Julia for Geophysical Fluid Dynamics: Performance Comparisons between CPU, GPU, and Fortran-MPI

Robert R Strauss[1], Siddhartha Bishnu[1], and Mark R. Petersen[1]

[1]Los Alamos National Laboratory

January 16, 2023

Abstract

Some programming languages are easy to develop at the cost of slow execution, while others are lightning fast at run time but are much more difficult to write. Julia is a programming language that aims to be the best of both worlds—a development and production language at the same time. To test Julia's utility in scientific high-performance computing (HPC), we built an unstructured-mesh shallow water model in Julia and compared it against an established Fortran-MPI ocean model, MPAS-Ocean, as well as a Python shallow water code. Three versions of the Julia shallow water code were created, for: single-core CPU; graphics processing unit (GPU); and Message Passing Interface (MPI) CPU clusters. Comparing identical simulations revealed that our first version of the single-core CPU Julia model was 13 times faster than Python. Further Julia optimizations, including static typing and removing implicit memory allocations, provided an additional 10–20x speed-up of the single-core CPU Julia model. The GPU-accelerated Julia code is extremely fast, with a speed-up of 230-380x compared to the single-core CPU Julia code if communication with the GPU occurs every 10 time steps. Parallelized Julia-MPI performance was identical to Fortran-MPI MPAS-Ocean for low processor counts, and ranges from 2x faster to 2x slower for higher processor counts. Our experience is that Julia development is fast and convenient for prototyping, but that Julia requires further investment and expertise to be competitive with compiled codes. We provide advice on Julia code optimization for HPC systems.

Julia for Geophysical Fluid Dynamics: Performance Comparisons between CPU, GPU, and Fortran-MPI

Robert R. Strauss[1], **Siddhartha Bishnu**[2], **Mark R. Petersen**[2]

[1]Center for Nonlinear Studies, Los Alamos National Laboratory, NM, 87545, USA
[2]Computational Physics and Methods Group, Los Alamos National Laboratory, NM, 87545, USA

Key Points:

- Unstructured-mesh shallow water models were created in Julia for single-core CPU, single-node GPU, and multi-core CPU clusters using MPI.
- Julia-MPI performance ranges from 2x faster to 2x slower than Fortran-MPI. Julia on GPUs is significantly faster than on CPUs.
- Julia development time is quick for prototyping, but requires more time to develop performant code; specifically, static typing is required.

Corresponding author: Mark R. Petersen, `mpetersen@lanl.gov`

Abstract

Some programming languages are easy to develop at the cost of slow execution, while others are lightning fast at run time but are much more difficult to write. Julia is a programming language that aims to be the best of both worlds—a development and production language at the same time. To test Julia's utility in scientific high-performance computing (HPC), we built an unstructured-mesh shallow water model in Julia and compared it against an established Fortran-MPI ocean model, MPAS-Ocean, as well as a Python shallow water code. Three versions of the Julia shallow water code were created, for: single-core CPU; graphics processing unit (GPU); and Message Passing Interface (MPI) CPU clusters. Comparing identical simulations revealed that our first version of the single-core CPU Julia model was 13 times faster than Python. Further Julia optimizations, including static typing and removing implicit memory allocations, provided an additional 10–20x speed-up of the single-core CPU Julia model. The GPU-accelerated Julia code is extremely fast, with a speed-up of 230-380x compared to the single-core CPU Julia code if communication with the GPU occurs every 10 time steps. Parallelized Julia-MPI performance was identical to Fortran-MPI MPAS-Ocean for low processor counts, and ranges from 2x faster to 2x slower for higher processor counts. Our experience is that Julia development is fast and convenient for prototyping, but that Julia requires further investment and expertise to be competitive with compiled codes. We provide advice on Julia code optimization for HPC systems.

Plain Language Summary

Scientists who write programs for supercomputers try to satisfy two requirements: the code should be both fast and easy to understand. These requirements are often in conflict, because fast programs use special libraries that add extra lines to the code and make it less readable. Supercomputers also change over time—for several decades, they had thousands of identical CPUs (each similar to a desktop), but in the past decade they include CPUs accelerated by graphics processing units (GPUs). This added hardware complexity results in more complex software. Here we test a relatively new programming language, Julia, which is designed to be simpler to write, but also to be fast on advanced computer architectures. We find that Julia is both convenient and fast, but there is no free lunch. Our first attempt to develop an ocean model in Julia was relatively easy, but the code was slow—it was 70 times slower than a long-standing ocean model written in Fortran. After several months of further development and experimentation, we did indeed create a Julia code that is as fast on supercomputers as the Fortran ocean model.

1 Introduction

A major concern in computer modeling is the trade-off between execution speed and code development time. In general, programs in scripting languages like Python and Matlab are faster to develop due to their simpler syntax and more relaxed typing requirements, but are limited by slower execution time. On the other end of the spectrum, we have compiled languages like C/C++ and Fortran, which have been extensively used in scientific computing for many decades. Programs in such languages are blessed with faster execution time, but are cursed with stricter and more cumbersome syntax, leading to slower development time. The Julia language strikes a balance between these two categories (Perkel, 2019). It is a compiled language with execution speed similar to C/C++ or Fortran, if carefully written with strict syntax (Lin & McIntosh-Smith, 2021; Gevorkyan et al., 2019). It is also equipped with a more convenient syntax and features, such as dynamic typing, to accelerate code development in prototyping. To this day, the majority of scientific computing models are programmed in compiled languages, which execute fast but can take months, if not years, to develop. In this paper, we investigate the feasability of writing Julia codes for computational physics simulations, since a Julia program can not only ensure high performance but also less development time in the initial stages. We develop a shallow water solver in Julia and compare its performance to an equivalent Fortran code.

An additional complication in choosing the best language is that layers of libraries have been added to C/C++ and Fortran to accommodate evolving computer architectures. For the past 25 years, computational physics codes have largely used the Message Passing Interface (MPI) to communicate between CPUs on separate nodes that do not share memory, and OpenMP to parallelize within a node using shared-memory threads. With the advent of heterogeneous nodes containing both CPUs and GPUs, scientific programmers have several new choices: writing kernels directly for GPUs in CUDA (Bleichrodt et al., 2012; Zhao et al., 2017; Xu et al., 2015); adding OpenACC pragmas for the GPUs (Jiang et al., 2019); or calling libraries such as Kokkos (Trott et al., 2022) that execute code optimized for specialized architectures on the back-end, while providing a simpler front-end interface for the domain scientist. All of these require additional expertise, and add to the length and complexity of the code base. Julia also provides an MPI library for parallelization across nodes in a cluster, and a CUDA library to parallelize over GPUs within a node. We have written shallow water codes in Julia that adopt each of these parallelization strategies.

In recent years, shallow water solvers such as Oceananigans.jl (Ramadhan et al., 2020) and ShallowWaters.jl (Klöwer et al., 2022) have been developed in Julia. These codes employ structured rectilinear meshes to discretize their domain, and are equipped with capabilities for running on GPUs to achieve high performance. Here we conduct a comparison on unstructured-mesh models, using the Fortran code MPAS-Ocean (Ringler et al., 2013) as a point of reference. MPAS-Ocean employs unstructured near-hexagonal meshes with variable resolution capability and is parallelized with MPI for running on supercomputer clusters. We developed a Julia model employing the same spatial discretization of MPAS-Ocean, and capable of running in serial mode on a single core, or in parallel mode on a supercomputer cluster or a graphics card. We discuss the subtle details of our implementations, compare the speed-ups attained, and describe the strategies employed to enhance performance.

