Analysis of a Contractile Torus Simulation in Titanium

Sabrina A. Merchant[1]

[1]Computer Science Division, University of California, Berkeley, CA 94720

Abstract

The purpose of this paper is to analyze the use of the Titanium language, a high-performance Java dialect, and parallel programming practices on an application of the Immersed Boundary (IB) Method for simulating biological processes. We will compare two Titanium implementations of the IB Method in terms of performance, developer productivity, and use of Titanium features. The first implementation makes use of Titanium's scientific computing and parallelization features. The second uses Java features for data structures and Titanium for parallelism. We analyze the libraries using a contractile torus simulation as a sample application, a simple model of elastic fibers suspended in viscous, incompressible fluid. The implementation of the simple model precedes the realization of a more complex model of the mammalian heart, discussed in the final section of the paper.

1. Overview of Titanium

The Titanium language extends Java and includes functionality for scientific computing applications, such as a global address space, parallelization primitives, and multi-dimensional arrays. An implementation of Titanium was developed at Berkeley that is publicly available. Titanium is optimized for scientific applications such as Multigrid [3] and particle simulations. The current driving application is a simulation of the mammalian heartbeat using the Immersed Boundary Method.

Berkeley's implementation does not require special hardware support. Programs written in Titanium can run on shared-memory and distributed memory architectures, as well as clusters of shared memory multiprocessors. The compiler translates Titanium into C with lightweight messaging layers. Currently, the compiler runs on the Cray T3E using Shmem, the IBM SP using LAPI, the Compaq/Quadrics clusters using Shmem, and on the SGI Origin 2000 using POSIX threads. In addition, an MPI implementation of the runtime system provides portability across essentially any cluster.

Titanium features can be grouped into two categories parallelization features and support for scientific computing. Titanium's support for parallelism includes a global address space, communication and synchronization primitives, and local/global references. Titanium's support for scientific applications includes multi-dimensional arrays, immutable classes, memory management control, and compatibility with scientific computing libraries written in other languages.

1.1 Parallelism

Titanium uses a Single Program Multiple Data (**SPMD**) model of computation, so each processor runs a copy of the same program. Processes may work independently by branching on process-specific data, and there is no implicit synchronization. The Titanium compiler prevents deadlock on global synchronization primitives by ensuring that all processes execute the same sequence of global synchronizations.

In addition, Titanium support for parallelism includes global synchronization, a global address space, and processor communication methods. **Global synchronization** is done with the barrier method, which has the following properties: a process will wait at the barrier until all other processes reach the barrier, and all processes execute the same sequence of barriers. The compiler performs global synchronization analysis by examining **single**-valued variables, variables that have the same value on every process. For example, conditional expressions with barriers must be single-values, meaning that all processes will take the same branch.

Communication between processes occurs using the broadcast or exchange methods. The **broadcast** method is a one-to-all communication, while the **exchange** method is an all-to-all communication. Both methods have implied global synchronization or barrier of all processes.

Because of the global address space, following a reference can be quite expensive on some machines where it leads to a communication event. The performance cost is significant on distributed-memory architectures, where local pointers are accessed faster and take less space than global pointers. To make this cost visible to the programmer, without forcing explicit communication, the Titanium language offers two kinds of references: **global** and **local**. A remote reference may point to either local or remote data, while a local one points to probably local data. All references are global by default, which simplifies porting of threaded Java code to Titanium.

1.2 Support for Scientific Applicatoins

Titanium arrays are true **multi-dimensional arrays**, independent of Java arrays. They are indexed by points, which are grouped into index sets called domains. Titanium has a rich domain calculus for determining subdomains, transposes, and intersections. To loop over the elements of Titanium arrays, the iterator *foreach* is used. Titanium does not specify an order of iteration for the *foreach* loop allowing the compiler to reorder iterations for optimizations. Distributed arrays are not handled explicitly but can be created using the exchange operation. Most applications have a specific data organization across processors; therefore one can create a distributed array structure tailored to their application.

Immutable classes were introduced into the language to increase performance of small objects. Objects in Java are accessed through references, which add overhead that reduces performance especially when many small objects are used in a program. Restrictions on immutable classes, such as final non-static fields and no inheritance, allow the compiler to pass them by value similar to C structs. Immutable classes can then effectively treated as Java primitive types and arrays of immutable objects can be stored contiguously in memory.

Memory management in Titanium is done using **region-based memory management**. Objects and data are allocated in a private or shared region, and regions are freed with a region-delete operation. Region-based memory management allows explicit programming for locality and performs better than garbage collection on distributed-memory machines.

In addition, Titanium allows C code to be linked to Titanium code using Titanium's native C interface. The **native** interface allows compatibility with scientific computing libraries. Also in cases with non-uniform access to Titanium arrays, C kernels can be written and interfaced to Titanium.

2. History of IB Implementations

The Immersed Boundary Method is a mathematical method for simulating the fluid dynamics of elastic material immersed in viscous incompressible fluid. The method was invented by Charles Peskin and Dave McQueen at the Courant Institute at NYU [1]. The method has been used in several simulations of biological processes including platelet clotting[7], cochlea function in the ear [4], and of particular interest the mammalian heart [2]. The original implementation of the IB method was written in Fortran 77 by Dave McQueen and is the code that several other implementations have been validated against. Following this implementation, other simulation writers modified the code for their model system. Recently, there was an effort by Nat Cohen (NYU) to design a library for the IB method general enough for all IB simulations. Cohen's library was written in Fortran with shared memory vectorization and has been used to model a contractile torus and human heartbeat. However, the Fortran implementation is limited to use on shared memory machines, such as the C90 and SGI Origin. This limits the problem size and ability to parallelize over hundreds of processes.

Titanium expands these limits because it can be used on distributed memory architectures using up to 300+ processes allowing a theoretically large problem size (1024^3+) and performance improvement. The first version of the Titanium Immersed Boundary Method (**Tigibs**) library was written by Siu Man Yau. Three simulations were written using his library: the contractile torus, cochlea plates, and mammalian heart. [8] He used much of Titanium's scientific computing support and parallelism. Subsequently, we added several optimizations to Tigibs that will be discussed in the Section 4.

The second version in Titanium was written by Ed Givelberg [4]. Givelberg's use of the Immersed Boundary Method was driven by simulation of cochlear function in the inner ear. He ported his C version of the method to Titanium in the form of a general library. Givelberg's **IB** library handles more generic materials such as discs, which are necessary for cochlea simulation. The IB library uses Java arrays and independently written domain calculus and uses Titanium for parallelism and native interfaces. Currently the IB library is being optimized and the cochlea application is being written using the library. We have implemented a contractile torus simulation using the IB library, which we will use to analyze the library and compare with the same simulation written with the Tigibs library.

We will compare the two libraries – Tigibs and IB – in terms of readability, performance and scaling. First, we give an overview of the immersed boundary method, then detail the differences between the two implementations, present a performance model of both implementations, next discuss the performance of the two versions, and finally we discuss the transition to a complete heart simulation.

3. Overview of IB Method

To demonstrate Titanium as used by a scientific application, we use a model system, the contractile torus. The torus consists of thousands of elastic fibers shaped into a torus structure that is covered by a grid of fluid. An elastic fiber behaves as a group of springs attached to each other to form a circle. Each spring exerts force according to the law $\boldsymbol{F = kx}$, where $\boldsymbol{k}$ is the spring constant and $\boldsymbol{x}$ is the displacement, on the spring preceding and following it. The fiber also exerts force on the fluid, the NS equa-

tions are solved to find the fluid velocity, and then the fibers are moved according to the fluid velocity.

