A Survey of Performance Optimizations for Titanium Immersed Boundary Simulation

Hormozd Gahvari, Omair Kamil, Benjamin Lee, Meling Ngo, Armando Solar

Computer Science Division, U.C. Berkeley
CS267 Final Project, Spring 2004

Abstract. This paper surveys a series of optimizations for an Immersed Boundary Method simulation of a mammalian heart implemented in Titanium. In particular, we review four optimizations for (1) Load balancing, (2) Aggregating communication, (3) Second Order Method, (4) Fluid partitioning in two dimensions. These four optimizations were implemented separate, but each suggests significant potential to improve execution time for larger Immersed Boundary Method simulations.

1 Introduction

We survey a series of optimizations for an Immersed Boundary Method simulation of a mammalian heart implemented in Titanium, a parallel dialect of Java. In particular, we four optimizations: (1) Load balancing by breaking fibers, (2) aggregating communication for sources and taps, (3) implementing an alternative second order method, and (4) partitioning the fluid cube in two dimensions. These optimizations generally seek to improve the performance of an immersed boundary simulation by reorganizing communication and improving scalability to a large number of processors.

The Immersed Boundary Method independently solves the Navier-Stokes equations that govern fluid behavior and elastic material behavior in the Eulerian and Lagrangian reference frames, respectively. A particular application of the Immersed Boundary Method is the a simulation of a mammmalian heart. The original simulation employed a $128x128x128$ fluid grid and over 100,000 fibers on the Cray C90. In this paper, we survey performance optimizations to enable the simulation of a larger fluid grid with more processors.

Effective load balancing will be critical to the scalability of a larger Immersed Boundary Simulation. We suggest a scheme to distribute fiber points among the processors, effectively distribute the computation since each processor must compute on the same number of fiber points. This scheme must also minimize the communication between disconnected fiber points in the same fiber where the costs of communication are approximated by the costs of the cuts in the graph. We find that our graph partitioning schemes are effective in reducing execution by as much as 15%.

Taps take sample measurements of fluid pressure or velocity and averages these samples for points in the region covered by the tap. Data for points not local to a given processor must be communicated. Similar communication also occurs when updating source points. Current implementations communicate data for each point. We suggest aggregating the data before communication to reduce associated overhead. Aggregating communication can improve execution time for tap sampling and source spreading by as much as a factor of $5x$.

We propose an implementation of the Second Order Method, an alternative solution method, for the heart simulation. We present the equations of motion for this alternative method. This Second Order Method offers improved accuracy that may allow a coarser time step in the simulation.

Lastly, we present a scheme to increase the processor scalability of the heart simulation by partitioning the fluid cube in two dimensions. This technique requires modifications to the FFT in the Navier-Stokes solver and the preceding exchange of ghost planes. We show linear speedups for the FFT while preliminary analyses indicate comparable execution times for ghost plane exchanges.

These four optimizations were implemented separately, but each suggests significant potential to improve execution time for larger Immersed Boundary Method simulations. Future work will be the integration of these techniques into a single implementation of the heart simulation.

2 Immersed Boundary Method

The Immersed Boundary Method was developed by Charles Peskin in the late 1970s. In the words of Peskin: "The Immersed Boundary Method is both a mathematical formulation and a computational method for the biofluid dynamic problem. In the immersed boundary formulation, the equations of fluid dynamics are used in an unconventional way, to describe not only the fluid, but also the immersed elastic tissue with which it interacts."

The basic idea of the method is to independently solve the Navier-Stokes equations that govern the behavior of fluid over a regular grid, and the equations governing the behavior of elastic material in a Lagrangian reference frame. To understand the difference between the Eulerial reference frame under which the Navier-Stokes equations are solved versus the Lagrangian reference frame over which the equations governing the behavior of the elastic material are solved, consider the difference between studying fluid flow by putting a sensor at a certain point in the fluid and measuring the fluid velocity at that point (Eulerian) versus studying fluid flow by putting a buoy in the fluid and examining the buoy's drift in the fluid. The key to making the method work is to link the Lagrangian and Eulerian components of the fluid through a "smoothed version of the Dirac delta function." Through these functions, the elastic forces generated

by the elastic material can be applied to the fluid, and the fluid velocities can be interpolated to determine how the material is going to move and deform.

2.1 Applications

The immersed boundary method has been used in a number of applications from simulations of the inner ear to studies of swimming that involve a wide range of organisms. The study of fluid flow in the mammalian heart, one of the most famous uses of the immersed boundary method, is the method's motivating application.

2.2 Implementations

In 1993, Charles Peskin and Dave McQueen were able to simulate a synthetic model of the heart using a $128x128x128$ fluid grid and over 100,000 fiber points. The model ran on a Cray C90 at the Pittsburg Supercomputing Center. There is currently an ongoing effort at the University of California, Berkeley to develop a scalable massively parallel implementation of the heart simulation that will allow us to use much finer fluid meshes and more detailed heart models. The application is difficult to parallelize efficiently because the combination of the Eulerian and Lagrangian representations implies the grid representing the elastic material is going to move through the domain while the fluid grid is static. In a distributed memory machine where both fluid and material grids are distributed, the fiber grids and the corresponding fluid grids with which they must interact may reside in different processors for every interaction step. This separation of fibers and fluids makes the interaction phase very communication intensive. Furthermore, the baseline implementation of the immersed boundary method use an FFT based Navier-Stokes solver that has problems scaling to many hundreds of processors. Several of these issues in method scalability are addressed in this paper's discussion of optimizations for the immersed boundary method applied to the heart simulation.

3 Load Balancing

Heart fibers consist of a list of fiber points and the fluid surrounding the heart is represented by a $128x128x128$ lattice. Each fiber is a one-dimensional array of points that reside in 3-space within the $128x128x128$ grid. We can imagine these points in space as a graph, where edges exist between those points that are adjacent to each other on the same fiber. Fiber points must behave according to the tensions in the fiber and each point in a fiber may be affected by a neighboring point. Force calculation is done by looking at a neighboring point. The forces of the fiber points are then projected onto the fluid lattice where the fluid velocity and pressure are updated before updating the fiber point positions.

3.1 Breaking Fibers

Currently the fibers are being passed out in a round-robin fashion among the processors. Each processor that does not contain a valve flap will receive fibers, passed out in chunks of ten. Although this scheme distributes the fibers evenly, it does not evenly distribute the fiber points since each fiber consists of a different number of points. In order to achieve load balancing, we must split the fibers and evenly distribute the number of fiber points among the processors. This scheme will distribute the amount of work evenly since each processor must compute forces on the same number of points.

