Parallel Modeling of Fish Interaction

Lamia Youseff[a] Alethea Barbaro[b] Peterson Trethewey[a,b]
Bjorn Birnir[b,c,d] John Gilbert[a]

[a]Department of Computer Science,
[b]Department of Mathematics,
[c]Center for Complex and Nonlinear Science
University of California
Santa Barbara, CA 93106 USA

[d]University of Iceland
107 Reykjavk ICELAND

Abstract

This paper summarizes work on a parallel algorithm for an interacting particle model, derived from the model by Czirok, Vicsek, et. al. [13] [3] [14] [4] [5]. Our model is particularly geared toward simulating the behavior of fish in large shoals. In this paper, the background and motivation for the problem are given, as well as an introduction to the mathematical model. A discussion of implementing this model in MATLAB and C++ follows. The parallel implementation is discussed with challenges particular to this mathematical model and how the authors addressed these challenges. Both static and dynamic load balancing were performed and are discussed. Finally, a performance analysis follows, using a performance metric to compare the MATLAB, C++, and parallelized code.

1 Introduction

The capelin is a species of pelagic fish which lives in the northern oceans. There are several stocks in the Northern Atlantic ocean; we are particularly interested in the stock which migrates in the seas around and north of Iceland. This stock forms large shoals off the northern coast of Iceland generally made up of billions of individuals. Each year, the mature portion of this stock undertakes an extensive migration to feed on the zooplankton whose population swells during the vernal phytoplankton bloom to the northeast of Jan Mayen [15] [16]. In the fall, the fish return to the northern coast of Iceland and the portion of the stock which undertook the feeding migration swims around Iceland to the southern coast. The fish then spawn and the adults die. The young drift with the tidal current to mature off the northern coast.

The goal of our work is to model the life cycle and migration route of this particular stock of capelin with schools of up to a million individuals. The Icelandic fishing industry is interested in an accurate model of the migration of this stock because the capelin is important to the region both economically and ecologically. The fishing industry fishes the stock for export, but this stock is one of the main food sources for many of the larger, more economically valuable fish in the vicinity. Hence, it is important that the stock not be overfished. Because the migration route of the capelin seems to be highly dependent on ocean temperature and currents, it is difficult to find the stock at a given time and therefore difficult to gauge the number of capelin during a given year [2]. The stock in other oceans has been catastrophically depleted by miscalculation of the number of fish in a stock when the fishing industry has happened across very dense pockets of fish during years when the stock is small. It is therefore extremely important to keep careful track of the location of the various parts of the stock of the capelin to avoid similar catastrophes in this region.

Other groups are also working on numerical simulations to reproduce this migration, see [8] [10] [9]. These groups have obtained reasonable spawning migrations using a comparatively small number of inter-

acting particles representing superindividuals. Their models include currents, temperature gradients, and a forcing term which simulate a homing instinct to draw the fish to the feeding and spawning grounds at the correct times. We are interested in simulating the migration with a quantitatively accurate number of fish and without these forcing terms. This paper addresses necessary architecture for simulating

One problem which is common to models of interacting particles is that it is computationally expensive to simulate a suitably large number of particles. Simulating many particles is important, however, because local information can become global information via local interaction if there are adequately many particles. In this paper, we address one possible solution to this problem. Our model is based on the model described in [1], which in turn is derived from the model in [8]. In this paper, we discuss transitioning the code from the MATLAB implementation discussed in [1] to sequential C^{++} and eventually to MPI. We discuss the various challenges we encountered parallelizing the code and the strategies we used to address these challenges, including geographic division of our space, ghost fish, shadow oceans, and static and dynamic load balancing.

This paper is organized as follows. In the next section, we describe our mathematical model and present the necessary background on the fish interation schemes in our model, and the formulas governing it. In Section 3, we describe our computational implementation of the mathematical model, including the serial model in Matlab and C++, as well as the parallel model in MPI. Section 4 discusses the major challenge in any interacting particle model, including our model. In this section, we detail our approach to the load balancing problem, including our statis and dynamic mythodologies we employed. We quantify the performance of our model in section five, and conclude the paper in the following section.