2 Methods

2.1 Equation Set & TRiSK-Based Spatial Discretization

Our Julia model solves the shallow water equations (Cushman-Roisin & Beckers, 2011) in vector-invariant form. This is sufficiently close to the governing equations for ocean and atmospheric models to be used as a proxy to test performance with new codes and architectures. The equation set is

$$\boldsymbol{u}_t + qh\boldsymbol{u}^\perp = -g\nabla\eta - \nabla K, \tag{1a}$$

$$\eta_t + \nabla\cdot(h\boldsymbol{u}) = 0, \tag{1b}$$

where $\boldsymbol{u}$ is the horizontal velocity vector, $\boldsymbol{u}^\perp = \boldsymbol{k}\times\boldsymbol{u}$, h is the layer thickness, η is the surface elevation or sea surface height (SSH), $K = |\boldsymbol{u}|^2/2$ is the kinetic energy, and g is the acceleration due to gravity. If b represents the topographic height and H the mean depth, then $\eta = h + b - H$. Moreover, if f denotes the Coriolis parameter, and $\zeta = \boldsymbol{k}\cdot\nabla\times\boldsymbol{u}$ the relative vorticity, then the absolute vorticity, $\omega_a = \zeta + f$, and the potential vorticity, $q = \omega_a/h$. The term $qh\boldsymbol{u}^\perp$ is the thickness flux of the PV in the direction perpendicular to the velocity, rotated counterclockwise on the horizontal plane. Ringler et al. (2010) refer to it as the non-linear Coriolis force since it consists of the quasi-linear Coriolis force $f\boldsymbol{u}^\perp$ and the rotational part $\zeta\boldsymbol{u}^\perp$ of the non-linear advection term $\boldsymbol{u}\cdot\nabla\boldsymbol{u}$. We spatially discretize the prognostic equations in (1) using a mimetic finite volume method based on the TRiSK scheme, which was first proposed by (Thuburn et al., 2009), and then generalized by (Ringler et al., 2010). This method was chosen to horizontally discretize the primitive equations of MPAS-Ocean while invoking the hydrostatic, incompressible, and Boussinesq approximations on a staggered C-grid. Since this horizontal discretization guarantees conservation of mass, potential vorticity, and energy, it makes MPAS-Ocean a suitable candidate to simulate mesoscale eddies.

Our spatial domain is tessellated by two meshes, a regular planar hexagonal primal mesh and a regular triangular dual mesh. Each corner of the primal mesh cell coincides with a vertex of the dual mesh cell and vice versa. A line segment connecting two primal mesh cell centers is the perpendicular bisector of a line segment connecting two dual mesh cell centers and vice versa. Regarding our prognostic variables, the scalar SSH η is defined at the primal cell centers, and the normal velocity vector $\boldsymbol{u}_e$ is defined at the primal cell edges. The divergence of a two-dimensional vector quantity is defined at the positions of η, while the two-dimensional gradient of a scalar quantity is defined at the positions of $\boldsymbol{u}_e$ and oriented along its direction. The curl of a vector quantity is defined at the vertices of the primal cells. Finally, the tangential velocity $\boldsymbol{u}_e^\perp$ along a primal cell edge is computed diagnostically using a flux mapping operator from the primal to the dual mesh, which essentially takes a weighted average of the normal velocities on the edges of the cells sharing that edge. Interested readers may refer to Thuburn et al. (2009) and Ringler et al. (2010) for further details on the mesh specifications.

At each edge location $\boldsymbol{x}_e$, two unit vectors $\boldsymbol{n}_e$ and $\boldsymbol{t}_e$ are defined parallel to the line connecting the primal mesh cells, and in the perpendicular direction rotated counterclockwise on the horizontal plane, such that $\boldsymbol{t}_e = \boldsymbol{k} \times \boldsymbol{n}_e$. The discrete equivalent of the set of equations (1) is

$$\left(u_e\right)_t = F_e^\perp \widehat{q}_e - \left[\nabla\left(g\eta\right)_i + K_i\right]_e, \tag{2a}$$

$$\left(\eta_i\right)_t = -\left[\nabla \cdot F_e\right]_i, \tag{2b}$$

where $F_e = \widehat{h}_e u_e$ and $F_e^\perp$ represent the thickness fluxes per unit length in the $\boldsymbol{n}_e$ and $\boldsymbol{t}_e$ directions respectively. The layer thickness h_i, the SSH η_i, the topographic height b_i, and the kinetic energy K_i are defined at the centers $\boldsymbol{x}_i$ of the primary mesh cells, while the velocity u_e are defined at the edge points $\boldsymbol{x}_e$. The symbol $\widehat{(.)}_e$ represents an averaging of a field from its native location to $\boldsymbol{x}_e$. The discrete momentum equation (2b) is obtained by taking the dot product of (1b) with $\boldsymbol{n}_e$, which modifies the non-linear Coriolis term to

$$\begin{aligned} \boldsymbol{n}_e \cdot \widehat{q}_e \widehat{h}_e \boldsymbol{u}^\perp &= \widehat{q}_e \widehat{h}_e \boldsymbol{n}_e \cdot (\boldsymbol{k} \times \boldsymbol{u}) = \widehat{q}_e \widehat{h}_e \boldsymbol{u} \cdot (\boldsymbol{n}_e \times \boldsymbol{k}) \\ &= -\widehat{q}_e \widehat{h}_e \boldsymbol{u} \cdot \boldsymbol{t}_e = -\widehat{q}_e \widehat{h}_e u_e^\perp = -F_e^\perp \widehat{q}_e. \end{aligned} \tag{3}$$

Given the numerical solution at time level $t^n = n\Delta t$, with Δt representing the time step and $n \in \mathbb{Z}_{\geq 0}$, the Julia model first computes the time derivative or tendency terms of (2) as functions of the discrete spatial and flux-mapping operators of the TRiSK scheme. Then it advances the numerical solution to time level t^{n+1} using the forward-backward method

$$u^{n+1} = u^n + \Delta t \mathcal{F}\left(u^n, h^n\right), \tag{4}$$

$$h^{n+1} = h^n + \Delta t \mathcal{G}\left(u^{n+1}, h^n\right), \tag{5}$$

where $\mathcal{F}$ and $\mathcal{G}$ represent the discrete tendencies of the normal velocity and the layer thickness in functional form, and the subscripts representing the positions of these variables have been dropped for notational simplicity.

The following sections introduce the new codes that were created for this study. Three versions of the Julia code were written (Strauss, 2023): the base single-core CPU version, an altered version for GPUs with CUDA, and a multi-node CPU implementation with Julia-MPI. These were compared against existing Fortran-MPI and Python versions of shallow-water TRiSK models. All use a standard MPAS unstructured-mesh file format that specifies the geometry and topology of the mesh, and includes index variables that relate neighboring cells, edges, and vertices. All models have an inner (fastest-moving) index for the vertical coordinate and were tested with 100 vertical layers to mimic performance in a realistic ocean model.

2.2 Single-Core CPU Julia Implementation

The serial-mode implementation on a single core involves looping over every cell and edge of the mesh to (a) compute the tendencies, i.e. the right-hand side terms of the prognostic equations (2) and (b) advance their values to the next time step. The tendencies can be functions of the dependent and independent variables as well as spatial derivatives of the dependent variable. The serial version of our model is the simplest one from the perspective of transforming the numerical algorithms into code.

In order to highlight differences in formulation, we provide a Julia code example for the single tendency term from (2) for the SSH gradient $-g\nabla\eta$, which is discretized as $-\left[g\nabla\eta_i\right]_e$. We then add a vertical index k to mimic the performance of a multi-layer ocean model, but each layer is trivially redundant. In a full ocean model this term would be the pressure gradient, and would involve the computation of pressure as a function of depth and density. For the single-core CPU version, the Julia function computing the SSH gradient is

Listing 1: Julia example for serial CPU

```julia
velocity_tendencies!(sshGradient, ssh, ...)

function velocity_tendencies!(sshGradient, ssh, ...)
    for iEdge in 1:nEdges
        cell1 = cellsOnEdge[1,iEdge]
        cell2 = cellsOnEdge[2,iEdge]
        for k in 1:nVertLevels
            sshGradient[k,iEdge] = - gravity / dcEdge[iEdge]
                * ( ssh[k,cell2] - ssh[k,cell1] )
        end
    end
end
```

Here `cellsOnEdge` is an array of index variables describing the mesh that points to the cells neighboring an edge, and `dcEdge` represents the distance between the centers of adjacent cells sharing the edge on which the normal velocity tendency is computed. In the actual code all the tendency terms are computed within this function, but here we only show the ssh gradient as a brief sample.