It will not do to just arbitrarily split the fibers and distribute the fiber points. We also want to minimize the amount of communication by minimizing the amount of cuts we make on the fibers, since information exchange must now occur because the fiber pieces live on different processors. It is not necessarily true that the lowest cost cut will equal the volume of communication, but finding the best cut will give us an approximation of the amount of communication involved. In order for force calculations to occur, each fiber point must know the location of its neighbor, and if its neighbor resides on another processor then communication must occur. Therefore we want a mapping of the fiber points onto the processors such that each processors contains more or less the same number of points, the number of non-contiguous sub-domains is minimized, and the cost of the edge cut is minimized. To do this we would need to solve the graph partitioning problem, but since the graph partitioning algorithm is NP-complete, we will use an approximation algorithm that creates high quality partitions.

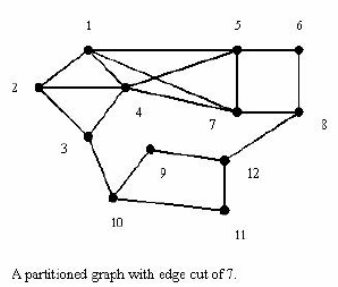


Fig. 1. Graph Partitioning: Partition 1 (1,2,3,4), Partition 2 (5,6,7,8), Partition 3(9,10,11,12). Partition 1 sends data for points 1,4 to partition 2. Partition 2 sends data for points 5,7 to partition 1. Partition 1, 3 exchanges data for points 3,10 and partition 2,3 exchanges data for points 8,12. For an edge cut of 7,8 communications are incurred.

In addition to the edges that already exist within a fiber, we also want to create edges for points that are close enough. All points exert a force on a $4x4x4$ cube of fluid cells that surrounds the fiber point. This force is added to each cell and depends on the distance of the cell from the fiber point. Therefore cells that are a distance of 4 or greater will not by affected by the fiber point. After the velocities have been calculated, the velocity of the fiber point is updated. If two points are close enough such that their fluid cells overlap, then they will have an affect on each other since their velocity updates depend on their surrounding fluid cells. Therefore we want these points to reside on the same processor; otherwise we will have to incur communication costs.

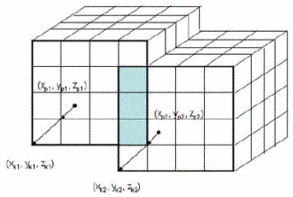


Fig. 2. Points with overlapping fluid cells

In order to create meaningful partitions, we must weight the edges of the graph accordingly. It is important that fiber points on the same fiber reside on the same processor and it is also important for fiber points with overlapping fluid cells to be on the same processor. Therefore we will weight the edges as such: if two points are adjacent on the same fiber, then the edge between those two points will have a weight of 64. In addition, if two points have overlapping fluid cells, then the edge between those two points will have a weight that is proportional to the amount of overlap. If the two points happen to be adjacent on the same fiber, then we add this additional weight to the existing weight. This scheme gives more weight to those points that are adjacent on the same fiber.

The initial algorithm we used to create the graph required looking at every other point for each fiber point. We first create edges for points within the same fiber. In order to create edges between points with overlapping fluid cells, we had to look at all other 500,000 fiber points. For an n2 algorithm, this problem size proved to be too large, generating over 5 gigabytes of data after 20 hours of computation. We then came up with an algorithm that limits the number of points (within a constant) that each point must examine.

The first algorithm we devised begins by sorting all points according to their position in space. We set a window size of N and for each point p, we look at

the N closest points to p. If any of those points are within a distance of 4, then we create an edge between those two points with weight equal to the volume of overlap. This algorithm gives us better performance than the n2 since the search space has been greatly reduced. Also, we can still achieve high connectivity because we are still looking at nearby points.

The second algorithm we devised creates 1283 buckets to store points. Each bucket corresponds to a cell in the fluid lattice and is identified by the coordinates of a point in that cell. For example, the point in (x, y, z) belongs to the bucket identified by the tuple (floor(x), floor(y), floor(z)). Hence each point belongs to a particular bucket. Once we are finished separating all points, we look at points in neighboring buckets to determine the extra edges to be added. Visually, this corresponds to looking at the cells that are adjacent to the six faces of the fluid cell.

The partitioning scheme is the deciding factor in the amount of communication and the load balancing. As mentioned above, we would like to partition the points such that each processor gets the same number of points as well as achieving the highest connectivity possible within a partition. For this we used Metis, which is a software package for partitioning large irregular graphs and large meshes as well as computing fill-reducing orderings of sparse matrices. The partitioning algorithm used in Metis is a recursive multi-level partitioning algorithm. The algorithm proceeds in a series of coarsening and refining phases, where the graph is coarsened by collapsing vertices and edges, partitioning the smaller graph, and uncoarsening the graph to refine the partition. Hence, Metis provides a tool that will create balanced partitions with least cost edge cuts and partitions with as much connectivity as possible.

After the points have been partitioned, the implementation proceeds by having processor 0 read through the partition file once to determine the processors on which the points will reside. As we are processing the points, we keep a current fiber segment. Points are added to the current fiber segment until we find a point that belongs to a different segment, in which case we will store the current fiber segment in a vector, create a new fiber segment, and continue processing points. Because all points are associated with a particular partition and with a particular fiber, we can easily determine when to create a new fiber segment. These segments are marked with the processor number on which they are to reside, so that all once the fibers are partitioned, each processor can grab its local copy.

An interesting optimization that we made was to split fibers on points rather than between points. Hence each segment will receive a copy of the end point of the previous segment. The idea behind this is to avoid the extra communication incurred when computing forces between points on the same fiber.

3.2 Evaluation

Below are the results for runs on Seaborg. Version 0 refers to the original implementation where fibers are passed out in a round-robin fashion. Version 1 refers to the algorithm in which points are first sorted before creating the graph, and Version 2 refers to the implementation where the graph is created using the spatial algorithm. All runs were done using 32 processors across two nodes. The times reported are in seconds.

	Version 0	Version 1	Version 2
ComputeForce	0.758221	0.770179	0.76153
SpreadForce	32.432274	32.126202	32.096788
cache.pack force to mbox	32.544188	32.17033	32.144611
mbox.send force	32.771772	32.234151	32.200682
Copy slab to workspace	0.585233	0.440549	0.441231
Locate Taps and CalcSources	0.898316	0.740991	0.740942
Upwind	4.158366	4.416908	4.226749
SolveTransformedEqns	1.668562	1.65828	1.670548
Copy fluid velocity	0.713853	0..719766	0.714237
Set Taps and copy Workspaces	1.44449	1.441897	1.4394
NS solver	47.730553	42.334408	41.885495
Pack velocity slabs	57.383379	43.967843	43.176888
mbox.send velocity	57.569192	44.027209	43.231476
Cache.unpack velocity from mbox	57.628768	44.05131	43.255839
Total move	83.803861	70.383689	69.551618
Time step	525.555373	446.100521	441.26001

Fig. 3. LoadBalancing: Reference on Seaborg

Both Version 1 and Version 2 are 15% faster than Version 0 in terms of total running time per time step. The most noticeable improvements are in the force updates of the fiber points, namely in sending velocity data and updating the position of the points (Pack velocity slabs, mbox.send velocity, cache.unpack velocity, and move). Because the fiber points are localized on each processor, the amount of communication needed between neighboring points has decreased. Also, the algorithm used in Version 1 to create the graph is comparable to the one used in Version 2. Although it is not as accurate, the approximation is decent when considering the faster running time.

