Uniform Load Balancing in PhysBAM's Fluid Simulation via Threading

Chinmayee Shah, Hang Qu, Saket Patkar, Daniel Perry

June 10, 2013

1 Introduction

PhysBAM is a physics based simulation library developed at Stanford University by Prof. Ron Fedkiw's group. It is capable of simulating a wide range of phenomenon like compressible and incompressible fluids, rigid and deformable solids, coupled fluids and solids, fracture and fire.

For this project, we focussed on simulating free surface flows on a single multiprocessor machine. Dividing the simulation domain into partitions, with a process per partition (as in the existing PhysBAM's MPI based implementation) can result in a skewed work load across processors. For example, one process may be in charge of regions that mostly contain air, resulting in little actual work being done, when that process is scheduled. A naive way to overcome this is to have more partitions and processes, so that work gets distributed more evenly between processors. However, this can result in large amount of communication on chip, as all ghost values need to be synchronized. Also, large number of partitions does not work well with stages like projection, which use techniques such as preconditionining to solve implicit equations. We used threading, instead of MPI, to overcome problems like load imbalance and different requirements for stages like advection and projection.

Using threading, we divided the domain into small partitions, assigning a thread to each partition. Once a thread is done with one partition, it moves to another partition. This allows us to divide work more evenly, while eliminating communication for synchronization of ghost values. For stages like preconditioning in projection, which do not perform well with a lot of subdivision, we use fewer threads to operate on the same data. For advection, we partition the complete domain, testing for presence of water, thus avoiding wasteful work. For projection, we partition regions as they are solved, so as to speed up the solve on each region. We studied the effects of varying the chunk sizes. We also measured CPU idle time for our method and compared it with PhysBAM's MPI implementation to prove the efficacy of our method.

In the following sections, we describe the changes that we made to PhysBAM for some major steps in the water simulation, our observations and conclusions.

2 Advection

Advection performs an update on various field quantities such as velocity, particle positions, signed distance and density, by moving the quantities over the velocity field. A field $\vec{q}$ advected over a velocity field $\vec{u}$ is updated as,

$$\frac{\partial \vec{q}}{\partial t} + \vec{u} \cdot \nabla \vec{q} = 0 \tag{1}$$

The PhysBAM water simulation that we are studying performs semi-Lagrangian advection for velocity and signed distance, and second order Runge-Kutta advection for particles. In all cases, advection of a quantity residing on a face, cell center or vertex needs access to neighboring values of the same quantity, as well as velocity.

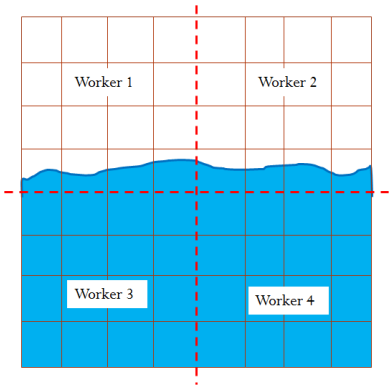

Figure 1: PhysBAM's way of dividing work among 4 workers.

2.1 Problems with current implementation

For the purpose of this project, we restrict ourselves to velocity advection. PhysBAM's current implementation performs advection over the entire domain being simulated. Such a strategy is quite wasteful for free surface flows where a large portion of the domain is covered by air, since one is interested in advection of quantities only near and below the fluid surface. We suspect that the main reason behind using this strategy is the fact that, when running with multiple processors, the overall speed is determined by the speed of the slowest processor. So, if one processor's domain is filled completely with fluid, optimizing other processors for air does not result in any speedup. On the contrary, additional

branches to check for air or fluid, for every cell, will only make the slowest processor even slower.

The imbalance in work (and fluid) distribution arises from the way PhysBAM divides the simulation domain. Currently, PhysBAM exploits parallelism through MPI processes, even when running on a single machine. In order to distribute work to each process, it divides the domain into uniform, fixed disjoint pieces, equal in number to the number of MPI processes (workers) invoked (see Figure 1). While such a division may be extremely cache coherent, it may result in a very uneven work distribution. Also, such an allocation scheme is static – once a region is allocated to a worker, the process stays in charge of the region for the entire simulation.

