A Parallel Implementation of a Fluid Flow Simulation using Smoothed Particle Hydrodynamics

Keith D. Vertanen

Major Professor: Dr. Michael J. Quinn

A Research Paper submitted to
Oregon State University
in partial fulfillment of the requirements of the degree of
Master of Science

Department of Computer Science
Oregon State University
Corvallis, OR 97331

Presented: June 24, 1999

Contents

1 Abstract

The reaction of fluid or gas flowing around an obstacle is a common engineering problem. Computer simulations are often used to measure and visualize the physical processes involved. In this report, we will discuss a parallel implementation of a simulation using the Smooth Particle Hydrodynamics (SPH) approach to fluid flow. We will discuss the design decisions made during development, problems encountered during implementation, and parallel performance issues.

Section 2 will introduce the fluid flow problem, the SPH technique, and our project motivations. Section 3 will discuss our first O(N^2) serial version of the simulator. Section 4 covers our improved O(N) grid-based serial version. In section 5, we will develop the MPI parallel simulation, discuss debugging issues, and provide results from our performance tuning. Section 6 describes our Java application client used to control the simulation. In section 7, the Java/C++ socket library used for communication with the client is discussed. We will conclude in section 8 with an evaluation of our successes and failures during the project and look at future directions for this work.

2 Introduction

2.1 Overview of the problem

The study of the behavior of fluids upon encountering an obstacle is an important one. The reaction of the fluid and the forces exerted upon the obstacle have practical implications on the design of many items such as bridge supports, hydrofoils, pipelines, fish ladders, etc. The complexity of the fluid dynamics problem makes it difficult or impossible to exactly solve for physical forces of a complex object in a flow. Approximate solutions can be obtained by construction and measurement of prototypes placed in a flow, or by use of a computer simulation.

Since usage of prototypes can be prohibitively time-consuming and expensive, many have turned to computer simulations to provide insight during the engineering process. The use of computer simulations can also be a valuable aid for students learning fluid mechanics. Students can observe the mathematical theories in action and have the ability to alter simulation setup and parameters much more easily than one could with a real-world experiment.

2.2 The SPH method

There are two common types of numerical methods for doing fluid simulations: the Eulerian approach (or grid method) and the Lagrangian approach (or particle method) [21]. Our simulation will utilize a Lagrangian approach called Smoothed Particle Hydrodynamics (SPH). Using a Lagrangian approach allows us to use particles (or more accurately, interpolating nodes) to simulate the fluid rather than a more complicated grid-based computation. Particle methods such as SPH have proved to be better able to simulate complicated systems and handle unexpected situations. On the downside, particle methods tend to be less accurate then correctly tuned grid based techniques.

SPH was originally created to study galaxy formation and other cosmology related problems [12] [8]. It has since been utilized to study a variety of fluid dynamic problems including viscous flows [19] and free surface flows [13]. For the details of the physics involved in SPH, the reader is referred to [3] [14]. For precise details of the physics used in our particular implementation, see [17]. For the purposes of this paper, we will abstract the SPH technique to the following high-level algorithm [18]:

1) Set initial conditions.
2) If not the first time step, predict thermal energy, velocity, and density for each particle.
3) Calculate pressure and speed of sound for each particle.
4) For each particle *i* do

    For each particle *j* in the neighborhood of particle *i* do

        Calculate change in density *j* causes on *i*.

        Calculate change in artificial viscosity *j* causes on *i*.

        Calculate change in velocity *j* causes on *i*.

        Calculate change in thermal energy *j* causes on *i*.

5) Update each particle's thermal energy, velocity, position and density.
6) Increment time.
7) Goto step 2.

2.3 Project motivation

There are numerous commercial fluid dynamic software packages available [10], but few are designed for students (Dynaflow's 2Dflow [11] being an exception). We found none that were freely available and none utilizing the SPH simulation approach.

We found several serial SPH implementations [15][1], but lacking powerful workstations, these serial versions would only be usable on simulations of limited scope. Using SPH to simulate fluid flow across an obstacle was a problem very amenable to solution by parallel distributed memory machines.

At our department, we could utilize SWARM, a Beowulf [2] cluster of 80 Pentium IIs running the MPI [7] message passing API. We found only one MPI parallel implementation of SPH [5]. This implementation was designed with non-uniform cosmology simulation in mind, not with a more uniform flow of a fluid past an obstacle. We contacted the authors, but source code was also not available for general use.

Our project was thus motivated by the following two goals:

1) Provide an easy-to-use tool for students to experiment with a SPH simulation.
2) Provide an easy-to-understand parallel MPI implementation of SPH that might be reused and extended by others in the future.

3 Initial serial version

3.1 Design considerations

Our initial efforts in developing the SPH simulation were focused on creating the computational code necessary to carry out the physical interactions between particles in the simulation. We developed the code and data structures to provided the easiest method of implementing the SPH physics equations. This focus had both advantages and disadvantages. The primary advantages were:

1) Allowed us to gain insight in to the intricacies and interdependencies present in the physics.
2) Quickly obtain simulation results showing that our computational code was producing “reasonable” results.
3) Demonstrate whether the problem was computationally intensive enough to justify a more sophisticated and/or parallel implementation.

The disadvantages to focusing just on the physics problem were:

1) Little time devoted to a well thought out design for the software infrastructure behind the physics.
2) Designing code and data structures without concern about the scalability to larger problem sizes.
3) Development of code and data structures that would be difficult to convert into parallel code.

3.2 Implementation and results

Our strategy of plunging straight into programming the project succeeded in some respects and failed in others. Within two weeks, we had a working simulation that was producing plausible results. We had developed classes to represent the fundamental objects in our simulation: smooth particles and boundary shapes. We had also coded functions to perform the calculations representing all the physics of SPH.

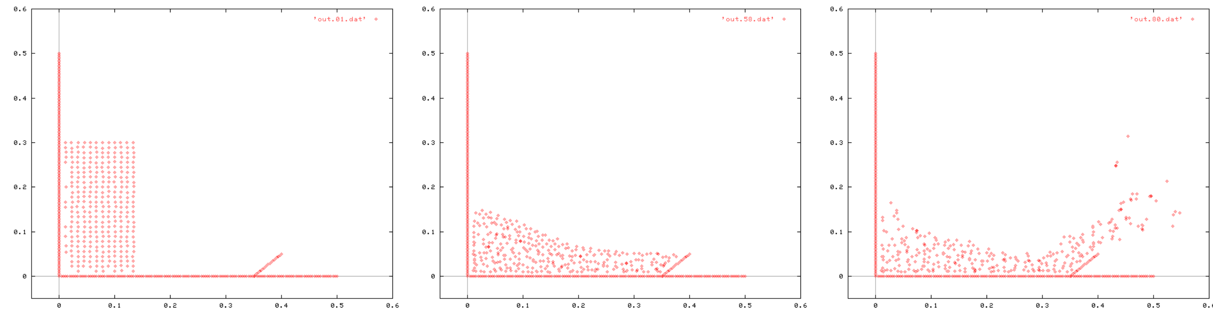


Figure 1: First successful simulation, an imitation of Monaghan's breaking dam.

Lack of a well thought out class design before coding resulted in a fairly degenerate class structure and code distribution in the software. Nine separate classes were utilized in this initial version. A total of 2524 lines of code were created with 1216 (almost 50%) in the `World` class. The `World` class encompassed all the physics, reading and storage of simulation parameters, output generation, and simulation initialization.

In an effort to make the code as simple as possible, various inefficiencies were introduced. In SPH, each particle only interacts with particles within a certain distance (the smoothing length). To make this version as simple as possible, we checked each particle against every other particle. We did not employ any sort of grid or sorting of particles to avoid an all-pairs comparison. For N particles, this leads, of course, to an $O(N^2)$ algorithm. In addition to this basic algorithmic inefficiency, there were also numerous redundant physical calculations present in the code.

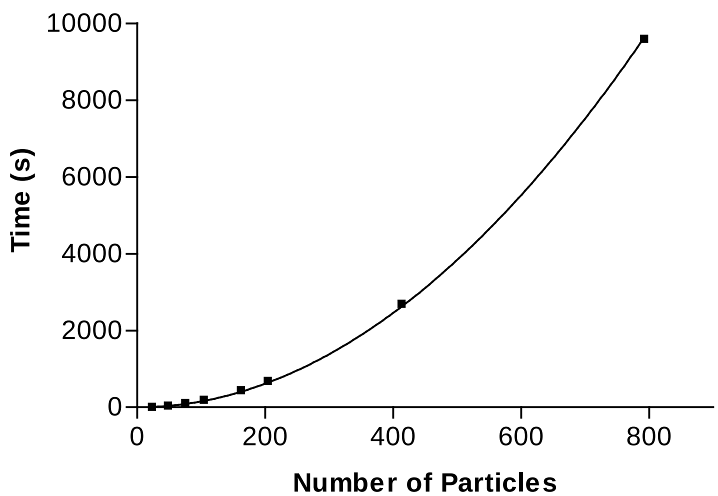


Figure 2: Initial version performance curve: $y = 0.0153x^2$ sec

A timing study was performed on the first version to determine the constant term and to allow us to estimate the wall-clock time required to run bigger simulations. Varying the number of particles, the simulation was run on an unloaded Sparc-20 workstation. The results (figure 2) showed that the running time (for 12,500 time steps) could be expressed by the equation:

$$y = 0.0153x^2 \text{ sec}$$

For physically accurate simulations, we expected to need simulations with on the order of 10,000 particles and 100,000 time steps. A simulation of this magnitude equates to 141.6 days of computation. Clearly this was going to be much too slow.

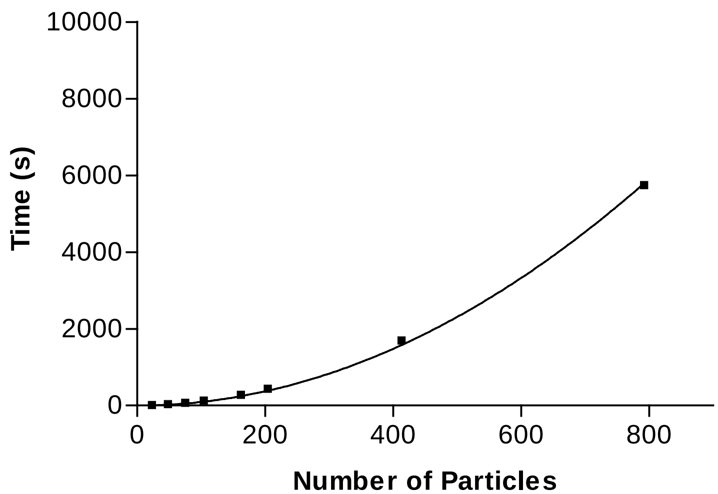


Figure 3: Optimized version performance curve: $y = 0.00923x^2$ sec

An optimized version of the simulation was created, eliminating redundant physical calculations by caching results in global variables. While this saved us a significant number of cpu cycles, it also noticeably reduced the readability of our code. The optimization was successful in reducing the constant term to 0.0093. Our target simulation of 10,000 particles now took just 86.1 days. Reducing the constant term was of course not the solution to our efficiency problem, we needed to address the $O(N^2)$ nature of our algorithm.

Probably the biggest advantage of doing a "quick-and-dirty" version was that by having a working example of how it *could* be designed, we were better able to see how it *should* have been designed. In our first attempt, we had failed to split up tasks into appropriate classes and had failed to use an appropriate algorithm to avoid an all-pairs computation between the particles. It was now time to throw away this version, address the deficiencies, and start from scratch.

4 Improved grid version

4.1 Design considerations

After our experience with the first version of the simulation, we had a much better idea of what needed to be done and how a new version could improve upon the old. We developed the following goals for the new version:

1) Calculation of particle interactions needs to be O(N).
2) There should be better separation of tasks between different classes.
3) Interface to particle interactions should be atomic. We should be able to call one method to update a particle against another.
4) Data structures and algorithms should be reusable in a parallel implementation with a minimum amount of modification.
5) The code needed to be more readable. Others should be able to use and extend our simulation in the future.

We also discussed the details of the implementation. It was apparent that a more complicated class structure was going to be necessary. This lead us to the following questions:

1) What classes were needed and why?
2) Which class (or classes) contained the actual particle data?
3) Which classes needed access to data in another class? How would they get to it?
4) What classes could we add for an easy transition to parallel code?
5) Which class would do the computations?

Table 1 outlines the primary classes we decided upon to address the above questions.