2 Background

2.1 Fish interaction

In our model, fish interact with each other locally. All fish are identical and there are no fish which are designated as "leaders," i.e. having more information or behaving differently from the other fish. Each fish interacts only with fish within a certain finite region and ignore information from all other fish. Generally speaking, each fish tries to head away from fish which are too close, align with fish which are reasonably close, and head toward fish which are too far away. Avoiding fish which are too close averts collisions, aligning with neighbors allows the fish to form cohesive schools to help avoid predation and offer hydrodynamic advantages, and getting closer to fish which are far away helps fish avoid being alone and encourages the formation of schools [11]. This type of behavior has been observed in schooling fish by biologists and has been shown to be motivated by vision and the lateral line, a sense organ which detects pressure changes and runs down the side of many species of fish including the capelin[12].

2.2 Zones of interaction

In our model, we simulate these effects using the zones of interaction shown in Figure 1. Three zones are defined by three concentric circles around every fish: the *zone of repulsion*, the *zone of orientation*, and the *zone of attraction*. The smallest circle is the *zone of repulsion* of the fish at the center of the diagram. The annulus between the zone of repulsion and the larger circle is the *zone of orientation* and the annulus outside the zone of repulsion and inside the largest circle is called the *zone of attraction*. Fish try to head toward fish in their zone of attraction, try to align in speed and direction with fish in their zone of orientation, and try to head away from fish in their zone of repulsion. They do this by taking an average of all these (often conflicting) desires; for details, see Section 2.3. In our implementation, each zone is given equal weighting and the radii of the different zones is a parameter which can be adjusted to create individual variations among the fish.

2.3 Mathematical Model

Our model is derived from the interacting particle model first presented by Czirók, Vicsek, et. al., see [13] [3] [14] [4] [5], and later adapted by [8] and then [1]. Fish change their directional heading and speed at each time step by reacting to nearby fish through the spherical zones of interaction described in Section 2.2. The algorithm for updating the kth fish's speed is

$$v_k(t + \Delta t) = \frac{1}{N} \sum_{i=1}^{N} v_i(t)$$

where there are N fish inside the zone of orientation of fish k. Letting ϕ_k be the directional heading of the kth fish, we update its directional heading at each time step according to the following rule:

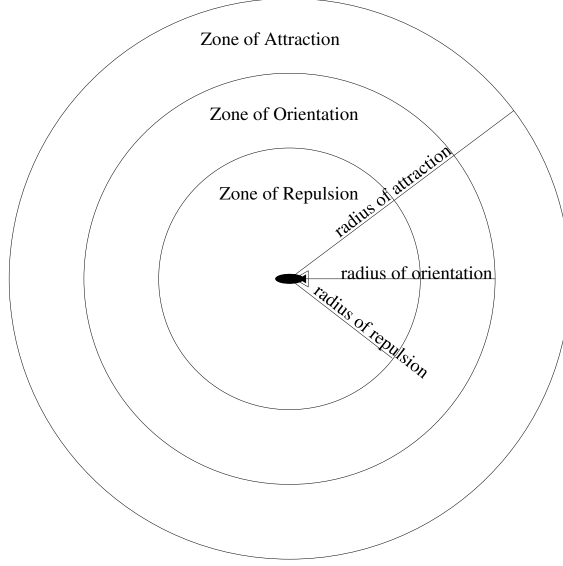

Figure 1. This figure depicts the zones of interaction of a fish in our simulation.

$$\cos(\phi_k(t+\Delta t)) = \frac{1}{R+N+A}\Big(\sum_{r=1}^{R}\frac{x_k(t)-x_r(t)}{(x_k(t)-x_r(t))^2+(y_k(t)-y_r(t))^2} + \sum_{n=1}^{N}\cos(\phi_n(t)) + \sum_{a=1}^{A}\frac{x_a(t)-x_k(t)}{(x_k(t)-x_a(t))^2+(y_k(t)-y_a(t))^2}\Big)$$

$$\sin(\phi_k(t+\Delta t)) = \frac{1}{R+N+A}\Big(\sum_{r=1}^{R}\frac{y_k(t)-y_r(t)}{(x_k(t)-x_r(t))^2+(y_k(t)-y_r(t))^2} + \sum_{n=1}^{N}\sin(\phi_n(t)) + \sum_{a=1}^{A}\frac{y_a(t)-y_k(t)}{(x_k(t)-x_a(t))^2+(y_k(t)-y_a(t))^2}\Big)$$

Here, R, N, and A are the number of fish in the kth fish's zone of repulsion, orientation, and attraction, respectively. The indices r, n, and a run through all the fish in their respective zones. The algorithm uses the speed and directional heading from these calculations to move each fish according to its velocity as follows:

$$\begin{pmatrix} x_k(t+\Delta t) \\ y_k(t+\Delta t) \end{pmatrix} = \begin{pmatrix} x_k(t) \\ y_k(t) \end{pmatrix} + v_k(t)\begin{pmatrix} \cos(\phi_k(t)) \\ \sin(\phi_k(t)) \end{pmatrix}\Delta t. \tag{1}$$