2.3 SIMD GPU Julia Implementation

GPUs are very powerful tools for SIMD (Same Instruction Multiple Data) computations: they have thousands of independent threads, which can execute the same operation at the same time with different input values. Since we numerically solve the same prognostic equation for (a) the SSH at every cell center $\boldsymbol{x}_i$, and (b) the normal velocity at every edge $\boldsymbol{x}_e$ of the mesh, a GPU is a logical tool to employ for our computations. By placing subsets of cells and edges on different threads of the GPU, we can perform the tendency computations, and advance the prognostic variables at once in parallel rather than looping over every cell and edge, which would scale in wall-clock time according to the size of the mesh.

We wrote CUDA kernels for an Nvidia GPU using the CUDA.jl library for computing the tendencies and advancing the prognostic variables to the next time step. The code for the single-core implementation can be converted to CUDA with surprising ease by removing the `for` loop over the cells and edges, and instead performing the underlying computation on a single cell or edge:

Listing 2: Julia example for GPU with CUDA

```julia
CUDA.@cuda blocks=cld(nEdges, 1024) threads=1024 maxregs=64
    velocity_tendencies_cuda!(sshGradient, ssh, ...)

```

```julia
function velocity_tendencies_cuda!(sshGradient, ssh, ...)
    iEdge = (CUDA.blockIdx().x - 1) * CUDA.blockDim().x
        + CUDA.threadIdx().x
    cell1 = cellsOnEdge[1,iEdge]
    cell2 = cellsOnEdge[2,iEdge]
    for k in 1:nVertLevels
        sshGradient[k,iEdge] = - gravity / dcEdge[iEdge]
            * ( ssh[k,cell2] - ssh[k,cell1] )
    end
end
```

Each cell and edge of the mesh will be designated to a different thread on the GPU. The computation for a single cell or edge will run on a single thread, and a CUDA method will be used to map the index of the thread to the index of the cell (i) or edge (e), at which the prognostic variable is being updated. To execute this method over all threads on the GPU, we use a CUDA macro to call our kernel and designate the number of threads to use, which should be equal to the number of cells or edges in the mesh. Note that the inner computation of `pressureGradient` is identical for the CPU and CUDA kernal codes.

2.4 CPU/MPI Julia Implementation

Rather than iterating through every cell or edge of the mesh, we may parallelize the simulation with multiple processors by assigning to each processor a portion of the mesh, a process called domain decomposition. However, the computations of some spatial operators may require information from the outermost cells of the adjacent processors. So, we need the neighboring processors to communicate these pieces of information with each other. To ensure an efficient communication, we include an extra ring or "halo" of cells around the boundary of the region assigned to each processor, which overlaps with the region assigned to adjacent processors. We do not compute the tendencies of the prognostic variables in the halo region of a processor. In fact, we cannot perform this operation without information in an additional ring of halo cells, which is not assigned to the processor under consideration. So, we obtain the updated values of the prognostic variables in the halo region by communication with adjacent processors, which contain these halo cells in their interior, and update the prognostic variables in them.

A number of crucial modifications are necessary to implement this parallelization scheme. For instance, the simulation methods are amended so that each process (rank) only performs computations for the set of cells or edges assigned to it. We use the MPI communication channel (comm) to receive the updated values of the prognostic variables in the halo region of a processor from adjacent processors which advance these variables. Similarly, we send the updated values of the prognostic variables along the outermost region of the above-mentioned processor to adjacent processors, for which these variables belong in the halo regions. For the TRiSK-based spatial discretization and the forward-backward time-stepping method, the halo region consists of only one layer (one halo ring) of cells.

Listing 3: Julia example for CPU with MPI

```julia
# each process executes the following, receiving a different value
# on each rank:
comm = MPI.COMM_WORLD
rank = MPI.Comm_rank(comm)

myCells = cells_for_rank(mesh_file, rank, partition_file)
myEdges, myHaloEdges = edges_on_cells(myCells)

velocity_tendencies!(myEdges, sshGradient, ssh, ...)
update_halo_edges!(sshGradient, myHalodEdges, rank, comm)
```

```
function velocity_tendencies!(myEdges, sshGradient, ssh, ...)
    for iEdge in myEdges
        cell1 = cellsOnEdge[1,iEdge]
        cell2 = cellsOnEdge[2,iEdge]
        for k in 1:nVertLevels
            sshGradient[k,iEdge] = - gravity / dcEdge[iEdge]
                * ( ssh[k,cell2] - ssh[k,cell1] )
        end
    end
end

function update_halo_edges!(data, edgesInMyHalo, rank, comm)
    for neighborRank in find_neighbors(rank, comm)
        MPI.Irecv!(data[edgesInMyHalo,:], neighborRank, 0, comm)
        edgesToNeighbor = find_halo_overlap(rank, neighbor, comm)
        MPI.Isend(data[edgesToNeighbor,:], neighborRank, 0, comm)
    end
end
```

Here `myCells` and `myEdges` are the lists of cells and edges in the local domain, owned by the rank running this code, plus its halo.

2.5 CPU/MPI Fortran Implementation

The baseline comparison code for this study is the Model for Prediction Across Scales (MPAS-Ocean) (Ringler et al., 2013; Petersen et al., 2015), which is written in Fortran with MPI communication commands. It is the ocean component of the Energy Exascale Earth System Model (E3SM) (Golaz et al., 2019; Petersen et al., 2019), the climate model developed by the US Department of Energy. In this study, the code is reduced from a full ocean model solving the primitive equations to simply solving for velocity and thickness (1). Thus the majority of the code is disabled, including the tracer equation, vertical advection and diffusion, the equation of state, and all parameterizations. In order to match the Julia simulations, we employ a forward-backward time-stepping scheme, exchange one-cell-wide halos after each time step, compute 100 layers in the vertical array dimension, and use the identical Cartesian hexagon-mesh domains (Petersen et al., 2022).

MPAS-Ocean is an excellent comparison case for Julia because it is a well-developed code base that uses Fortran and MPI, which have been standard for computational physics codes since the late 1990s. The highest resolution simulations in past studies used over three million horizontal mesh cells and 80 vertical layers, scale well to tens of thousands of processors (Ringler et al., 2013) and have been used for detailed climate simulations (Caldwell et al., 2019). MPAS-Ocean includes OpenMP for within-node memory access, and is currently adding OpenACC for GPU computations, but these were not used for this comparison to Julia-MPI on a CPU cluster.

2.6 CPU Python Implementation

In addition to MPAS-Ocean, we compare the performance of the Julia shallow water code against an object-oriented Python code Bishnu (2022). The Python code solves the rotating shallow water system of equations using two types of spatial discretizations: the TRiSK-based mimetic finite volume method used in MPAS-Ocean, and a discontinuous Galerkin Spectral Element Method (DGSEM). The code offers a number of standard predictor-corrector and multistep time-stepping methods, including those analyzed for ocean modeling in Shchepetkin and McWilliams (2005).

The Julia shallow water code was first written by translating this Python code into Julia syntax. While the Julia code was expanded for parallelization and performance, the Python code was further developed to serve as a platform for conducting a verification suite of shallow water test cases for the barotropic solver of ocean models. Each of these test cases in the Python code verifies the implementation of a subset of terms in the prognostic momentum and continuity equations, e.g. the linear pressure gradient term, the linear constant or variable-coefficient Coriolis and bathymetry terms, and the non-linear advection terms. Bishnu et al. (2022) and Bishnu (2021) provide detailed discussions on these test cases along with specifics of the numerical implementation, the time evolution of the numerical error for both spatial discretizations and a subset of the time-stepping methods, and results of convergence studies with refinement in both space and time, only in space, and only in time. Out of all of these test cases, only the linear coastal Kelvin wave and inertia-gravity wave test cases were implemented in the Julia code for the current study.