Class name	**Purpose**	**Contains objects of type**

`World`	Main controlling class of the simulation. Responsible for reading in the simulation parameters, initializing and storing `SmoothParticles` and `BPShapes`, telling `Grid` to run each time step and outputting the result.	`Parameter`
		`BPShape`
		`Grid`

`Grid`	Contains the `Cells` which track which part of the `Grid` every free particle is at. Responsible for moving particles between `Cells`, deleting particles that move out of the simulation, adding new particles. The `Grid` also contains an array of all the `SmoothParticles.`	`SmoothParticle`
		`Cell`
		`Parameter`

`Cell`	Holds a collection of `SmoothParticles` that represent the free and boundary particles in a particular region of the simulation. Stores in a `FastLinkedList` the indices of the particle in the `Grid` array.	`FastLinkedList`
`SmoothParticle`	The actual particle object in our simulation. Contains all the physical attributes of the particle, knows how to update these attributes when interacted with other `SmoothParticles.`	
`BPShape`	A collection of non-moving `SmoothParticles` that make up some boundary shape (a cylinder or a line).	`SmoothParticle`
`FastLinkedList`	Provides a linked list with multiple current pointers for use during our inter-particle calculations.	
`Parameter`	Contains all the simulation constants and parameters.	

Table 1: Main classes of our grid-based serial version.

4.2 Implementation and results

The coding of the new grid-based version went fairly smoothly, taking about two weeks to implement and debug. Careful consideration was given to what implications the choices made during implementation would have on a future parallel version. This slowed down the development process, but it was hoped this would yield significant time savings in the future.

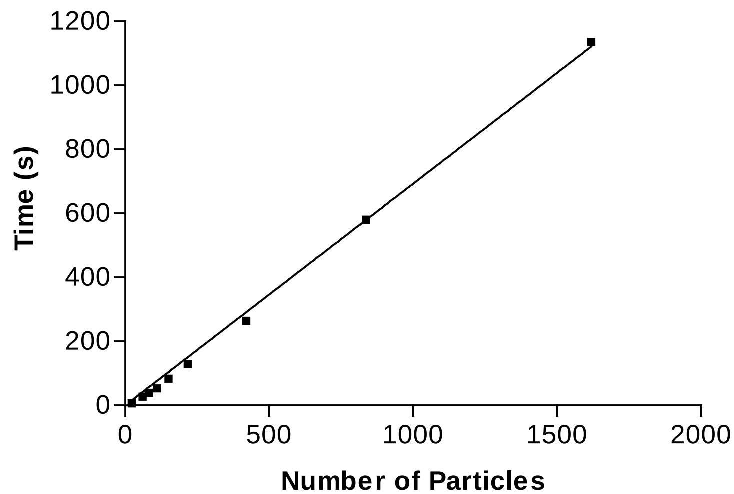


Figure 4: Grid based version performance curve: $y = 0.694x$ sec

Another timing study was performed on the new grid-based version. Our new algorithm design was O(N) and the empirical results bore this out, yielding a curve of:

$$y = 0.694x$$

Our target simulation of 10,000 particles and 100,000 time steps could now be done in 15.4 hours. The overhead of maintaining the grid information was increasing our constant term significantly, but the grid version still beat the $O(N^2)$ version for all simulations with more than 75 particles. Despite our reduction of complexity, the simulation was still proving to be computationally intensive enough to warrant a parallel implementation.

5 Parallel version

5.1 Design

5.1.1 Domain decomposition and inter-processor communication

While we had been careful to plan for future parallel version in our design of the serial grid version, the parallelization was still a formidable task. The first decision that needed to be made was exactly how to split the problem up among the processors. We had a two-dimensional grid of cells containing our particles. The particles could in general be assumed to be flowing from left to right during the course of the simulation.

We decided that we would divide the total simulation grid into horizontal slices, giving one slice to each processor. Each slice would consist of smaller version of the global grid, containing the same number of grid cell columns, but fewer grid cell rows. This had the advantage of being simple, utilizing the same grid construct used in our serial version. It would also hopefully minimize the number of particles travelling across processor boundaries. The prevailing flow of the fluid would tend to keep particles on the same processor.

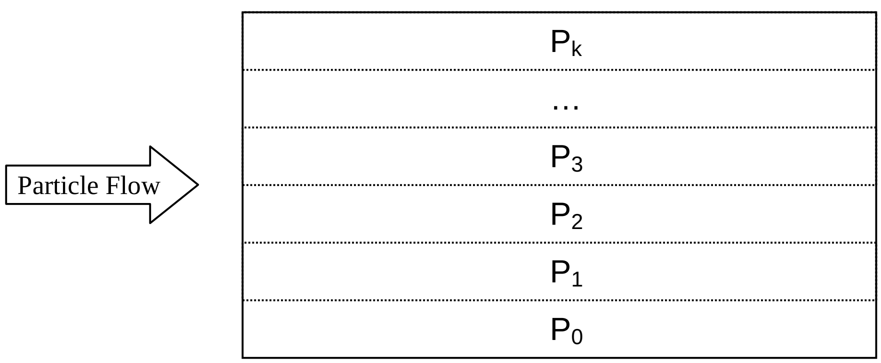


Figure 5: Division of our simulation grid among processors.

The main difficulty that now needed to be addressed was what was going to occur along processor boundaries. The particles near the top or bottom of a processor's slice would need to interact with particles that did not exist on that processor. The processor would either need to contact its neighboring processor and ask for the necessary information, or it would need to include some sort of overlap in particles near the boundaries.

We first considered the technique of selectively sending and receiving information along the boundary. This would require that a processor tell its neighbor which cells it needed particles from to complete its computation. While this strategy could reduce the overall quantity of information travelling between processors over the boundary (since a processor may only need a subset of the particles in a boundary row), it would also involve more complicated logic to determine exactly what needed to be sent. In addition, since one could typically expect that the boundary row would be uniformly full of particles, this would require the neighboring processor to send virtually all of the particles present in its boundary row to the other processor.

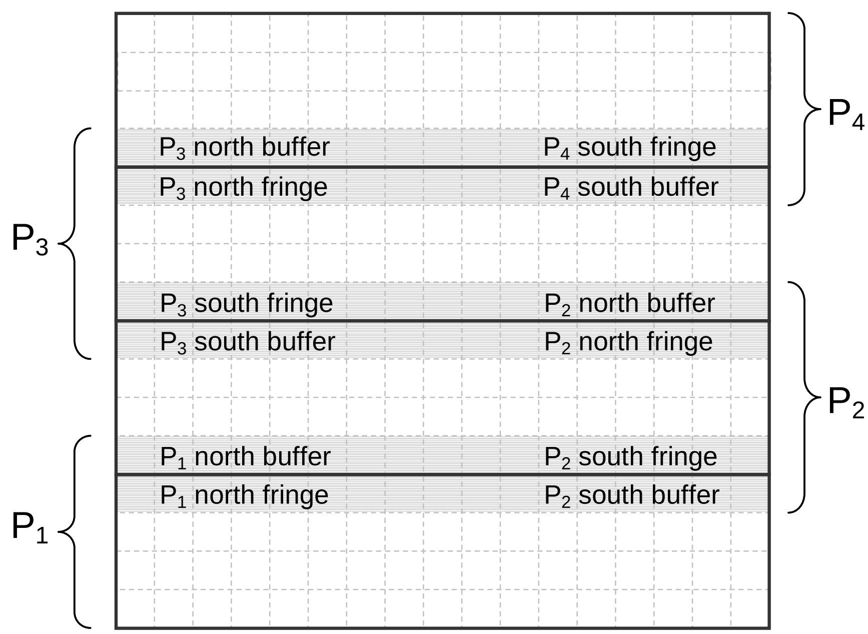
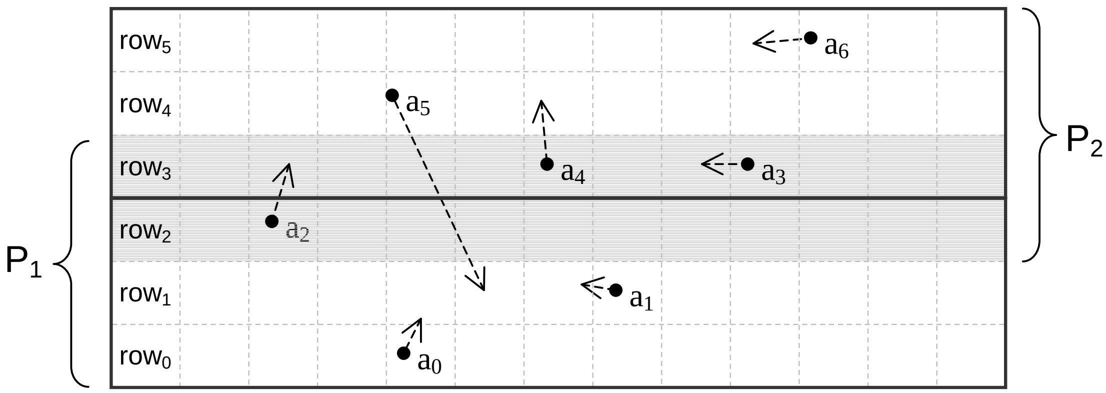


Figure 6: Example showing which rows each processor would contain.
Shaded rows are duplicated on adjacent processors.

Since we didn't expect much benefit from a more selective communication technique, we decided to solve the boundary problem by enforcing overlapping rows between adjacent processors. Each processor would have a set of rows that it was entirely responsible for maintaining. In addition, it would have *buffer rows* along the top and bottom of its section of grid. The buffer rows would allow a processor to update everything in its grid except for these buffer rows. After each time step, adjacent processors would exchange their top and bottommost rows for which they were responsible. These rows adjacent to the buffer rows we'll *call fringe rows*.

Besides exchanging buffer rows after each time step, processors may also need to communicate additional particles. *Migrant particles* are particles that have moved either into a processor's buffer row or completely out of a processor's rows. These migrants must be sent to the adjacent processor to insure consistency. The following example will help illustrate:



P_1	P_2
Has copies of particles: a_0 – a_4.	Has copies of particles: a_2 – a_6.
Updates particles in row_0 – row_2: a_0, a_1, a_2	Updates particles in row_3 – row_5: a_3, a_4, a_5, a_6
Deletes particles in row_3: a_3, a_4	Deletes particles in row_2: a_2
Sends particles in fringe row_2 to P_2: empty	Sends particles in fringe row_3 to P_2: a_3
Sends migrants to P_2: a_2	Sends migrants to P_1: a_5*
Receive particles in row_3 from P_2: a_3	Receive particles in row_2 from P_1: empty
Receive migrants from P_2: a_5	Receive migrants from P_1: a_2
Has copies of particles: a_0, a_1, a_2, a_3, a_5.	Has copies of particles: a_2, a_3, a_4, a_6.
Ready for next time step.	Ready for next time step.

* Note: normally particles should not jump across multiple rows. Such behavior would indicate that the simulation is running too rapidly. The system outlined above will handle multiple row jumps as long as an entire processor is not skipped. If a particle does skip over an entire processor, it will be deleted from the simulation.

5.1.2 Other design considerations

We now had an overall idea of how the simulation would be split among the processors. The remaining design phase involved issues related to how to implement this idea. The following design points were developed:

1) Server will consist of a master process and a number of slave processes. The master would have the following responsibilities:
    a) Communication with the client.
    b) Initializing of the simulation.
    c) Initializing of the slaves.
    d) Collecting results from the slaves.

    The slave processes would do the following:
    a) Compute the new particle positions for each step of the simulation.
    b) Communicating necessary information with adjacent slave processors.
    c) Periodically send the master an update of all the slave's particles.
2) Isolate in separate classes, routines for communication from the master to the slave, from the slave to the master, and from the master to the client.
3) At least initially, favor simple communication schemes that will be easier to get working correctly.
4) Reuse as much functionality as possible from the serial version. Aside from communication of fringe rows, each slave could be considered to be running its own serial simulation.