It is interesting to note that Version 1 creates an additional 34170 fiber points (7.06% more), while Version 2 creates 30043 fiber points (6.21% more). The creation of more fiber points indicates that Version 1 is creating more fiber segments, which in turn slows down the performance as more segments will incur more communication costs. While partitioning the fibers helps with load

	Version 0	Version 1	Version 2
ComputeForce	0.032075	0.037904	0.037989
SpreadForce	0.359096	0.391023	0.394634
cache pack force to mbox	0.398675	0.405619	0.412732
mbox send force	0.569192	0.469246	0.468845
Copy slab to workspace	0.005553	0.005366	0.005806
Locate Taps and CalcSources	0.061956	0.069067	0.740942
Upwind	0.030559	0.035103	0.067116
SolveTransformedEqns	0.013364	0.012898	0.013729
Copy fluid velocity	0.011281	0.013936	0.015206
Set Taps and copy Workspaces	0.122017	0.127954	0.127931
NS solver	1.067794	0.977093	0.977676
Pack velocity slabs	1.164268	0.995194	0.991934
mbox send velocity	1.346457	1.053045	1.044248
Cache.unpack velocity from mbox	1.384439	1.068352	1.059137
Total move	1.626746	1.304554	1.295438
Time step	10.206199	8.498164	8.467624

Fig. 4. LoadBalancing: Optimized on Seaborg

balancing, we can try another optimization that will minimize communication. If a fiber point is spreading force on fluid cells that do not reside on the same processor, then the force data must be sent to the owning processor. By placing the fiber points on the same processors as the fluid cells in which they affect, we can effectively reduce the communication. Unfortunately, this scheme allows for poor load balancing because the fiber points are clustered in the center of the fluid grid. Much work remains in finding the best weighting scheme that will produce the best partitions. Other improvements can be made in the graph creation algorithm itself by finding ways to produce better quality graphs without sacrificing running time. Our current implementation does not take valve flaps into consideration, opting instead to place the six valve flaps on the middle six processors and distributing the partitions among the rest of the processors. Future work would include splitting the valve flaps to achieve better even load balancing.

The figures below demonstrate the effectiveness of the graph partitioning schemes employed. Each figure shows fibers from a different set of processes. Fibers local to different processes are colored differently. Note that the colors are concentrated together, indicating that our partitioning schemes logically preserve spatial locality.

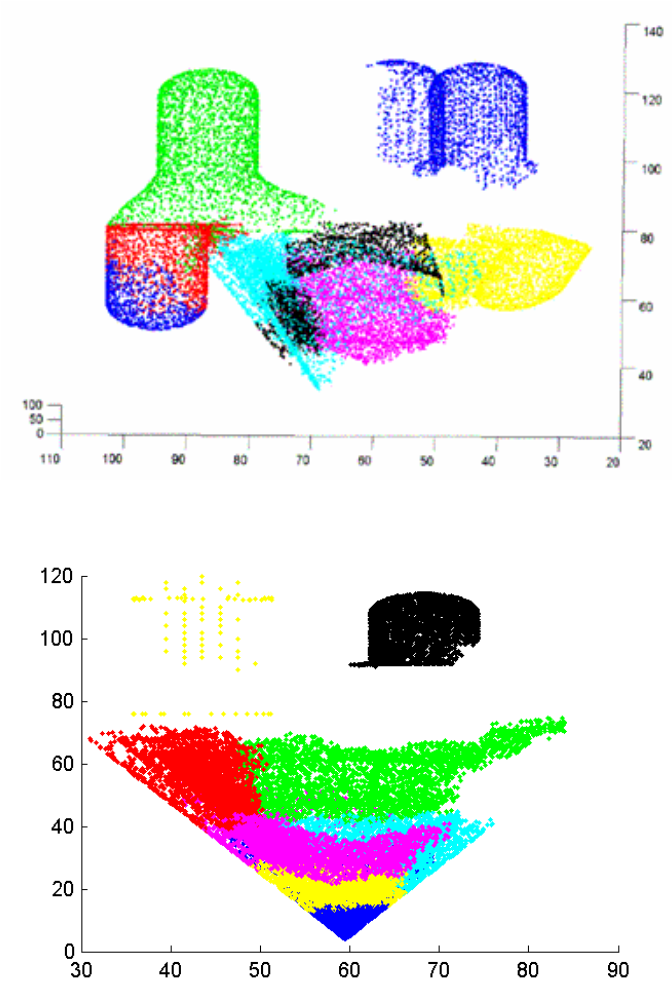


Fig. 5. Load Balancing: Processors 0 to 6 (above) and Processors 7 to 19 (below)

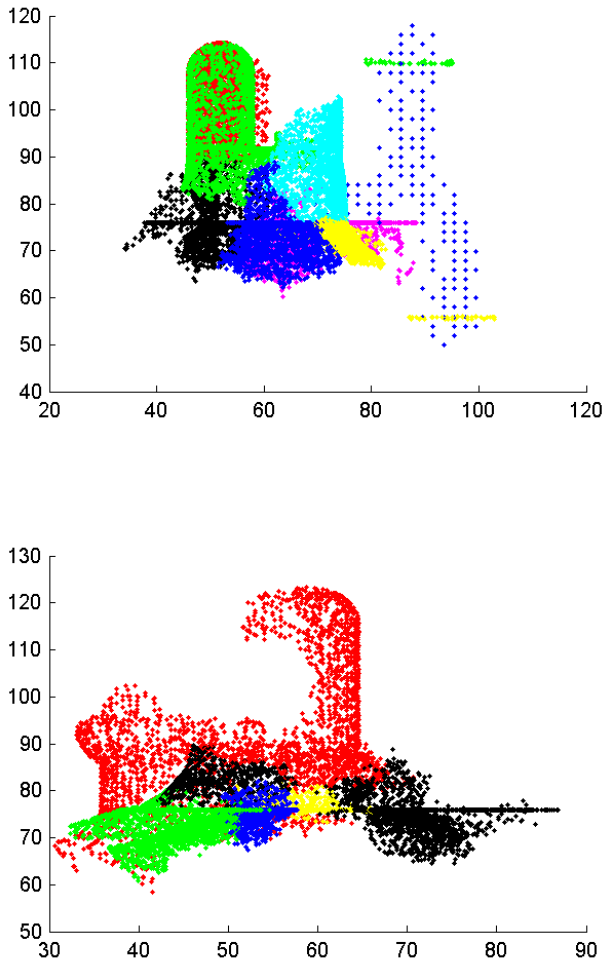


Fig. 6. Load Balancing: Processors 20-26 (above) and Processors 27-31 (below)

