# Nemo: A parallelized Lagrangian particle-tracking model

Benjamin Jones

December 12, 2013

**Abstract**

Lagrangian particle-tracking models are a computationally intensive, but massively parallelizable method for investigating marine larval dispersal processes, seed dispersal of plants, or a variety of other material transport processes. In order to fully capture the distribution of potential dispersal patterns, highly efficient models with the capacity to simulate tens of millions or more particles are needed. Traditionally, this goal has been achieved using models written in Fortran and parallelized using Message Passing Interface (MPI). This project is an investigation of a variety of techniques for parallelizing Lagrangian particle-tracking models, including using MPI, OpenMP, and graphics processing units (GPUs). This report summarizes the implementation of each parallelization strategy, compares the performance of each implementation, and concludes with suggestions for further development that could be done on this model.

## 1 Introduction

Transport of propagules in the ocean and atmosphere by currents and wind drives a variety of biotic and abiotic processes. Marine population connectivity, or the movement of individuals among the geographically separated subpopulations that compose a metapopulation, is driven by larval dispersal for many species and is a key determinant of population dynamics [9]. Plant seeds are often carried by the wind and control landscape structure [9]. In the event of disasters, understanding the transport of contaminants such as oil is crucial to damage assessment and the design of cleanup efforts [7]. Coupling high resolutions flow fields to Lagrangian particle-tracking models offers a potential avenue for exploring each of the above processes along with many others.

Particle-tracking models, which may be alternatively referred to as agent-based or individual-based models, are a computational technique to predict the likely dispersal patterns of propagules in the atmosphere and ocean. By integrating the trajectory of many particles, the models are able to capture the distribution of transport patterns. The Lagrangian attribute of their title reflects that the coordinate system of each particle is local, and moves in time and space with the particle. In contrast to advection-diffusion models that invoke the continuum hypothesis, individual-based models retain the stochastic nature of each particle. They provide an intuitive framework for incorporating growth rates, mortality, or directed movement of each particle in response to the particle's local environment, *e.g.* [6, 14]. In exchange for this focus on the individual, particle-tracking models have greater difficulty capturing the tails of the dispersal distribution than do advection-diffusion models [19].

By simulating a sufficiently large number of particles, the distribution of potential dispersal pathways may be captured to a level of accuracy that is appropriate to answer the research question. The number of particles required varies by the question being asked and system being studied, but can often number in the tens of millions or more including calibration runs *e.g.* [20]. Simulating this number of particles requires thousands of processor hours with highly efficient and scalable models. Reducing the runtime of particle-tracking models by a small amount can result in substantial savings on computational resources, or present the opportunity to operate more sophisticated models with the same resource requirements. Traditionally, the requirement for speed and efficiency has been interpreted as a mandate to use Fortran and Message Passing Interface (MPI), e.g. [15]. Recent advances in compiler technology and hardware architecture offer alternative strategies for parallelization that I investigate as part of this project.

Open Multi-Processing (OpenMP) is a specification for a shared memory, multi-threaded approach to high performance computing that is particularly well suited to the incremental parallelization of serial software. Through the addition of preprocessor directives, the programmer can specify that the compiler should spawn a number of threads and split the work of a particular operation across them. For

example, adding the directive `#pragma omp parallel for` prior to a for loop in C or C++ instructs the compiler to divide the iterations of the for loop across threads. The programmer retains responsibility for ensuring that no iteration of the loop is reliant on data produced during other iterations, but is relieved from explicitly creating and destroying threads. The number of threads used is determined at runtime, and usually defaults to the number of CPU cores. If race conditions exist, they may also be noted by designating specific regions as critical or operations as atomic. Because the compiler adds the platform specific operations to spawn and destroy threads and most compilers support OpenMP on the major operating systems, adding OpenMP retains cross platform compatibility that may be lost with platform specific multithreading libraries. The directives may be added to serial programs without otherwise altering the code, so OpenMP is an ideal candidate for incrementally parallelizing existing serial programs. However, OpenMP is a multithreading specification, and so memory is shared across all threads. Due to the cost of large, shared memory systems, this limits OpenMP parallelized programs to small to moderately sized problems.

An alternative to OpenMP for distributed memory parallelization is MPI. Whereas OpenMP may be used to parallelize portions of a program, MPI requires the entire program to be parallelized. For software that is already written to operate serially, this requirement can lead to substantial effort and major changes. At minimum, an MPI program must call `MPI_Init()` to create a set of processes and `MPI_Finalize()` to destroy them. The MPI specification includes a set of subroutines for sending data among the processes that are running within the MPI parallel environment. When calling `mpirun` to create the parallel environment, the user can set the number of processes with the flag `-np`. Because MPI uses multiple processes and distributed memory, it may be run on arbitrarily large numbers of nodes, making it suitable for use on the largest supercomputers. As with OpenMP, implementations of MPI exist for the major operating systems.