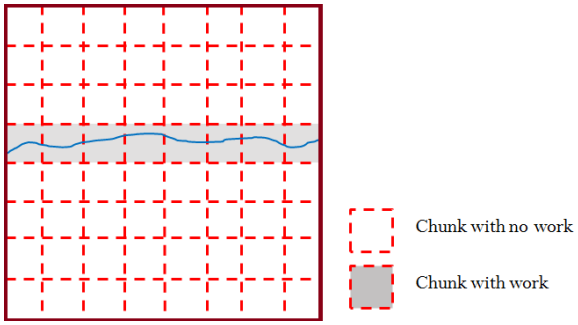

Figure 2: Dividing the domain into smaller chunks.

2.2 Work distribution through threading

To alleviate the problem of unbalanced distribution of work, we decided to exploit multi-core opportunities through threads instead of processes. Threading allows us to allocate work to processors on the fly – once a thread completes one piece of work, it can move to the next piece of work in the queue. We divide the simulation domain into small chunks, such that number of chunks is much larger than number of threads, or available cores (see Figure 2). We use the Intel TBB library to split the domain into small pieces, and `parallel_for` and `parallel_reduce` constructs to schedule these pieces of work. We cannot perform such dynamic distribution of work with PhysBAM's current MPI implementation, because a domain once allocated to a process, cannot be changed for the entire simulation, and large number of small chunks does not work well for projection, which involves implicit equations.

Once we were able to distribute work dynamically using threads, we eliminated wasteful work by performing advection only in regions with water. To achieve this, we pass the signed distance to the advection code. We perform advec-

tion only in regions where signed distance is below a positive threshold – thus restricting the operation to near and below the surface. While this adds an additional branch to the advection code, the speed-ups that we obtained overshadow the cost of branching.

2.3 Domain division

PhysBAM lays out the data on the grid into a single 1D array. The index of a point (x, y, z) in a $m \times n \times k$ grid is given by $(x \times m + y) \times n + z$. We restricted ourselves to partitioning the data by partitioning this array – resulting in partitions along the x dimension.

2.4 Results

We performed a series of tests with different initial conditions for water, and compared the performance of our threaded implementation with PhysBAM's MPI implementation. The tests were run on an Intel laptop, equipped with a i7-2820QM processor, that has 4 cores, with 2 hyperthreads per core. We chose 4 different initial conditions, that we believe are representative of the conditions in a single-phase fluid simulation. Tests 1 and 2 have water in approximately half the domain, test 3 has an extremely skewed distribution of water while test 4 has water over almost the entire domain. Figure 3 provides a snap-shot of the cross section of the initial conditions, for these tests. All tests were run for 3d domain.

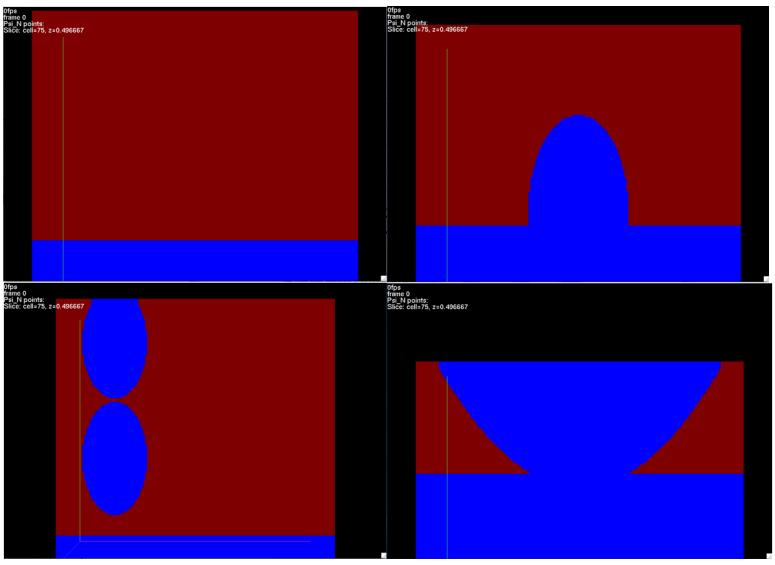

Figure 3: Top left image shows the cross section of the initial condition for test 1, top right for test 2, bottom left for test 3 and bottom right for test 4.

To make a fair comparison, we also modified the MPI code to perform advection only near and below the water surface. We ran tests at various resolutions, using 1, 2 and 4 workers, results summarized in Table 1. Figure 4 compares the performance for a $250 \times 250 \times 250$ grid, for 2 extreme cases (tests 3 and 4).