3 Interaction Simulation Model

3.1 Matlab and C^{++} Models

In [1], Barbaro, Birnir and Taylor implement the model described in Section 2.3 in MATLAB. For completeness, we began our analysis with this implementation. In this code, there is no sorting of the fish, and every fish computes its distance to every other fish at every time step to see if it needs to react to the other fish. This slows the code significantly as the number of fish increases, making it prohibitively expensive to model the number of individuals necessary for realistic simulations of shoals of the capelin. It is therefore necessary to move the code to another platform such as C^{++}.

3.1.1 Class Architecture

Our model consists of three main classes: Fish, Ocean and World. The Fish class stores coordinate and velocity data for a fish, the Ocean is meant to represent a single body of water, whereas a World is a bigger body of water composed of several connected oceans. Each fish stores an x and y coordinate for its location in the world. A fish stores its velocity as the cosine and sine of its direction angle together with a non-negative speed. The Ocean class has a member

variable "fish" which is an array of Fish living on that ocean. For a performance improvement, we sort the list of fish on an ocean by x coordinate, and when fish on an ocean interact, they need only compare their positions with fish nearby in the sorted list. This saves us from an "all-to-all" comparison for proximity detection. The Ocean class has member functions which iterate through the fish, updating their velocities and moving them. The Ocean class does not handle any MPI communication. The World class contains a 2-dimensional array of oceans. Member functions of the World class iterate through the oceans and instruct each ocean to interact or move its fish. The World class handles the communication of the fish between oceans both locally and over a network with MPI. In our implementation the oceans in a world are either connected in a torus or not according to a flag in the World class. If the torus flag is set to false, fish which traverse the boundary of the world disappear from the simulation. When the torus flag is set to true, the top edge of the world is identified with the bottom edge of the world and likewise left and right, so fish which traverse a boundary of the world teleport to the other side of the world.

3.1.2 Local Communication

Even if a World is instantiated on a sequential machine, the oceans in that world do a network-style communication of fish. The motion and interaction of fish in a time step is computed one ocean at a time. Between time steps, the oceans inform each other of pertinent fish.

There are two phases of communication per time step:

1. Before the fish can interact, each ocean needs to be informed of sufficiently nearby fish on neighboring oceans. We do this by adding a copy of each fish in the neighboring ocean (provided it's within the largest radius of attraction of the boundary) with a flag set to indicate that the fish is a "ghost". These fish need to be present to affect other fish, but in the interaction phase of computation, ghost fish need not have their velocity updated.

2. When fish migrate to an ocean from a neighboring ocean, the migrant fish need to be removed from the source ocean's list and added to the receiving ocean's list.

In communication phase (1) where ghost fish are sent from ocean to ocean, we employ a trick that uses sorting to save computational effort. Each ocean needs to communicate ghost fish to its eight neighboring oceans in the 2-dimensional model. The sets of ghost fish destined for the various neighbors tend to intersect nontrivially. For instance, the ghost fish which need to get sent to the neighbor in the upper-right corner of an ocean also need to get sent to the ocean directly to the right. To avoid redundant computation, we first divide the ocean up into regions We assign to each fish the number of the region in which that fish is currently located. We then sort the entire array of fish by region number. Because of the ordering of the region numbers, the fish that need to get sent to any neighboring ocean will be contiguous in the sorted list. In communication phase (2), when migrating fish are transported to their target oceans, we use a similar scheme, except that this time we assign a number to the fish according to its destination ocean.

3.2 Parallelization

The challenges of parallelization lie mainly in (1) and (2) above. To address these challenges, when running in parallel each thread instantiates one world, the structure of which mimics the worlds on all other threads. To divide the work among threads, each thread is assigned a connected set of oceans to compute. On each thread, the oceans in the world which are not assigned to that thread begin each time step empty of fish. We call such an ocean a *shadow ocean.* During each time step, we first allow the fish in the world to interact, then we move the fish according to the rules described in Section 2.3. When moving the fish, the processor simply iterates through all its assigned oceans, and for each ocean advances every fish in that ocean by its velocity. This might move fish into the processor's shadow oceans, so a network communication pass follows which moves the fish in each of a processor's shadow oceans into the identical ocean on the proper thread. Before the fish interact with each other to update their velocities, the oceans need to be informed of ghost fish. So, on each thread, the world iterates through all oceans, and performs a local communication of ghost fish. Ghost fish spill into shadow oceans, so a network communication pass follows which sends the ghost fish in each shadow ocean into the identical ocean on the proper thread. Once all ghost fish are in place, the real fish can update their velocities. Once updated, all ghost fish are removed before the next time step.