The number of timesteps varies with the problem, for example the heart simulation is performed in 57,000 timesteps while the torus takes 512 timesteps. Each timestep consists of four phases of the Immersed Boundary Method: calculating the fiber force, spreading the force to the fluid, solving the Navier-Stokes (NS) equation for fluid velocity, and moving the fibers at the local fluid velocity. At the commencement of the timestep, the fibers have been displaced and the process is repeated in the next timestep. The phases are summarized here.

Fiber Force Calculation: Each fiber is represented as a set of points linked together by the springs. We calculate a force for each point using an elastic spring law described above. The force will act to pull the point's neighbors towards it or push its neighbors away.

Spread Force: In this phase, each fiber point will update the 4x4x4 grid of fluid cells surrounding it by adding the fluid force that it will exert on them. The amount of force exerted on the fluid cell by a fiber point is calculated as a smoothed Dirac Delta function of the fiber force evaluated at the fluid cell. The force that a fluid cell carries at the end is the sum of force exerted on it by nearby fiber points.

NS Solver: In the NS Solver, we first calculate the right-hand side of the NS equation, using nearest-neighbor updates on the fluid force grid. Then we take an FFT of the left-hand side, and find the velocities in Fourier space, followed by an inverse FFT to translate the velocity grid back to normal space in 3d.

Interpolate Velocity: Finally, the fiber velocity is calculated from its surrounding fluid velocity, the same 4x4x4 grid of fluid points, as a sum of smoothed Dirac Delta functions of the fluid velocities evaluated at the fiber points. The fiber points are moved into a new position, based on their velocities.

4. Comparison of Implementations

Both implementations use the same force calculation and fiber activation, but the implementations differ in how the calculations are performed. We detail the differences between the implementations for each of the phases. The discussion is presented in phases of the IB method and split into detail of the **Tigibs** library and then the **IB** library.

4.1 Fiber Structure

Fibers are represented as space curves immersed in a rectangular fluid grid. A fiber is a cyclic, ordered set of fiber points residing in 3-dimensional space within the domain of the fluid grid. Each fiber point is connected to two other fiber points in the fiber set, and each connection is conceptually a spring.

Tigibs: The fiber is represented by a doubly-linked list of fiber points. Each fiber point object consists of its position, its velocity, the force exerted on it by the two neighboring points, and pointers to its neighboring fiber points in the linked list structure.

Fibers often cross process regions depending on the choice of fiber partition. To address the representation of fibers that cross boundaries, we introduced another data structure for storing portions of fibers that reside on a process, which we call a fiber fragment. A fiber fragment object consists of a pointer to the first and last fiber point in that fragment, as well as pointers to the fragment's immediate neighbors on different processes. Each process owns a Java Vector of its fiber fragments.

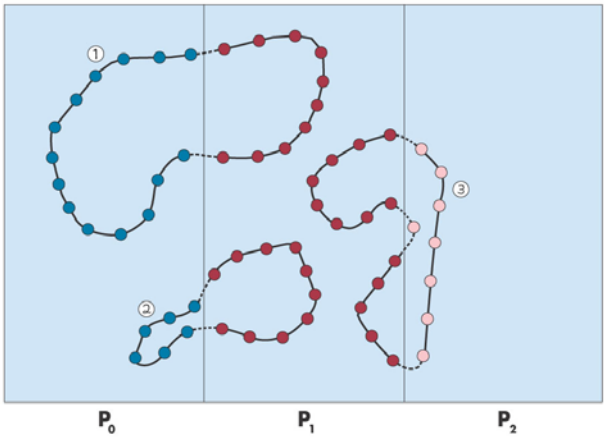

Figure 1: Distributed fiber structure.

Figure 1 shows the distribution of fiber fragments in this library. Fiber points owned by the same process are in the same color. The partition shown is one where the fiber fragments are owned by the owner of the underlying fluid grid. This partition allows us to take advantage of locality, decreasing communication in the interaction phases. However, load balance is poor as the fibers are concentrated in the center of the fluid domain. Also, the force calculation requires iteration over all points in a fiber. A partition where fibers cross process boundaries increases communication in this phase. One partitioning strategy takes into account locality and load balance, called egg slicer, which assigns fiber fragments to the corresponding fluid grid owner but balances the number of fiber points on each process. Another partitioning strategy does not allow fibers to cut across processes, called spaghetti.

Figure 3 shows both partitioning strategies for the force calculation of the heart simulation. The spaghetti partition performs better during the force calculation but badly

for the interaction phases. Egg slicer works best overall. The optimal partitioning strategy is easier to decide for the more regular fiber structure of the torus. Cross-sectional fibers should be kept intact, while longitudinal fibers should be cut for locality, called the pizza partition.

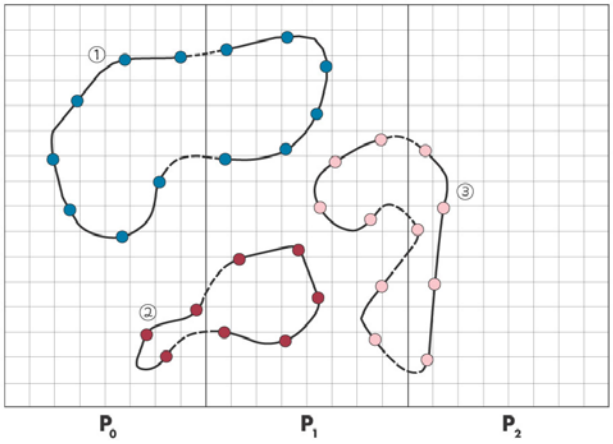

Figure 2: IB fiber structure.

IB: Figure 2 shows the fiber structure for the IB library. The fluid is cut into slabs and assigned to processes, and the fibers are uncut when assigned to a process. The different colors represent the ownership of each fiber. In this figure, each fiber is assigned to a different process. Currently, locality is not taken into account, however the number of fiber points is approximately constant for all processes. To improve performance and scalability, fibers close to each other should be distributed together. The fibers points are maintained in an array rather than in a linked list. Also, the library requires that the entire fiber be located on the same process, therefore no communication occurs in this phase.

4.2 Force Calculation

Tigibs: In the force calculation, each fiber point accesses its two neighbors, which requires communication between processes only when the fiber point is at the beginning or end of a fiber fragment. The communication can be reduced by introducing immutables classes to encapsulate the coordinates. We use immutable classes to represent the position, velocity, and force data as a tuple of three doubles. This representation allows one to reduce the number of global accesses by a factor of three. However, the use of immutable classes adds overhead when we want to change the immutable's value. We must initialize a new immutable object, copy the new values into the object, and replace the old immutable object. We would like to retain all the performance properties of immutables but allow variables to be updated.

Figure 3 demonstrates the performance increase with immutables for the heart simulation. This optimization works solely when communication is involved but does not affect the spaghetti partition, where no communication occurs.