Test	Scale	1 proc	1 thread	2 procs	2 threads	4 procs	4 threads
Test 1	150	0.34	0.38	0.18	0.19	0.18	0.11
	200	0.80	0.87	0.42	0.46	0.41	0.24
	250	1.5	1.74	0.8	0.92	0.8	0.49
Test 2	150	0.55	0.54	0.28	0.30	0.27	0.17
	200	1.27	1.33	0.70	0.68	0.64	0.39
	250	2.45	2.66	1.30	1.38	1.30	0.76
Test 3	150	0.37	0.40	0.29	0.21	0.18	0.14
	200	0.86	0.95	0.68	0.49	0.43	0.30
	250	1.60	1.88	1.30	0.98	0.80	0.58
Test 4	150	1.02	1.00	0.55	0.51	0.33	0.30
	200	2.44	2.40	1.29	1.22	0.77	0.70
	250	4.64	4.54	2.42	2.41	1.50	1.33

Table 1: The numbers represent the time for advection, in seconds, averaged over 3 frames, each consisting of several time steps. Threaded code performs better than MPI code with 4 workers, for tests 1, 2 and 3. Both codes give comparable performance for test 4.

Threaded code with 1 worker performs worse, compared to MPI with 1 worker, for most cases. We believe that this is due to the overhead, added by Intel TBB library (splitting and scheduling, that is performed even for a single thread). For most cases, threaded implementation performs comparable to, or better than MPI version for 2 and 4 workers. When the imbalance in work distribution is high, as in test 2 and 3, threads performs up to 40% faster than MPI. When the distribution of water is more uniform, as in test 4, 2 and 4 threads perform comparable to MPI. We believe that with larger number of cores per machine, the difference in performance of threaded and MPI code will become even more.

2.5 Remarks

Conceptually, signed distance and particle advection can also be threaded using the same approach as velocity. However, advecting these quantities requires modifying certain PhysBAM data structures, and requires a significant amount of more effort, compared to velocity advection. Hence, we ignore these stages for the purpose of this project. For instance, particle advection performs several list operations for updating particle positions. Threading particle advection requires us to either maintain a separate list per thread and synchronize the lists, or maintain a global list and use locks to perform updates on the list. This makes the threaded approach inefficient. MPI implementation eliminates

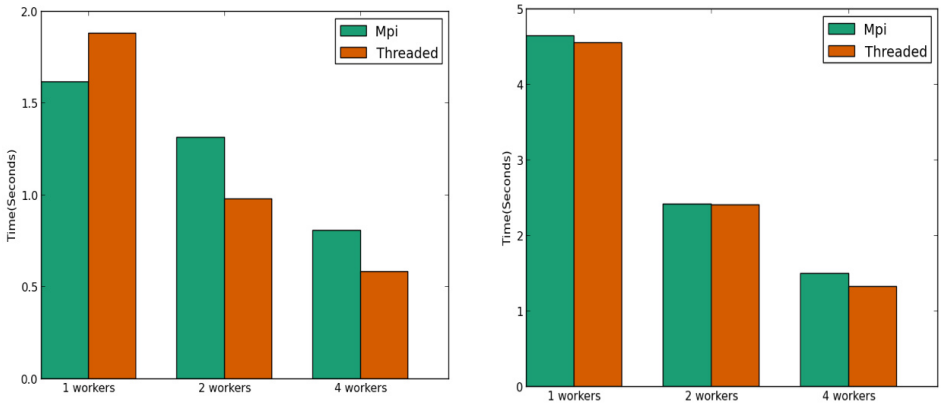

Figure 4: Left plot shows performance for test 3, right plot shows performance for test 4, for velocity advection.

the need for synchronization by having different processes deal with different regions, and maintaining a separate list for each region. We feel that we can use a separate list per chunk, similar to MPI, and still have the benefits of uniform and dynamic work distribution, by using threads and a reasonable number of chunks. However, this requires further investigation, and we leave it for future.

3 Projection

Projection is the final stage in a simulation step. It operates on velocity and pressure fields, and makes the fluid velocity divergence free. To do this, projection computes a pressure such that applying it to velocity field makes velocity divergence free, as follows,

$$\vec{u}^{n+1} = \vec{u} - \frac{\Delta t}{\rho}\nabla p \tag{2}$$

$$\nabla \cdot \vec{u}^{n+1} = 0 \tag{3}$$

Applying equation 3 to equation 2 results in a linear system, with as many equations (or variables) as number of cells. Using the computed pressure, $\vec{u}^{n+1}$ can be updated with equation 2. PhysBAM first performs a flood-fill operation to find out the number of connected regions, and then solves each region using preconditioned conjugate gradient method (PCG).