By organizing the procedure in this way, we made a healthy barrier between the computation of fish motion and the inter-thread communication of fish. The movement and interaction code can run blind to the

fact that network communication need be performed, and shadow oceans automatically do the job of accumulating ghost fish or migrant fish (depending on the phase of computation) into a list. All the communication code does is transmit the entire contents of each shadow ocean to the thread to which that ocean belongs. In fact, transmission of the entire contents of an ocean from one thread to another is exactly what our dynamic load balancing scheme requires, so the same code can be used for that as well, see Section 4.

4 Load balancing

One of the major challenges in any interacting particle implementation is the distribution of the workload among different parallel threads at runtime. Therefore, one crucial design decesion in our implementation was to add load balancing capability to our simulation. Due to the dynamic nature of the problem in hand, the workload assigned to each thread in the system is neither balanced at startup time of the application as different ocean gets varing number of fish, nor during its run time as fish migrate from one ocean to another. Therefore, adding static and dynamic load balancing to the simulation was essential in allowing a larger number of fish simulation with limited computational resources.

4.1 Static Load Balancing

As we outlined earlier, each thread owns a number of oceans, and its workload is determined by the number of fish interacting inside those ocean, and their total communication characteristics. Based on the simulation instance and the startup configurations, the workload between the different threads at startup may be not balanced. Static load balancing in our model is, therefore unavoidable in order to distribute the workload among the different threads before the fish start to interact.

In nature, the load balancing challenge in computational models of interacting particles is a subset of the weighted graph-partitioning problem. The problem is NP complete, but there exits different approximation that would solve the problem at reasonable orders of complexity [6, 7]. To simplify the problem, we opt to distribute the workload among thread in a snake-like distribution, where each group of oceans has a reasonably comparable portion of the total workload. The goal of our simplified solution was to decrease the communication between the different threads running on different processors without losing computational advantages. Figure 6(i) illustrates how we distributed different oceans to different processors at start-up. Oceans having the same color run on the same thread. The oceans' distribution tries to minimize the number of edges between different threads, and thus minimize the network communication between them. Another possible work distribution theme was to distribute the oceans in blocks, as shown in Figure 6(ii). However, to implement this distribution we had to enforce restrictions on the x and y dimensions of the world dependent on the number of oceans and the number of threads. Furthermore, we aimed for a sub-optimal load-balanced distribution of the initial workloads among processors, since our dynamic load balancing -which we discuss in more details in the following subsection- would do more load balancing to the different threads workloads.

4.2 Dynamic Load Balancing

Because of the interactions between the fish described in Section 2.2, the fish gather into schools and then move through the world as schools. This means that in order to avoid having some processors being overloaded while other processors remain idle, we had to deploy some implementation of dynamic load balancing to our simulation. Our goal for dynamic load balancing is for each processor to do a relatively comparable amount of work during each iteration.

We first decided what constitutes "work." Because the majority of the computations during each time step occur while updating each fish's velocity, we chose to create a variable, interactionCounter, which keeps track of the number of interactions that occur during a given time step in each ocean. At the beginning of every time step, interactionCounter is set to zero on each ocean and we increment it whenever a fish finds another fish within its zone of attraction, since then the fish interacts with this fish. This definition of work is more informative than keeping track of the number of fish in a given ocean, since fish which are far enough apart will not interact and thus will not affect the amount of time the processor is spending on computations as much as the same number of fish placed close together.

Once "work" is defined, there are several options for how to dynamically load balance the simulation. We could divide the world into differently-sized oceans depending on the density of fish within each region. However, the data structure which would need to be communicated between processes would be very complex, because it would have to be able to capture an arbitrary division of the world into oceans. Also, it might result in quite a bit of communication since a processor would most likely be given only a small piece of a very densely populated region, and fish on that processor would be interacting with many fish on other processors, requiring a larger number of ghost fish to be sent between processors.