Class name	Purpose	Contains objects of type	Contains pointers to

`Master`	Provides main processing loop for master process.	`ClientLink`
		`MasterMPI`
		`World`	

`Slave`	Provides main processing loop for slave processes.	`SlaveMPI` `SlaveWorld`	
`ClientLink`	Handles communication between the master and the client via a socket connection.	`Server`	`World` `MasterMPI`
`MasterMPI`	Handles communication to the slaves via MPI routines. Provides methods to initialize slaves, send commands to slaves, and receive updates from slaves.		`World`
`SlaveMPI`	Handles communication to the master via MPI routines. Provides methods to receive commands from master, send and receive particles with adjacent slaves, and send master periodic updates.		`SlaveWorld`
`World`	Master's class controlling overall simulation grid. Provides methods to initialize the `Grid`, to write output to disk, and to perform a time step in the simulation.	`Grid` `Parameter`	
`SlaveWorld`	Slave's class controlling its portion of the simulation grid. Provides methods to perform a step in the simulation and to pump in additional particles into the `Grid`.	`Grid` `Parameter`	

Grid	Grid will be used by the master process for initialization and compilation of results. Contains the Cells which track which part of the Grid every free particle is contained in. The Grid contains an array of all the SmoothParticles. The Grid will handle free space management of this array.	SmoothParticle Cell Parameter FastLinkedList	
SlaveGrid	Subclass of Grid used by slave processes. Includes methods to read and delete particles in the fringe rows, track particles heading to adjacent processors, obtain double arrays containing particle information, and to perform calculations on the particles.	SmoothParticle Cell Parameter FastLinkedList	
Cell	Holds a collection of SmoothParticles that represent the free and boundary particles in a particular region of the simulation. Stores in a FastLinkedList the indices of the particle in the Grid array.	FastLinkedList	
SmoothParticle	The actual particle object in our simulation. Contains all the physical attributes of the particle, knows how to update these attributes when interacted with other SmoothParticles.		
BPShape	A collection of non-moving SmoothParticles that make up some boundary shape (a cylinder or a line).	SmoothParticle	
FastLinkedList	Provides a linked list with multiple current pointers for use during our inter-particle calculations.		

`Parameter`	Contains all the simulation constants and parameters.		

Table 2: Main classes of the parallel version

5.2 Implementation

Once the design was complete, modifications began on the serial code. Rather than try and implement everything at once, a step-wise process was employed. In the first step, a version was designed that could receive the simulation parameters from the client. After receiving these parameters, the master MPI process would run the simulation in serial without utilizing any of the slave processes. This step allowed development and testing of the classes responsible for the client-master socket communication.

In the second step, the master process initialized a single slave process. The master sent the slave all the necessary data and the slave would then run the simulation in serial. This allowed development and testing of the classes involved in master-slave communication, as well as a modified `World` class (`SlaveWorld`) that the slaves would utilize.

The next step was the hardest: implementing the actual methods and data structures necessary for a parallel solution. This involved methods to handle sending and receiving particles between slave processors, deletion of particles within a processors fringe row, free space management of the fixed arrays holding particle data, and the division of the initial simulation grid by the master. At this phase, the problem of compiling results from all the slave processors was ignored; each slave simply wrote its output to a separate file.

Finally, code was added to handle compilation of results. The slaves would periodically send all their rows (except for buffer rows) to the master. The master could then compile a complete global grid. Initially this was output to a single data file, but later it would be sent over the socket link to the client application for display.

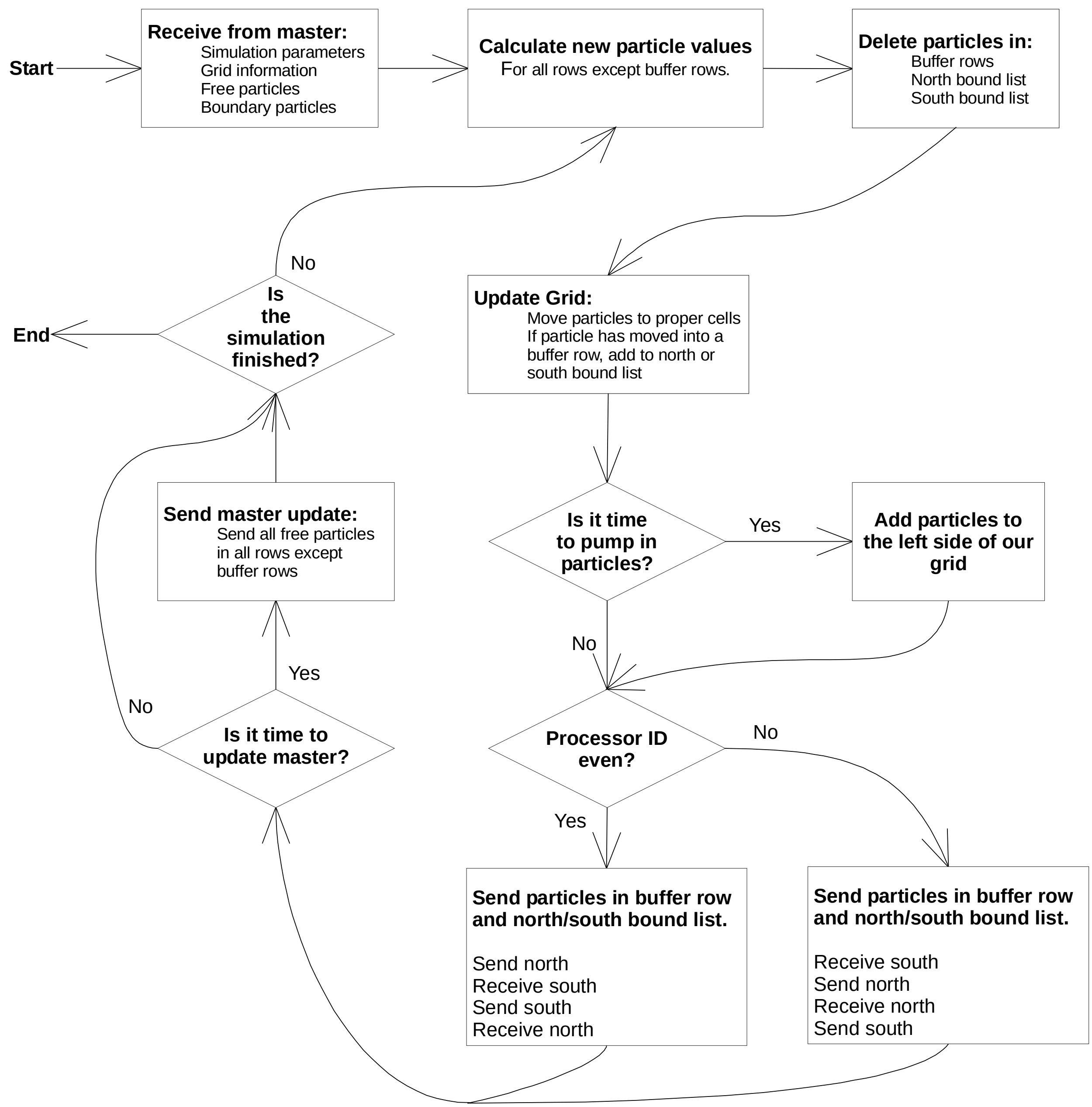


Figure 7: Flow chart of the operation of a slave process.

5.3 Debugging

5.3.1 Initial problems

A running parallel version of the code was produced fairly quickly from the serial program. As we began to use the parallel version however, problems soon began to arise.

Initially we had numerous bugs that caused the program to core dump. These turned out to be some of the easiest to fix; most were related to improper and out of range array indices. It was a simple matter to track the line of code responsible for the core dump, then backtrack out to explain why the index was bad. The nature of these errors dictated that they would occur early in the simulation; one can't get away with using out of range array indices for very long.

During this first stage of debugging, the code was modified to make use of the MPI debugging routines `MPE_IO_Stdout_to_file`. This allowed the standard output of each process to be redirected automatically to a file. From this point on, we would be able to place debugging print statements in our code and have them show up in the file corresponding to each node in our network. These print statements were our primary debugging tool, we did not employ any parallel debugging tools.

After the initial rounds of bugs were eliminated, the bugs became more insidious. The next problem encountered involved the "bad" behavior of some of our particles during a simulation. Particles would occasionally disappear, reappear and just act plain crazy. It was often the case that the behavior would occur as particles crossed from one processor to another. Initially this led us to believe that there was a mistake in how particles were being passed among processors. But after much investigation, it was finally discovered that particles leaving a processor were having their array index added twice to our free space linked list. This caused incoming particles to overwrite each other in unpredictable and incorrect ways.

5.3.2 Searching for the cause of "cancer"

All seemed well for a few weeks, but as longer and bigger simulation were tried, a new problem appeared. After a simulation had been running normally for quite some time, almost all the particles would suddenly disappear. At first the disappearance seemed fairly instantaneous, but this was due to the infrequency of our output to animation frames. We modified our code to increase the frequency of output frames around the time of failure. In this way, a finer grained picture of the events surrounding the failure could be seen. As we can see in figure 8, the disappearance of particles appears to start in localized regions and then spread to surrounding particles, eventually leaving only a few random particles bouncing erratically around the grid. This deadly spreading behavior of the bug earned it the affectionate title: the cancer.

Finding the cause of the cancer was no easy undertaking. The main problem was that failure seemed to occur only after the simulation had been running for tens of thousands of time steps. If one was not clever in how and when debug messages were displayed, output files would soar to hundred of megabytes.

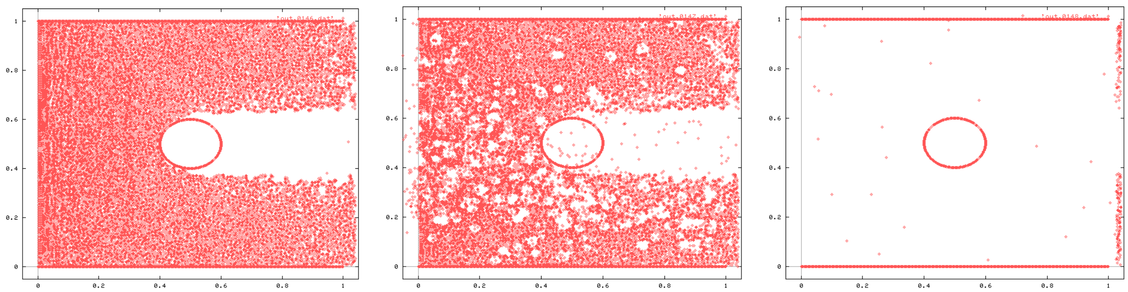


Figure 8: Example of catastrophic propagation of bad values in the simulation.

It was observed that some particles were jumping large distances in a single time step, jumping over several grid rows. This led to the discovery that these particles had very large velocity values. These high values would in turn effect any other particles that updated themselves against the offending particle. It was in this manner that the error seemed to propagate through all the particles. Once a particle had a high velocity, it would fly off the grid and be deleted, but not before “infecting” other particles with bad velocity values.

In an attempt to correct the problem, the code was modified to delete particles if their velocity became too great in magnitude. This effectively cauterized the cancer, leaving a big hole in the grid of particles, but preventing it from catastrophically propagating. This was of course simply treating the symptoms of the problem and not the actual cause.

The real breakthrough occurred when a small example was found in which a cauterization event occurred. With only around 100 particles in the simulation, it was easy to track the motion and deletions involved in the event. It was noticed that during the time step before deletion of particles began, a single particle had bounced backwards and crossed over the line x = 0. A test fix was applied, deleting all particles once they reached x = 0. This fixed the test case problem, but eventually the cancer still reappeared. A further fix was applied, deleting all particles that reached a x-coordinate of 1.0. Cancer had now been eliminated, but we understood little about the actual cause of the bug.

After several days of further analysis, we realized that the particles in the outermost cells of the grid (those with x-coordinates of less than 0.0 and more than 1.0) were being updated differently from other particles in the simulation. In our algorithm, some formulas are applied sequentially to all particles in our arrays (they do not depend on values of their neighbor particles). Other formulas are computed only between particles in adjacent cells. These outer cells do not have neighbors on one side and therefore were being skipped during particle-particle interactions.

This inconsistent treatment of the border cells led to progressively bad values for particles within adjacent cells. Now that we understood the cause of the cancer, the most reasonable solution was the one we discovered earlier: deleting particles when they reached the outermost cells.

5.3.3 Verification using different number of processors

After our experience with the cancer, our faith in the correctness of the parallel version was shaken. It was decided to verify that the results of the simulation running on multiple slaves matched the results of the simulation running on a single slave. The single slave version was a good benchmark as it did no complex inter-processor communication, it only periodically sent all of its particles to the master for output. Just by examination of the output animations, it could be seen that the multiple slave animation and the single slave animation were not exactly the same.

We needed more accurate tools to measure how and when the two simulations diverged. It would be handy to be able to track particular "problem" particles throughout the simulation. To do this, a unique ID field was added to every particle. A scheme was developed that allowed new particles flowing into the simulation to be assigned ID's in such as way that a particular particle would have the same ID regardless of the number of processors the simulation was running on.