3.1 Limitations with current implementation

With PhysBAM's MPI implementation, it is not possible to change the regions allocated to processors, as the simulation progresses. As a result, one processor

may contain a large portion of a water component, while another processor may host only a small portion of the region. This results in an imbalance, both at the preconditioning stage, and the conjugate gradient stage of PCG. Another drawback with the current implementation is that, there may be processors that host no water, which may end up doing no work – when they could have shared some of the work, if they had access to the data.

3.2 Threading implementation

It turns out that splitting the domain into a large number of chunks does not work well for the preconditioning part in PCG. If a region is split into n chunks, PCG first constructs an n block incomplete Cholesky LU factorization, and then performs forward and backward solve at the beginning of each iteration using the computed L and U (lower and upper triangular) matrices. Making n larger results in larger number of blocks, making the preconditioning weak, and resulting in larger number of iterations. Hence, we decided to keep the number of blocks, n, equal to the number of workers. For a machine with larger number of cores, we can make n less than the number of workers too. Although this seems similar to what PhysBAM currently does, there is one important difference. We make n blocks of same size, for each region (color). In the MPI implementation, since a region may be divided across processors, number of blocks and block size depends on the way the region is divided across processors, and may result in blocks of extremely different sizes. As mentioned before, some cores that do not host any water from a solution region may end up wasting CPU cycles, performing no useful work. We use the `task_group` construct from Intel's TBB to run parallel forward and backward solves.

Other operations such as sparse matrix-vector multiplication, dot product, vector addition and calculating maximum component of a vector are highly parallelizable. These operations constitute most of the conjugate gradient part of PCG. We parallelize these operations by splitting matrices and vectors into small chunks. This is different from the splitting performed in advection, in the sense that the matrix and vectors corresponding to each solution region, rather than the entire domain, are split. We use `parallel_for` and `parallel_reduce` constructs from Intel's TBB library to perform splitting and to schedule the chunks. Splitting the matrices and vectors into chunks, equal to the number of workers, is sufficient to distribute work evenly across the processors. The motivation behind splitting these structures into small chunks is to compensate for certain cores that may be slow due to reasons such as other ongoing tasks.

We did not parallelize some parts of projection due to lack of time. These steps include setting the boundary conditions, constructing the actual system matrix and constructing the preconditioner (incomplete Cholesky factorization). These operations can be parallelized in a manner similar to MPI's imeplementation, but require modifying PhysBAM's data structures. Also, these operations are outside the main PCG iteration loop, and hence, cause only a small performance

Test	Scale	1 proc	1 thread	2 procs	2 threads	4 procs	4 threads
Test 1	150	1.2	1.46	0.71	1.19	0.76	1.11
	200	3.0	3.47	1.67	1.97	1.80	1.83
	250	5.92	6.81	3.3	3.87	3.63	2.95
Test 2	150	2.25	2.60	1.26	1.48	1.44	1.38
	200	5.33	6.15	3.01	3.50	3.47	2.67
	250	10.46	12.07	5.83	6.82	7.02	5.31
Test 3	150	1.08	1.24	1.10	1.13	0.97	1.09
	200	2.77	3.20	2.65	2.69	2.81	2.59
	250	5.90	6.08	5.26	3.98	5.26	3.75
Test 4	150	4.54	5.24	2.51	2.96	2.14	2.24
	200	10.77	12.40	6.03	7.03	5.14	5.55
	250	22.54	25.26	12.03	14.39	10.55	10.87

Table 2: The numbers represent the time for projection, in seconds, averaged over 3 frames, each consisting of several time steps.

hit in large systems.

3.3 Domain division

As mentioned before, we have implemented 2 different kinds of division for projection. First, we split the region being solved into n blocks, where n equals number of workers, and perform Cholesky factorization on each block. We then use these n blocks to perform forward and backward solve, in parallel within the PCG iteration loop. The rest of the loop consists of matrix-vector multiplication, dot product, vector sum and maximum component calculation. We calculate chunk size as a function of the number of cells in the solution region (keeping a fixed, minimum chunk size), pass it to TBB and use TBB's built-in range splitting feature, to parallelize the conjugate gradient operations.

3.4 Results

We measured performance for the threaded implementation for the same test cases as advection in Section 2.4, Figure 3. The tests were run on an Intel laptop having i7-2820QM processor. The results, averaged over 3 frames, are summarized in Table 2. Figure 5 compares the performance of threaded code with the MPI implementation for 2 extreme cases, for a $250 \times 250 \times 250$ grid.