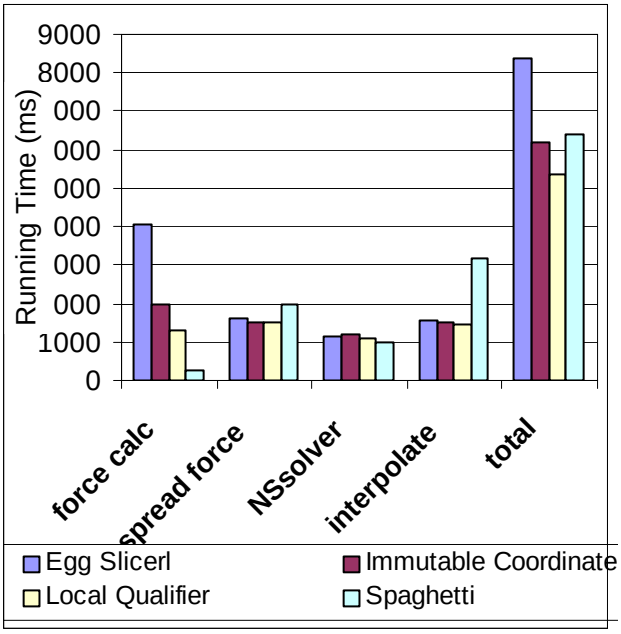

Figure 3: Optimizations on Tigibs.

Another optimization that we introduced is the explicit use of the local qualifier. All references in Titanium are global by default, and the compiler automatically infers that some references are local, but local qualifier inference is not fail-proof. To gain all benefit associated with local qualification, the programmer should explicitly declare the variable as local. In the torus, the linked list fiber representation includes local and global pointers. Neighboring fiber points that lie at process boundaries are left as global pointers, while internal fiber points are explicitly declared local. The local qualifier reduces the access time and increases the application's performance slightly. Figure 3 also shows the performance increase for local qualification in this phase.

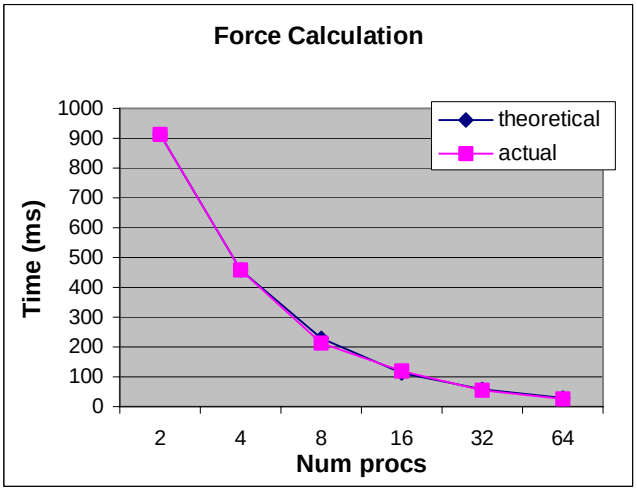

Figure 4: Performance model for force calculation.

IB: The IB library requires that the entire fiber be located on the same process, therefore no communication occurs in this phase. The amount of time spent in this phase depends entirely on how many fiber points are allocated to each process. The best partitioning strategy would be to equally distribute the fiber points.

To develop a performance model for this section, we assume that the number of fiber points on each process is equal. First we look at the trend in computation cost for this phase in Figure 22. The figure shows that this phase has approximately 100% parallel efficiency. Since no communication occurs, computation is the dominating cost. The computation includes floating point operations as well as memory operations. The ratio of floating point operations to memory operations is approximately 0.45. Since the cost of memory operations is so high, we use the mflops rate to determine the performance model for this phase. We used the following formula to model the running time in this phase:

Time = (number of floating point operations per fiber point) x (number of fiber points) / (serial mflops rate)

Figure 4 shows that the model is very close to the actual behavior in this phase.

4.3 Spread Force

This phase consists of taking the force calculated at the fiber points and projecting the force to the fluid domain. The force is projected using a Dirac delta function that spreads the force from the fiber point to a 4x4x4 region of fluid surrounding the fiber point in a 3d distribution.

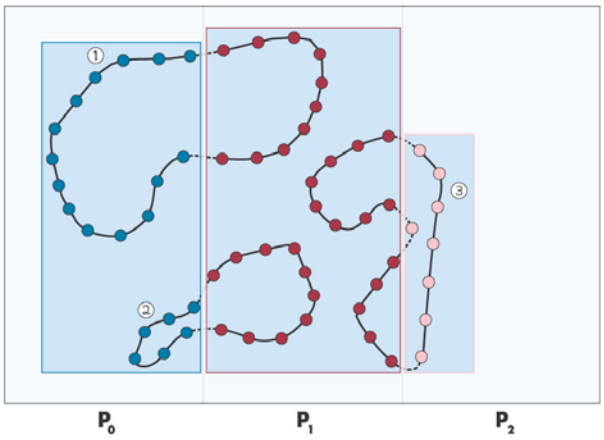

Figure 5: Bounding boxes.

4.3.1 Tigibs Spread Force

Tigibs uses a bounding box algorithm to perform the delta function interaction. This interaction phase consist of three subphases: setting up the bounding boxes, doing the core of the computation, and communicating results to the appropriate process. First each process determines its bounding box, the smallest fluid region that contains all of the points in its list. The dimensions of the bounding box plus padding is used for the domain of the 3d workspace, which will hold the force spread with the delta function after the native method call. Figure 5 demonstrates the bounding boxes for a simplified case. The fluid is partitioned into slabs and the fluid owners are labeled with Pi. The fiber colors denote the fiber's owner. The blue boxes covering all fiber points owned by a processor denotes the area that is the bounding box. This subphase is memory intensive and has few floating point operations. It's performance will be determined by cache behavior and memory performance.

The core of the interaction code is written in C for performance. The native code performs a for loop through all the points in the array. Within the loop, the indices of the 4x4x4 fluid region corresponding to each point are calculated. The delta functions weights are determined. The product of the weight and force is assigned to the portion of the workspace indexed. The regions may overlap; therefore the force for the fluid point is the sum of all the force components corresponding to that fluid location. Figure 6 shows P_0's bounding box and demonstrates spreading force from points to the fluid bounding box. Note that the core of the interaction was written in C because random access to titanium arrays showed low performance, however recent optimization to random access improves the performance but is still four times slower than the same code written in C. This subphase is computationally intensive and consists mainly of floating point

operations. It is expected to scale well and its performance will be determined by the speed of floating point operations, somewhat by cache behavior, and load balance.

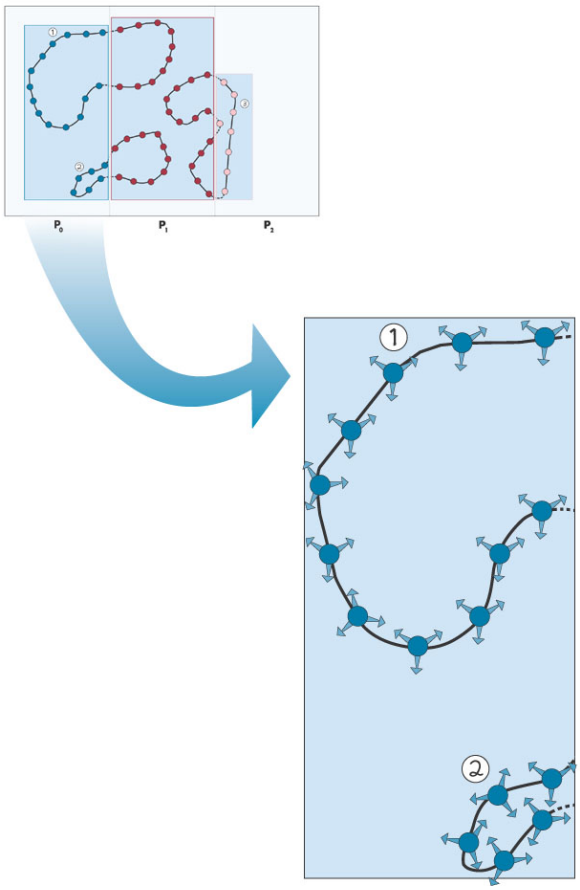

Figure 6: Spreading force to fluid.