With the ID field in place, a debug file with the particle ID number and all physical particle attributes could be outputted. A Java program was written that would read in the debug files from two separate simulation runs. Since the order of the particle in the debug file varied with the number of processors, the particles had to be sorted by ID fields. A comparison could then be done between particle to detect any difference between particle positions. Using these tools, the following bugs were found in the parallel implementation:

1) Failing to send all relevant particle attributes between processors.
2) Redundant particles were being pumped into processor buffer rows.
3) Some migrant particles were being "bounced" back to the sending processor, allowing a double update for a single time step.

Even after fixing these errors, it was still observed that occasionally data files would differ by very small amounts (on the order of 10^{-6}). The output data file was increased to include all 15 digits of precision afforded by doubles. This allowed us to see that values did get off by small amounts in the last decimal place from early in the simulation.

Number of time steps	**Average particle position error** Different linked list orders.	**Average particle position error** One slave versus four slaves, same linked list order.
5,000	8.956×10^{-17}	0.000
10,000	8.131×10^{-15}	0.000
15,000	3.278×10^{-10}	0.000
20,000	1.901×10^{-08}	2.630×10^{-16}
25,000	5.903×10^{-09}	1.299×10^{-14}
30,000	1.884×10^{-08}	1.346×10^{-12}
35,000	4.755×10^{-07}	2.699×10^{-10}
40,000	7.262×10^{-05}	2.533×10^{-08}

Table 3: Magnitude of error introduced by order of computation or by number of processors.

We discovered that while particles were undergoing the same interactions, these interactions were occurring in different orders. The differences we saw in the last decimal place were being caused by rounding effects related to the order of the summations carried out by the simulation. In the version running on multiple processors, the linked lists containing the contents of a cell would become reversed in comparison to the version running on a single slave. We changed our linked list implementation so this would not occur and the difference was reduced substantially. The single and multiple processor runs still do differ, indicating that there are some calculations that are still occurring in different orders.

As we can see from table 3, the magnitude of error introduced by our order of computation was very small. It is however growing exponentially as simulation time increases. While the differences caused by order of operations may only affect the least significant digits, the differences are propagated as they are used repeatedly in all future calculations.

After our successful experience using this comparison tool to find bugs, we continued to use it to verify any code changes against a standard data set. In this way, we could be assured that the changes did not introduce any errors into the simulation (at least not any errors that produced noticeable differences in the values of our particles).

5.4 Optimizations

5.4.1 Initial performance and improvements

Once the bulk of the debugging was complete, the next phase was to measure the parallel performance of our implementation. Timing studies were performed, holding the simulation size constant at 10,000 particles while varying the number of processors used. Our measure of parallel performance will be parallelizability [16]. If the time it takes the parallel program to run on one processor is T_1 and the time it takes to run on p processors is T_p, then the parallelizability P is given by:

$$P = T_1 \; / \; T_p$$

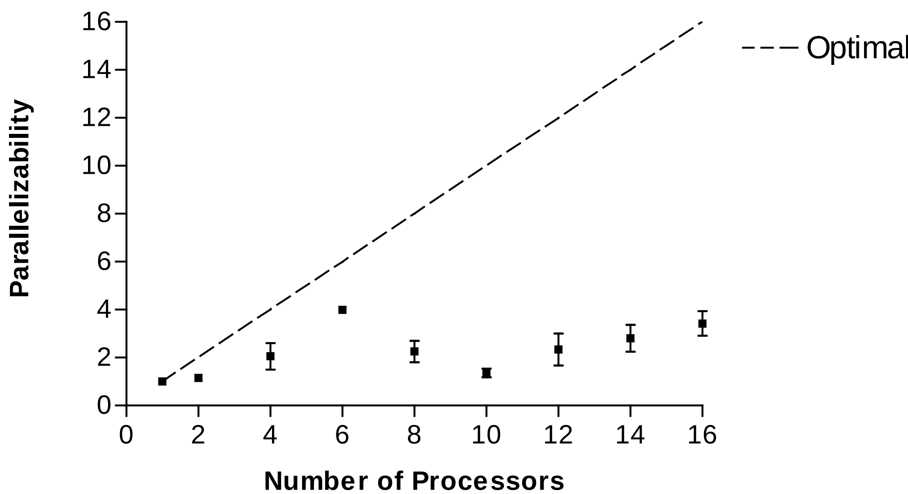


Figure 9: Parallelizability of the initial parallel version.

As one can see from figure 9, the parallelizability of the initial implementation was very poor. Running on 16 processors, we obtained a speedup of less than threefold. It was discovered that the slaves were often waiting for the master to finish writing an output frame to disk. The master was outputting to a text file and the file I/O was terribly slow. This text file output was only a temporary construct in the simulation; eventually the data would travel to the client for display. In the meantime, the text file output was replaced with a routine to output the frame to disk as a binary SGI image file.

In addition to the faster file I/O, optimization was also done on the size of packets carrying particle information between slaves and between slaves and master. The size of a particle travelling between slaves was reduced from 14 doubles to 10. The size of particles travelling between a slave and the master was reduced even further, from 14 doubles to 3. Also, the blocking MPI_Send calls used by the slaves were changed to buffered non-blocking MPI_Ibsend calls.

5.4.2 Investigating problem size

These improvements did provide somewhat better parallelizability, but we were still far from optimum. One possibility was that the simulation of 10,000 particles being used was not large enough. Increasing the problem size might well improve our parallelizability. A study was carried out measuring the efficiency of the program. If P_k is the parallelizability of the program running on k processors, then the efficiency is given by:

$$E = P_k / k$$

The simulation was run for 5,000 to 50,000 particles with a varying number of processors. Figure 10 shows that the efficiency drops off sharply for all problem sizes as the number of processors is increased. There does appear to be a sweet spot around 30,000 particles, but in general the efficiency was quite bad for any more than four processors. Our poor parallelizability does not appear to be a result of too small of a problem size.

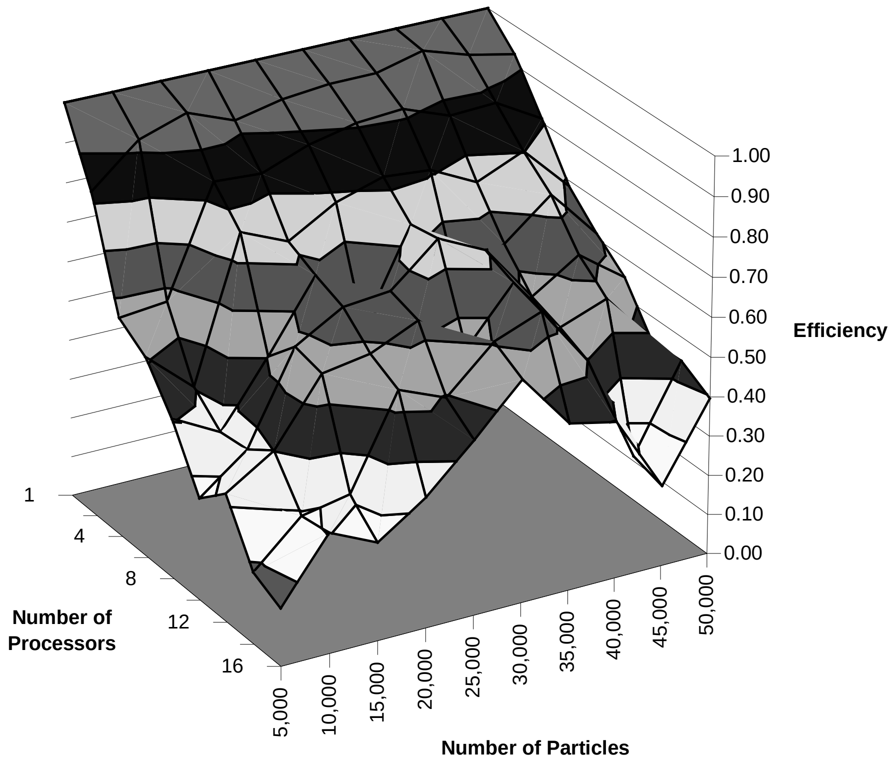


Figure 10: Efficiency of the simulation with varying number of particles and processors.

The generally improving trend as we go from 5,000 to 30,000 particles can be explained by the relative amounts of computation and communication we do for different problem sizes. For small number of particles, each processor can calculate new values for its particles quite quickly. But it must then communicate and synchronize with other processors. As we increase problem size, the computation costs grow quickly as each processor is responsible for more particles. Our communication costs grow more slowly since as the number of particles increase, the size of our grid cells decrease, thus the fringe rows that need to be communicated don't contain significantly more information.

Unfortunately, as we increase beyond 30,000 particles, our efficiency drops off. A more in depth study was carried out, measuring the parallelizability, average inter-slave packet size, percentage idle time per slave, and percentage communication time per slave. This study showed that the average packet size was increasing with problem size, but not at an alarming rate. The percentage idle time and percentage communication time were both decreasing with increased problem size.

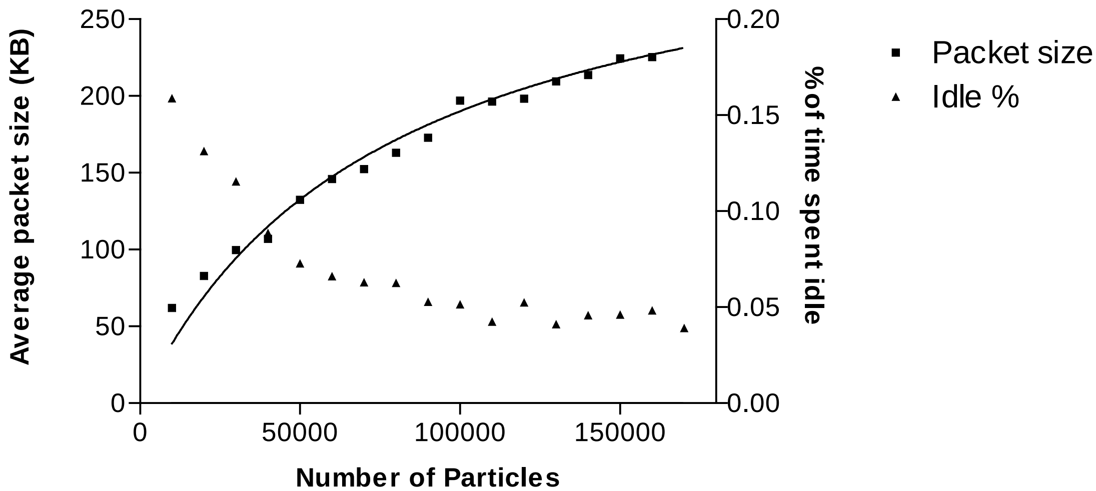


Figure 11: Average packet size and idle time as problem size grows.

We could not ascertain the exact cause of the performance drop. The increase we see in packet size is possibly significant. It could be that we exceeded some buffer or packet size limit, incurring additional delay in the sending, reception or transmission of messages. Other factors such as memory management or cache utilization may be at work, it is difficult to know for sure.

5.4.3 Switching to blocking sends

In the initial round of optimizations, the original blocking `MPI_Send` calls were replaced by their non-blocking cousin `MPI_Ibsend`. The user buffered version `MPI_Ibsend` had to be used since the normal non-blocking `MPI_Isend` ran out of buffer space and resulted in erroneous values being transmitted. The non-blocking call allowed the slaves to send their update messages north and south without waiting for synchronization with the adjacent slaves. It also allowed the slaves to send the master update messages without waiting for the master to be ready.

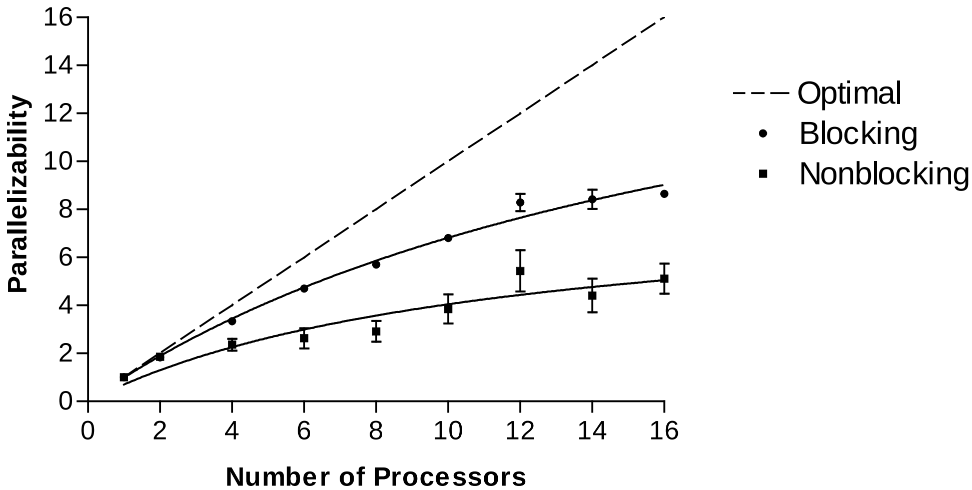


Figure 12: Performance impact of using blocking and non-blocking MPI sends.