The computational requirements of real-time rendering of 3D graphics in computer games led to the creation of an alternative form of massive parallelism, the Graphics Processing Unit (GPU). Instead of improving performance by increasing the clock speed of the CPU, GPUs put many slower cores on a single chip. Although each GPU core runs slower the CPU cores, the combination of hundreds of them provides solid performance for repeatedly performing the same task on many independent pieces of data. The GPU is an entirely separate unit from the CPU and uses a different address space, so GPUs are often more limited in memory availability. For example, whereas the CPU on my test platform has 16GB of memory, the GPU has only 1GB. In addition, the GPU does not have direct access to files. APIs such as NVIDIA's proprietary Compute Unified Device Architecture (CUDA) or the industry standard Open Computing Language (OpenCL) provide subroutines to move data between the CPU and GPU and to start kernels on the GPU. GPUs have been successfully used for scientific problems including hydrodynamic simulation using finite differencing [12], smoothed particle hydrodynamics [8], and computation of finite time Lyapunov elements [5].

The objective of this project is to develop insight into promising parallelization strategies for Lagrangian particle-tracking models. I developed a small Lagrangian particle-tracking model that captures the overall components of larger models actively used in research and implemented OpenMP, MPI, and CUDA parallelization in it. In addition, I included a multiprocessing approach that manually forks worker processes and uses shared memory regions for communication among the workers.

# 2 Nemo

The practice of Lagrangian particle-tracking modeling can be broadly divided into 3 broad stages (Fig 1, left). Initially, a hydrodynamic model is calibrated with empirical observations to simulate the currents in a region of interest. Some examples of models that may be used include the Regional Ocean Modeling System (ROMS) [16,17], Finite Volume Community Ocean Model (FVCOM) [4], or HYbrid Coordinate Ocean Model (HYCOM) [3]. The choice of hydrodynamic model and procedures to calibrate them are beyond the scope of this project; it is sufficient to know that these models produce high resolution flow fields. The smallest processes that can be resolved by these flow fields are limited by the mesh size of the hydrodynamic model. When diffusive processes are relevant to the hypothesis being tested, they may be simulated through the addition of a stochastic component. There is little consensus regarding the parameterization of the diffusion term in particle-tracking models and it is sometimes modeled as a linear function of mesh size [13]. The Lagrangian particle-tracking model integrates the trajectories over time including advection by the simulated currents, subgrid scale diffusion, and particle behavior. Some examples of particle behavior may include ontogenetic migration or natal homing of marine larvae [14,18]. Finally, postprocessing and analysis relates the particle trajectories back to original hypothesis being

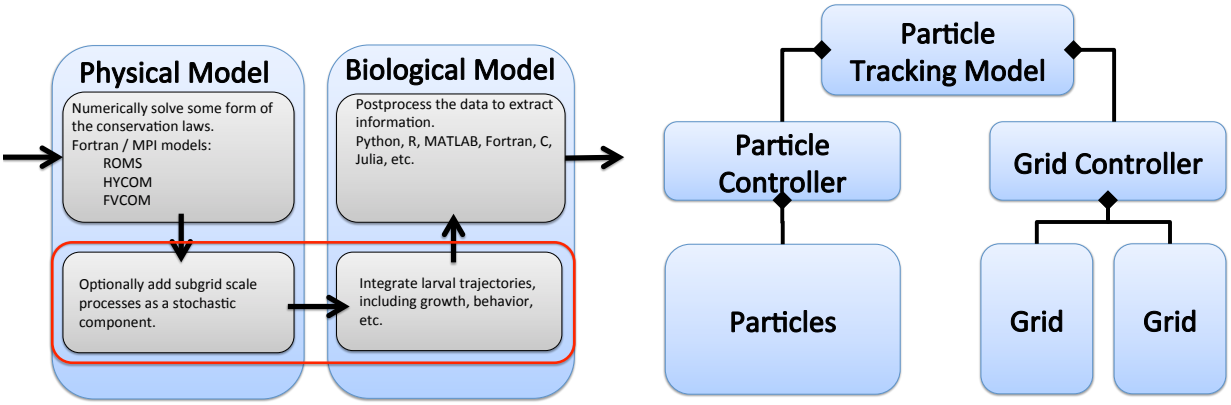



Figure 1: Left: An overview of the process of biophysical modeling of oceanographic systems. Right: An overview of the major classes in Nemo and their relationships with one another.

tested. The computationally intensive tasks are often performed in compiled Fortran or C code, and the production of visualizations in scripting languages such as Python, R, or MATLAB. The advent of just-in-time compiled languages such as Julia or the PyPy implementation of Python may present the opportunity to postprocess and analyze the data in a single dynamic programming language [1, 2].

Nemo is the model that I developed for the purposes of testing each method of parallelization and implements the second stage of biophysical modeling: integrating particle trajectories. It is written in C++ using an object oriented design (Fig 1, right). The integration of particle trajectories and inclusion of biological processes falls under the purview of the particle controller (ParticleCtl) class. Reading the archived forcing fields and interpolating them in time and space to the particle locations is handled by the grid controller (GridCtl) class. The particle tracking model (PTM) itself retains ownership of the one or more ParticleCtls and GridCtls and ensures that they are properly configured and able to communicate with one another. In the serial version, this communication is achieved by passing each ParticleCtl a pointer to a GridCtl instance and allowing it to call the GridCtl's public methods. The communication methods used for the parallel models are described below.