Each process owns a vertical slab of the fluid grid; therefore the forces on the grid need to be updated by adding the forces in the bounding boxes overlapping the vertical slab. To accomplish this, each process gets pointers to the bounding boxes for all processes. For each process, it copies the remote workspace from the destination process to a local buffer and adds the forces in the local copy to their vertical slab.

Also, note that communication occurs only in the last loop and it is all-to-all. The amount of communication is determined by the size of each process's bounding box and the amount of overlap with other processes' fluid slabs. The fiber points could be anywhere in the fluid domain, not only within that process's fluid slab. Therefore, the bounding box could lie entirely within the process's fluid slab, overlap two or more processes' fluid slabs, lie completely within another processes' fluid slab, or cover the entire fluid domain. The communication is determined by the spread of the points. Since the fluid is partitioned into vertical slabs, vertical fibers located near each other will perform the best in this phase, while horizontal fibers and fibers distant from each other will perform worst. A good fiber partitioning strategy would be to group points that are near each other to minimize the size of the bounding boxes. Also, an alternative method would be to not cut the vertical fibers for high performance in the force calculation, and cut the horizontal fibers across processes.

4.3.2 IB Spread Force

The fluid domain is partitioned into slabs, similar to Tigibs, but it is partitioned further into cubes of size 4x4x4 as shown in the grid in Figure 7. The processes maintain a list of cubes that are being used. The shaded

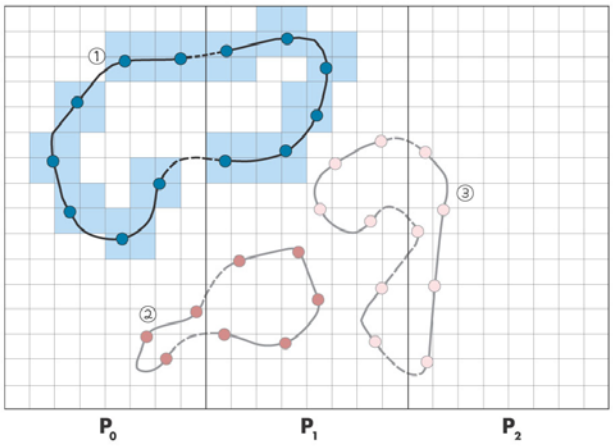

Figure 7: Packing the force cubes.

cubes in Figure 7 represent the cubes that correspond to the fiber indicated. Each processor has a group of complete fibers, which they have calculated the spring forces on that need to be spread to the fluid domain. Every fiber point in the fiber distributes its force to the fluid force field.

The algorithm used to spread the force at each point has the following subphases: spreading force, packing force cubes, sending mail, and updating force slabs. **Spreading the force** consists of determining the delta function weights for each 4x4x4 region surrounding each fiber point, determining which cubes surround each point, and incrementing the force field by the product of the weight and the fiber point's spring force. Each process maintains a list of cubes the fiber points interact with, therefore no redundant cubes are saved. This subphase is mostly computationally intensive but has a large number of memory operations because of determining the cube lists. To model this subphase's performance, we use a similar equation as the force calculation:

Time = (number of floating point operations per fiber point) x (number of fiber points) / (serial mflops rate)

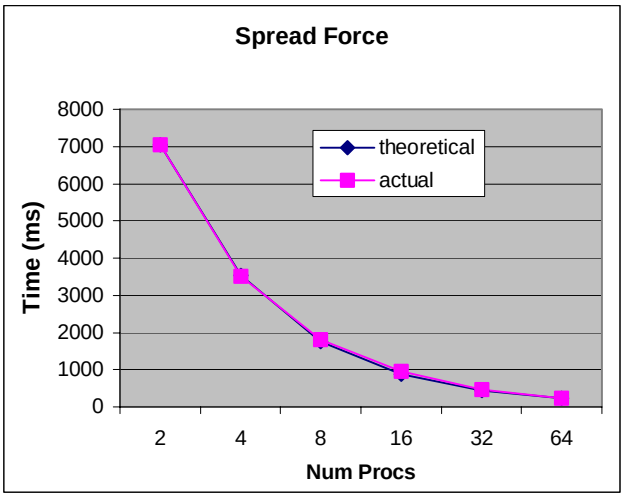

Figure 8: Performance model for spread force.

The number of fiber points is proportional to the running time when the weights are computed, but the number of cubes are relevant for maintaining the cube list. The number of cubes depends on the spread of the fiber points owned by each process. We simplify the model by assuming that the number of cubes is also proportional to the number of fiber points, and we use number of fiber points as the metric for performance. Increasing the number of processes helps to a certain extent, when the width of the fluid slabs are very narrow, causing most of the fluid to be communicated.

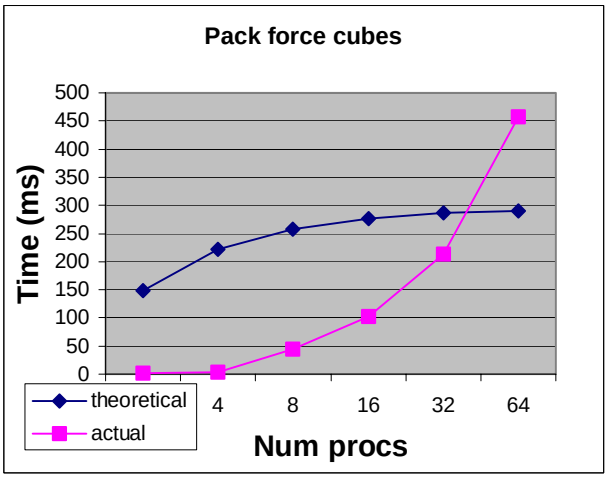

Figure 9: Performance model for packing/unpacking.

Packing force cubes consist of flattening the cubes into a one-dimensional Java array that can be communicated to other processes. The one-dimensional array is referred to as a mailbox. Packing is a memory intensive operation. The number of cubes and the caching behavior determines the memory cost. To determine the number of cubes, we assume that the maximum number of cubes are used. We assume that the memory operations hit in the L1 or L2 cache. Figure 5 shows the model versus the actual number. The model in this subphase is not very accurate because we estimate the number of cubes. If we had a better estimate for the number of cubes, the model would fit the data better. The equation used to determine running time in this subphase follows:

Time = 3 x (number of cubes) x (size of cube) x (L2 cache latency)

So far, no communication has occurred. The blocks that overlap other processor's fluid slabs must be sent to those processes and their force fields updated. This communication is done using a Mailbox system. First, the blocks are condensed into a Java array that is sent to the correct processor using the System.arrayCopy operation, an operation referred to as **sending mail**. The cost of this subphase is determined by the network of the machine. We could use an alpha-beta model, however this does not model the behavior well. We believe that the barrier time is the limiting factor in this subphase, and we model the performance by doing a linear regression of sending mail time versus number of processes, Figure 10.