What we chose instead was to start with a preset

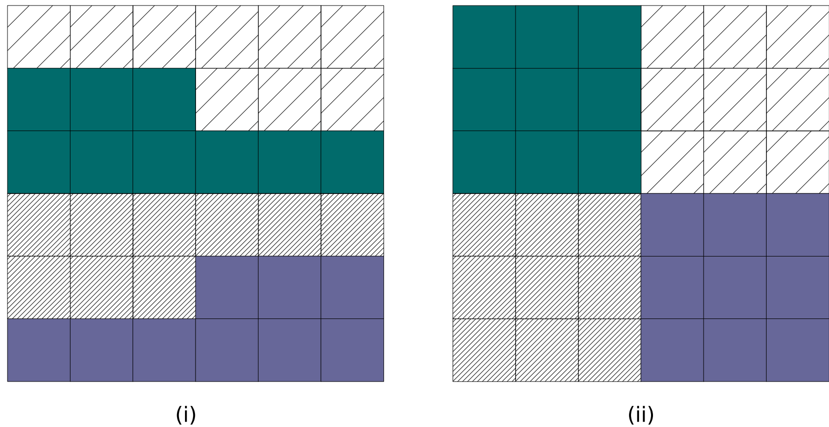

Figure 6. The figure illustrates possible distribution of oceans to different MPI threads at application start-up. Oceans in the same color run on same thread. (i) Left figure shows our snake-like distribution, while (ii) right figure shows another theme of distributing the oceans by blocks amoung threads

array of oceans and to initially deal them out as described in Section 4.1. We keep this array throughout the simulation, but we allow ourselves to reassign oceans to processors as time goes on. We choose a counting number N, and run our dynamic load balancing code every N time steps. Our dynamic load balancing algorithm is run by a master node which iterates through the processors and computes how much work is done by each processor by adding up the interactionCounter from the previous time step of every ocean belonging to that processor. It then finds which processor is doing the most work and calls that processor *processorHigh*. Then it computes the amount of work done by processorHigh's neighbors, i.e. the processors which have an ocean bordering an ocean belonging to processorHigh. The neighboring processor which is doing the least amount of work is labeled as *processorLow*. The algorithm runs through each ocean belonging to processorHigh which shares an edge with an ocean on processorLow and finds out which ocean processorHigh can give to processorLow in order to make the work done by these two processors closest to equal.

It is important that processorHigh gives its ocean away to a processor bordering this ocean since their sharing an edge means that ghost fish and migrating fish at this edge will not have to be communicated via the network. This choice therefore reduces the amount of communication necessary between processors. Giving away only one ocean each time the dynamic load balancing module runs means that the communication overhead for the dynamic load balancing algorithm is fairly low. A large choice for N will mean that whole oceans are communicated less frequently, which will reduce this overhead; however, it is important to balance the cost of this communication with the benefit of having each processor work at equal capacity. We present the performance of the load balancing module, and its communicational requirements in the following section.

5 Performance Analysis

The ultimate goal for this application was to enable us to simulate the interaction of fish in large shoals in a shorter execution time. Therefore, we decided to define an application-specific metric of performance to reflect the motivation for the simulation. The metric measure the amount of time taken to execute one fish interaction with surrounding fish per iteration. We also measure the total execution time of the simulation as a functions of the total number of fish in the world and the number of iterations. We show the impact of porting our implementation on leveraging the simulation time per fish per iteration between the matlab code, the C++ code and the MPI code with load balancing.

We evaluated the sequential performance of both Matlab and C++ as a base performance measure for our experimentation. In the Sequential performance