The presented projection numbers correspond to the total time for Cholesky factorization (computing preconditioner), forward and backward solve, and the conjugate gradient operations. We have omitted the time corresponding to flood-fill operation and construction of A matrix, since these operations are easily parallelizable, but require touching a lot of PhysBAM data structures.

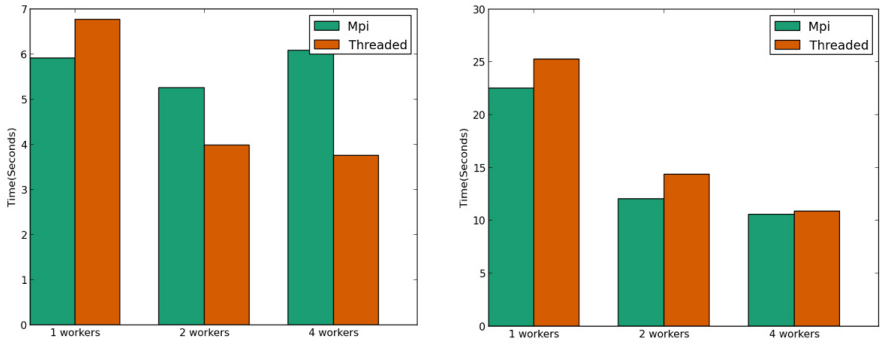

Figure 5: Left plot shows performance for test 3, right plot shows performance for test 4, for projection.

We can parallelize Cholesky factorization for an n-block diagonal matrix, over n workers. However, we could not implement the change due to time constraints. We expect the performance numbers for smaller systems ($150 \times 150 \times 150$ and $200 \times 200 \times 200$) to improve slightly, with the modification.

With present threaded implementation, we obtain performance comparable to, or better than MPI, for 4 workers. We obtain a performance improvement of up to 30% for the test cases that we built. We believe that the threaded code still suffers a slight performance hit due to overhead from Intel TBB, and serialized preconditioner computation, resulting in poor performance with 1 and 2 workers.

4 Varying chunk size

We performed several tests with different chunk sizes for advection, and conjugate gradient operations in projection, for scale $250 \times 250 \times 250$. For advection, we changed the size of chunks along the outermost dimension (x in our case). For projection, we increased the number of chunks, as a multiple of the total number of workers. This implies $\frac{l}{k \cdot n}$ size chunks for l-size vector, with n threads, and where k is an integer. We imposed a minimum limit on the chunk size, to ensure that we do not split the domain into extremely tiny regions.

We varied k from 1 to 4, and did not see any significant difference in the performance for projection. For advection, we varied the chunk size from 1 to 128, and saw a performance dip for large sizes. This can be attributed to the fact that as chunk size increases, number of chunks decreases. When total number of chunks is less than the number of workers, performance deteriorates, due to less parallelism. Also, with large chunks, work distribution across threads gets skewed, in cases where water is not distributed uniformly across the domain.

For the test cases that we built, sizes less than 16 were sufficient to distribute work in a fair manner. Advection performance was insensitive to variation in chunk size below 16. This number varied slightly for different tests. Tests 1 and 4 were insenitive to any variation in chunk size, as long as enough chunks were available. We can attribute this to the uniform distribution of water along x dimension, in these cases. Tests 2 and 3 were more sensitive to chunk size, due to different distribution of water along x axis, but were insensitive to size variation below 16.

5 CPU utilization

Uniform distribution of work should improve average CPU utilization, as well as reduce variance in utilization across different CPUs. To study how threading affects CPU utilization, we measured CPU idle time for our threaded implementation and PhysBAM's MPI implementation. Since we did not parallelize the entire water simulation, we measured idle time for only the steps we parallelized. However, this still includes serialized incomplete Cholesky factorization in the threaded code. Since all steps do not require syncrhonization, a CPU with less amount of work can move ahead to the next step in the MPI code, giving a false, high utilization number. For instance, a process with little water may move ahead to the next step, while another process is still advecting velocities. To isolate readings for only the parallelized steps, we inserted an MPI barrier right before and after advection and projection. We believe this is still representative of the actual CPU utilization, because the faster process would eventually have to wait for the slower process during some phase that requires syncrhonization, such as global reductions or exchange of ghost data.