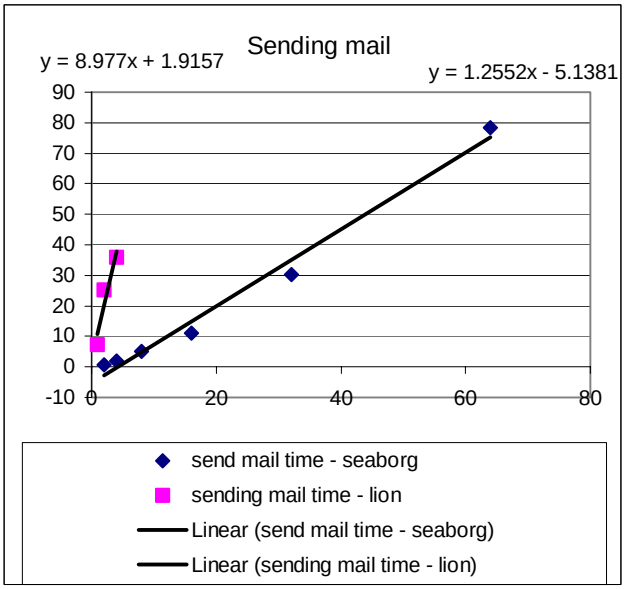

Figure 10: Linear regression for sending mail.

The communication in this phase is an all-to-all communication. The amount of data to be communicated depends on the number of blocks that overlap other processor's fluid slabs. Like the Tigibs interaction phase, the

spread of the points determines the amount of communication. Figure 11 shows the performance model for this subphase on an IBM SP.

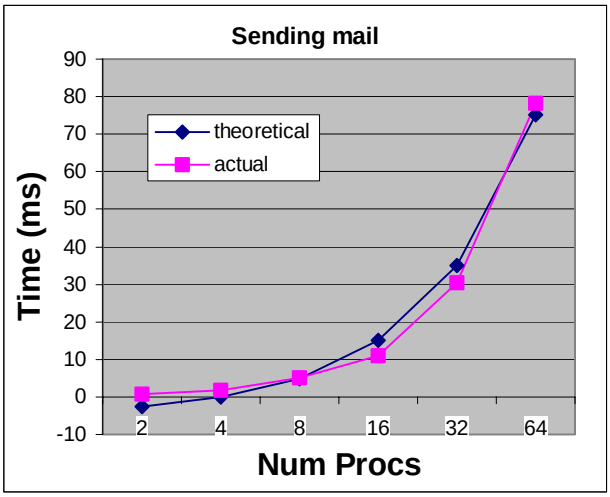

Figure 11: Performance model for sending mail.

Updating force slabs consists of the owners of the fluid slabs updating its own fluid slab after it unpacks the message sent from the overlapping process. This phase consists of memory operations as well as floating point operations. The parallel efficiency is low (Figure 22) and memory operations dominate, therefore we use a similar model to packing:

Time = 3 x (number of cubes) x (size of cube) x (L2 cache latency)

This model would be more accurate if the floating point operations were taken into account. (Figure 9)

4.4 NS Solver

In this phase, the NS equations are solved for the fluid velocity using the fluid force determined above. This phase consist of transforming the knowns into fourier space by three forward FFTs, the equations are solved explicitly and then three inverse FFTs are done to transform the unknowns (the velocities) back to normal space.

Tigibs: This library uses FFTW, a fast FFT implementation in C, to do the forward and inverse FFTs. A Titanium based FFT is also implemented for platforms that do not support FFTW. The FFT is expected to perform well, with time complexity *O (n log n).* The solving of the equations after the transformations is embarrassingly parallel and is expected to scale well. There is also communication of the boundaries between the slabs before the equation solve.

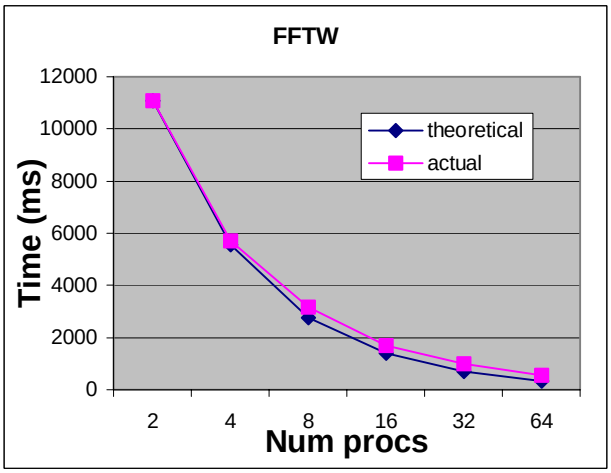

Figure 12: Performance model for the ffts.

IB: This library uses FFTW explicitly for the **FFT**'s, however it is written to easily interface with other FFT libraries. The number of floating point operations is approximately *C x n log n*, where *C* is a constant chosen to be 1.5. The serial mflops for the FFT is 160 Mflops. The performance for the FFTs can be can be modeled using the following equation:

Time = 6 x (C x n log n) / (serial mflops)

After the FFTs, the boundaries of the slabs are communicated and the equations are solved explicitly. The parallel efficiency is fairly high, so we model the performance as if it were embarrassingly parallel.

Time = (number of floating point operations per fluid point) x (number of fluid points) / (serial mflops rate)

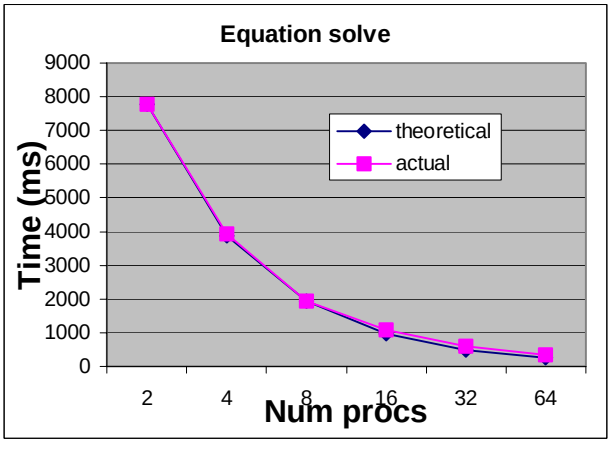

Figure 13: Performance model for the NS equa-

tion solve.

The model would be more accurate if the communication was taken into account.

4.5 Interpolate Velocity

After solving for the fluid velocity in the NS Solver, we need to interpolate the fluid velocity to the fiber point locations using the same Dirac delta function above. We sample the velocity of the surrounding 4x4x4 region and sum the products of the delta function weights and fluid velocity.

4.5.1 Tigibs Interpolate Velocity

The algorithm used is the opposite of spread force. First, we determine the bounding box, initialize a 3d array the size of the bounding box, and copy the fluid velocity into it. This part requires communication as the bounding box will likely overlap with other processes. The partitioning of fibers is important here as well; the bounding box could cover the entire region if partitioned badly. (Figure 14) This is the only communication in this phase. It's performance depends on the size of the bounding boxes. A possible optimization is to retain the bounding box size from the spread force phase and reuse it here, reducing the set up time but not affecting the communication time.

The core of the interaction is again in a native C method. The difference from the spread force phase is that the delta function weights are multiplied by the workspace velocities, and the sum of the products in the 4x4x4 surrounding region is put into the point array. Again, this subphase is computationally intensive and depends on the number of points and the speed of floating point operations and random access to arrays. Finally, using Titanium we copy the velocities from the point array back to the fiber points.