While not used in this study, a number of libraries exist to accelerate Python for various architectures. These include Numba and PyCuda for GPUs, mpi4py for CPU clusters, and Cython for single-CPU acceleration. Numba (Lam et al., 2015) is an open-sourced Anaconda-sponsored NumPy-aware optimizing compiler, which translates Python functions to fast machine code at runtime using the remarkable industry-standard LLVM compiler library. PyCUDA (Klöckner et al., 2012), written in C++ (the base layer) and Python, provides access to Nvidia's CUDA parallel computation API from Python. Mpi4py (Dalcín et al., 2005, 2008), provides Python bindings for the Message Passing Interface (MPI) standard. As an alternative, one can 'cythonize' an existing Python code by providing static type declarations and class attributes, that can then be translated to C++/C code and to C-Extensions for Python. Cython is an optimising static compiler for both the Python programming language and the extended Cython programming language. It is designed to offer C-like performance with code mostly written in Python with additional C-inspired syntax. The rotating shallow water Python code Bishnu (2022) is currently undergoing cythonization. Cythonized codes can further be accelerated on GPUs using Nvidia's HPC C++ compiler, and the C++ Standard Parallelism (stdpar) for GPUs (Srinath, 2020). However, the extent of additional modifications and enhancements required to bring GPU-accelerated C++ algorithms to the Python ecosystem may not always be a reasonable investment of time. As we will see in later sections, a serial Julia code, which already achieves the performance of a fast compiled language, does not require extensive modifications to be parallelized on GPUs or multiple cores, and is therefore more convenient than python for high-performance scientific computing applications.

3 Results

3.1 Model Verification

Each serial and parallel implementation of the shallow water model described in the previous section was verified for accuracy with convergence tests against exact solutions. We obtained the expected second-order convergence of the various TRiSK-based spatial operators on a uniform planar hexagonal MPAS-Ocean mesh. The operators included the gradient, the divergence, the curl, and the flux-mapping operator used to interpolate the tangential velocities from the normal velocities (Figure 1). The formulation of these operators is shown in Figure 3 of Ringler et al. (2010). Once the operator tests were complete, the linearized shallow water equations were verified against exact solutions for the coastal Kelvin wave and inertia-gravity wave cases, as described in Bishnu et al. (2022) and Bishnu (2021). With refinement in both space and time, we observe the expected first-order convergence of the numerical solution (Figure 1), spatially discretized with the second-order TRiSK scheme, and advanced with the first-order forward-backward time-stepping method (Bishnu, 2021).

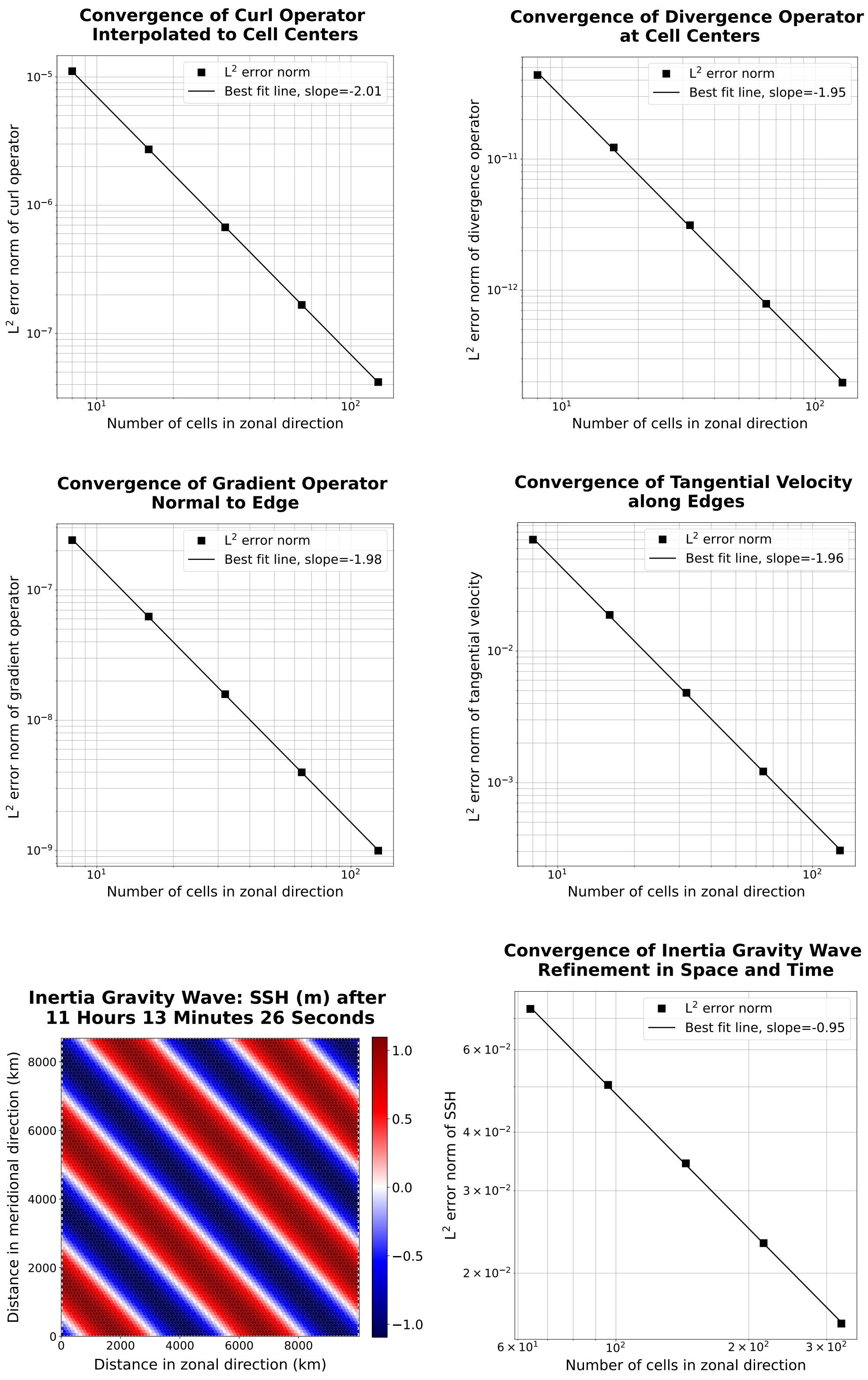


Figure 1: The first two rows show convergence plots of the TRiSK-based spatial operators for the newly-developed Julia code. Tests were run with both CPU and GPU implementations, and identical results were obtained. The slope of -2 indicates the expected second-order convergence. The third row shows a snapshot of the inertia-gravity wave test case, and the convergence plot of the numerical solution with refinement in both space and time.

3.2 Acceleration of Julia with GPU Hardware

The Julia serial CPU version of the shallow water model was compared against the Julia CUDA library GPU version and the reference Python CPU code (Table 1 and Figure 2). Tests were conducted on the Darwin cluster at Los Alamos National Laboratory, using a single node equipped with Intel Cascade Lake CPUs (Gold 6248 with a clock rate of 2.5 GHz and 27.5M Cache) and the Nvidia Quadro RTX 8000 "Turing" GPU architecture (4608 CUDA cores, 16.3 TFLOPS peak single precision performance, 48 GB GPU memory, and GPU memory bandwidth of 672 GB/s). All performance tests described in this and the following sections used the coastal Kelvin wave test case on a planar hexagon mesh with the linear shallow water equations and 100 vertical layers. Samples are averaged over ten trials. All codes use double-precision (8 byte) real numbers, and performance tests do not include the time for initialization, input/output, or generating plots.

In our first version of the Julia single-core CPU code, we did not take any special steps for code optimization, and it was already 13 times faster than Python. Julia and Python both have dynamic typing, but Julia has the ability to go much faster since it also supports concrete typing. Julia is compiled, but hides it cleverly by compiling on the fly based on what datatypes are provided at run time. It supports a hierarchical abstract typing system, allowing for semi-specified types, such as "Any", which all types extend and is the default if no type is specified (thus acting like python), or "AbstractArray", which can be occupied at run time with any Array-like data.

After the initial Julia development, further effort was put into optimization, which led to a 10–20 times speed-up for the CPU-serial code. The changes included optimizing for memory management by tracking down and reducing unnecessary allocations that contributed significantly to the run time, as well as making all types and subtypes concrete rather than abstract, to minimize on-the-fly compilation. These improvements are explained in more detail in section 4.

