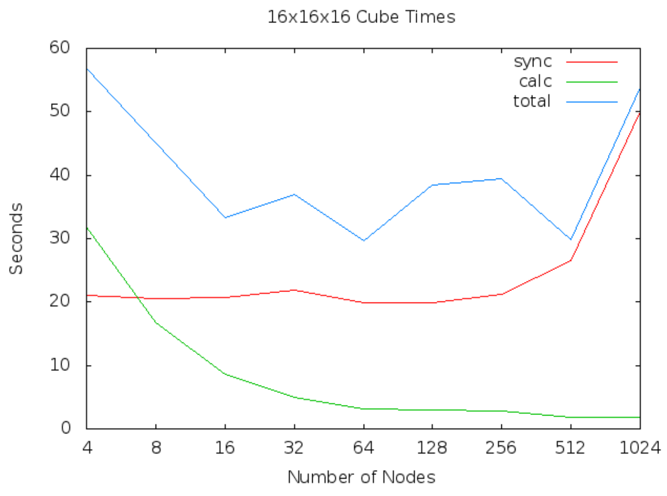

Figure 16: 16x16x16 Times - File I/O Enabled

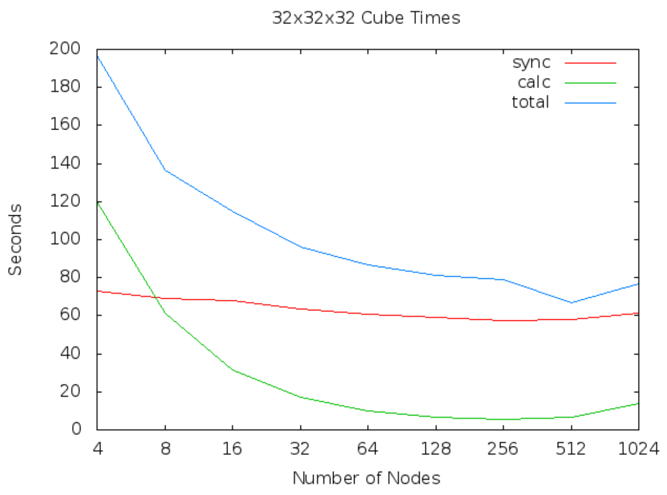

Figure 17: 32x32x32 Times - File I/O Enabled

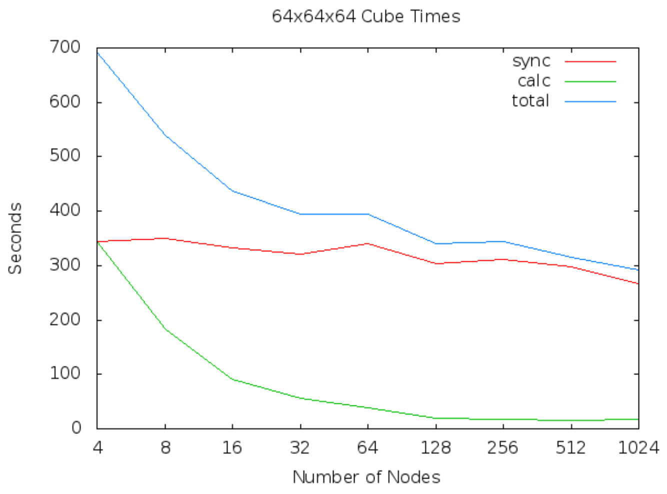

Figure 18: 64x64x64 Times - File I/O Enabled

However, in the next series of tests, we decided to disable our global gather-based file I/O to simulate the scaling of the program as if MPI file I/O were properly implemented. As the following graphs show, the scaling improves drastically.