The main routine for all versions of Nemo begins with the creation of a PTM. The PTM reads the JSON configuration file and creates the particle and grid controllers. The constructor for each grid or particle controller accepts a Boost PropertyTree as read from the configuration file and a mechanism for communicating with other workers that is specific to the parallelization technique. Each particle controller allocates memory for the particles under its control, and each grid controller initializes the grids that it controls. The grids are ranked heirarchically within each grid controller, so it is possible to nest local higher resolution grids within lower resolution regional or global grids. The PTM then calls the run method on each particle controller and is responsible for deleting the grid controllers when the run is complete.

The numerical methods used in Nemo are acceptable for the purposes of testing runtime performance, but not sufficient for use in practice. The integration uses a first order Euler method, which is inadequate for oceanographic research. Second and fourth order Runge-Kutta methods have been implemented for the serial version of the model, but not yet tested on the parallel versions. The forcing fields are linearly interpolated in time and space. Ideally the spatial interpolation would be done with a bicubic method. Diffusion is not yet implemented. Despite these limitations to the use of Nemo for particle-tracking, it is still sufficient for testing the performance of various parallelization strategies.

## 2.1 OpenMP parallelization

The OpenMP parallelization required the fewest modifications to Nemo. With the addition of three lines of code, I parallelized the temporal and spatial interpolation and the particle integration. Because the NetCDF libraries used for IO operations are not thread safe, this process represents the changes that would likely be made as a first attempt to incrementally parallelize an existing serial model. This parallelization required the addition of two `#pragma omp parallel for` directives.

In the initial version of Nemo, the data accessor methods in the Grid class accepted a request time and automatically read in the data from disk or interpolated between timesteps if necessary. Because there is a single slot for storing the current timestep in the Grid class, this design creates a race condition. One thread may request data at timestep $t$. After the data at timestep $t$ has been loaded, but before the Grid has completed the time consuming process of temporally interpolating each velocity, a second thread may request data at the previously loaded timestep $t_0$. If the second thread's request is fulfilled before the first thread updates the Grid to note that time $t$ was loaded, then the second thread will be given data from $t$ believing that it is from $t_0$. Alternatively, the first thread could first update the current time to $t$, then temporally interpolate, but that strategy would just reverse the race condition so that the first thread gets the wrong data and potentially result in corrupted results from calls to the NetCDF libraries. One potential solution would be to mark the temporal interpolation region as critical and allow only 1 thread to enter it at a time. Although this solution removes the race condition, it precludes the possibility of parallel temporal interpolation, because all but one thread would block while a single thread performed the interpolation of all elements in serial. Another potential solution would be to load single grid elements only when needed instead of the entire mesh, but this is inefficient for large number of particles because there are likely to be many particles within a single element. The solution I chose was for the particle controller was to first request that the grid controller load the data at a specific point in time from the main thread, then integrate all the particles for that timestep in parallel.

## 2.2 MPI parallelization

The MPI parallelization required more substantial modifications to the PTM, ParticleCtl, and GridCtl classes, but did not require modification to the Grid or Particle classes. Instead of the main routine first creating a PTM instance, the MPI version first calls `MPI_Init`. Node number 0 then creates a PTM class that reads the configuration file and initializes the model, and all other nodes call the blocking subroutine `MPI_Recv`. The master node sends 3 pieces of information for each worker node. First, a single integer that tells the worker to either become a grid controller or a particle controller. Second, the size of the configuration data for the worker node. Third, a JSON string with the worker node configuration. For the particle controllers, it also sends the node number for a grid controller where the particle controller can send forcing requests, evenly dividing the particle controllers across grid controllers. Each grid controller immediately enters an infinite loop of waiting for requests and sending the results back to the requesting node, and each particle controller immediately begins to integrate the particle trajectories. Upon completion, the particle controllers notify the master node, which then sends a request to the grid controllers that causes them to terminate.

## 2.3 CUDA parallelization

To run Nemo on a GPU, I used NVIDIA's CUDA functions and compiler. The choice to use CUDA instead of the cross platform OpenCL was motivated in part because the test platform has a NVIDIA GPU and in part because it supports double precision arithmetic. The CUDA version of Nemo is operationally very similar to the serial version, but required additional pointer arithmetic and memory allocation to smoothly move data between the hard disk, main memory, and GPU memory. During each timestep, the CPU configures the pointers in the Grid and ParticleVector classes to point to device memory, copies them to the device, then launches a kernel perform the spatial interpolation and particle movement. The new particle locations are copied back to the CPU and written to disk. This process involves an excessive amount of data copying and only uses global memory in the device, both of which could be improved in future versions.

In order for the Grid and ParticleVector to interface properly on the GPU, the ParticleVector was given a pointer to the Grid and allowed to access it directly instead of through its controller. Also, each class instance is serialized prior to moving it to the device and after recovering it from the device. The serialization process happens in two steps. The first step is to allocate a single contiguous block of memory for the class and copy the class itself and all dynamically allocated members to this block. The second step is to update the pointers that used to reference dynamically allocated memory to instead be offsets to the start of the block. Because this second step does not involve accessing the memory locations referenced, it does not matter whether the reference location is the actual location of the class, or its future location when copied to the device. The process of serializing the memory happens only once before the first timestep, but serializing the pointers happens any time an instance is copied to or from the device.