We found the CUDA GPU implementation to be *significantly* faster than the single-core implementation. Because the memory transfer between the CPU and GPU takes many orders of magnitude longer than the actual on-GPU computations, we split them out in Table 1 and Figure 2. The memory transfers require between 0.015s and 0.68s and scale with the array size, while the GPU computations alone are extraordinarily fast, at 0.00027s for the 512x512 resolution case, and do not scale with resolution. This shows the power of GPUs, where computations alone can run over 40,000 times faster on the GPU than the CPU, but this speed-up is substantially diminished by the memory transfer time. Still, codes that are designed with a small memory footprint and limited memory transfer can greatly benefit from GPU computations. Strategically reducing array precision to 4-byte or even 2-byte reals for certain variables allows higher-resolution domains to fit on GPUs (Ye et al., 2022; Klöwer et al., 2022). In addition, single-precision floating point numbers (CUDA `Float32` data type) calculations may execute significantly faster than `Float64` (*Introduction to CUDA*, 2022). We did not leverage `Float32` in this work, but it shows that GPU simulations could run even faster than the results shown here.

Summing the GPU memory transfer and compute for the 10 timestep performance test, the GPUs were 229 to 386 times faster than the single CPU (Table 2). This compares to published studies of ocean models that show a speed-up from CPU to GPU ranging from 5–50 (Bleichrodt et al., 2012; Zhao et al., 2017; Xu et al., 2014), and a speed-up of up to 1556x for a GPU/CUDA Based Parallel Weather and Research Forecast Model (WRF) (Mielikainen et al., 2012). Note that our speed-up factor could be increased substantially by transferring data from the GPU to CPU less frequently. For a low-resolution ocean model with 30-minute time steps, the speed-ups in Table 2 correspond to collecting data every 10 time-steps, which is 5 hours of model time. One could instead collect data for analysis every 100 time-steps (∼2 days), and that would result in a GPU speed-up of 2290 to 3860, because the compute time is negligible compared to the memory transfer. On the other hand, if model communication is required frequently for surface data forcing or coupling with atmospheric and sea ice components, the speed-up is drastically reduced. For example, if memory must be transferred between the CPU and GPU every time step, the speed-ups range from 23—39. The point is that GPU performance is wholly dependant on the GPU communication frequency.

	128x128	**256x256**	**512x512**
Python, CPU	3.08E+03	1.31E+04	4.96E+04
Julia, CPU-serial (unoptimized)	2.25E+02	8.64E+02	3.86E+03
Julia, CPU-serial (optimized)	1.12E+01	7.43E+01	3.33E+02
Julia, GPU, total	4.90E−02	2.03E−01	8.64E−01
transfer to GPU	2.98E−02	1.16E−01	4.58E−01
compute on GPU	2.51E−04	2.67E−04	2.67E−04
transfer back to CPU	1.53E−02	9.54E−02	6.84E−01

Table 1: Wall clock duration (seconds) of performing ten timesteps with 100 layers on an Intel Cascade Lake CPU or an NVidia Turing GPU.

	128x128	**256x256**	**512x512**
Python, CPU	274	177	149
Julia, CPU-serial (unoptimized)	20	12	12
Julia, CPU-serial (optimized)	1	1	1
Julia, GPU	**229**	**366**	**386**

Table 2: Speed-up (bold) or slow-down (non-bold) factor compared to the optimized CPU-serial Julia version at the same resolution. GPU speed-ups are based on transferring arrays between GPU and CPU every ten time steps.

GPU threads are grouped into threadblocks (or just “blocks”) for efficiency. While calling the kernel function, we must specify the number of blocks and number of threads per block (the “block size”), as shown in listing 2. Within the kernel, we obtain the index of the block and thread, multiply the block index by the block size, and add the thread index to compute a global index. There is a maximum possible block size, but we can choose any smaller value to execute the kernel with. The block size does have an effect on how quickly the kernel runs, so we benchmarked the evaluation time of the same kernel run with different block sizes, as shown in Figure 3. Smaller block sizes run faster on the GPUs by 15%. This is interesting to note, but GPU compute time is so small compared to the memory transfer time that thread tuning has little impact on the overall simulation time.

3.3 Julia-MPI versus Fortran-MPI

Julia and Fortran codes were compared on multi-node CPU clusters, where both used MPI for communication between processors. Comparisons were made with domains of 128, 256, and 512-squared grid cells solving the shallow water equations. All timing tests were conducted for 10 time steps and repeated 12 times on each processor count, spanning 2 to 2048 processors by powers of two. The vertical dimension included 100 layers to mimic ocean model arrays and provide sufficient computational work on each processor. Separate timers report on computational work versus MPI communication within the time-stepping routine. The i/o, initialization, and finalization time is excluded.

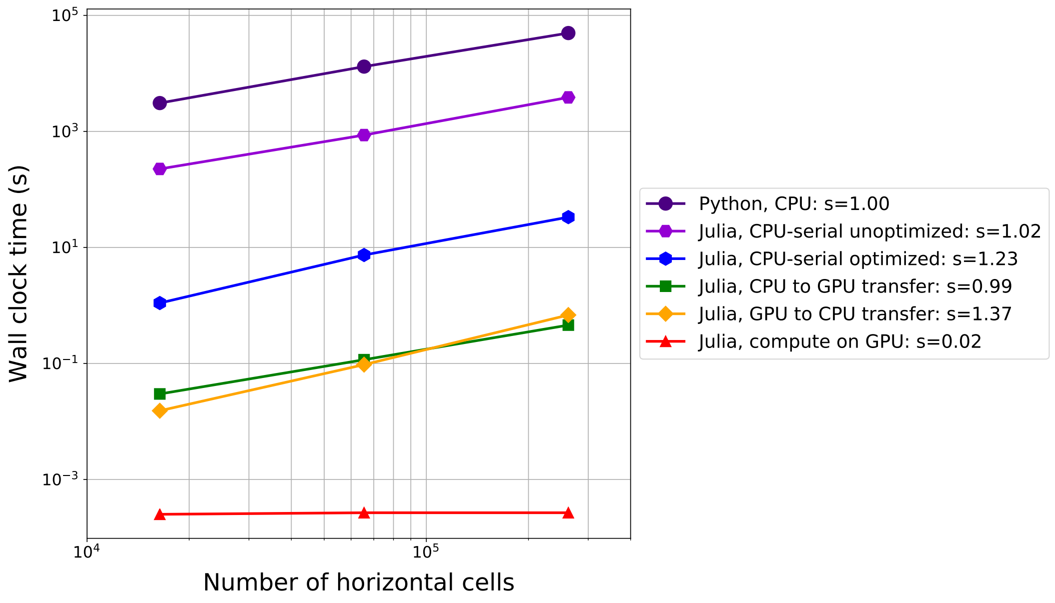


Figure 2: Timing data from Table 1, comparing ten timesteps of the Kelvin Wave test case on an Intel Cascade Lake CPU or an NVidia Turing GPU. The log-log slope, shown as s in the legend, is 1.0 for perfect scaling.

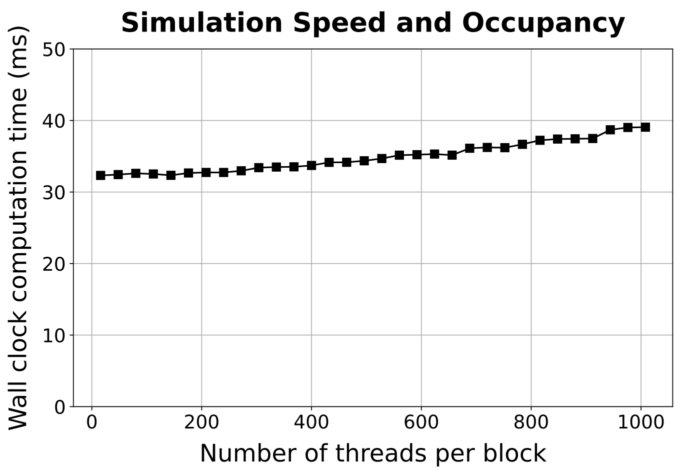


Figure 3: The same kernel was executed with the same data but different block sizes and the average execution time over 1000 runs was recorded. Fewer threads per block results in faster execution times on the GPUs.

Simulations were conducted on Cori-Haswell at the National Energy Research Scientific Computing Center (NERSC). Cori-Haswell consists of 2,388 nodes in 14 cabinets, using Intel Xeon Processor E5-2698 v3 with a clock rate of 2.3 GHz. Each processor has 32 physical threads per node and two hyper-threads per core, with 128 GB of memory per node. The interconnect is a Cray Aries with Dragonfly topology and > 45 TB/s global peak bisection bandwidth. The Julia-MPI and Fortran-MPI tests were both run with up to 32 ranks per node.