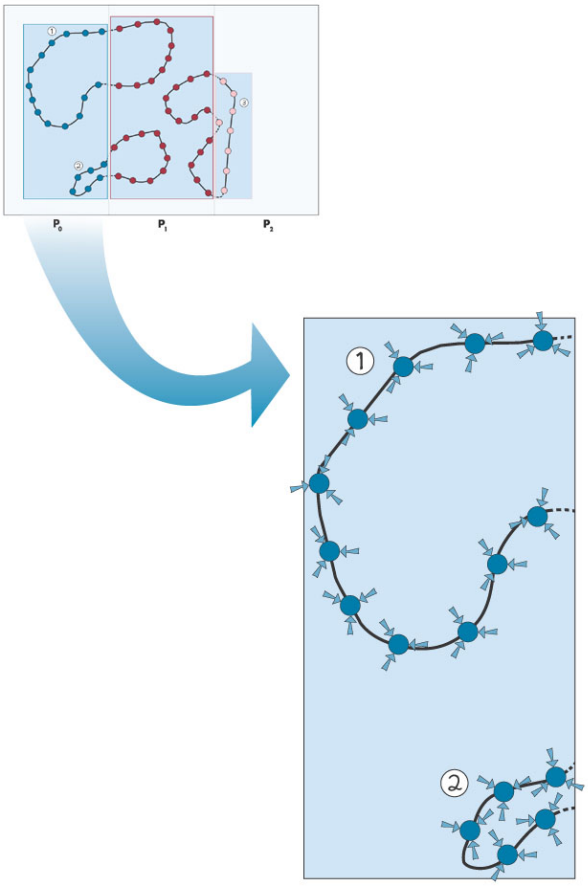

Figure 14: Interpolating velocity to fibers.

Note that the communication in this phase is similar to the spread force phase. The larger the spread of the fiber points, the larger the size of the fluid to be communicated. The partitioning strategy optimized for this phase will be similar to spread force.

4.5.2 IB Interpolate Velocity

The velocity solved during the NS Solver phase must be sampled and interpolated to each of the fiber points. This phase consists of several steps: packing the velocity cubes, sending mail, unpacking the velocity cubes, and moving the points. The velocity is stored in the fluid slabs after the fluid solve. The velocity is first copied from the slabs to the mailbox, referred to as **packing the velocity cubes**. The same communication descriptor is used as in the spread phase, and the data structure is reused here. The cubes are copied from the fluid slabs and flattened into the mailbox structure described previously. The model of this subphase is similar to packing force cubes in spread force (Figure 9), although the running time is lower because the cubes are reused here:

Time = 3 x (number of cubes) x (size of cube) x (L2 cache latency)

The mailbox is then communicated to other processes, referred to as **sending mail**. Again, we use a linear regression of sending mail time versus number of processes. (Figure 11)

After the mail is sent, the processes **unpack the cubes**. Again, this is a memory intensive operation and we use the same model as the packing operation, Figure 9:

Time = 3 x (number of cubes) x (size of cube) x (L2 cache latency)

Finally, the delta weights are calculated for every point in the 4x4x4 region surrounding each fiber point, and the product of the weight and velocity are added to the fiber point's position, called **moving points**. According to Figure 22, the parallel efficiency is close to 100% because it is a computational phase. We use the same model as the other computational phases (Figure 15):

Time = (number of floating point operations per fiber point) x (number of fiber points) / (serial mflops rate)

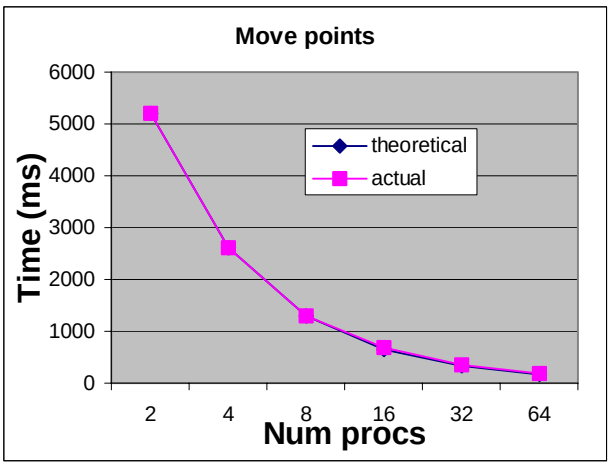

Figure 15: Performance model for moving points.

4.6 Complete Performance Model

Putting all the subphases together gives us an accurate performance model of the behavior on two machines described in the next section. Figure 16 shows the performance model versus the actual data for the 256^3 problem on Seaborg. Figure 17 shows the same for the 128^3 problem on Lion-XL.

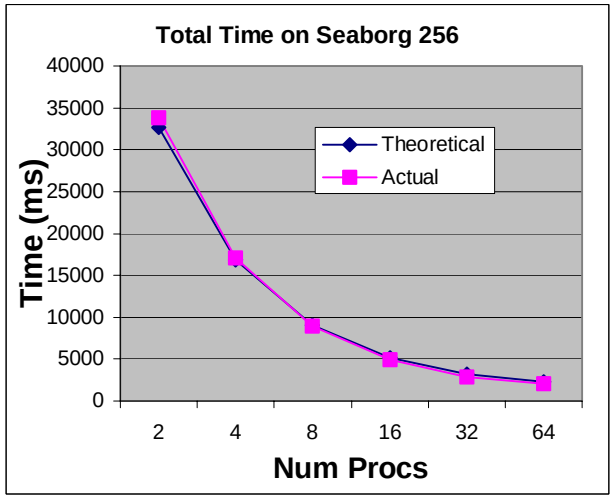

Figure 16: Performance model versus actual on Seaborg.

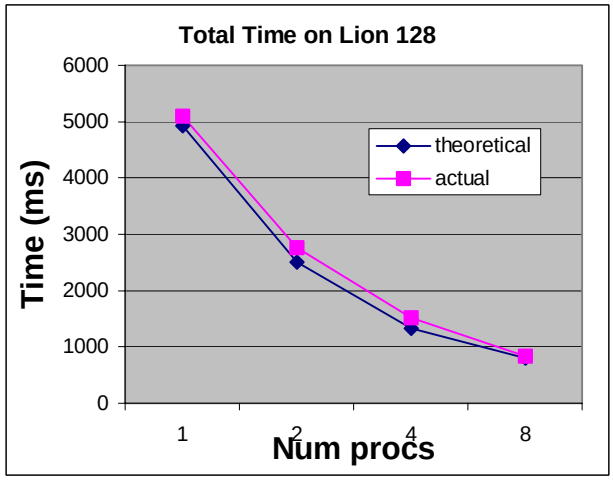

Figure 17: Performance model versus actual on Lion-XL.

5. Performance

The torus simulation was run on an IBM SP (Seaborg) at NERSC and a Dell PC cluster (Lion-XL) at Pennsylvania State University. The specifications for both machines are given in the diagram below:

Machine	**Seaborg**	**Lion-XL**
Type	IBM SP	Dell PC
Number of CPUs per Node	16	2
Number of Nodes	416	176

CPU Clock speed	365 MHz	2.4 GHz
CPU FP results/clock	4	
CPU Peak Performance	1.5 Gflops	
Communication Latency	60 micro-	
Communication BW	160 MB/s	
Network	MPI	Quadrics

Table 1: Machine specifications.