Without the bottleneck of the faulty file I/O, we see that the scaling continues improving as we allocate more nodes to the problem. Because of this, we decided not to implement any type of dynamic node-cell allocation sizing. The scaling is a direct result of the neighbor-to-neighbor communications paradigm that we utilized. If a master/slave configuration were used, the synchronization times would increase just as they did with the gathering file I/O tests seen in *Figure 14: 16x16x16 Times – File I/O Enabled, Figure 15:32x32x32 times – File I/O Enabled*, and *Figure 16: 64x64x64 Times – File I/O Enabled.*

In the following series of graphs, we also included both single and two core run times to fully compare the performance and scalability of our implementation. In the best case scenario, we observed a performance increase of 80x when scaling from 1 to 1024 cores.

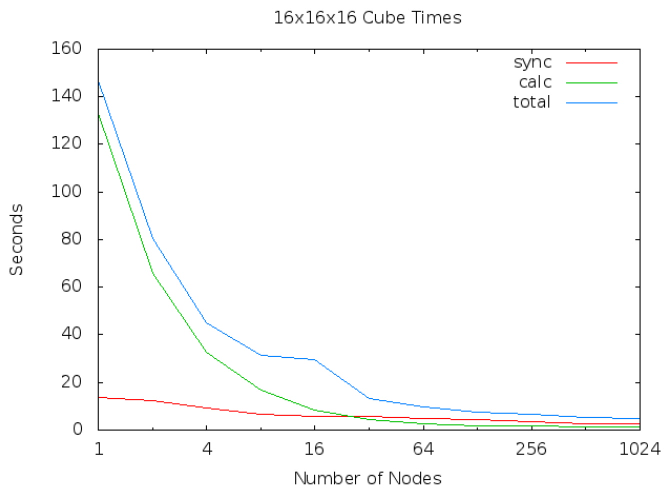

Figure 19: 16x16x16 Times - File I/O Disabled

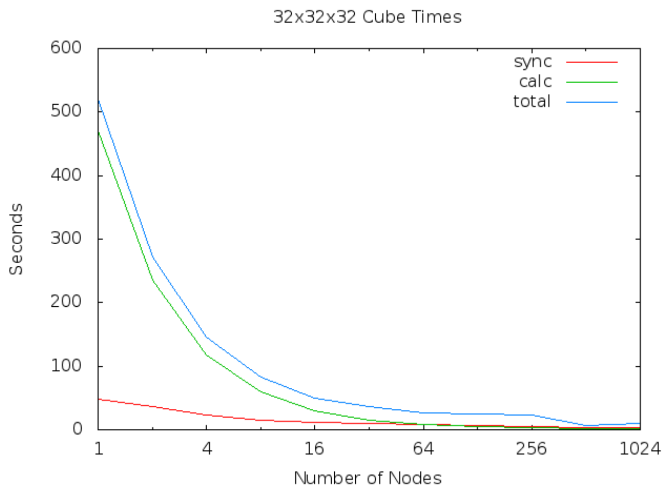

Figure 20: 32x32x32 Times - File I/O Disabled

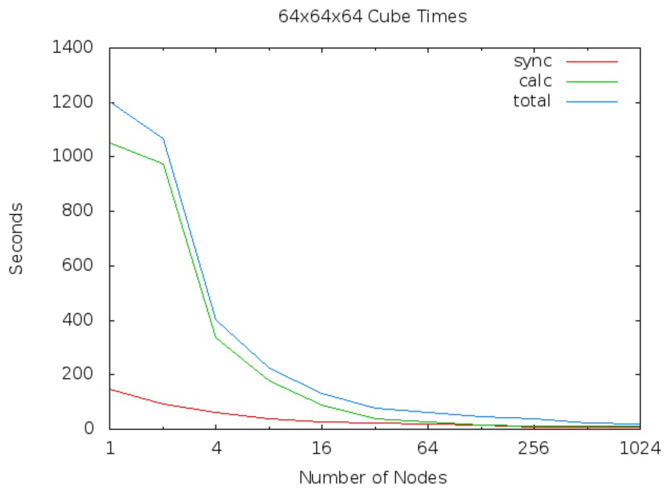

Figure 21: 64x64x64 Times - File I/O Disabled

5. Limitations

Unfortunately, our method does have some limitations. The surface code for our method does not function properly, and produces unrealistic, albeit interesting results.