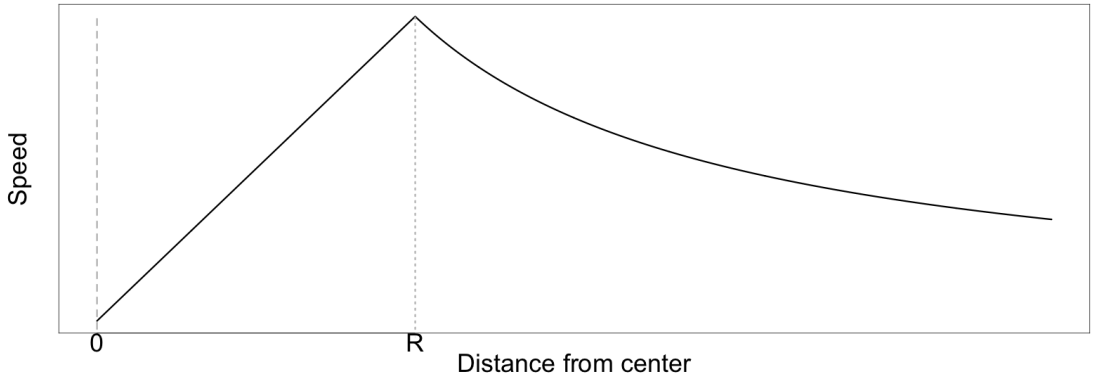



Figure 2: The velocity magnitude of a Rankine vortex along a radial transect. Within the radius $R$, the velocity increases linearly with radius. Outside of $R$, the velocity magnitude decays with radius.

### 2.4 Lower level parallelization

One additional well intentioned, but ill informed, attempt at parallelization was also implemented. This version of Nemo was operationally identical to the MPI version, but used `fork` system calls to spawn new processes and shared memory regions to communicate among workers. Communication with the master node relied on POSIX signals. Although I included this version in the performance tests, I found that it did not contribute any benefits not already present in other versions and am in the process of removing it now. Because this version used `fork` calls to create new process, I refer to it as the fork version.

## 3 Test framework

The performance testing took place on an analytic flow field. The Rankine vortex is an idealized model of an eddy that is characterized by 2 parameters. Within a radius $R$, the eddy demonstrates solid body rotation. Outside of $R$, the velocity magnitude decays. The absolute speed of the eddy rotation is defined by the circulation $\Gamma$. In addition to the rotational aspect described by (1), I included a horizontal translation along the positive x axis at rate $k$. The test grid was configured to simulate a 300km diameter eddy with 0.25m/s flow 150km from the center and moved horizontally at 0.1m/s as may be expected for a western boundary current eddy [10]. The grid spanned 1000km by 500km and hourly output was written for 60 days. One particle was released from and the velocity vectors were written at each point on a low resolution 4km and high resolution 1.5km lattice grid.

$$u_\theta(r) = \begin{cases} \Gamma r/(2\pi R^2) & r \leq R, \\ \Gamma/(2\pi r) & r > R. \end{cases} \tag{1}$$

The first set of tests sought to compare the performance of each parallel method. Each version of Nemo was compiled using the optimization levels -O0, -O1, -O2, -O3, and -O4. Each model was 3 times, using the 4km grid to simulate the particles for 15 days with an hourly timestep. The fastest version of each model was chosen and an additional 7 tests were run, for a total of 10 runs with each of the fastest versions. The results were then standardized against the median runtime of the serial version. During these tests, the MPI and low level parallelizations were limited to 1 grid controller and 1 particle controller, so they would be expected to operate more slowly than the serial version due to the overhead of interprocess communication. In the second set of tests, I profiled each version of the model to determine where the runtime was being spent. Each method was classified as an IO operation, associated with physical forcing, related to the particle integration, or a communication routine. A mosaic plot captures the proportion of time that each version of the model spent in each category using the 1.5km grid. Finally, I tested the scalability of the MPI version of the model by running it with 1, 2, 4, and 8 particle controllers and 1, 2, 4, and 8 grid controllers.

All of the tests were run on a mid 2012 Retina Macbook Pro running OS X 10.8.5. The test platform has an 2.3 GHz Intel Core i7 with 4 physical cores and 2 threads per core. It has an NVIDIA GeForce GT 650M graphics chipset with 384 900MHz cores. The main memory consists of 16GB of 1600MHz memory