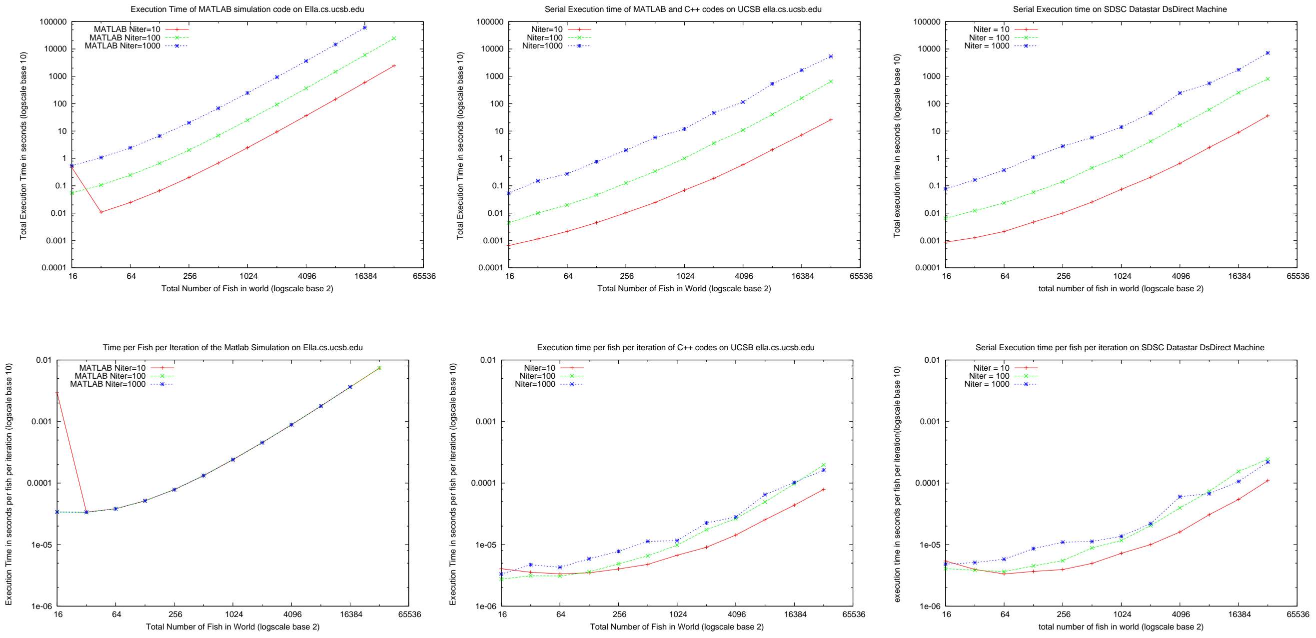

Figure 7. The performance comparison between Matlab code and C++ implementation. The upper row represents the matlab performance while the lower is C++ performance. The left column shows the Total time of execution of the simulation as a function of number of fish in oceans for different number of iterations. The right column illustrates the time in seconds needed by different implementation to simulate a fish per time iteration. Notice that the scale of number of fish for Matlab code is much smaller than the scale used in C++ measurements.

experiments, we deployed our code on a Intel Pentium 4 with a dual core; each core is 2.60GHz CPU, 512KB L2 cache size and 1GB of main memory. Furthermore, we deployed our C++ sequentional code on Datastar machine. DataStar is an IBM terascale machine at San Diego Supercomputing Center (SDSC). The machine has Power4 processors, with pipelined 64-bit RISC chips with two Floating-point Units. Each Power4 is a 1.5GHz with 16 GB of main memory, and has a two-way L1 (32 KB) cache, a L2 (0.75 MB) cache which is four-way set associative. There is also an 8-way L3 cache on each node (16 MB per processor).

Although Matlab code was more compact and less in number of lines than C++ code, Its performance lagged behind the more complicated C++ code. Figure 5 portrays the performance distinction. The figure comprise of six subfigures. The first row of subfigures represent the total execution time of the simulation (on y-axis) as a function of the total number of fish in the world, while the second row portrays the time per fish per iteration (on the y-axis) as a function of the total number of fish in the world (on x-axis). The red, blue and green curve reflect the 10, 100 and 1000 iterations simulations respectively. The two left-most subfigures are the matlab performance, while the two middle subfigures and the two left-most subfigures are the C++ performance on the Intel Pentium machine, and on SDSC Datastar respectively. Notice that the y-axis scale of the subfigures is not the same.

From figure 5, we can clearly observe that the matlab code on Pentium machine was 10X slower than C++ code performance o both Pentium and Datastar machines. Using a curve fitting methodology, the slope of the two curve are different (*what are the slopes?*). However, we observe that the performance difference between the curves for the different number of iterations is constant and is not impacted by the total number of fish in the world. In addition, regarding the subfigure in the bottom left corner of figure 5, the matlab code is observed to have a non-linear growing time of simulating fish per iteration, which reflects the poor scalability of the matlab code. On the other hand, the middle and right-hand bottom subfigure reflect a stepper and slower growth of the performance time per fish per iteration as the number of fish increases for the C++ code. This performance analysis illustrate the ef-