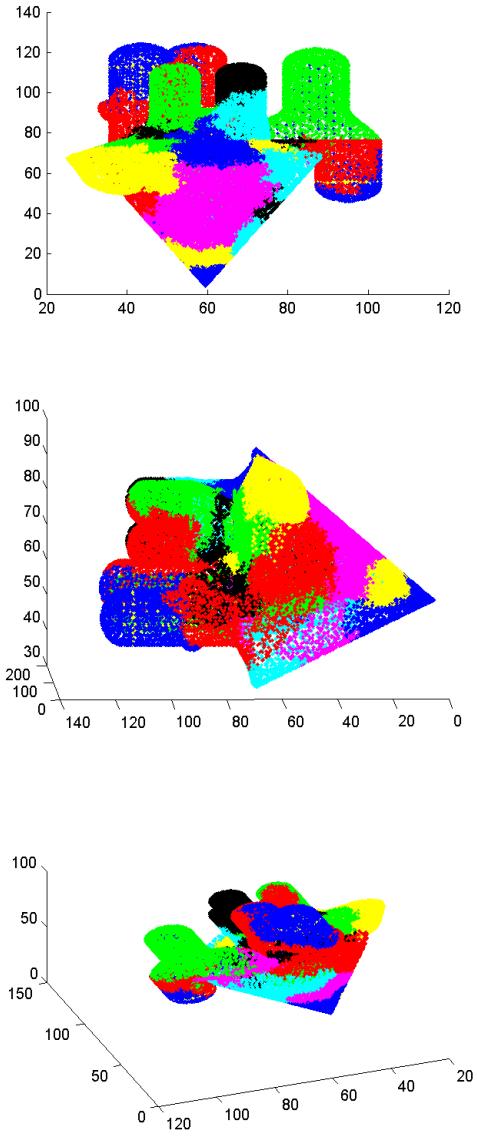


Fig. 7. Load Balancing: All processors from varying angles, excluding processors for valves

4 Taps and Sources

Taps are special objects covering regions in the fluid grid. Taps are to take a sample measurement of the pressure or velocity in the fluid. The average value is calculated across the region covered by the tap. In the heart simulation, taps are placed both inside the heart (which is suspended in the fluid) and in the fluid outside of the heart.

The implementation includes a reservoir with a particular pressure. The pressure of the reservoir is equal to some delta greater than the pressure at a fixed point in the fluid grid but outside of the heart. The reservoir is connected to the heart by imaginary hoses that have some resistance. The amount of fluid flowing through these hoses is based on the difference between the pressure in the reservoir, and the pressure at the opening of the hose inside the heart. The flow is also dependant on the resistance inside the hose, as well as the mass and velocity of the fluid inside the hose. Based on these factors, we can calculate the exact flow over a set period of time.

Sources are located at the opening of the hoses in the heart. Sources are specialized taps where fluid is generated or absorbed. The imaginary hoses connect these sources to the reservoir. It is important to note that the hoses are not part of the heart model. From the point of view of the simulation the only part of this system that is relevant is the fluid coming out of the sources. Another object is created outside of the heart that acts as a drain. During the expansion phase of the heart, fluid enters the system from the sources inside the heart and exits from the drain. The reverse occurs when the heart contracts. This process is illustrated in the figure below.

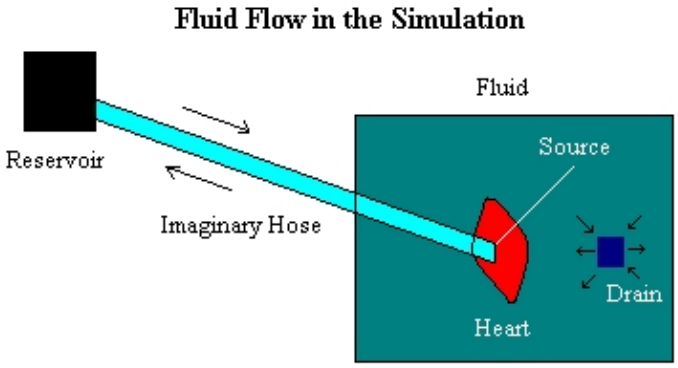


Fig. 8. Fluid Flow

A sample pressure value is computed in the source over the region covered by the source, which is then used to calculate the flow out of or into the source. In addition to this sample, sources also spread out the flow over the region.

4.1 Communication

Each tap/source is assigned to a single process. This process performs all of the calculations related to this tap, including the calculation of the sample and the spreading of the flow for sources.

The fluid grid in the heart simulation is divided into slabs in a single dimension. For the heart simulation, a typical run involves a grid size of 128 cubed, run on 32 processors. In this case, each process is assigned a fluid slab of size 4x128x128. Taps typically cover an 8x8x8 by region, and can span across as many as three slabs. This is illustrated by the figure below.

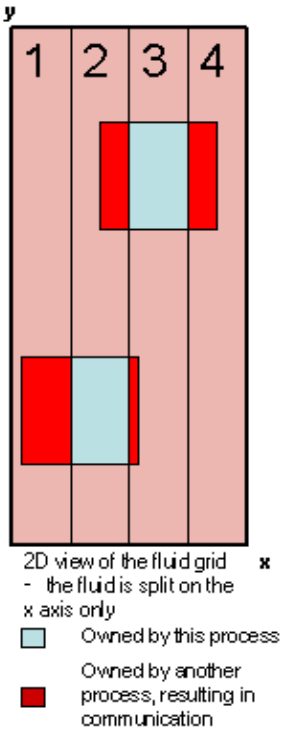


Fig. 9. Bulk Communication

In the original code, the process that owns a tap computes the sample value by reading each point of the tap one by one. For each point that is located in the fluid slab of a foreign process, the owning process reads the point from the other process and adds it to the running total for the sample. A message is sent for every foreign point.

A similar problem exists for the spread calculation. In this case, the owning process updates each source point individually. The points in the source are update based on current source strength, the amount of fluid coming out of the source. A point is read by the owning process and incremented by some value, resulting in a read and write. In addition, the total amount of fluid coming out of the sources is calculated by a reduction. Each process owns part of the drain, which utilizes this value.

Not all processes own sources and taps. This results in considerable load imbalance, as some processes must wait at barriers while others perform these calculations.