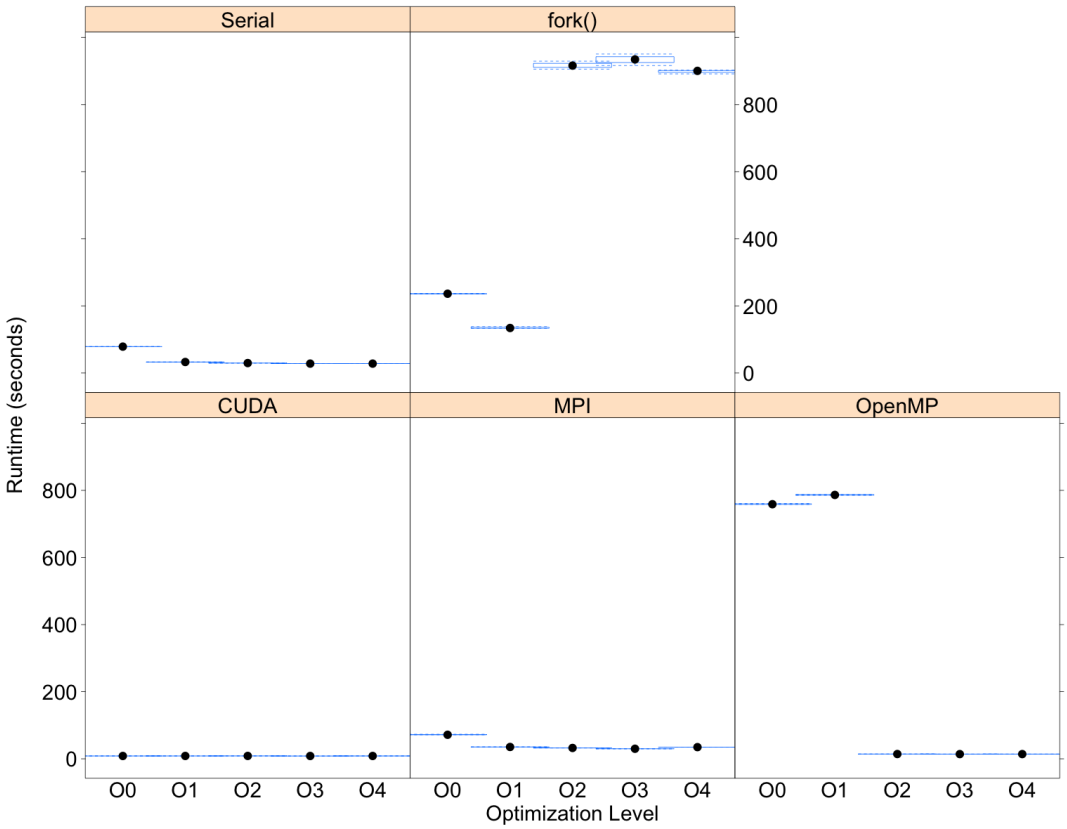


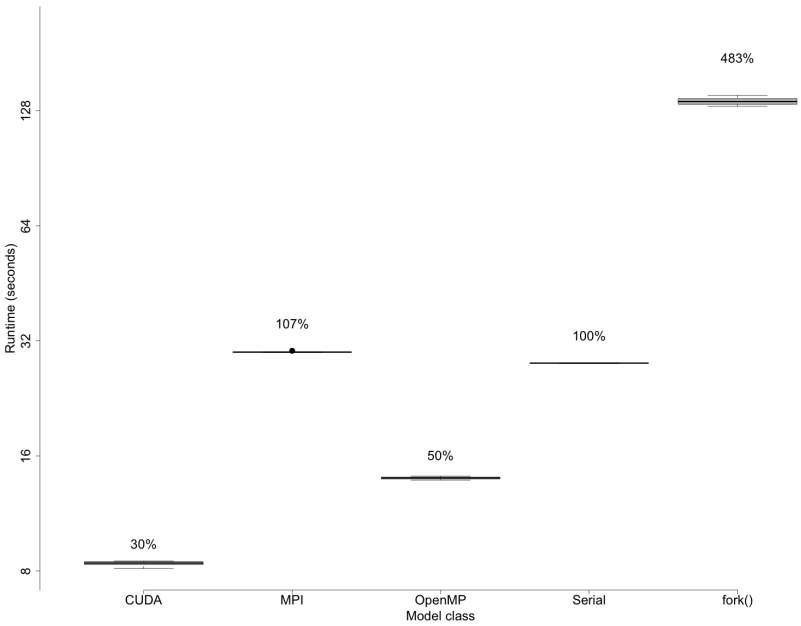

Figure 3: Three runs were conducted with each optimization level and model version. The results of those test are depicted here.

and the GPU has 1024MB of VRAM. No effort was made to optimize the code for this architecture. The compiler used was GCC 4.2, Apple LLVM version 5.0 with OpenMPI version 1.7.3, or NVIDIA nvcc V5.5.0. All tests were conducted with single precision operations due to limitations of the GPU. All of the tests were conducted using commit number 1950fb39e6e41598a1f126c79869877e55c10600 on the master branch of the git repository at https://github.com/btjones16/nemo.

# 4 Test Results

The first round of testing showed very similar performance across all runs for the CUDA, MPI, and serial versions, and vastly differing performance between runs for the OpenMP and fork versions. Splitting the results along both the version and optimization level, it becomes apparent that these differences arise because the -O2, -O3, and -O4 optimized fork models are slow, and the -O0 and -O1 OpenMP models are slow (Fig 3). Comparing only the fastest optimization levels, the OpenMP parallelization on 8 threads is approximately twice as fast as the serial version and the CUDA version just over 3x faster (Fig 4). The MPI version incurs a 7% overhead for communication, and the fork version is nearly 5 times slower.