The scaling plots in Figure 4 show that the Julia-MPI and Fortran-MPI models have identical performance at two cores; Julia-MPI is faster by up to a factor of two for mid-range core counts; and Fortran-MPI is 2x faster than Julia-MPI at higher ranges, depending on the resolution. For both languages, computation scales well with processor count, while communication does not, and communication progressively requires a much larger fraction of time at higher processor counts (Figure 5). Once computations are optimized, communication, which is fixed by the interconnect speed, will remain a bottleneck regardless of the language. At the lowest resolution of 128x128, there is insufficient work beginning at 512 processors (which corresponds to 32 grid-cells per processor), and timing is dominated by communication, resulting in poor scaling above 512 processors. Communication times in Julia are much more variable than in Fortran across samples and processor counts, as shown in the right column of Figure 4. When measuring computation time without communication (Figure 4, right column), Julia-MPI scales nearly perfectly, while Fortran-MPI computational time drops off from perfect scaling at 8 and 16 cores. This produces the Julia times that are 2x faster for the total times for mid-range processor counts of 16 and higher. Overall, Julia performance on CPU clusters is extremely competitive with Fortran. Once the high-level codes have been optimized, the "winner" between Julia and Fortan will likely depend on the details of the MPI libraries and hardware.

4 Optimization Tips for Julia Developers

Julia serves the dual purpose of a prototyping language as well as a production language. Not only can we construct quick-to-write but slow-performing code (although still significantly faster than other development languages, as we saw with comparison to python) to demonstrate an idea, we can also spend a bit more time to carefully construct an optimized code to achieve performance on par with Fortran. Julia's ability to act as a prototyping language can be attributed to one of its key features: dynamic typing. Just like Python, variables may be initialized without defining their types. However, Julia is also endowed with a static typing feature, even though it is optional. If the variable types are statically defined in a concrete fashion, performance is greatly improved. Julia activates its dynamic typing feature with an "Any" type which could be any type at run time. So, Julia must compile parts of the code on the fly (*Eval of Julia code*, 2016). A method involving an "Any" type is compiled at run time for whatever type is actually provided during execution (called just-in-time compiling). The implication is that without static typing, performance will greatly suffer from compilation during run time. Additionally, with concrete types, the Julia compiler may optimize the code much further than if it is compiled for an unknown type.

When first creating the MPAS shallow water core in Julia, we did not specify the array types, and let Julia assign them the "Any" type:

```julia
struct MPAS_Ocean
    layerThickness
    normalVelocity
    ...
end
```

However, by concretely defining these variables to be floating point arrays, we gain a substantial performance boost:

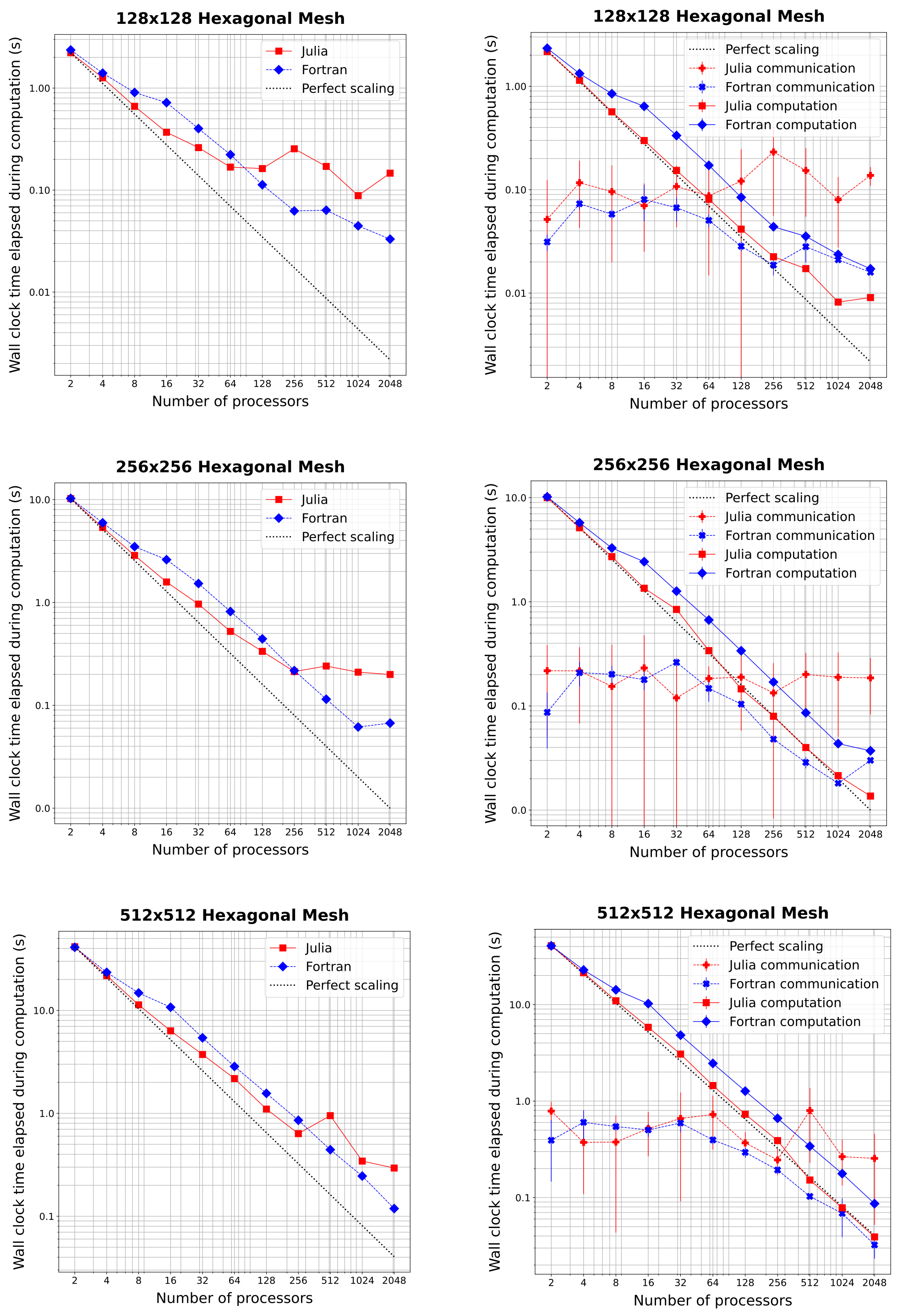


Figure 4: Wall clock time versus the number of processors to simulate 10 steps of the coastal Kelvin wave test with 100 layers. Left column shows total time without i/o; right column splits MPI communication and computation. Vertical lines display the standard deviation of communication times.