It was just assumed that this would yield superior performance. This optimization was bundled with the reduction of packet size and the net result yielded a performance improvement. But if the blocking and non-blocking sends are compared holding other things constant, the surprising finding is that the blocking sends are significantly faster. The reason for this is unclear; MPICH's implementation of the non-blocking must be inefficient in some way.

5.4.4 Master reception order

It was also found that performance could be improved by allowing the master to receive update messages from the slaves in any order. Allowing the master to start receiving incoming packets as soon as possible reduced the time slaves had to wait for their blocking sends to complete.

An initial attempt was made using MPI_Probe to determine which slave's message was waiting and then executing a MPI_Recv with the waiting processor's number. While this method did not hang, sometimes the master would receive redundant message from one processor and no messages from another. What exactly was going on inside the MPI implementation was again not clear. The problem was resolved by eliminating the MPI_Probe call and using an increasing message tag number for each master-slave update step.

5.4.5 Exact lazy updates

Another likely candidate for the cause of our lackluster parallel performance was the inter-processor communication done at every time step. In order to prove this theory, the simulation was modified to allow "lazy" updates along processors boundaries. Instead of communicating buffer rows and migrating particles after every time step, the simulator could be set to communicate only every k^{th} time step.

These initial tests of lazy updates were purely for performance measurement, no care was taken to assure the correctness of the implementation. Figure 13 shows that the lazy updates do indeed improve performance, the parallelizability curves begin to approach optimal as we update less and less frequently.

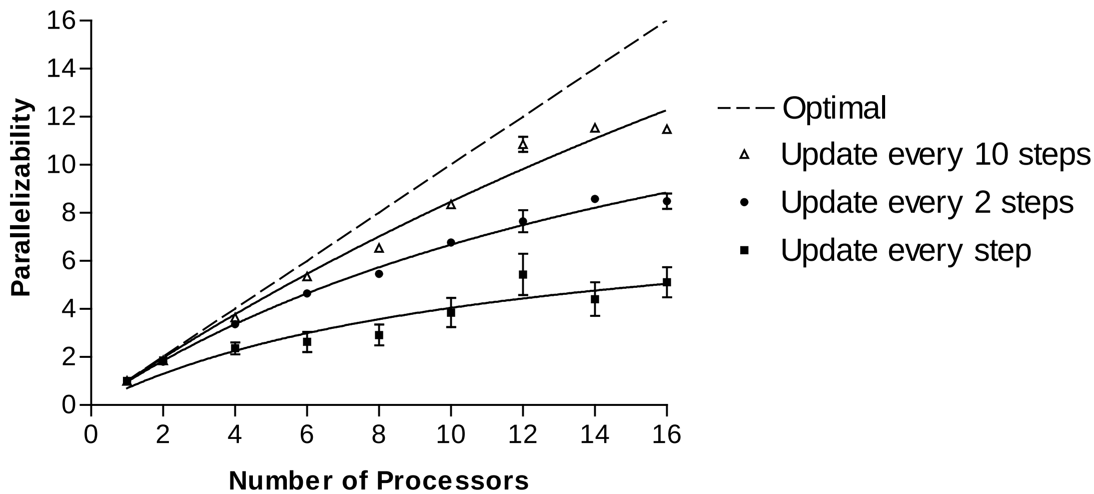
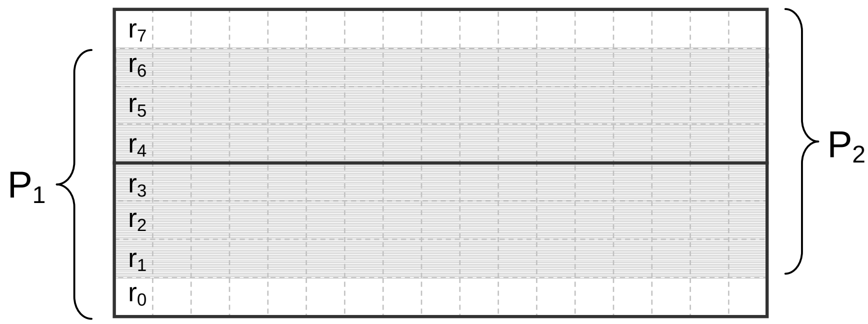


Figure 13: Parallelizability with different frequencies of inter-processor updates.

Since there was indeed performance to be gained by reducing the update frequency, we investigated how to achieve this without sacrificing the accuracy of our simulation. Our solution was to increase the amount of row overlap between adjacent processors. The following example will illustrate the technique. P_1 and P_2 will start out with identical copies of the shaded rows and will run for two time steps without sending an update message.



Time	**Actions**
t_0	P_1 can exactly update rows r_0 - r_5 P_2 can exactly update rows $r_2 – r_7$
t_1	P_1 can exactly update rows $r_0 – r_3$ P_2 can exactly update rows $r_4 – r_7$
t_2	P_1 sends P_2 rows r_1, r_2, r_3 and any particles that moved from r_3 to r_4 in t_1 P_2 sends P_1 rows r_4, r_5, r_6 and any particles that moved from r_4 to r_3 in t_1 P_1 and P_2 now have a synchronized grid again.

Figure 14: Example showing how to maintain synchronized grids without updating every step.

Note that we had to overlap three rows to extend updates to every two time steps. Initially it was thought an overlap of two rows would be sufficient for this. Consider the case that P_1 starts with only the rows r_0 - r_5. In t_0 P_1 can update exactly rows $r_0 – r_4$, but during this time step a particle could migrate from r_5 to r_4. Only P_2 would know about this as P_1 is unable to update this row (due to its lack of rows above r_5). Since our whole goal is to avoid communication during this step, P_2 cannot tell P_1 about this migrant particle. Therefore in t_1, P_1 can no longer be assured that its update of $r_0 – r_3$ will be completely accurate.

We also need to make the assumption that particles do not jump over multiple rows during a single time step. If this were to be allowed, we might need to communicate after every time step in order to send these fast moving particles to the responsible processor. A particle moving this fast is an indication that there is something wrong with the simulation and should not normally occur. We choose to simply print a warning message and delete any particle trying to jump too far.

This technique can be used to push out the need for communication as far as we wish to go. Each additional time step will require two more overlapping rows. The downside is that the greater the number of overlapping rows, the more computation each slave must do per time step. In addition, when updates do occur, the messages exchanged will be longer, proportional to the number of overlapping rows.

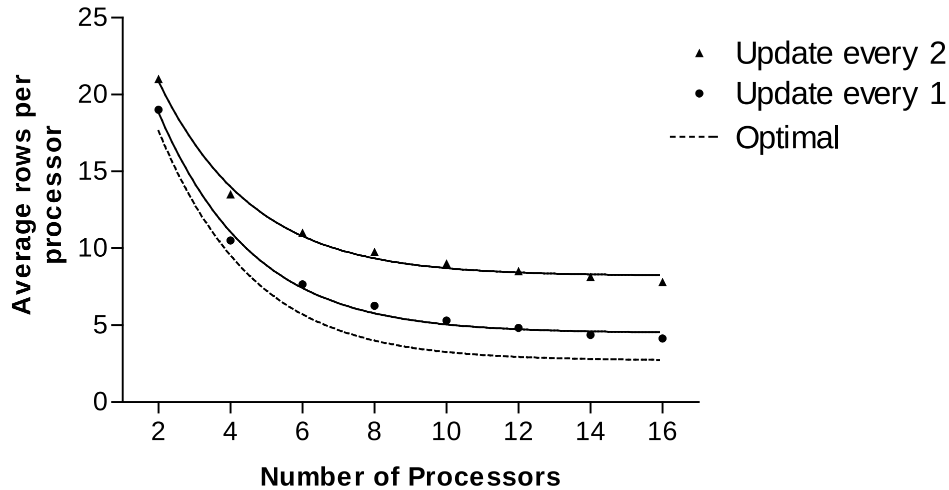


Figure 15: Comparison of the average number of rows needed per processor.

As figure 15 shows, for a simulation of 10,000 particles, the extra row overhead incurred in order to push out updates to every other time step is significant. A version of the simulation was implemented using this increased row overlap to measure whether the advantages of less frequent communications would outweigh the costs of more computations.

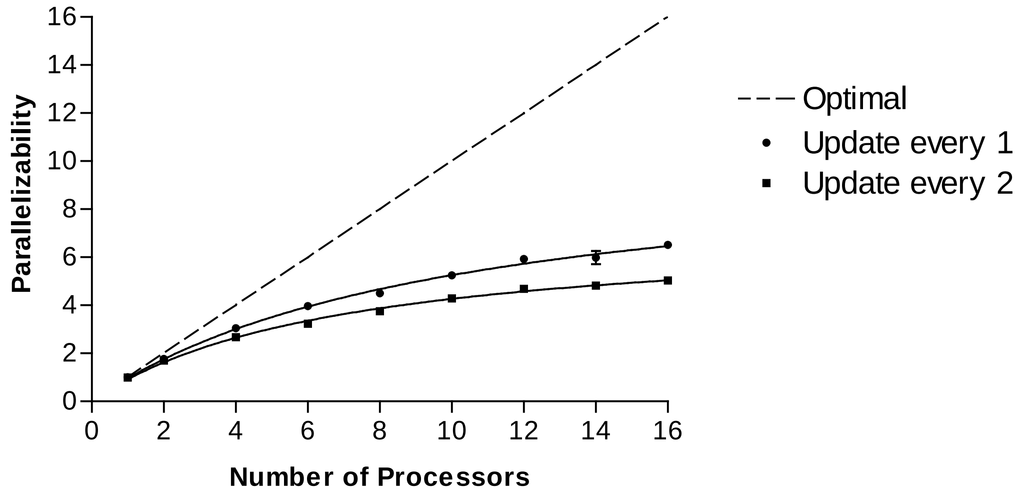


Figure 16: Performance of exact lazy updates.

Unfortunately, as figure 16 demonstrates, pushing out the communication by increasing the row overlap actually reduced our performance. For 10,000 particles, there is just too much redundant computation resulting from the extra rows on each processor. It is possible that for larger simulation sizes, this method could prove more beneficial. At this point, we scrapped this exact method of lazy updates; it added significant complexity to our code and did not perform well for our target simulation size.

5.4.6 Non-exact lazy updates

While we failed to obtain a well performing method to do exact lazy updates, if we were willing to accept some amount of error in our simulation, we could employ a simpler policy. The simplest policy was similar to the one we used initially to demonstrate the performance benefit of lazy updates. We would only update every k^{th} step, on steps in-between, use the old values of particles in the buffer rows. If particles needed to migrate between processors, we would simply hold them until we reached the next update step.

Our first attempt resulted in highly divergent results when lazy updates were employed. By analyzing the animations and using our particle comparison utility developed during the debugging phase, we discovered there were some subtle problems with the technique. We had to modify our routines responsible for pumping new particles into the simulation and utilize two linked lists to track migrant particles (one for the last time period and one of the current time period).

We measured the average positional difference between a version updating every time step and one updating less often. Eventually the number of particles between the simulation compared would differ. We chose to track the overall number of particles missing between the two runs, but we did not include these particles in our calculation of the average positional difference.

Time steps	**Update every 2**		**Update every 4**		**Update every 8**	
	Position error	Missing particles	Position error	Missing particles	Position error	Missing particles
5,000	1.556×10^{-5}	0	5.879×10^{-5}	0	1.089×10^{-4}	0
10,000	1.050×10^{-3}	3	1.643×10^{-3}	2	1.696×10^{-3}	4
15,000	3.204×10^{-3}	14	4.219×10^{-3}	24	4.122×10^{-3}	25
20,000	5.004×10^{-3}	50	6.235×10^{-3}	58	5.748×10^{-3}	49
25,000	6.408×10^{-3}	47	7.934×10^{-3}	69	7.229×10^{-3}	58
30,000	8.076×10^{-3}	74	9.668×10^{-3}	68	9.185×10^{-3}	71
35,000	9.931×10^{-3}	82	1.169×10^{-2}	105	1.161×10^{-2}	86
40,000	1.268×10^{-2}	105	1.431×10^{-2}	116	1.444×10^{-2}	125

Table 4: Comparison of error introduced by simple lazy updates.