In the serial version, IO operations consume 92% of the runtime. Under the CUDA and OpenMP versions, this proportion is vastly decreased, and the process of locating particles on the grid and interpolating velocity vectors becomes the dominant process. This does not necessarily mean that the IO operations took less time since the IO operations were not parallelized but the other methods were. The MPI and fork methods spent 87% and 77% of their runtime in communication related methods and the vast majority of that time was spent in spin conditions. However, this result is largely an artifact of limiting the test cases to a single grid controller and a single particle controller. Each of these processes is blocked in a `MPI_Recv` call while the other is running, and the master process is blocked throughout the duration of the run. A better estimate of the communication overhead for MPI process is the 7% from




l

Figure 4: Left: The runtime of all versions and all optimization levels of Nemo. Right: The runtime for the 10 tests conducted with the fastest optimization level for each model.

the timing results displayed in Figure 4. Excluding communication time, all 4 parallel versions spent the most time processing physical data.

Running the MPI version with multiple particle controllers and multiple grid controllers revealed that the ideal configuration for this grid size and number of particles is 1 grid controller per particle controller. With higher numbers of particle controllers, it is not a poor choice to have 2 particle controllers per grid controller. Even though additional grid controllers are not assigned particle controllers and sit idle for the duration of the run, have an excess of grid controllers causes slowdowns in performance. For example, running 1 particle controller with 1 useful and 7 idle grid controllers wastes system resources and takes over twice as long as a single grid controller. The balance between the number of grid and particle controllers is highly dependent on the number of particles and grid size and would ideally be dynamically balanced during the model run.

# 5 Discussion

The development and implementation of Nemo highlighted a number of considerations that go into parallel programming. I ran into stumbling blocks concerning the choice of language, overall design of Nemo, and choice of libraries.

I had initially written Nemo in Python, but rapidly encountered stumbling blocks that convinced me to switch to C++. The global interpreter lock in CPython restricts the possibilities for multi-threaded programming because only 1 byte code instruction can be executed at a time. This limitation, and that the initial tests were approximately 3 orders of magnitude too slow, convinced me to switch to a compiled language. Fortunately, or perhaps unfortunately, this stumbling block occurred early in the semester, allowing me sufficient time to rewrite the model in C++, but also before we had completed any assignments with Julia. The model loops over the same few lines of code millions of times or more, so the overhead associated with compiling the model on the fly would be minimal relative to the overall runtime. Revisiting the choice of language, it would interesting to explore the possibility of using Julia to write high performance scientific models.

The first parallel version of Nemo was a multiprocessing version with a more strict object oriented design. There was a Particle class, and each timestep iterated through all of the particles, obtaining the forcing data and advancing the particle prior to moving on to the next particle. In the parallel version, this strategy resulted in 2 context switches per particle per timestep. Although a single context switch

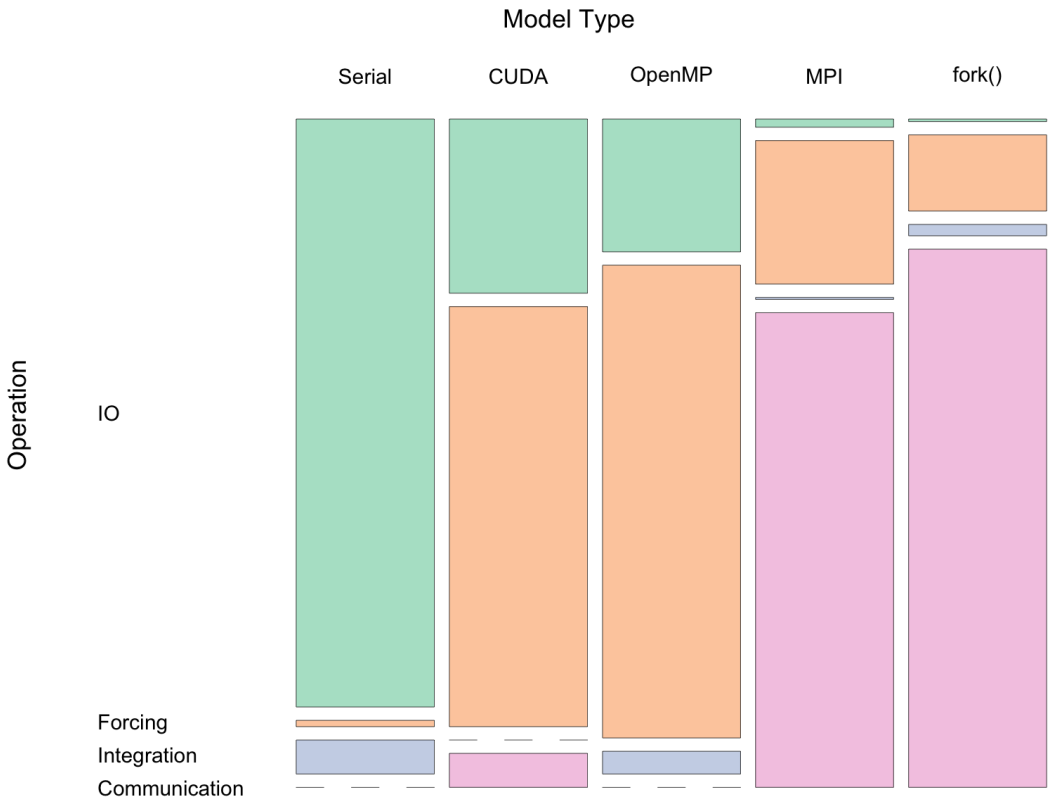



Figure 5: Each method was classified into one of 4 categories. The size of each box corresponds to the proportion of profiling samples that were in each category. Sampling occurred across all cores, processes, and threads, so parallel versions may have more samples in a particular category even if the wall clock time was lower.