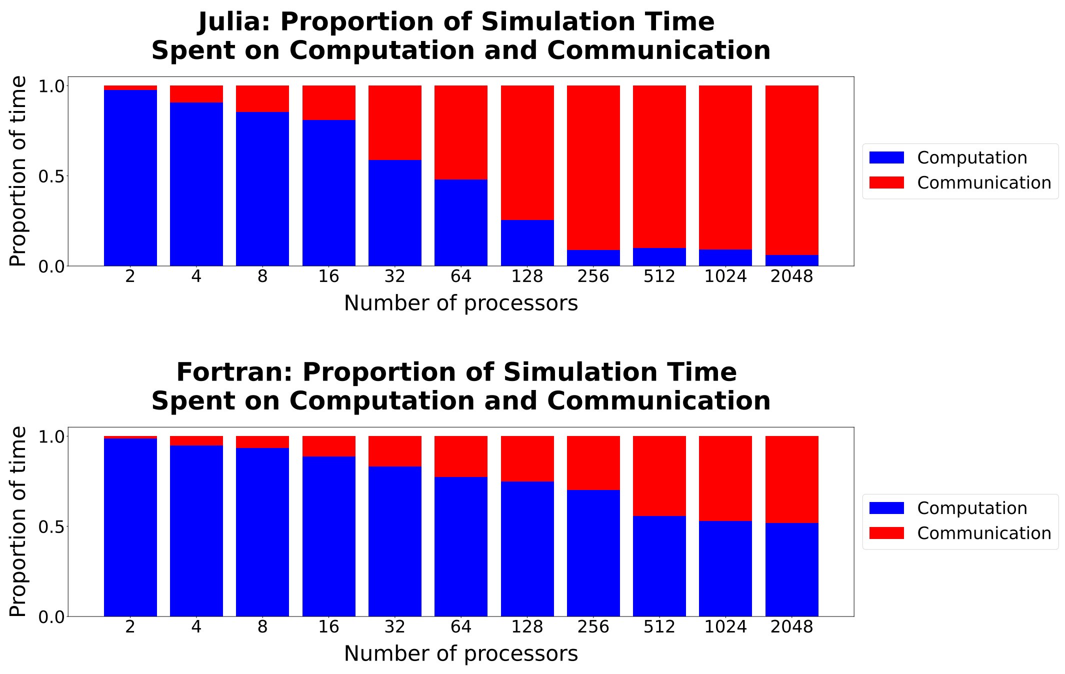


Figure 5: Comparison of the proportion of time spent in computation (blue) versus communication (red) in Julia-MPI (top) and Fortran-MPI (bottom) on the 128x128 hexagonal mesh. The relative time spent in communication increases dramatically at high processor counts.

```julia
struct MPAS_Ocean
    layerThickness::Array{Float64}
    normalVelocity::Array{Float64}
end
```

When parallelizing for the graphics card, a different array type is used that is suited for GPUs. We tried defining an abstract array type that encompasses both the CPU and GPU data types, so that `CUDA.CuArray`s and regular `Array`s could be used interchangeably, allowing the model to be run on the GPU or CPU at will. We also used an abstract type specification on the contents of these arrays `F <: Float`, meaning any type extending the abstract floating point type can be used at runtime.

```julia
struct MPAS_Ocean
    layerThickness::AbstractArray{F <: Float}
    normalVelocity::AbstractArray{F <: Float}
end
```

This approach seems like it should be performant, since the types are defined before run time. However, abstract types, like an Any type, slow down execution since at run time they may actually be a different type that extends the abstract type (`CUDA.CuArray` or `Array`), meaning the compiler is doing just-in-time compiling. Similarly, specifying an inexact element type (`F <: Float`) rather than a concrete type (`Float64`) is very inefficient.

Instead, two separate structures should be defined concretely when running on GPUs versus CPUs:

```julia
struct MPAS_Ocean_CUDA
```

```
    layerThickness::CUDA.CuArray{Float64,2}
    normalVelocity::CUDA.CuArray{Float64,2}
end

struct MPAS_Ocean
    layerThickness::Array{Float64,2}
    normalVelocity::Array{Float64,2}
end
```

Now the array types are concrete, element types are concrete (`Float64`), and the number of dimensions is specified (`Float64,2`). This code no longer has the advantageous feature of being able to switch between running on the CPU and GPU on the fly. However, the execution speed is massively improved. We found that making this change from abstract to concrete array types sped up computation by a factor of 34x.

The key in optimizing Julia code, we found, was reducing allocations. Memory allocation significantly slows down execution. And it is not always obvious what seemingly innocent actions may allocate memory. For example, simply reading a pair of values from an array with two columns:

```
cell1Index, cell2Index = cellsOnEdge[:,iEdge]
```

can allocate significant memory. In one test, this one line (executed repeatedly throughout the simulation) allocated 408 KiB. This is because the line is really creating a tuple, not directly reading each column into the two scalar variables. If we separate this into two lines to enforce only using scalars and not allocating tuples or arrays:

```
cell1Index = cellsOnEdge[1,iEdge]
cell2Index = cellsOnEdge[2,iEdge]
```

then this cuts allocations to zero—making this line almost instantaneous, and dropping the time spent on the whole tendency calculation from 198 μs to 99 μs. That means this line alone was responsible for about 50% of the computation time, when it could be rewritten to take no time at all.

There are likely many inconspicuous lines like this lurking in one's Julia code, slowing it down substantially. Additionally, even one overlooked field which is not concretely typed may significantly slow execution. Luckily, Julia is equipped with a tool to quickly locate such memory-hoarding lines. This tool is called `@code_warntype`. Prefixing a function call with it will print out a color-coded list breaking each line down to individual memory operations:

```
@code_warntype calculate_normal_velocity_tendency!(mpas)
```

It helpfully highlights inexact types and memory allocations with red, pointing a user right to the lines and fields that need to be optimized. This feature alone makes Julia very powerful for high-performance applications, significantly speeding up development time to optimize a model's performance.

Another very helpful tool when optimizing Julia code is `--track-allocations`, a command line option that can be added to any Julia execution as follows:

```
$ julia --track-allocations=user ./anyJuliascript.jl
```

A new file is created at `./anyJuliascript.jl.XXX.mem` (where `XXX` is some unique number). This file contains each line of the script prefixed by the number of memory allocations created by that line, giving a line-by-line breakdown of where allocations occur.

5 Conclusion

As new programming languages and libraries become available, it is important for model developers to learn new techniques and evaluate them against their current methods. This is particularly true as computing architectures continue to evolve, and long-standing languages such as C++ and Fortran require additional libraries to remain competitive on new supercomputers.

In this work, we created three implementations of a shallow water model in Julia in order to compare ease of development and performance to standard Fortran and Python implementations. The three Julia codes were designed for single-CPU, GPU-enhanced single CPU, and parallelized multi-core CPU architectures. Julia-MPI speeds were identical to Fortran-MPI at low core counts, 2x faster for mid-range, and 2x slower at higher core counts. Julia-MPI exhibited better scaling than Fortran-MPI for computation-only times, and more variability for communication times.

The most surprising result of this study was the speed of computations on the GPUs—a speed-up of 40,000 to over 100,000 times compared to the CPU. Of course, this comes with the caveat that memory transfer between CPU and GPU can take thousands of times longer than the computation, up to 0.5s at our highest resolution. So the key is to transfer memory to and from the GPU as little as possible, which is a well-known practice. If one can fit the full resolution of a computational physics domain within the memory of a single graphics card and sample results rarely, GPUs offer extraordinary speed-ups. For climate models, a single low-resolution component may well fit into GPU memory if the developers are careful with their memory footprint. The difficulty is that including ocean, atmosphere, land, and sea ice components requires the use of multiple nodes, and inter-node communication will keep the model slow, regardless of the GPU speed. Higher-resolution domains will need many nodes for each component and present the same problem.

The shallow water equations are simple enough for rapid development and verification, yet contain the salient features of any ocean model: intensive computation of the tendency terms, a time-stepping routine, and for the parallel version, interleaved halo communication of the partition boundary. Indeed, this layout, and the lessons learned here, apply to almost all computational physics codes.

This work specifically tests unstructured horizontal meshes, as opposed to structured quadralateral grids. Unstructured meshes refer to a neighbor's index using additional pointer arrays, so require an extra memory access for horizontal stencils. In structured grids, the physical neighbors are also neighbors in array space ($i + 1, j + 1$, etc), which leads to more contiguous memory access patterns that are easier for compilers to optimize. Our results show that unstructured meshes do not present any significant challenge in either Fortran or Julia. The use of a structured vertical index in the inner-most position and testing with 100 layers provides sufficient contiguous memory access for cache locality.

In the end, we were impressed by our experience with Julia. It did fulfill the promise of fast and convenient prototyping, with the ability to eventually run at high speeds on multiple high performance architectures—after some effort and lessons learned by the developers. The Julia libraries for MPI and CUDA were powerful and convenient. E3SM does not have plans to develop model components with Julia, but this study provides a useful comparison to our C++ and Fortran codes as we move towards heterogeneous, exascale computers.