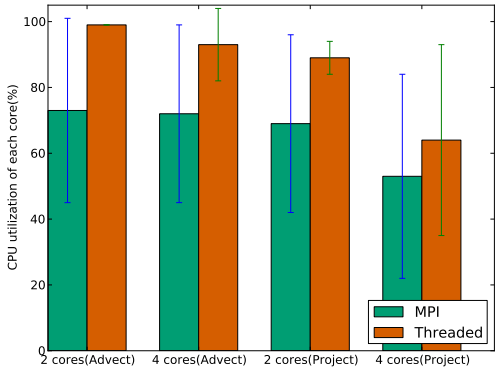

Figure 6: CPU utilization in percentage, for test 3, averaged over all running cores for each step. The error bars show the standard deviation, averaged over several points.

As shown in Table 3 and Table 4, threaded implementation maintains an average utilization of over 90% during advection and over 85% for projection, for most cases. As the number of workers increases to 4, threaded code outperforms MPI code in terms of utilization. Figure 6 compares utilization during velocity advection and projection phases, for MPI and threaded code, averaged over running CPUs. The error bars represent the standard deviation in utilization of each CPU. CPU utilization and standard deviation are averaged over several measurements. We see that the standard deviation for CPU utilization is high for MPI implementation. This is because of the large variation in the amount of water that each processor hosts, resulting in a large variation in the amount of work each processor does.

Test	2 procs	2 threads	4 procs	4 threads
Test 1	98	99	66	97
Test 2	100	100	69	96
Test 3	73	99	69	89
Test 4	100	100	93	97

Table 3: CPU utilization % for advection, averaged over several time steps.

Test	2 procs	2 threads	4 procs	4 threads
Test 1	98	92	61	85
Test 2	98	92	66	86
Test 3	72	93	53	64
Test 4	98	93	93	87

Table 4: CPU utilization %, for projection, averaged over several time steps.

Test 3 gives a low CPU utilization for projection for both, threaded and MPI code. Nevertheless, threaded code still performs better than the MPI code. We feel that after parallelizing Cholesky factorization, we should be able to get even better performance, mainly because there are 3 small regions over which projection occurs.

6 Conclusions and future work

Through this project, we showed that it is possible to achieve better overall CPU utilization through more uniform work distribution, for fluid simulations like free surface flows, by dividing work into small chunks. Even with a simple splitting strategy and a straight-forward scheduler, we obtained up to 30% to 40% improvement over the MPI implementation. In cases with more uniform

distribution of water throughout the simulation domain, we obtained performance comparable to MPI for large grids and large number of workers. We did not see much variation in the performance with variation in chunk size, as long as chunks were small enough to distribute work evenly. We found the overall CPU utilization to be higher in threaded cases with same number of workers as MPI, agreeing with the conclusion that we can achieve better work distribution and CPU utilization through small chunks of work.

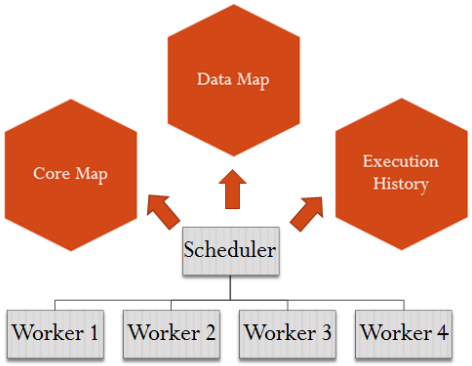

Figure 7: A scheduler allocating work to 4 workers.

In future, we would like to implement threading for other parts of PhysBAM like advection of signed distance and particles, flood-fill, construction of matrix and computation of pre-conditioner, other particle level-set operations, kinematic evolution and application of forces, and extrapolation. We should see a further improvement in projection performance, especially for small systems, with pre-conditioner computation parallelized. We would like to try a different splitting strategy (domain-wise, that is, splitting along x, y and z) and investigate if that affects the performance of advection due to better locality. We would like to study how our code performs in a heterogeneous environment when all cores are not of same configuration.

We would also like to use another threading library like pthreads, with our own scheduler, and see if we can reduce the overhead for fewer workers, thus reducing the overhead from threading. We would like to build a more sophisticated scheduler, wherein we can specify a policy based on information about execution history and data locality, and extend the idea of uniform work distribution to a cluster of machines, with multiple cores per machine and possibly heterogeneous machines (see Figure 7). We would like to build in fault tolerance into the system, where a controller can roll back the simulation if a processor or machine fails, recovering any lost data.