4.2 Optimizations

Initially, the goal was to convert the communication for taps and sources into bulk communication. An array of points would be sent between processes, resulting in a few instances of communication for each tap rather than one for each point in the tap.

This method was not implemented completely, as we realized that communication could be minimized if the all processes sharing a tap performed the calculations on their part of the tap. In the case of sample for all taps, the sample is essentially a sum of values computed on each point of the tap. Each individual process could compute a local sum, and a reduction would compute the total sample. For spreading the flow in a source, the current source strength for each source can be passed to all processes, and each process can update its own points corresponding to the source.

4.3 Implementation

We create an immutable class (a light tap) that includes all of the information required to calculate the sample or spread. These light taps include the location of the center of the tap, the location of one corner of the tap, and the size of the tap. If the tap is also a source, it includes the current source strength. These light taps are exchanged between all processes prior to the spread calculation every iteration. The exchange is performed every iteration because the taps move around.

Each process updates the points it owns corresponding to a particular source based on the source strength stored in the sources light tap. The total amount of

fluid leaving the sources can be computed by each process using the light taps, removing the reduction. The other values are used to compute the local sample. Global samples are calculated using reductions.

4.4 Evaluation

Performance numbers were taken from running a pipe simulation. This pipe code is used by the heart simulation as described above. The optimizations for taps and sources do not affect other parts of the heart simulation.

The simulation was run on Seaborg, a cluster machine consisting of 16 processor SMP nodes. The machine consists of 375 MHz Power 3 processors with a peak performance of 1.5 Gflops per processor. This simulation was run on a single node.

Our implementation shows better load balance for all sizes. Note that the new implementation includes reductions in each sample calculations, accounting for the equivalent average and maximum sample times. For smaller grids, including the 128x128x128 used by the heart, our implementation is significantly faster for spread. For larger grids, sources have a greater likelihood of existing in a single fluid slab.

Maximum/Average Sample Time (ms) on 16 Processors

	Original	Modified	Original (optimized compile)	Modified (optimized compile)
64x64x64	51 / 6.69	11 / 11	20 / 2.81	4 / 4
128x128x128	49 / 6.69	12 / 12	20 / 2.75	4 / 4
256x256x256	50 / 5.75	15 / 15	19 / 2.25	4 / 4

Spread Time (ms) on 16 Processors

	Original	Modified	Original (optimized compile)	Modified (optimized compile)
64x64x64	125	43	44	5
128x128x128	296	225	50	11
256x256x256	1511	1514	96	57

Fig. 10. Performance on Seaborg

5 Second Order Method

5.1 Second-Order Method

Also implemented was the formally second-order accurate method of McQueen and Peskin [1]. The main difference between this method and the previously

implemented one [3] is that the movement of the fibers and the computation of the fluid velocity and pressure use two steps each instead of just one. The overall solution cycle proceeds as follows:

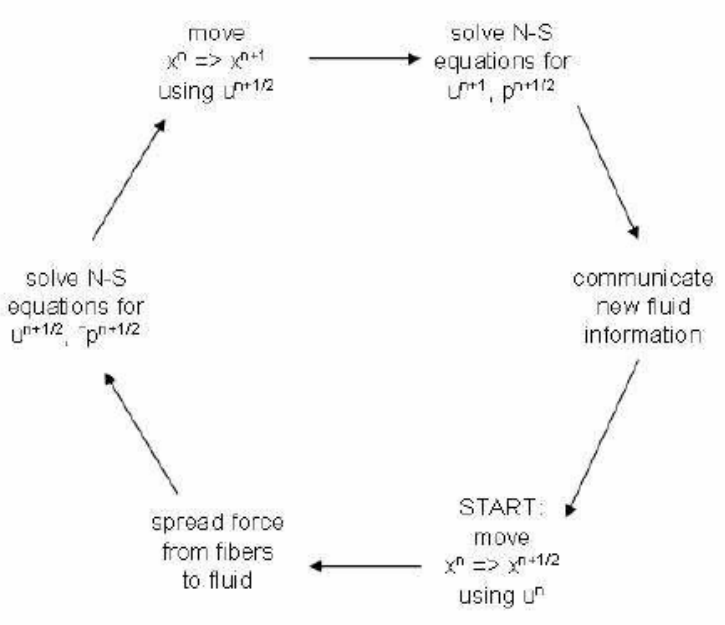


Fig. 11. Solution cycle for second-order method. This cycle is repeated during each timestep.

A thorough treatment of the mathematical details of every step can be found in [1] and [2]. The key steps, though, are the ones for moving the fluid, as it is the fluid velocities that are used to determine how far the fibers move in each

step (the fluid velocities themselves are computed based on forces exerted by the fibers on the fluid, but the procedure for computing forces in the second-order method remains the same as it does in the first-order method). This section will discuss the implementation details of this step and show results from both the first and second-order methods.

5.2 Equations of Motion

The equations used to compute the fluid velocities $\boldsymbol{u}$ and fluid pressure p given the fluid density ρ, fluid viscosity μ, and force $\boldsymbol{f}$ are the well-known Navier-Stokes equations [1]:

$$\rho(\frac{\partial \boldsymbol{u}}{\partial t} + \boldsymbol{u}\cdot \nabla \boldsymbol{u}) + \nabla p = \mu \nabla^2 \boldsymbol{u} + \boldsymbol{f} \tag{1}$$

$$\nabla \cdot \boldsymbol{u} = 0 \tag{2}$$

This assumes there are no sources or sinks. If this is not the case, then a term $\frac{1}{3}\mu\nabla q$ is added to the right-hand side of the first equation to reflect the contribution of the sources and sinks to the fluid velocity. From now on, we will assume the presence of sources and sinks.

5.3 Numerical Solution

The Navier-Stokes equations are discretized as follows [1]. First, a half-step velocity $\boldsymbol{u}^{n+\frac{1}{2}}$ and an intermediate fluid pressure $\tilde{p}^{n+\frac{1}{2}}$ are computed based on the previous timestep:

$$\rho[\frac{u_i^{n+\frac{1}{2}} - u_i^n}{\frac{\Delta t}{2}} + \frac{1}{2}(\boldsymbol{u}^n \cdot \boldsymbol{D}u_i^n + \boldsymbol{D}\cdot(\boldsymbol{u}^n u_i^n))] + D_i\tilde{p}^{n+\frac{1}{2}} = \mu L u_i^{n+\frac{1}{2}+f_i^{n+\frac{1}{2}}} + \frac{1}{3}\mu D_i q \tag{3}$$

$$\boldsymbol{D}\cdot\boldsymbol{u}^{n+\frac{1}{2}} = 0 \tag{4}$$

Here $\boldsymbol{D}$ is a centered finite difference, and L is a "tight" Laplacian operator, both defined in [1]. Following this step, the velocity $\boldsymbol{u}^{n+1}$ and pressure $p^{n+\frac{1}{2}}$ for the next step are computed based on both the half-step and previous-step velocities:

$$\rho[\frac{u_i^{n+1} - u_i^n}{\Delta t} + \frac{1}{2}(\boldsymbol{u}^{n+\frac{1}{2}}\cdot\boldsymbol{D}u_i^{n+\frac{1}{2}} + \boldsymbol{D}\cdot(\boldsymbol{u}^{n+\frac{1}{2}}u_i^{n+\frac{1}{2}}))] + D_i p^{n+\frac{1}{2}} = \frac{1}{2}\mu L(u_i^n + u_i^{n+1}) + f_i^{n+\frac{1}{2}} + \frac{1}{3}\mu D_i q \tag{5}$$

$$\boldsymbol{D}\cdot\boldsymbol{u}^{n+\frac{1}{2}} = 0 \tag{6}$$

The equations are solved in the Fourier domain with the help of FFT's. This makes the solution a lot easier as the velocities can be obtained by solving a sequence of small 4×4 linear systems whose solution can be calculated by hand instead of solving a large linear system whose size is the volume of the fluid cube.

5.4 Results

The second-order method was tested on a simulation of fluid flow through a pipe with one source and one sink. The test was run on the NERSC seaborg cluster using 16 processors on one SMP node and a $128 \times 128 \times 128$ fluid cube, and its results were compared with the previously implemented first-order method. Through the first ten timesteps, the results of the second-order method and the first-order method were similar, but after that, the second-order method started violating the CFL condition [4] used to check the convergence of the method. The following plots of the fluid velocities after ten timesteps for both the first and second-order methods shed some light on what is going on.

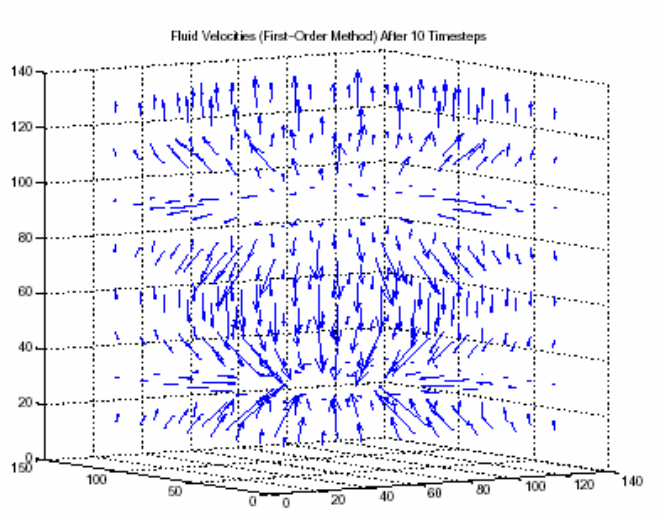


Fig. 12. Fluid velocities for the first-order method after ten timesteps.

Notice the large arrows pointing up and away from the fluid source at the top of the second-order plot that are absent from the first-order plot, which did not have any issues with the CFL condition. This indicates some sort of bug in either the velocity calculation or force distribution steps. Efforts to correct the problem are underway, but as of yet have not been successful.

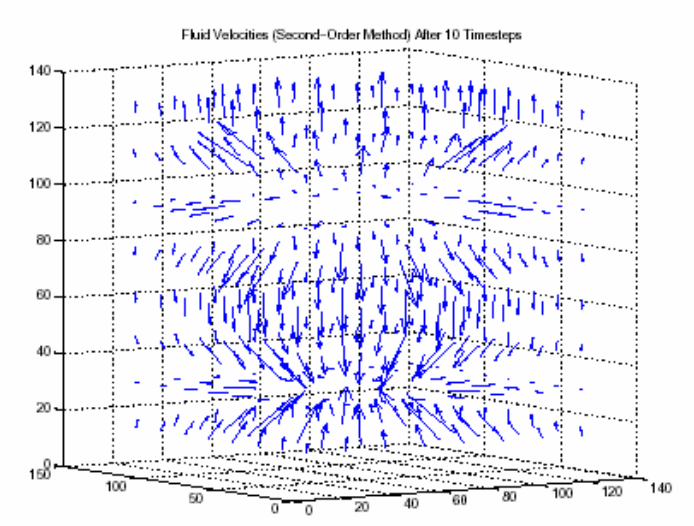


Fig. 13. Fluid velocities for the first and second-order methods after ten timesteps.

6 2-D Fluid Partitioning

The baseline heart simulation is constrained to a maximum of 32 processors for $128x128x128$ fluid cube. More importantl, for an $NxNxN$ problem, it is only possible to use $N/4$ processors, a major limitation for large N. The reasons for this restriction are two-fold. First, the current version of the code uses FFTW, which only allows for a slab based partition. Second, the code works more efficiently when it can generate $4x4x4$ cubes entirely within a processor, which means there is a loss of efficiency when there are fewer than 4 slabs per processor. For these reasons, a major goal of this project is to increase the number of processors as N^2. Two modifications to the baseline were needed to achieve this goal. Ghost planes must be exchanged in two different directions corresponding to the partitioned dimensions. Since FFTW only allows slab partitions, we implemented our own FFT starting from the 1-dimensional FFTW.

6.1 Ghost Planes – Baseline

The baseline implementation of the immersed boundary simulation partitions the fluid in only one dimension (the x-dimension). This partitioning effectively separates the fluid cube into a set of fluid slabs where the number of slabs is equal to the number of Titanium processes.

The Navier-Stokes solver requires the first and second derivatives of the fluid velocities, performing a four-point stencil on every fluid point in the cube. The stencil operation for fluid points on the slab boundary requires ghost planes to provide the velocities of adjacent points not owned by the local process.

With 1-D fluid partitioning in the x-dimension, each process must ghost planes for fluid points in adjacent slabs. These ghost planes are updated at the beginning of every call to the Navier-Stokes solver. This update logically requires communication between adjacent fluid slbas and physically requires communication between processes. After ghost planes are exchanged, each process performs a 3-D Fourier transform on these values and solves the transformed equations.

The velocities and forces for each point in the fluid cube are reprsented in a 1-dimensional array. Points in the z-dimension are contiguous, lines of points in the y-dimension are contiguous, and planes of lines in the x-dimension are contiguous. Thus, the indirection when indexing points in the array increases from z to y to x. This 1-dimensional representation of a 3-dimensional fluid results in the allocation of a ghost plane at the beginning of the array corresponding to boundary points on the previous process and a ghost plane at the end of the array corresponding to the boundary points on the next process.