As you can see in table 4, the simple implementation of lazy updates results in significant errors in the particle positions. If you consider that we typically had particles in the range (0.0, 0.0) to (1.0, 1.0) and displayed our results at a screen resolution of 640 x 480, a positional difference of around 1.0×10^{-3} would be sufficient to alter the plotted position of a particle. After only 10,000 simulation steps, even updating every two steps was not sufficient to attain this accuracy.

Our quantitative results showed that this simple method of lazy updates introduced noticeable amounts of error. But perhaps these numeric errors would not make a noticeable difference in the overall qualitative look of the simulation. To substantiate this claim, animations were created using update frequencies of 1, 2, 4, and 8. When the animations were compared side-by-side, little difference could be detected in the overall simulation. The fluid behaved in the same way regardless of the frequency of our updates.

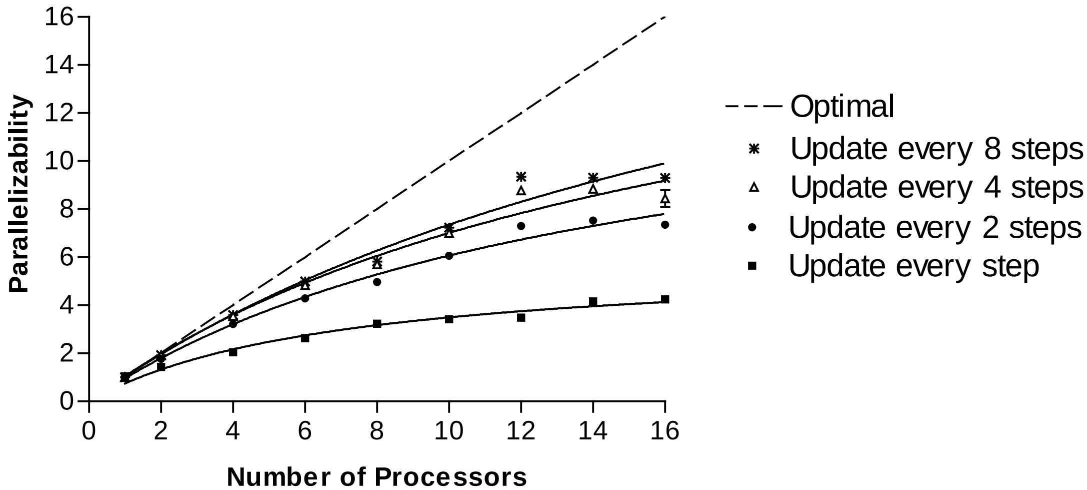


Figure 17: Performance of non-exact lazy update method.

Parallelizability timing tests were run for a simulation of 10,000 particles using update frequencies of 1, 2, 4, and 8. As figure 17 shows, updating every 2 steps produced a big gain in parallelizability. Updating every 4 steps improved things slightly more, while updating every 8 steps didn't seem to provide much further benefit.

In all cases, as the number of processors increased beyond 12, there was no further reduction in the overall time required for the computation. We believe this is the result of the increasing number of redundant rows as we increase the number of processors. For our standard simulation of 10,000 particles, the entire grid consists of 37 rows. For 16 slave processors, each slave is responsible for around 4 rows, 2 of these being redundant overlapping buffer rows.

Ideally we would like to reduce the amount of redundancy in the grid. Unfortunately this is not possible since the size of the grid cells is determined by the physics related to the number of particles in the simulation. The size of a cell represents the cut-off distance for physical interactions of our particles (the so-called smoothing length). The only way we can increase the total number of rows and achieve a reduction in the redundancy is to increase the total number of particles in the simulation.

We ran another set of parallelizability tests using 30,000 particles, the sweet spot discovered in our efficiency contour plot (figure 10). Update frequencies of 1, 2, 4 and 8 were tested. We found that as expected, this number of particles provided much better performance even without using lazy updates. As the update frequency was increased, the trend was for the parallelizability curve to improve, but not by large amounts (curves for frequencies 2 and 4 were omitted from figure 18 for clarity).

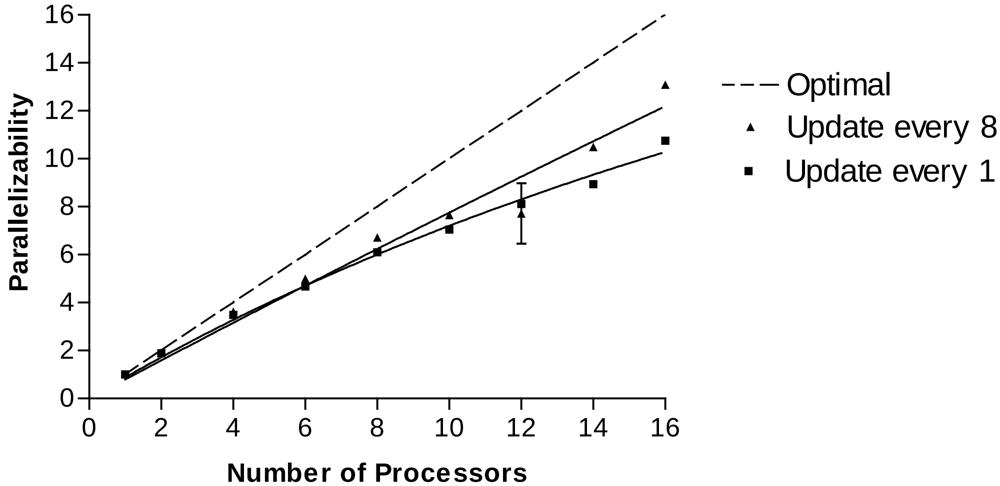


Figure 18: Performance using simple lazy updates and 30,000 particles.

Comparing the results for lazy updates with a simulation size of 10,000 particles (figure 17) and 30,000 particles (figure 18), we see that the necessity for using the lazy update feature is closely coupled to the problem size. For 10,000 particles, simulations without lazy updates perform very poorly and large gains are made by utilizing lazy updates. For 30,000 particles, simulations without lazy updates perform quite well and lazy updates only provide a small performance benefit. This is likely due to the relative amounts of time the simulation is spending in communication versus computation. For larger number of particles, each processor is spending a higher percentage of its time in computation, minimizing the overall effect of delays due to communication and synchronization.

5.4.7 Relative processor loading

Another factor that could be harming our performance is the relative amounts of work each processor is performing. Our program splits up the work as evenly as possible in the beginning, but does not reevaluate that decision during the simulation. If one processor is performing slower than the rest, it will bring the whole simulation down to its speed since processors must synchronize with neighbors after every time step.

To investigate this potential problem, we instrumented our code to measure the percentage of time each processor was idle. If we let T be the total runtime of the simulation on a processor and T_c is the amount of time a processor spent calculating (includes everything except time spent waiting on blocking sends and receives), then our idle percentage is given by:

$$I = 1.0 - (T_c / T_p)$$

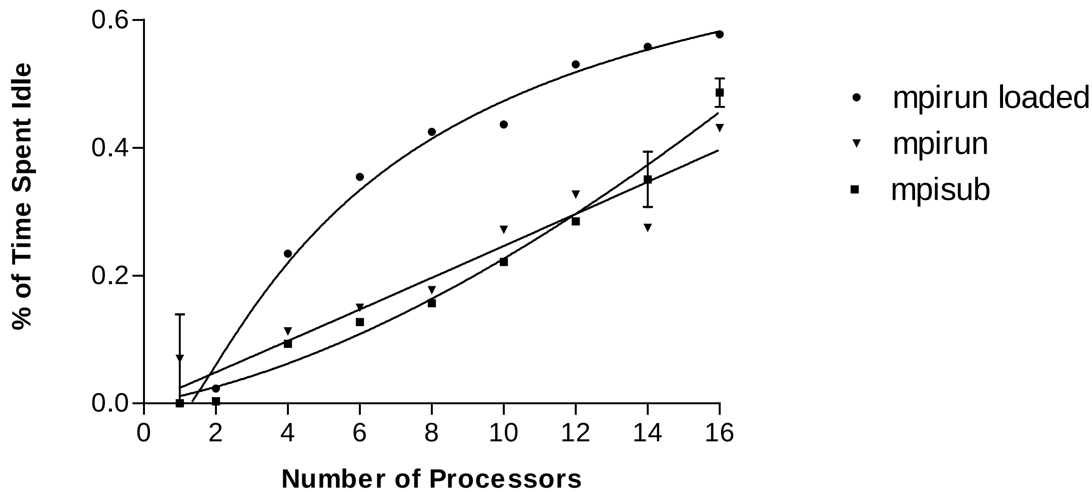


Figure 19: Idle time using different MPI dispatch mechanisms.

We started up our test simulation under three different environments. The first was our standard method using the `mpisub` command. The `mpisub` command goes through the cluster's LSF load balancing system. This assures us that the processors in our simulation will be unloaded by other `mpisub` jobs (though they could have users logged in locally taking up CPU cycles). This was compared with starting up the simulation using `mpirun`, bypassing the LSF system and loading onto processor regardless of jobs currently on those processors. Finally, we used `mpirun` but specified explicitly which processors we wanted, purposively loading two of these processors with another `mpisub` job.

Figure 19 shows that the `mpisub` job and the normal `mpirun` job have similar average idle time percentages. This indicates that at the time the `mpirun` job was running, all the nodes utilized were unloaded to begin with. When we forced the `mpirun` job onto several nodes that were loaded, we see the idle times increase dramatically. This demonstrates that our simulation's performance is highly sensitive to the relative performance of all the nodes in our MPI network.

To improve performance, we would need to implement some form of load balancing. One idea would be to have the master periodically redistribute the grid rows to all processors, giving fewer rows to processors that have been finishing last and giving more rows to those that have been finishing early. We decided against implementing load balancing in our simulation. We deemed that the added complexity would detract from the readable and reusability of code for future users of the software.

5.4.8 Performance effect of boundary shapes

Up to this point, all performance tests were conducted on a standard simulation world consisting of a cylinder in the middle of the fluid flow. Since our domain decomposition relied heavily upon the overall motion of particles travelling horizontally, the more vertical motion our fluid exhibited, the worse one could expect our simulation to perform. We conducted a set of experiments using different simulation worlds to measure how badly performance would degrade in more “difficult” situations.

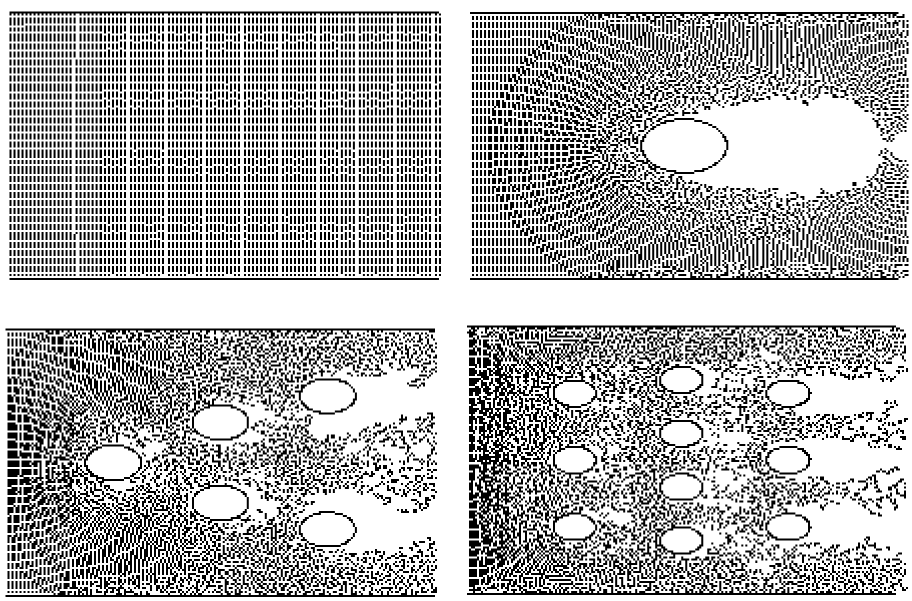


Figure 20: Four simulation worlds tested for performance.

Parallelizability tests were run on the four worlds with differing numbers of cylinders. The results were remarkably similar. The world with no cylinder performed slightly better that the other three. The worlds with 1, 5 and 10 cylinders all performed almost the same. Further investigation revealed that the average size of packets travelling between slave processors did not vary much. For 0 cylinders the average was 62K, 1 cylinder 58K , 5 cylinders 56K, and 10 cylinders 54K.