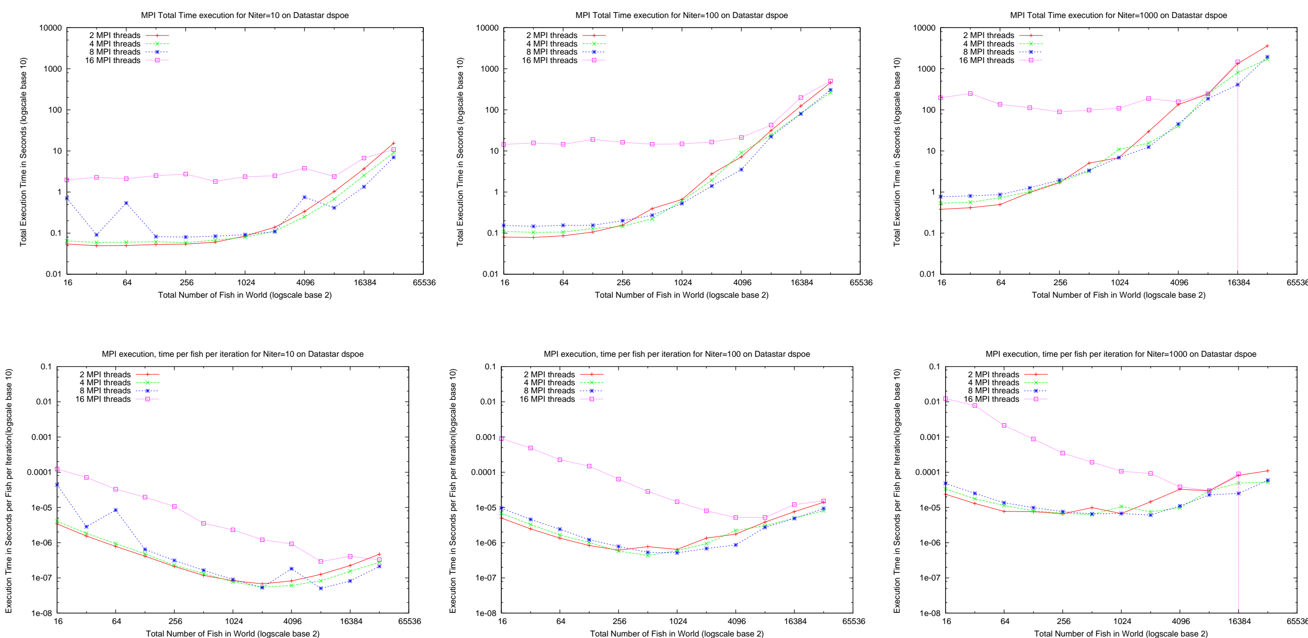

Figure 8. MPI performance curves using 2, 4, 8 and 16 mpi processes for parallel mpi simulation of the fish interaction C++ model. The two left-most subfigures are the performance characteristics for 10 iterations of the fish iteration simulation, while the two middle subfigures and the two right-most subfigures reflect the performance for 100 and 1000 iterations respectively.

ficency of C++ to our problem domain in comparison to matlab code.

In addition to the sequential performance characteristics, we present the performance attributes of the MPI parallel code. Figure 5 presents the performance of the MPI code with different number of MPI threads, and different iterations. The figure comprise of six subfigures. The first row of subfigures represent the total execution time of the simulation (on y-axis) as a function of the total number of fish in the world, while the second row portrays the time per fish per iteration (on the y-axis) as a function of the total number of fish in the world (on x-axis). The two left-most subfigures are the performance characteristics for 10 iterations of the fish iteration simulation, while the two middle subfigures and the two right-most subfigures reflect the performance for 100 and 1000 iterations respectively. For each of the subfigures, we show the performance using different number of MPI processes. The red, blue, green and purple curves demonstrate the performance of 2, 4, 8 and 16 MPI processes respectively. All of those parallel performance measurements were collected on SDSC Datastar machine.

From figure 5, we observed that the number of iterations impact the scalability of the mpi code, as we notice the variability of the slopes between the three figures on the upper row. As the number of iterations increase, the slope of the curve become steeper. This is caused by the increase in communicational load increase as the number of iterations increase. This impacts the efficiency of the MPI communications, and its interleaving with the computational load. This in turn impact the overall total execution time of the simulation as the number of iterations grow. Furthermore, notice that the 16-processes performance lags behind the other curves, especially that the problem instance has 16 oceans (i.e., every MPI process is executing one ocean). As the small number of fish and small number of oceans in the world are divided among the 16 processes, the computations is exceedingly short relative the amount of communication between the oceans. This imbalance in the communication and computations load results in the performance retardation, which is noticed by the purple curve of the 16 processes. However, as the number of fish in the world increases, the computational load of every ocean on each processor increase, and slowly attains the balance between the communications and computational loads.