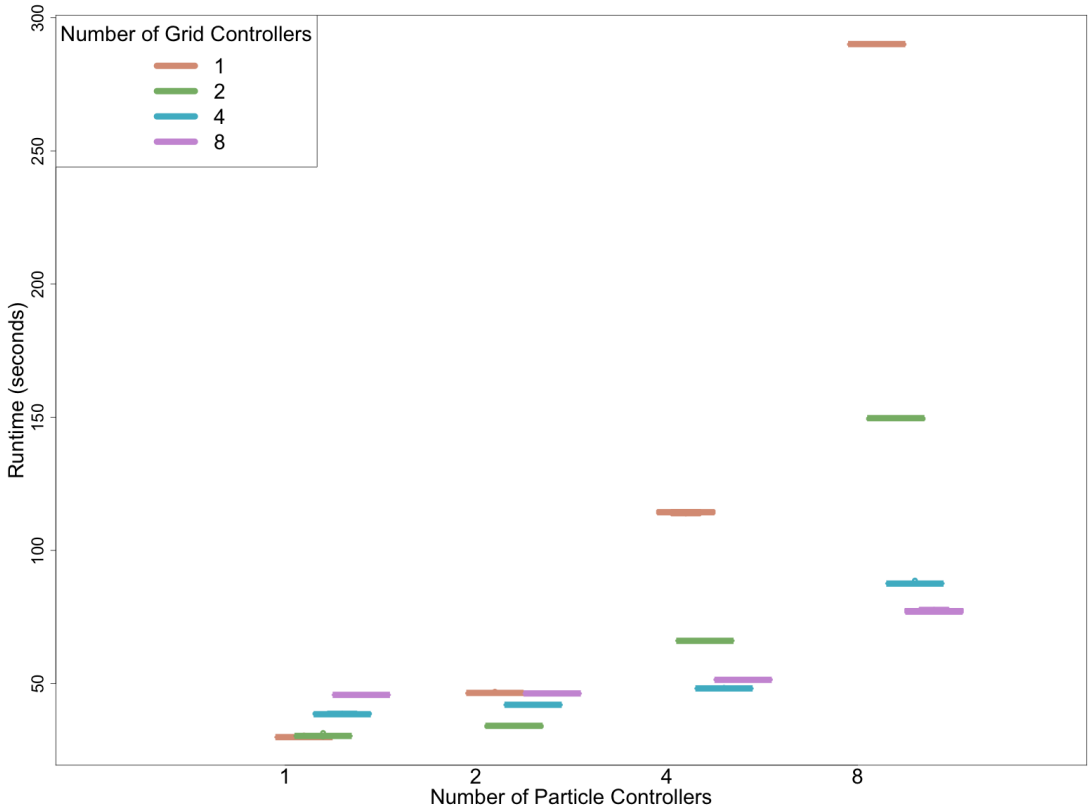



Figure 6: Timing for each of the MPI runs. Each particle controller was configured identically, so the runs with 2, 4, and 8 particle controllers simulated 2, 4, 8 times the number of particles.

may take only a few microseconds [11], only a few operations were performed between switches and the time spent switching dwarfed that spent doing useful work. By instead grouping the particles into a ParticleVector class that all requested forcing data together, then all moved together, it was possible to reduce the cost of context switching to a negligible amount. Although I have no evidence to support this, my working hypothesis for the slowdown in the fork version of the model when using -O2 or above optimization is that it is related to the context switching issue. If the compiler attempts to optimize caching by requesting data and moving each particle before advancing to the next particle, then the extra context switches could potentially lead to long runtimes. Since the compiler would not be aware that the grid controller and particle controller run in separate processes, it may not account for the interprocess communication in optimizing the code.

The choice of communication method in the fork version of the model also resulted in a substantial slowdown. The shared memory communication method used is a Interprocess Message Queue from the Boost libraries, which is a priority queue. Although the priority mechanism is not used by Nemo and all particles are coded with priority 0, the process of inserting new members into the queue may result in performance penalties. This problem could be alleviated with the use of a circular buffer in shared memory instead.

The next step in developing Nemo into a research is to refine the numerical methods and increase the stability. The spatial interpolation should use a bicubic or tricubic method, and integration should use a higher order method such as 4th order Runge-Kutta. Since the test model was run in a controlled environment and the size of the problem relative to the system resources was known in advance, none of the error codes returned by `malloc` or NetCDF library calls were checked, but they should be in practice.