The IBM SP is a distributed-memory machine and a latency bound machine. We expect bulk communication to perform well but smaller messages to perform badly. The Quadrics network on the PC cluster performs well on small messages.

5.1 Overall Performance

We compare the performance of several fluid grid sizes for both implementations. Figure 18 shows that IB outperforms Tigibs for every grid size, and the difference is

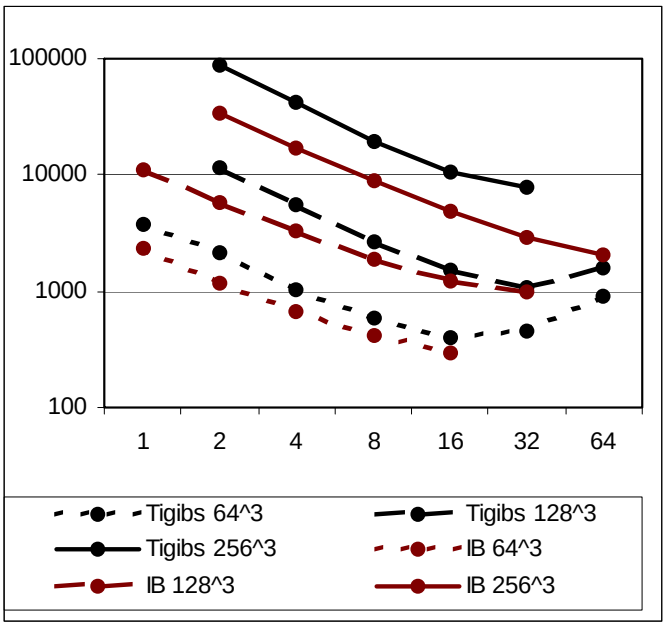

Figure 18: Total running time vs. Number of processors on Seaborg.

greatly pronounced in the largest problem size (256^3 fluid grid). As the number of processes increases, the running time of Tigibs decreases for problem sizes of 64^3 and 128^3 up to a point of diminishing returns, after which inter-node communication is too large to benefit from parallelization. This trend may be visible for 256^3 if we could run the problem on a larger number of processes. Although IB shows a small decrease in performance due to inter-node communication, running time monotonically decreases when the number of processes increases, thereby scaling better than Tigibs.

Comparing the performance on both machines for the 128^3 problem, Figure 9, shows that both implementations scale better on the PC cluster, possibly because of the faster Quadrics network.

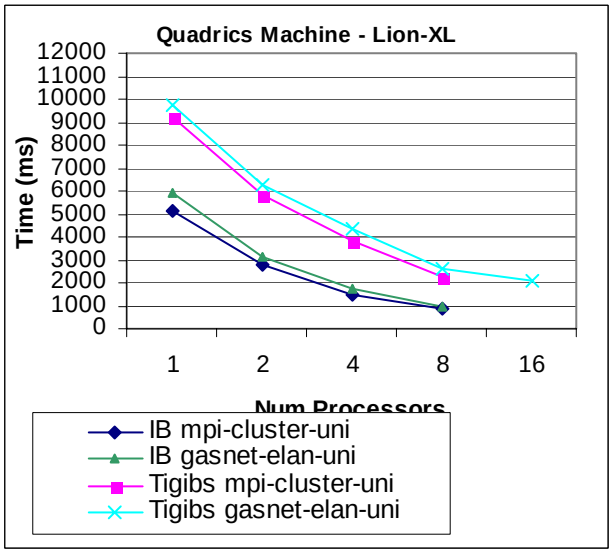

Figure 19: Running Time on the the PC cluster

5.2 Identifying Bottlenecks

Since IB clearly outperforms Tigibs, we focus our discussion on optimizations associated with the IB library. The first step in tuning the performance is identifying bottlenecks. To this extent, we look at the percentage of time in each phase of the algorithm and determine which optimizations will cause the greatest speedup.

Figure 20 shows the percentage of time spent in each phase for three different grid sizes. The smaller grid sizes (64^3 and 128^3) have similar bottlenecks, however the largest grid size (256^3) shows that the NS Solver is the major bottleneck with 44% of time. The bottleneck in the smallest grid size is the computation of the delta function (in spread force and move points-40%) followed by the communication (both send mails-28%). In the middle grid size the bottleneck is the communication (both send mails-39%) followed by the NS solver (21%). While the largest grid size's slowest phase is the NS solver (44%) followed by packing force cubes (23%).

Optimizing the delta function calculations by 50% will decrease the running time of the 64^3 problem by 20%, the 128^3 problem by 12%, and the 256^3 problem by 11%. While tuning the NS solver causes the greatest gain in performance of 9%, 11%, and 22% respectively for each grid size. Reducing the communication by 50% will allow each timestep to be run in 14%ms, 19%ms, 3%ms.

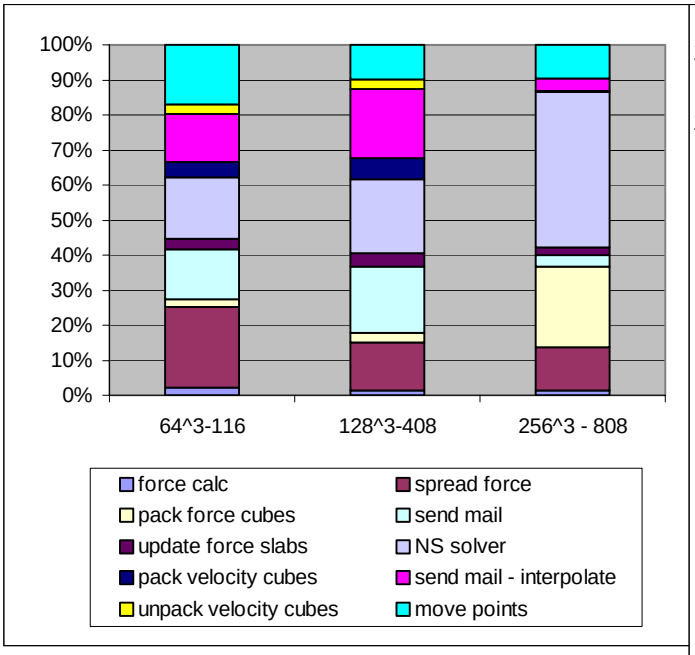

Figure 20: Percentage of time spent in phases.

Comparing the distribution of time in the phases for both machines shows that the communication time is less of a bottleneck and the NS solver becomes more important. Optimization of this phase will increase performance the most on the Quadrics machine.

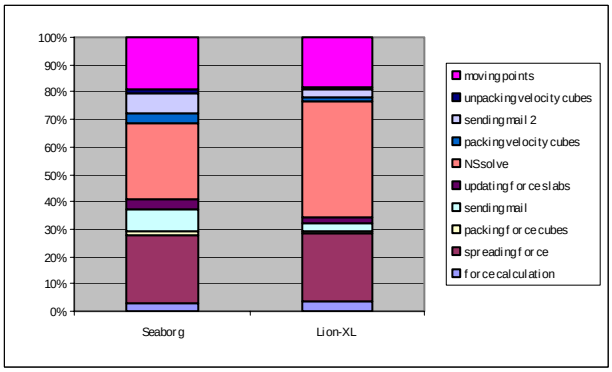

Figure 21: Percentage of time spent in phases on both machines.