The lower row of subfigures in figure 5 presents the time of execution per fish per iteration as a function of

the total number of fish in world. These curves shows that MPI code has a better scalability than the matlab and C++ codes, as shown in figure 5. In addition, the curves have a a knee, before and after which the time of simulation per fish per iteration is higher. This point is at 2048, 1024 and 512 fish for 10, 100 and 100 iterations respectively. At these points, the simulation achieves the most balanced state between the computational load and communicational load for this number of iterations. In addition, as the number of iteration increase, the balance point is achieved at a lower number of fish. This is also explained by the fact that the communication overhead increases by the number of iteration, and not at the same rate of increase of the computational load. This characteristics are specific to the architecture we run our simulation on, and would be different from one super-machine to another.

6 Conclusions

In this paper, we implemented a parallel version of the model of fish schooling described in [1]. Some of the challenges addressed in this implementation were ways to divide the problem domain among processors for parallelization, how to communicate between processors effectively and at minimal cost, and how to distribute work dynamically without excessive communcation overhead. Our architecture is an innovative compromise between fixed spatial allocation and completely variable-sized spatial allocation. Load balancing in the way described in section 4 allows us to keep messages between processors simple, but also equallizes the work done by the processors. A series of tests gave us emperical results indicating that not only is our C^{++} implementation a plasing improvement over the previous MATLAB implementation, and that our dynamic load balancing scheme actually gives a significant further performance benefit.

References

[1] A. "Barbaro, B. Birnir, and K. Taylor'. "discrete and continuous models of the dynamics of pelagic fish: Orientation-induced swarming and applications to the capelin", June 11, 2007. Center for Complex and Nonlinear Science. Paper Bio2.

[2] J. Carscadden, B. S. Nakashima, and K. T. Frank. Effects of fish length and temperature on the timing of peak spawning in capelin (*Mallotus villosus*). *Can. J. Fish. Aquat. Sci.*, 54:781–787, 1997.

[3] A. Czirók, H. Stanley, and T. Vicsek. Spontaneously ordered motion of self-propelled particles. *J. Phys. A: Math. Gen.*, 30:1375–1385, 1997.

[4] A. Czirók, M. Vicsek, and T. Vicsek. Collective motion of organisms in three dimensions. *Physica A*, 264:299–304, 1999.

[5] A. Czirók and T. Vicsek. Collective behavior of interacting self-propelled particles. *Physica A*, 281:17–29, 2000.

[6] U. Elsner. Graph partitioning - a survey, 1997.

[7] P. Fjallstrom. Algorithms for graph partitioning: A survey, 1998.

[8] S. Hubbard, P. Babak, S. Sigurdsson, and K. Magnússon. A model of the formation of fish schools and migrations of fish. *Ecological Modelling*, 174:359–374, 2004.

[9] K. G. Magnússon, S. Sigurdsson, and B. Einarsson. A discrete and stochastic simulation model for migration of fish with application to capelin in the seas around iceland. Technical Report RH-20-04, Science Institute, University of Iceland, 2004.

[10] K. G. Magnússon, S. T. Sigurdsson, and E. H. Dereksdóttir. A simulation model for capelin migrations in the north atlantic. *Nonlinear Analysis: Real World Applications*, 6:747–771, 2005.

[11] B. L. Partridge. The structure and function of fish schools. *Scientific American*, 246(2):114–123, 1982.

[12] B. L. Partridge and T. J. Pitcher. The sensory basis of fish schools: Relative roles of lateral line and vision. *Journal of Comparitive Physiology*, 135:315–325, 1980.

[13] T. Vicsek, A. Czirók, E. Ben-Jacob, I. Cohen, and O. Shochet. Novel type of phase transition in a system of self-driven particles. *Physical Review Letters*, 75(6):1226–1229, 1995.

[14] T. Vicsek, A. Czirók, I. Farkas, and D. Helbing. Application of statistical mechanics to collective motion in biology. *Physica A*, 274:182–189, 1999.

[15] H. Vilhjálmsson. *The Icelandic capelin stock: capelin, Mallotus villosus (Müller) in the Iceland-Greenland-Jan Mayen area.* Hafrannsóknastofnunin, Reykjavík, 1994.

[16] H. Vilhjálmsson. Capelin (*Mallotus villosus*) in the iceland-east greenland-jan mayen ecosystem. *ICES Journal of Marine Science*, 59:870–883, 2002.