Each process must communicate with adjacent processes to exchange ghost planes. This exchange occurs with each process pushing its boundary planes to adjacent processes and is implemented as an array copy from a process' local

fluid array to the destination process' fluid array, a remote data structure.

6.2 Ghost Planes – Modifications

Partitioning the fluid in two dimensions (the x- and y-dimensions) requires modifications to each process' set of ghost planes. In addition to the ghost planes from adjacent slabs in the x-dimension, each process must also maintain ghost planes from adjacent slabs in the y-dimension. These additional ghost planes require additional points in each process' local fluid array and additional communication to exchange these planes.

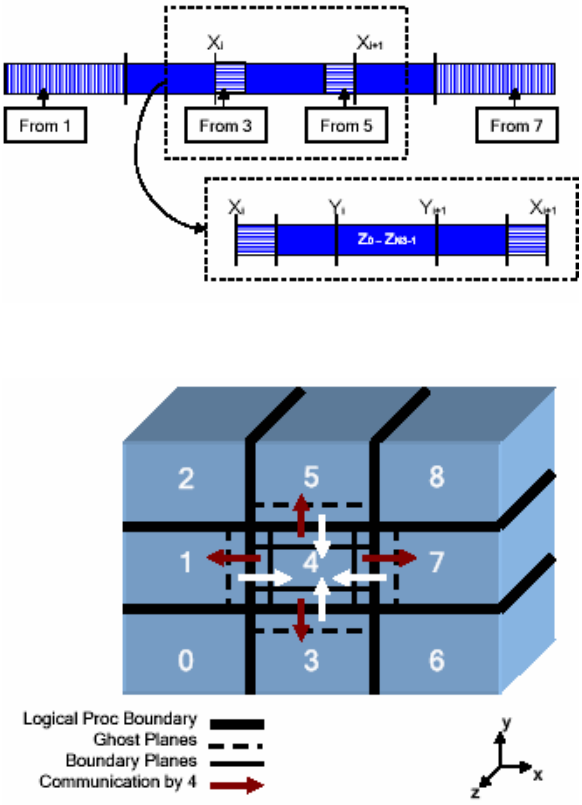


Fig. 14. Ghost Plane Modifications: Array representation (above) and exchange communication (below).

The boundary fluid points and the ghost planes in the y-dimension are interleaved in the fluid array, complicating the communication of these planes. The communication of these ghost planes must be aggregated into a send buffer to minimize communication between local and remote data structures. The send buffer is then communicated to a remote receive buffer. The receiving process will unpackage the buffered planes and integrate them into its local fluid array. The ghost planes in the x-dimension are exchanged after those in the y-dimension. This second exchange occurs as described in the previous section.

Despite the additional communication of ghost planes in the y-dimension, the increase in execution time is negligible because the exchange methods were converted to local Titanium methods and the runs employed the LAPI backend on the Seaborg cluster. These two considerations significantly improved performance for the more complex communication such that it required no more time than the relatively unoptimized exchange in the x-dimension.

6.3 Fast Fourier Transform – Scaling

Performing a 3D FFT requires performing three sets of FFTs, one for each dimension. The first dimension is straightforward because the data is already contiguous in that dimension. For the second dimension, the slab is transposed efficiently because each process can transpose its own slab independently. Thus, the second FFT is also performed over contiguous data. To perform the third and last FFT, each process obtains a set of contiguous data in the last dimension by performing an all-to-all exchange.

To obtain better scaling for the FFT, we decided to make the implementation SMP-aware. Given $PN1$ processes with $PN2$ threads per process, we partition our space in the x-direction so as to give $N1/PN1$ slabs to every process. Within the process (hereafter referred to as the node, since we only have one process per node), all threads will cooperate to manipulate their slab. The decision to make the code SMP-aware allowed us to use many more nodes while keeping the cost of all-to-all communication under control. However, this decision also introduced problems with false sharing which we were unable to resolve.

The modified FFT algorithm operates as follows:

1. **First FFT (First):** Each process has $N1/PN1$ slabs, so it has $N1*N2/PN1$ bars, to do FFT on. Each thread in the node will do an FFT on a contiguous chunk of $N1 * N2/(PN1 * PN2)$ bars, completely independently of all the other threads, so in theory, there is no reason not to expect a perfect speedup. However, we can see that in the case of 128*128*128, the scaling is not quite linear, and it actually becomes very bad when we get to 32 processors. One explanation for this could be false sharing. This is because on each node, the master thread will allocate a big array, and all the processors will be doing FFTs on contiguous chunks of that array. Thus, pairs of processors

will be doing FFTs on pieces of memory that are very close together.

2. **Slab Transpose (Lt):** For the slab transpose, we divide the slabs in each node among the threads. If there are more threads than slabs, sets of threads will cooperate to transpose each slab. The transpose itself is blocked to increase locality, however, we can see that the local transpose scales fairly well until it gets to the point where you have one thread per slab. When you have more than one thread per slab, however, performance drops noticeably, once again indicating potential false sharing problems. For the tests we ran, we had a block size of 4x4, but it is quite possible that increasing this block size reduce false sharing in this case, but in the interest of time, we were unable to perform that set of experiments.

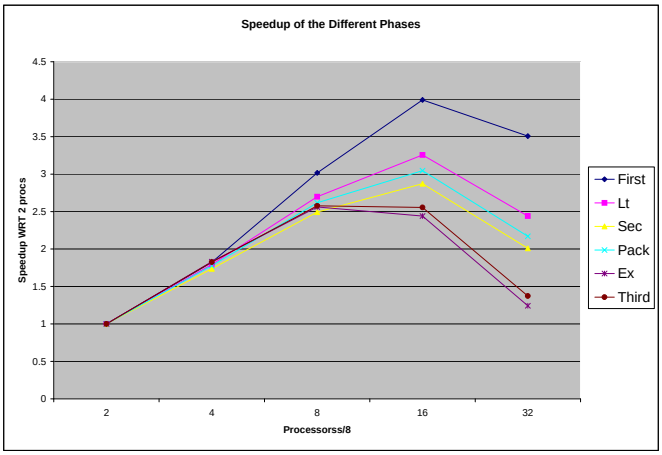
3. **Second FFT (Sec):** The second FFT behaves very similarly to the first one, except its much slower. This can be partly explained by the fact that the second FFT is not quite identical to the first one. This one is complex to complex, where as the first one is real to complex. However, in this case we are only doing a little over half as many FFTs, as before, so overall the amount of work should be the about the same.