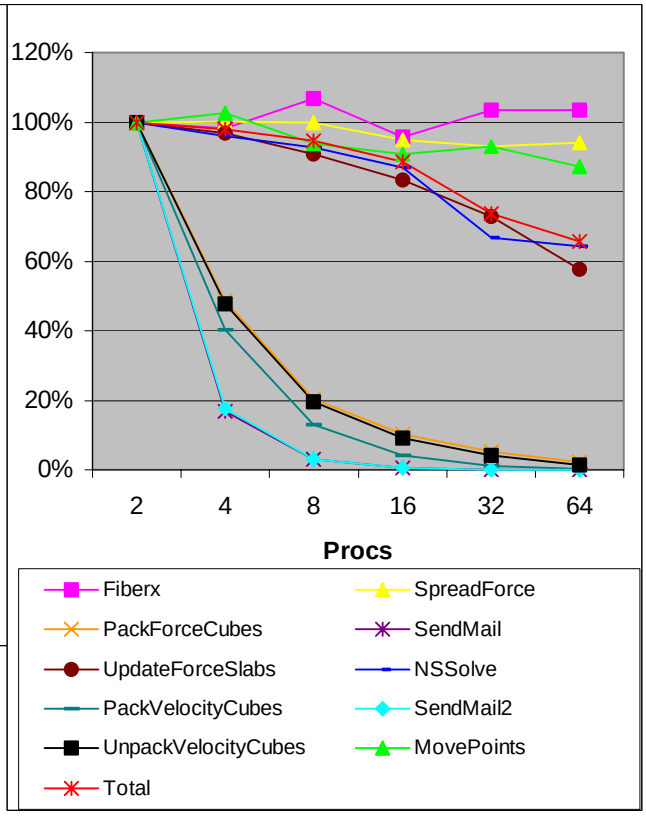

Figure 22: Computation cost in parallel efficiency.

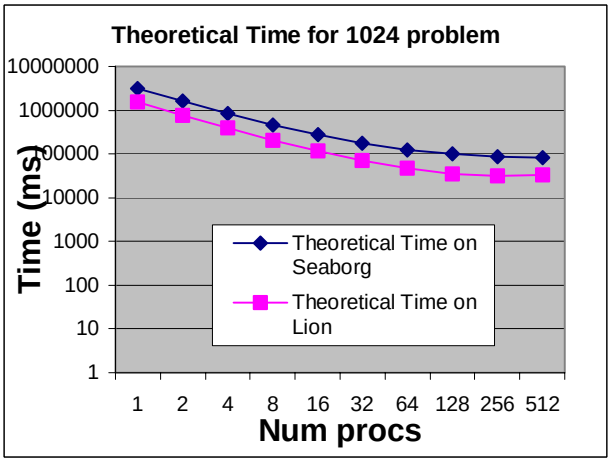

Figure 23: Performance model of 1024^3 torus.

5.3 Extending the Performance Model.

We have extended the performance model described in section 4 to predict the performance of a 1024^3 torus on both machines. Figure 23 shows a plot of the theoretical performance of the 1024^3 problem. On Seaborg, one timestep takes 83 seconds on 512 processors. On Lion, the same problem takes 33.6 seconds.

6. Future Work – Toward a Heart Simulation

The model torus sets the stage for simulation of the mammalian heartbeat. Heart simulation requires the fiber structure defined above and the immersed boundary method libraries described. In addition, the heart fiber activation varies from activation of torus fibers. Blood vessels, which act as sources and sinks, of fluid into and out of the heart also need to be added.

Heart fibers are activated according to the muscle layer the fiber belongs to and the timestep the simulation is in. The heart consists of twelve muscle layers that can be grouped into classes: atrial layer, ventricular layer, papillary layer, etc. The atrial layer is activated during systole (the first half of the simulation), the ventricular layer is activated during diastole (the second half of the simulation), and the papillary layer is activated at different times during the simulation. Fiber activation consists of decreasing the resting length and increasing the stiffness of the fiber, thereby causing contraction of that muscle layer.

The size of the heart differs from the torus. The current heart has a hard-coded grid size of 128^3 and approximately 6 M fiber points. However, higher resolutions will ultimately be needed to make new physiological discoveries from the simulation.

Sources and sinks are modeled as 4x4x4 regions in the heart that are connected by a pipe to an imaginary reservoir of fluid. For sources, the reservoirs push fluid into the region, and for sinks the reservoirs pull fluid from the region. A modified Ohm's law determines the volume flow rate into or out of the heart:

(pressure at source – pressure at reservoir)/ (resistance of the pipe)

The pressure at the reservoirs is the blood pressures of the cardiac vessels (the superior and inferior vena cavas, aorta, and pulmonary artery and vein). The resistance of the pipe is a parameter that the user can vary. Each source is surrounded by a ring of markers (points in space that move with the fluid but exert no force) that determine the location of the source.

The method of source simulation is detailed here. First, one must determine the average location of each group of markers. Then, sample the pressure, which is initially zero, of the 4x4x4 region surrounding the source point. The pressure is then used to determine the volume flow rate using the modified Ohm's law above. The divergence of the velocity, which is the average volume flow rate, is kept in a three-dimensional array the size of the fluid grid. The divergence array is zero everywhere except the regions where the sources lay and the overflow drain. The overflow drain, located in the first two planes at the edge of the fluid grid, is needed to maintain conservation of fluid flow in/out of the heart. The NS solver needs to be modified to include the divergence term, which adds two FFTs to the current implementation, a forward FFT for the divergence and an inverse FFT for the pressure. After the new velocities and pressure have been solved, we move the markers and begin the next iteration with finding the new location of the sources.

Currently, the heart simulation using the Tigibs library is in need of a numerically correct version of the sources and sinks in order to have a functional simulation. Sources and sinks have been added to the heart simulation but have a numerical bug. In addition, the heart simulation needs to be written using the IB library.

7. Conclusions

We have presented two implementations of the torus simulation, compared the simulations in terms of algorithmic organization, investigated the performance in detail, and developed a performance model. The performance model can be extended to the driving application, the heart. A heart with fluid grid resolution 1024 running for approximately 60,000 timesteps will take 22 days to complete. In order for this simulation to be feasible, performance will need to be increased by improving the distribution of the fibers as well as increasing the performance of the NS solver.

Acknowledgements

References

1. D. McQueen and C. Peskin. A general method for the computer simulation of biological systems interacting with fluids. In *Biological Fluid Dynamics*, 1995.
2. D. McQueen and C. Peskin. Shared-memory parallel vector implementation of the immersed boundary method for the computation of blood flow in the beating mammalian heart. In *Journal of Supercomputing*, 1997.
3. K. Datta and S. Merchant. Multigrid methods for titanium heart simulation. CS267 class project, Fall 2001.
4. E. Givelberg, J. Bunn and M. Rajan. Detailed simulation of the cochlea: recent progress using large shared memory parallel computers. In *Proceedings of the 2001 International Mechanical Engineering Congress*, 2001.
5. S. L. Johnsson and D. Mirkovic. Automatic performance

tuning in the UHFFT library. In *ICS*, 2001.
6. S. H. Chang. Titanium benchmarks. http://www. cs.berkeley. edu/~szuhuey/Titanium
7. A. Fogelson. A mathematical model and numerical method for studying platelet adhesion and aggregation during blood clotting. In *Journal of Computational Physics*, 1984.
8. S. Yau. Titanium generic immersed boundary software package. Master's thesis, University of California, Berkeley, Computer Science Division, 2001.