Additionally, as mentioned in the Scaling section of this paper, we were unable to implement a proper MPI file I/O system, so getting renderable data has a significant impact on the overall performance of the simulation.

One of the largest limitations that we had to contest with was the endianness of our local system versus the endianness on the Blue Gene/Q. Because we output rendered data as binary code, the endian mismatch caused the rendering program to be unable to display the recorded data.

To remedy this, we implemented on-the-fly byte order switching, and ensured that all of our data types were byte aligned on eight byte boundaries. Unfortunately, this caused the simulation to run roughly 100x slower on the Blue Gene/Q.

6. Additional Tools

To facilitate our final water simulation scene, we created a procedural city generator. This city generator creates a multilevel island with a specified number of buildings. The island has specified radii for each of the levels and creates the buildings at random locations with widths of either two or three.

It is also an important property for the fluid simulation that the number of obstacles is minimized, so the city generation script guarantees to have no duplicate obstacle cells, lessening the computational load on the simulation run.

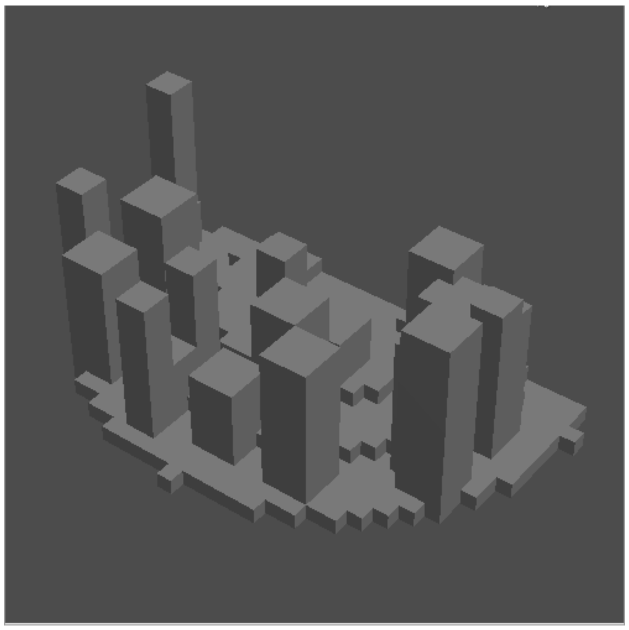

Figure 22: Small City Generation Example

7. Results and Discussion

We were able to create a large-scale marker and cell implementation with interesting visualizations. Of particular interest were the cell affinity visualizations, marker affinity visualizations, and obstacles. We also added shadow volumes and surface opacity to truly showcase the simulation's capabilities.

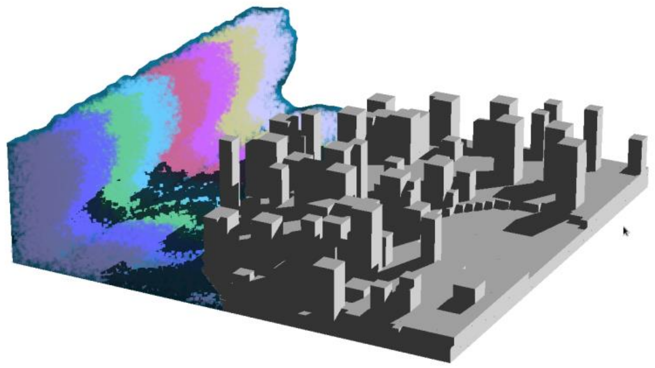
Figure 23: Large Shadowed City with Affinity Coloring