Since all of the parallel versions spent more time processing the physical data than integrating the particle trajectories, that would be my first objective for optimization. In addition, the amount of data transmitted between MPI nodes and copied from the host to device in the CUDA version can be vastly reduced. The MPI workers transmit a structure with x, y, u, and v both to the grid controller and back to the particle controller, but only x and y need to be transmitted to the grid controller and only u and v back. In the GPU version, the entire grid and set of particles is copied to and from the device every timestep. Using a small internal timestep and writing the particles to disk only every few timesteps could reduce the amount of particle data copied. Moving the temporal interpolation to the grid would reduce the amount of grid data copied and speed up the interpolation. Also, better use of shared and thread specific memory on the GPU may speed up the computations

At this point, Nemo is an interesting toy Lagrangian particle-tracking model that has taught me a great deal about the caveats of parallel computing, but a fair amount of additional work is required before it will be ready for use in oceanographic research.

# References


[1] Davide Ancona, Massimo Anacona, Antonio Cuni, and Nicholas D Matsakis. RPython: a step towards reconciling dynamically and statically typed oo languages. In *Proceedings of the 2007 Symposium on Dynamic Languages*, 2007.

[2] Jeff Bezanson, Stefan Karpinski, Viral B Shah, and Alan Edelman. Julia: a fast dynamic language for technical computing. *CoRR*, abs/1209.5:1–27, 2012.

[3] Rainer Bleck. An oceanic general circulation model framed in hybrid isopycnic-Cartesian coordinates. *Ocean Modelling*, 37:55–88, 2002.

[4] Changsheng Chen, Robert C Beardsley, and Geoffrey Cowles. An unstructured-grid finite-volume coastal ocean model (FVCOM) system. *Oceanography*, 19(1):78–89, 2006.

[5] Christian Conti, Diego Rossinelli, and Petros Koumoutsakos. GPU and APU computations of Finite Time Lyapunov Exponent fields. *Journal of Computational Physics*, 231(5):2229–2244, 2012.

[6] R. K. Cowen. Connectivity of Marine Populations: Open or Closed? *Science*, 287(5454):857–859, February 2000.

[7] J.C. Dietrich, C.J. Trahan, M.T. Howard, J.G. Fleming, R.J. Weaver, S. Tanaka, L. Yu, R.a. Luettich, C.N. Dawson, J.J. Westerink, G. Wells, a. Lu, K. Vega, a. Kubach, K.M. Dresback, R.L. Kolar, C. Kaiser, and R.R. Twilley. Surface trajectories of oil transport along the Northern Coastline of the Gulf of Mexico. *Continental Shelf Research*, 41:17–47, June 2012.

[8] Alexis Hérault, Guiseppe Bilotta, and Robert A Dalrymple. SPH on GPU with CUDA. *Journal of Hydraulic Research*, 48:74–79, 2010.

[9] Brian P Kinlan and Steven D Gaines. Propagule dispersal in marine and terrestrial environments: a community perspective. *Ecology*, 84(8):2007–2020, August 2003.

[10] I-Huan Lee, Dong Shan Ko, Yu-Huai Wang, Luca Centurioni, and Dong-Ping Wang. The mesoscale eddies and Kuroshio transport in the western North Pacific east of Taiwan from 8-year (2003–2010) model reanalysis. *Ocean Dynamics*, 63(9-10):1027–1040, July 2013.

[11] Chuanpeng Li, Chen Ding, and Kai Shen. Quantifying the cost of context switch. In *Proceedings of the 2007 workshop on experimental computer science*, number June, pages 13–14, 2007.

[12] Jason Mak, Paul Choboter, and Chris Lupo. Numerical Ocean Modeling and Simulation with CUDA. In *Proceedings of OCEANS*, 2011.

[13] Akira Okubo. Oceanic diffusion diagrams. *Deep Sea Research*, 18(152):789–802, 1971.

[14] Claire B Paris, LM Chérubin, and Robert K Cowen. Surfing, spinning, or diving from reef to reef: effects on population connectivity. *Marine Ecology Progress Series*, 347:285–300, October 2007.

[15] Claire B Paris, Judith Helgers, Erik van Sebille, and Ashwanth Srinivasan. Connectivity Modeling System: A probabilistic modeling tool for the multi-scale tracking of biotic and abiotic variability in the ocean. *Environmental Modelling & Software*, 42:47–54, April 2013.

[16] Alexander F. Shchepetkin. A method for computing horizontal pressure-gradient force in an oceanic model with a nonaligned vertical coordinate. *Journal of Geophysical Research*, 108(C3):3090, 2003.

[17] Alexander F. Shchepetkin and James C. McWilliams. The regional oceanic modeling system (ROMS): a split-explicit, free-surface, topography-following-coordinate oceanic model. *Ocean Modelling*, 9(4):347–404, January 2005.

[18] Erica Staaterman, Claire B Paris, and Judith Helgers. Orientation behavior in fish larvae: a missing piece to Hjort's critical period hypothesis. *Journal of Theoretical Biology*, 304:188–96, July 2012.

[19] Eric A Treml, Jason J Roberts, Yi Chao, Patrick N Halpin, Hugh P Possingham, and Cynthia Riginos. Reproductive output and duration of the pelagic larval stage determine seascape-wide connectivity of marine populations. *Integrative and Comparative Biology*, 52(4):525–537, 2012.

[20] James R Watson, Bruce E Kendall, David A Siegel, and Satoshi Mitarai. Changing seascapes, stochastic connectivity, and marine metapopulation dynamics. *The American Naturalist*, 180(1):99–112, 2012.