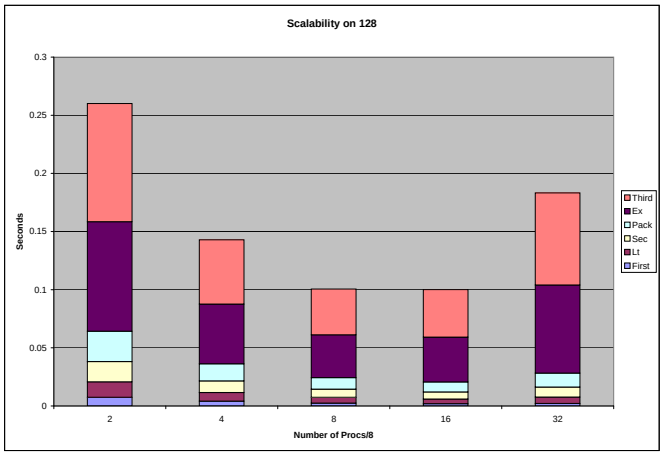
4. **Pack (Pack):** After the second FFT, we have to do the all-to-all exchange. The exchange is done in three parts. The fist part is the Pack. In this stage, the threads in a node pack into contiguous memory all the data that will go to different nodes. This is similar to the transpose, in that all the processors are accessing the same global array in order to rearrange it. False sharing is much more likely in this case, because all the processors are accessing interleaved portions of memory, and it is not blocked like the transpose is. For this reason, we can see that the performance and scalability are slightly worse than for the Local Transpose.

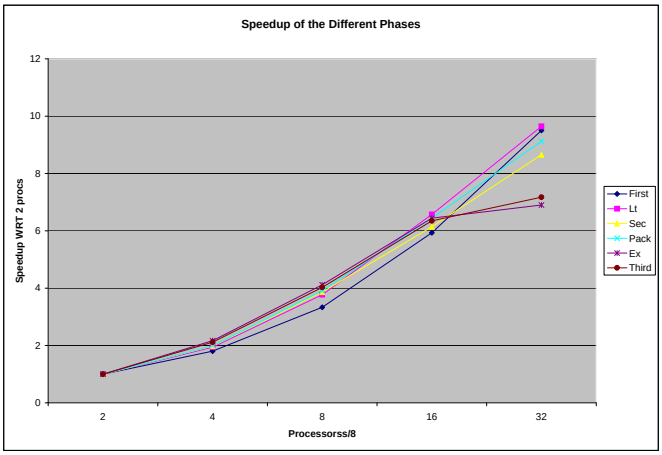
5. **Exchange (Ex):** The second two parts of the all-to-all exchange run simultaneously, and are both packaged under Exchange. At this stage, thread one of each node will ship data one by one to all the other nodes. At the same time, the other PN2-1 threads will be unpacking the data as it is being received, so that after this routine is completed, all the data will be in the proper layout.

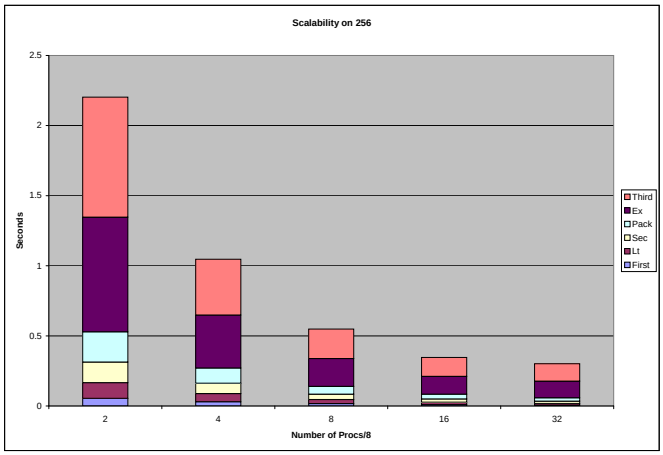
6. **Third FFT (Third):** The third FFT is identical in every way to the second FFT. For this reason, it is very surprising that whereas the second FFT takes almost 5 times more than the second one. Because this is a call to FFTW, it is difficult to tell exactly what is causing this performance degradation.

There is still quite a bit that we dont understand about the performance of the FFT. In particular, the fact that the second FFT is slower than the first one, and that the third FFT is so much slower than everything in the code is very unexpected.


Speedup of the Different Phases
Speedup WRT 2 procs
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
2
4
8
16
32
Processorss/8
First
Lt
Sec
Pack
Ex
Third


Scalability on 128
Seconds
0.3
0.25
0.2
0.15
0.1
0.05
0
2
4
8
16
32
Number of Procs/8
Third
Ex
Pack
Sec
Lt
First


Speedup of the Different Phases
Speedup WRT 2 procs
Processorss/8
0
2
4
6
8
10
12
2
4
8
16
32
First
Lt
Sec
Pack
Ex
Third


Scalability on 256
Seconds
Number of Procs/8
0
0.5
1
1.5
2
2.5
2
4
8
16
32
Third
Ex
Pack
Sec
Lt
First

7 Conclusions and Future Directions

We survey four optimizations for an Immersed Boundary Method simulation: (1) Load balancing by breaking fibers, (2) aggregating communication for sources and taps, (3) implementing an alternative second order method, and (4) partitioning the fluid cube in two dimensions. Preliminary performance analyses suggest significant potential for improving performance in existing implementations of the heart simulation. Furthermore, these techniques enable the simulation of larger problem sizes with more processors. Future work will be the integration of these separate techniques into a single implementation of the Immersed Boundary Method to explore the cumulative effects of these optimizations.

References

1. D.M. McQueen and C.L. Peskin. "Heart Simulation by an Immersed Boundary Method with Formal Second-Order Accuracy and Reduced Numerical Viscosity." In Proc. of the International Conference on Theoretical and Applied Mechanics (ICTAM), 2000.
2. C.L. Peskin. "The Immersed Boundary Method." Acta Numerica 11:479-517,2002.
3. C.L. Peskin and D.M. McQueen "A General Method for the Computer Simulation of Biological Systems Interacting with Fluids". Biological Fluid Dynamics, 1995.
4. J.W. Thomas. "Numerical Partial Differential Equations: Finite Difference Methods." Springer-Verlag, 1995.
5. "Cardiac Fluid Dynamics and the Immersed Boundary Method." http://www.npaci.edu/SAC/Collaborations/Peskin/reports/final.report.html.
6. G. Karypis and V. Kumar. "METIS: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices."
7. G. Karypis and V. Kumar. "Multilevel k-way Partitioning Scheme for Irregular Graphs." Technical Report 95-064, Department of Computer Science, University of Minnesota, 1998.
8. K. Datta and S. Merchant. "Multigrid Methods for Titanium Heart Simulation."
9. D.M. McQueen and C.S. Peskin. "Shared-Memory Parallel Vector Implementation of the Immersed Boundary Method for the Computation of Blood Flow in the Beating Mammalian Heart."
10. C.S. Peskin and D.M. McQueen. "A General Method for the Computer Simulation of Biological Systems Interacting with Fluids."