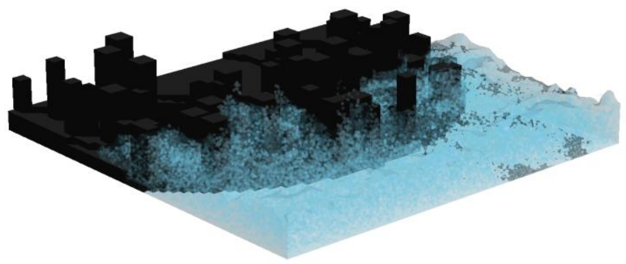
Figure 24: In-Progress City Fluid Simulation

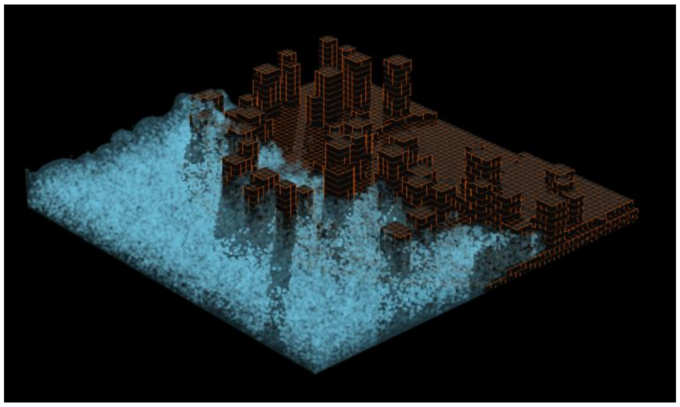
Figure 25: Tron.png

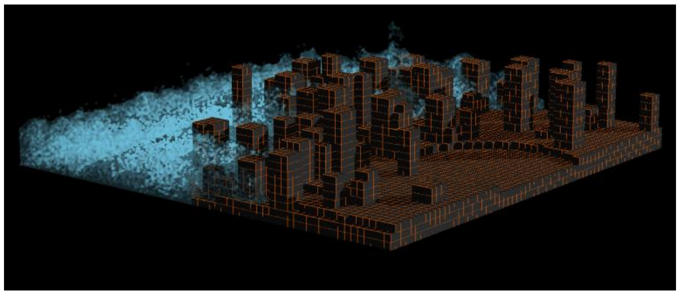
Figure 26: Tron2.png

For this project, we ended up utilizing pair programming for large portions, since much of our time was spent fixing bugs and algorithmic errors in our code. We learned much about properly using MPI and its API. Additionally, we learned that it is difficult to debug parallel programs.

Because there were many portions of the Marker and Cell code that had to be parallelized, it was difficult to track down issues in any single function. When more than one bug contributed to a simulation error, it was very difficult to track them down. Additionally, we learned that the synchronizations that are required in a problem of this nature can be difficult to optimize unless properly formulated.

In total, we estimate that we spent about 400 hours on this project between the two of us. Cumulatively, there were also nearly 400 commits to our git repo over the course of the project.

8. Future Work

We would like to complete the massively parallel implementation by including proper MPI file I/O. This would allow us to render extremely large jobs on the Blue Gene/Q and retrieve data when available.

Additionally, we would like to expand the simulation to include a non-uniform grid. By implementing some form of voxel-based simulation, we would be able to improve the simulations of smooth surfaces, buildings, and other interesting geometries that our current incarnation cannot reasonably handle.

Finally, we would like to implement dynamic load balancing so that particle computations can be better spread among the nodes. By doing this, we can ensure that we achieve the fastest possible run times when computing very large scenes.

9. References

[1] Averbuch, Ioffe, ISraeli, and Vozovoi, "Highly Scalable Two- and Three-Dimensional Navier-Stokes Parallel Solvers on MIMD Multiprocessors", *In Journal of Supercomputing*, Vol. 11, 1997

[2] Foster and Metaxas, "Realistic Animation of Liquids", *In Graphical Models and Image Processing,* 1996

[3] Keenan Crane, Ignacio Llamas, Sarah Tariq, “Real-Time Simulation and Rendering of 3DFluids.” *In GPU Gems 3*, 2009.

[4] Todisco, “Real-Time Fluids with Advanced Shaders”, 2012