The differences in packet size appear to have more to do with the overall number of free particles in the simulation than with the configuration of the shapes. As more shapes are added, the amount of space available to initialize particles is reduced. In the case of no cylinders, we had 200-400 more free particles. While there may have been more particles migrating between processors in worlds with shapes, the cost of these migrations were minor in comparison to the costs of sending the buffer rows (which need to be communicated in their entirety regardless of the movement of particles between processors).

We concluded that we didn't need to be too concerned with the overall configuration of the shapes in our world. Obviously it would be possible to construct degenerate examples that will cause our simulation to perform badly (i.e. creating a series of shapes funneling all the particles onto one processor), but we will assume our users would use more reasonable worlds.

6 Client application

6.1 Motivation and design

Scientific computer simulations, especially parallel ones, tend to be difficult to use programs that only the creators ever use. One of our project goals was to create an easy-to-use interface to our simulation software. This would allow others to experiment with the SPH method without having to learn the complexities of running parallel programs. Our goals for the interface were as follows:

1) Cross platform, at least to the point of being able to run on a variety of workstations.
2) A user should be able to connect to the server running on SWARM without needing a CS-Research account.
3) A novice user should be able to obtain a reasonable looking simulation run without knowing much about SPH or our software.
4) The user would be able to save and load simulation parameters so as to be able to repeat and modify past experiments.
5) The user should be able to save resulting animations for future viewing.

The first two of the above design goals directly influenced the direction of our implementation. In order to be as cross platform as possible, we choose to implement the client in Java. On our campus, the JDK Java interpreter would be available on any workstation a user would be likely to be use. A disadvantage to using Java would be that we could expect slower performance than from a similar compiled C program utilizing a GUI API such as Tcl/Tk.

Our second goal of utilizing our server on SWARM from a remote machine proved to be challenging. We could not assume that the user would have an account on our CS-Research network, and even if they did, they might not be familiar with starting parallel jobs on SWARM. Our server would therefore need to be already running on SWARM, waiting for a connection from the client. Therefore, we would need some method of communication between the client and the server.

Our first thought was to use MPI as the messaging API between client and server. This proved to be problematic for several reasons. First, MPI was only available on CS-Research machines and Department of Engineering workstations. While our target users would be likely to have access to a workstation running MPI, it would limit usage of the client to only these workstations. Secondly, due to security restrictions in place on the SWARM cluster, it was not possible for an engineering workstation to communicate via MPI with SWARM. Thus if we used MPI, the client would be limited to running on CS-Research workstations.

Using socket communication was another option for the client/server communication. An investigation revealed that such socket communication was not subject to the security restrictions that prevented MPI from functioning. Using sockets also had the advantage of enabling the client to be run from literally anywhere. The client could communicate from any machine with an Internet connection. The main disadvantage to this technique was that we needed to develop a socket communication library between Java and C++ (see section 7). An additional disadvantage was that it precluded the possibility of using an applet. We would need to open an arbitrary network connection, a Java security exception for unsigned applets. An applet could be signed to alleviate this restriction, but presently the process for signing applets is annoyingly complex. We chose to write our client as a Java application.

6.2 Implementation

Developing a GUI can be a very tedious business. Each graphic element must be placed in exactly the right location and hooked up to interact with the software and other on screen elements. We decided to utilize a tool to assist in the GUI development. After evaluating several other products, we choose Symantec's Visual Café as the easiest IDE tool for developing our Java client. Visual Café provides a rapid application development environment for Java applications and applets. With little starting knowledge, the overall layouts of the GUI can be created by dragging and dropping components. Events can be specified for certain components and Visual Café provides the necessary code stubs for the listeners.

It was discovered early on that a Java application created using the AWT toolkit did not port gracefully to other platforms. An application that looked perfect in Windows NT would be nearly impossible to use in X-windows due to truncated labels, buttons, and other visual anomalies. We switched to using the Swing library [4] for our GUI components. The look-and-feel of the Swing components could be set to be the same regardless of the operating system. While we still found that there were occasionally visual quirks between operating systems even with Swing, we could trust that our application would be at least useable between different platforms.

Implementation went very smoothly, it was a refreshing change to be using Java instead of C++. We could avoid all the details of the memory allocation and focus on the functionality of the program. We were also able to implement threaded classes for reception and playback for our animation. This allowed the user to switch between different views and frames of the simulation even while new data was being received and post-processed.

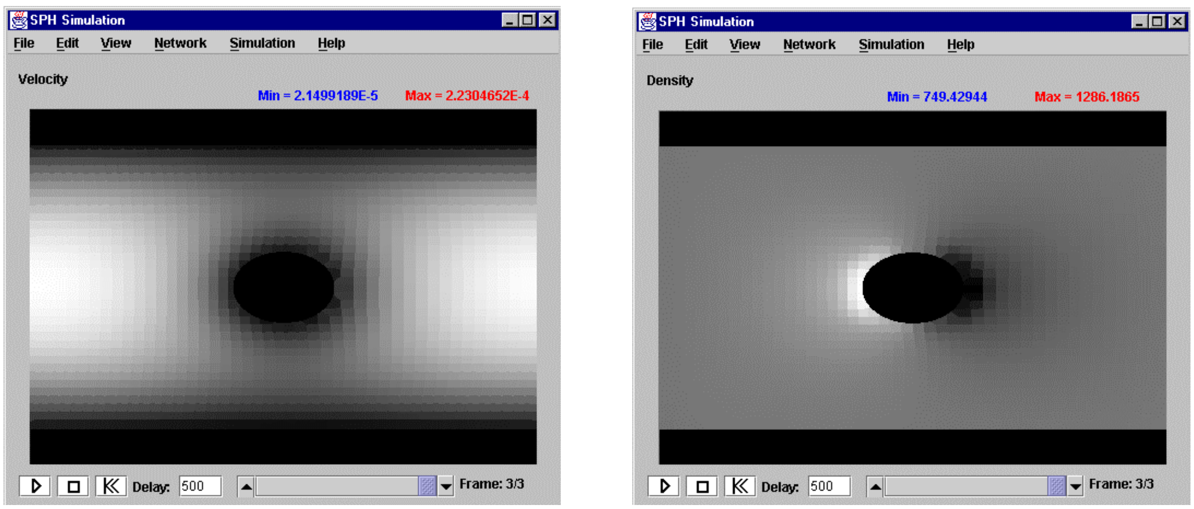


Figure 21: Velocity and density contour plot using the client application.

We decided the client should receive the floating-point data for each output frame of the simulation rather than bitmap images. This allowed the client to perform computations and visualization on the data without the need for constant server support. The client could either receive the data during a live connection to the server, or it could later load data files saved by the server. The disadvantage to this was that if the server was sending frames to the client very quickly, the client could hold up the whole simulation while it performed the generation of bitmap images from the floating-point data.

We added the ability to view not only the positions of our smooth particles, but also a contour plot of the density, pressure, and velocity. This aided greatly to our understanding of the physics of our simulation and helped us locate several mistakes in our formulas. We originally scaled the colors on our plots from blue to red based on the minimum and maximum values within a single frame. This was very misleading when the frames were displayed in sequence.

Unfortunately, there was no way to know a priori what the global minimum and maximums would be for a particular attribute. We settled for allowing the user to view plots using the local minimum and maximum and provided a rescale function so plots could later be recalculated using the global minimum and maximum.

7 Socket communication library

7.1 Design considerations

In order to facilitate communication between our Java client application and the MPI server running on SWARM, we needed a library for socket communication. A fairly exhaustive search of the web revealed plenty of existing socket libraries for C, a few for Java, but none that were designed to operate between the two languages. It was therefore necessary to design and implement our own socket communication library. We utilized the Java socket class and C socket functions for the low-level socket communication, building a higher level API on top of these. The design goals of the library were as follows:

1) Insulate the user from the low-level details of the socket communication.
2) Allow sending and receiving of a variety of data types: strings, bytes, integers, floats and doubles.
3) Provide similar interfaces to the library on both the Java and C sides.
4) Communication throughput sufficient to avoid a bottleneck in the client-server link.

7.2 Implementation

Development of the library began with learning the low-level socket calls needed in C and Java to achieve a simple 'Hello World' socket program. Modifying example code in [9] and [6], simple text socket messages could be sent between C and Java code. Once it was proved that the C and Java socket routines could indeed communicate, implementation of the actual library began.

Since the Java side would have to be designed in an object-oriented manner, in order to provide consistent interfaces, the C side would also need to be object-oriented. A `Client` class was created in Java, and a `Server` class in C++. The library would be used by calling various methods on a `Client` or `Server` object. Since both `Client` and `Server` objects would need to send and receive all the supported data types, identically named methods were created in both the Java and C++ classes. This identical naming allowed the library interface to be almost the same on both the `Client` and `Server`. The only interface difference was in the construction of the `Client` and `Server` objects: a `Client` object needs to know the port and system name to connect to, a `Server` object just needs to know the port to listen on.

Difficulties began to emerge when the first version of the library was tested for performance. The performance test involved repeatedly sending and receiving blocks of data between the `Client` and `Server`. While initial simple testing of the libraries had worked reliably, the performance test program often hung. The problem was in transmitting multiple messages in one direction. To illustrate the problem, take the example of the `Client` sending two messages consisting of 100 integers each. The `Server` receives the first message, but the socket calls may read slightly into the second message, throwing the extra integers away. When the `Server` then tried to receive the second 100 integer message, there would not be enough data and the program would hang.

The solution was to utilize acknowledge messages (ACKs) after completion of a send/receive. An ACK consisted of a single-byte message with the contents of the byte being irrelevant. After completion of each receive method, an ACK was sent to the other party. The receive method would then wait for an ACK from the other side. Symmetrically, after completion of each send method, an ACK was waited for from the other party. The send method would then respond by sending its own ACK. This simple addition eliminated the unreliable behavior, allowing the client and server to call any number of sends or receives in a row. The `Client` and `Server` programs must still have sends and receives balanced, but this is true of most communication APIs.

7.3 Performance issues and results

The first three of our stated design goals had now been met; we had an easy-to-use method for sending and receiving various types of data between our Java client and C++ server programs. Our last design goal was that our socket library must provide enough communication throughput to avoid becoming a bottleneck in our simulation infrastructure. We wanted a transfer rate fast enough to allow the server to stream the results of the simulation real-time to the client. Exactly the data rate needed was an unknown quantity until we had the SPH simulation implemented. We estimated that for a 640 x 480 animation running at 5 frames per second, one byte/pixel, a data transfer rate of 1.5 MB/s would be required.

Timing studies were performed, measuring the overall data transfer rate on large blocks of various data types. The initial results were disappointing. Our library yielded a 16 KB/s transfer rate between two machines connected by fast Ethernet (with a theoretical transfer rate of 12.5 MB/s). The initial version was performing 100 times slower than our target rate and 800 times slower than the theoretical rate.

The main source of the slowdown was the socket routines used on the Java side. Java's `Socket` class provides methods to send and receive the primitive types: `char, int, float`, and `double`. These methods can only send or receive one data item at a time. For large blocks of data, these methods were simply called multiple times. This probably resulted in extremely short and inefficient packets being sent over the network. The C side avoided this problem since its socket functions `send` and `recv` only accepted a pointer to an array of `char`s. An array of data items of any type could be sent or received by simply casting the array:

```
send(new_fd, (char *) buff, sizeof(double)*len, 0);
```

The Java side did have a `write` method capable of sending multiple bytes, but the data to be sent must be in the form of a byte array. In Java, converting an array of some other data type to an array of bytes wasn't as easy as a cast in C. For sending, the original data must be filtered through several streams to obtain a byte array:

```
ByteArrayOutputStream bytestream = new
ByteArrayOutputStream(len*8);
DataOutputStream = new DataOutputStream(bytestream);
for (int i=0; i<len; i++)
    out.writeDouble(vals[i]);
output.write(bytestream.toByteArray(), 0, bytestream.size());
```

In addition to changing to reading and writing blocks of bytes, the Java `InputStream` and `OutputStream` objects used were changed to `BufferedInputStream` and `BufferedOutputStream`. Benchmark tests were then done to test the overall transfer rate for various types of data (doubles, integers and bytes). The overall packet size was held constant at 64KB and 100 packets were sent from the server program on SWARM to a client program running on `jasper.cs.orst.edu.` These improvements resulted in 300-400 KB/s transfer rates for complex (non-byte) data types. Bytes could be sent much faster, 2.5 MB/s, indicating that the above routine converting between complex arrays and byte arrays is very costly.

Packet size / data type	Average transfer rate (KB/s), 5 trials	Standard deviation
8192 doubles	330.75	0.50
16384 integers	392.06	1.43
65536 bytes	2480.96	32.47

Table 5: Socket transfer rates between Java client and C++ server.

To test whether the interpreted Java was to blame for our slow performance, a version of the `Client` class was written in C++. The same communication benchmark tests where then run using the C++ `Server` and C++ `Client` classes.

Packet size / data type	Average transfer rate (KB/s), 5 trials	Standard deviation
8192 doubles	2050.34	85.30
16384 integers	2081.45	45.66
65536 bytes	3332.04	93.57

Table 6: Socket transfer rates between C++ client and C++ server.

Comparing the results in table 5 and 6, we see that for doubles and integers, a six-time increase in throughput is seen using C++ on both sides. However, the performance gain for bytes is much less pronounced. This provided further evidence that the performance problem with double and integers is in the translation of the original data into a byte array. The overall speed of Java does not appear to be a significant limiting factor.

The library so far was using only connection-oriented methods of network communication. While these connection-oriented services provide guaranteed delivery of messages, they also involve extra communication overhead. The `Client` and `Server` classes were further extended to allow sending and receiving of datagrams, connectionless network communications. A transfer rate of 3097 KB/s was obtained using a combination of datagrams and acknowledgement messages sent by guaranteed means. The acknowledgement messages were necessary to insure any packets that disappeared in transit would be resent.

While we now had achieved our target rate of 1.5 MB/s, the library was still performing well below theoretical limits. The reasons for this are not clear. It may involve details of the network and operating system that are limiting the throughput. It could be that the send and receive socket routines are still breaking up each large packet into many smaller, inefficient network messages.

7.4 Example application using socket library

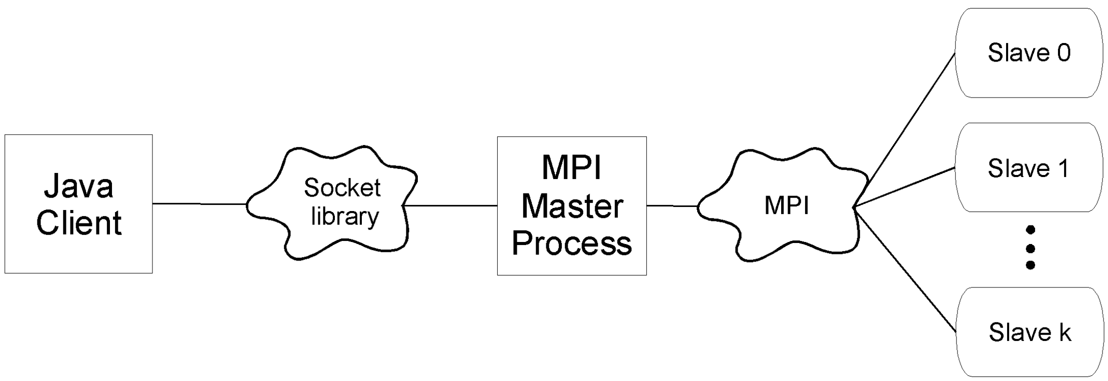


Figure 22: Communication architecture of the SWARM Performance Monitor.

With the underlying socket classes complete, a simple application was created to demonstrate the viability of our overall project communication architecture. Client and server programs were developed to do a simple performance benchmark on nodes of SWARM. The client program was designed to do the following:

1) Establish a socket communication link with a remote server system on a specific port.
2) Receive the names and a megaflop performance rating of all nodes running in the server's network.
3) Display the received data

The server program would be a MPI program consisting of a master process and a number of slave processes. The master would do the following:

1) Accept a socket connection from the client on a specific port.

2) Receive commands from the client. The client could tell the server to: run a performance test on all its nodes, shutdown the server completely, or close the socket connection and wait for a new one.
3) Inform the slave processes when they needed to perform a benchmark or if they were to shutdown.
4) Collect benchmark data from the slaves and transmit to the client.

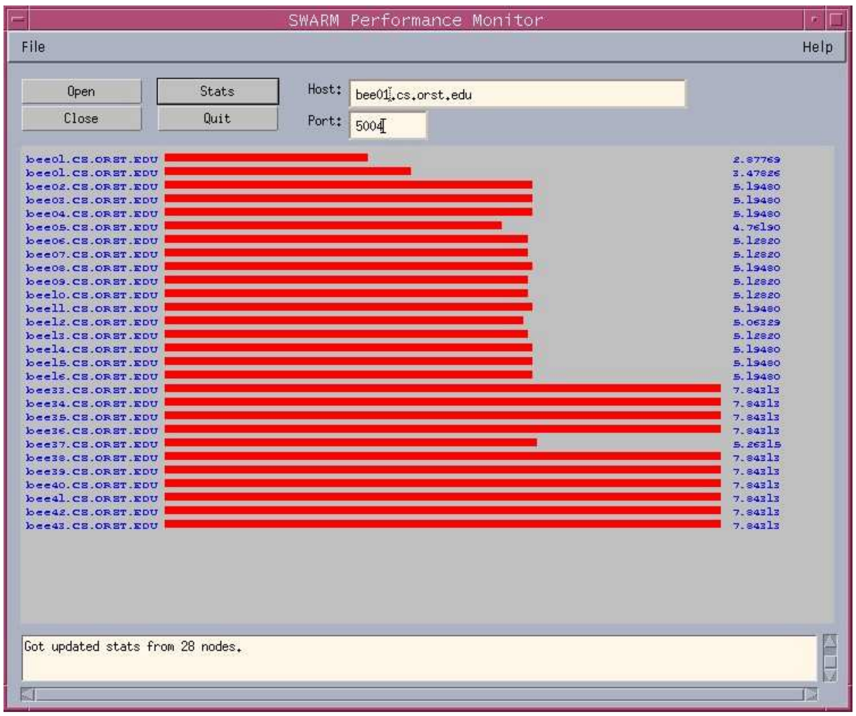


Figure 23: SWARM Performance Monitor application

Since the socket libraries developed may be of use to others, the code has been made available on the web [20]. In addition to the Java client, C++ server, and C++ client written for this paper, a Java server was also written to complete the set. Sample programs demonstrating the basic use of each class were created in both Java and C++ and a Makefile provided.

8 Conclusions

8.1 Meeting our objectives

When we began this project, our objectives were to produce an easy-to-use learning tool for students and also to provide an extendable parallel implementation of SPH. Did we meet these objectives?

For our simulation to be a useful learning tool for students, it should operate at interactive speeds. Someone just experimenting with the fluid dynamics problem is not going to be motivated enough to wait hours to see the results of a change in simulation parameters or object placement. As we discovered, the SPH method requires too much computation to provide the sort of speed necessary for an interactive simulation.

While our simulation may not be of much benefit to the average fluid dynamics student, it may be valuable to researchers in fluid dynamics. People unfamiliar with SPH can use our software to setup and run simulations, without needing to code the details of the method themselves. These users will be much more tolerant of the time involved to generate results and will probably have access to more power computing resources than the typical student.

We have succeeded in implementing the SPH method in parallel using MPI. As far as we know, we are the first to make freely available source code for an MPI parallel SPH simulator. We have gone through multiple revisions of our source code with the express purpose of improving the understandability for future users. While any software of this magnitude is bound to be complex, we have isolated the physics, MPI communication, and client-server communication such that modifications can be made by others without having to understand the entire program.

8.2 Lessons learned

The lure of quick coding can be an overpowering force. Often during the project we fell prey to this temptation, leading to poor design and numerous bugs. While at the time, it seems that formal design documents and procedures will unnecessarily slow down the development, I’m confident they would have resulted in a net time savings due to a much shorter debugging phase.

The experience of working on a software development team (albeit a small two-person team) has been a learning experience. Simple things like different naming styles for class methods and attributes can become a major headache. It became difficult to guess the correct combination of capitalization, underscores and abbreviations that were used inconsistently throughout the source. Standards for coding and commenting should be established early in the software development cycle.

We would have also benefited from a more sophisticated revision control system like RCS. The system we used involved periodically switching working directories, leaving copies of old source code intact. While we could back out to an older version, it was difficult to exactly enumerate the changes made between versions. RCS would have encouraged us to document changes during every check-in, providing a better audit trail. If changes were being made to code by both of us at the same time, we would periodically merge the changes using the Unix utility `diff`. RCS would have not helped much here as often changes were widespread enough that they would have required check-out of a large subset of the source files, precluding simultaneous development.

We learned that tuning the performance of a parallel program requires a lot of experimentation and guesswork. Sometimes the best results are obtained by counter-intuitive actions (such as using blocking send and receives instead of non-blocking). But the fundamentals are still the most important; reduce redundant computations and global synchronizations.

8.3 Future work

Further improvements could be made to the parallel performance of the simulation. Utilizing some sort of load balancing scheme would likely improve performance. It would also be interesting to investigate other distributions of the simulation grid to the processors. Our division into strips causes more cell overlap than a division into square blocks. Since we found that the amount of migration between processors had little effect on performance, a square decomposition may be the most efficient.

Performance could also potentially be improved by a more careful analysis of cache coherence issues. While we tried to order our calculation in such a way as to stay as local as possible, we still made several passes over our particle arrays. It may be possible to combine these loops and update a particle without these repeated passes. A tool to measure the cache hit/miss ratio of each processor's cache would be very helpful in such an optimization effort.

The client side application we developed in this project was basic. There is a lot of room to improve and expand the capabilities of this application. The shapes in the simulation world could be created and modified visually, rather than with the text-based editing method we employed. Additional visualization tools could be added such as tracking of individual particles, graphing physical attributes over time, and plotting forces felt on the obstacles.

We have provided a code foundation for others to extend and use our software for SPH based simulations. Our code may also be useful as a starting point for others solving n-body problems with a cut-off distance. The physics of SPH could be replaced by other equations to model different physical phenomenon. It is our hope that others will build upon our work, investigating new and interesting problems.

9 References

[1] Barnes, J., L.E. Hernquist, and P. Hut. *An Environment for Experiments in Stellar Dynamics.* BAAS 20, 1988.

[2] Becker, Donald J., et al. *Beowulf: A Parallel Workstation for Scientific Computation.* Proceedings, International Conference on Parallel Processing, 1995

[3] Benz, W. *Smooth Particle Hydrodynamics: A review*. Nato Workshop, 1989.

[4] *Creating a GUI with JFC/Swing.* http://java.sun.com/docs/books/tutorial/uiswing/index.html

[5] Davé, Romeel, John Dubinski, and Lars Hernquist. *Parallel TreeSPH.* New Astronomy, July 1997.

[6] Deitel & Deitel. *Java: How to Program*, Prentice-Hall, Inc. (1998)

[7] Dongarra, J., et al. *An Introduction to the MPI Standard*. Communications of the ACM, January 1995.

[8] Gingold, R.A. and Monaghan. Smoothed Particle Hydrodynamics: Theory and Application to Non-spherical Stars, MNRAS 181 (375), 1977.

[9] Hall, Brian. *Beej's Guide to Network Programming.* http://www.ecst.csuchico.edu/~beej/guide/net

[10] http://dir.yahoo.com/Business_and_Economy/Companies/Engineering/Mechanical_Engineering/Software/Fluid_Dynamics/

[11] http://www.dynaflow-inc.com/

[12] Lucy, L.B. *A Numerical Approach to the Testing of the Fission Hypothesis.* Astrophysical Journal 82 (1013), 1977.

[13] Monaghan J., et. al.. *Simulation of Free Surface Flows with SPH.* ASME Symposium on Computational Methods in Fluid Dynamics, Lake Tahoe, June 19-23, 1994.

[14] Monaghan, Smooth Particle Hydrodynamics. Annual Review of Astronomoy and Astrophysics 30, 1992.

[15] *Overview of SMOOTH*. http://www-hpcc.astro.washington.edu/tools/SMOOTH/

[16] Quinn, Michael J., *Parallel Computing: Theory and Practice.* McGraw-Hill, Inc., 1994

[17] Schlatter, Brian. *A Pedagogical Tool using Smoothed Particle Hydrodynamics to Model Fluid Flow Past a System of Cylinders.* MS thesis, Oregon State University, June 11, 1999.

[18] *Smoothed Particle Hydrodynamics at GMU.* http://www.science.gmu.edu/~ahaque/math/sph.html

[19] Takeda, Hidenori, et. al. *Numerical Simulation of Viscous Flow by Smoothed Particle Hydrodynamics*. Progress of Theoretical Physics 92 (5), 1994.

[20] Vertanen, Keith. *Java / C++ Socket Class*. http://www.cs.orst.edu/~vertanen/socket.html

[21] White, F.M. Fluid Mechanics, McGraw-Hill, 3rd edition.