Open Research

Three code repositories were used for the performance comparisons in this study. These are publicly available on both GitHub and Zenodo:

1. Julia Shallow Water code for serial CPU, CUDA-GPU, and MPI-parallelized CPU

GitHub: https://github.com/robertstrauss/MPAS_Ocean_Julia
Zenodo: https://doi.org/10.5281/zenodo.7493065

2. Python Rotating Shallow Water Verification Suite

GitHub: https://github.com/siddharthabishnu/Rotating_Shallow_Water_Verification_Suite.git. This study used the specific code version https://github.com/siddharthabishnu/Rotating_Shallow_Water_Verification_Suite/tree/v1.0.1
Zenodo: https://doi.org/10.5281/zenodo.7425628

3. Fortran-MPI MPAS Shallow Water code with Coastal Kelvin wave initial condition (Petersen et al., 2022)

GitHub: https://github.com/MPAS-Dev/MPAS-Model. This study used the specific code version https://github.com/mark-petersen/MPAS-Model/releases/tag/SW_julia_comparison_V1.0.
Zenodo: https://doi.org/10.5281/zenodo.7439134

The planar hexagonal MPAS-Ocean meshes used in this study for the numerical simulations and convergence tests of the coastal Kelvin wave and the inertia-gravity wave can be obtained from the Zenodo release of the Python Rotating Shallow Water Verification Suite Meshes at https://doi.org/10.5281/zenodo.7419817.

Acknowledgments

RRS gratefully acknowledges the support of the U.S. Department of Energy (DOE) through the Los Alamos National Laboratory (LANL) LDRD Program and the Center for Nonlinear Studies for this work. SB was supported by Scientific Discovery through Advanced Computing (SciDAC) projects LEAP (Launching an Exascale ACME Prototype) and CANGA (Coupling Approaches for Next Generation Architectures) under the DOE Office of Science, Office of Biological and Environmental Research (BER). MRP was supported by the Energy Exascale Earth System Model (E3SM) project, also funded by the DOE BER.

This research used computational resources provided by: the Darwin testbed at LANL, which is funded by the Computational Systems and Software Environments subprogram of LANL's Advanced Simulation and Computing program (NNSA/DOE); the LANL Institutional Computing Program, which is supported by the DOE National Nuclear Security Administration under Contract No. 89233218CNA000001; and the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the DOE under Contract No. DE-AC02-05CH11231.

References

Bishnu, S. (2021). *Time-Stepping Methods for Partial Differential Equations and Ocean Models* (Doctoral dissertation, Florida State University). doi: 10.5281/zenodo.7439539

Bishnu, S. (2022, December). *Rotating shallow water verification suite.* Zenodo. Retrieved from https://doi.org/10.5281/zenodo.7425628 doi: 10.5281/zenodo.7425628

Bishnu, S., Petersen, M., Quaife, B., & Schoonover, J. (2022, dec). *Verification suite of test cases for the barotropic solver of ocean models.* Authorea. Retrieved from https://doi.org/10.22541/essoar.167100170.03833124/v1 doi: 10.22541/essoar.167100170.03833124/v1

Bleichrodt, F., Bisseling, R. H., & Dijkstra, H. A. (2012, January). Accelerating a barotropic ocean model using a GPU. *Ocean Modelling*, *41*, 16–21.

Retrieved 2022-11-29, from https://www.sciencedirect.com/science/article/pii/S1463500311001661 doi: 10.1016/j.ocemod.2011.10.001

Caldwell, P. M., Mametjanov, A., Tang, Q., Van Roekel, L. P., Golaz, J. C., Lin, W., ... Zhou, T. (2019). The DOE E3SM coupled model version 1: Description and results at high resolution. *Journal of Advances in Modeling Earth Systems*, *11*(12), 4095–4146. doi: 10.1029/2019MS001870

Cushman-Roisin, B., & Beckers, J.-M. (2011). *Introduction to geophysical fluid dynamics: physical and numerical aspects.* Academic press.

Dalcín, L., Paz, R., & Storti, M. (2005). Mpi for python. *Journal of Parallel and Distributed Computing*, *65*(9), 1108–1115.

Dalcín, L., Paz, R., Storti, M., & D'Elía, J. (2008). Mpi for python: Performance improvements and mpi-2 extensions. *Journal of Parallel and Distributed Computing*, *68*(5), 655–662.

Eval of Julia code. (2016). Retrieved 2022-10-10, from https://docs.julialang.org/en/v1/devdocs/eval/#

Gevorkyan, M. N., Demidova, A. V., Korolkova, A. V., & Kulyabov, D. S. (2019, April). Statistically significant performance testing of Julia scientific programming language. *Journal of Physics: Conference Series*, *1205*, 012017. Retrieved from https://iopscience.iop.org/article/10.1088/1742-6596/1205/1/012017 doi: 10.1088/1742-6596/1205/1/012017

Golaz, J.-C., Caldwell, P. M., Van Roekel, L. P., Petersen, M. R., Tang, Q., Wolfe, J. D., ... Zhu, Q. (2019). The DOE E3SM Coupled Model Version 1: Overview and Evaluation at Standard Resolution. *Journal of Advances in Modeling Earth Systems*, *11*(7), 2089–2129. doi: 10.1029/2018MS001603

Introduction to CUDA. (2022). Retrieved 2022-12-13, from https://cuda.juliagpu.org/stable/tutorials/introduction/#A-simple-example-on-the-CPU

Jiang, J., Lin, P., Wang, J., Liu, H., Chi, X., Hao, H., ... Zhang, L. (2019). Porting LASG/ IAP Climate System Ocean Model to Gpus Using OpenAcc. *IEEE Access*, *7*, 154490–154501. (Conference Name: IEEE Access) doi: 10.1109/ACCESS.2019.2932443

Klöckner, A., Pinto, N., Lee, Y., Catanzaro, B., Ivanov, P., & Fasih, A. (2012). PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation. *Parallel Computing*, *38*(3), 157–174. doi: 10.1016/j.parco.2011.09.001

Klöwer, M., Hatfield, S., Croci, M., Düben, P. D., & Palmer, T. N. (2022). Fluid simulations accelerated with 16 bits: Approaching 4x speedup on a64fx by squeezing shallowwaters.jl into float16. *Journal of Advances in Modeling Earth Systems*, *14*(2), e2021MS002684. Retrieved from https://agupubs.onlinelibrary.wiley.com/doi/abs/10.1029/2021MS002684 (e2021MS002684 2021MS002684) doi: https://doi.org/10.1029/2021MS002684

Lam, S. K., Pitrou, A., & Seibert, S. (2015). Numba: A llvm-based python jit compiler. In *Proceedings of the second workshop on the llvm compiler infrastructure in hpc* (pp. 1–6).

Lin, W.-C., & McIntosh-Smith, S. (2021, November). Comparing Julia to Performance Portable Parallel Programming Models for HPC. In *2021 International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)* (pp. 94–105). St. Louis, MO, USA: IEEE. Retrieved from https://ieeexplore.ieee.org/document/9652798/ doi: 10.1109/PMBS54543.2021.00016

Mielikainen, J., Huang, B., Huang, H.-L. A., & Goldberg, M. D. (2012, August). Improved GPU/CUDA Based Parallel Weather and Research Forecast (WRF) Single Moment 5-Class (WSM5) Cloud Microphysics. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, *5*(4), 1256–1265. (Conference Name: IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing) doi: 10.1109/JSTARS.2012.2188780

Perkel, J. M. (2019, August). Julia: come for the syntax, stay for the speed. *Nature*, *572*(7767), 141–142. Retrieved from http://www.nature.com/articles/d41586-019-02310-3 doi: 10.1038/d41586-019-02310-3

Petersen, M. R., Asay-Davis, X. S., Berres, A. S., Chen, Q., Feige, N., Hoffman, M. J., ... Woodring, J. L. (2019). An Evaluation of the Ocean and Sea Ice Climate of E3SM Using MPAS and Interannual CORE-II Forcing. *Journal of Advances in Modeling Earth Systems*, *11*(5), 1438–1458. doi: 10.1029/2018MS001373

Petersen, M. R., Bishnu, S., & Strauss, R. R. (2022, December). *Mpas-ocean shallow water performance test case.* Zenodo. Retrieved from https://doi.org/10.5281/zenodo.7439134 doi: 10.5281/zenodo.7439134

Petersen, M. R., Jacobsen, D. W., Ringler, T. D., Hecht, M. W., & Maltrud, M. E. (2015, February). Evaluation of the arbitrary Lagrangian–Eulerian vertical coordinate method in the MPAS-Ocean model. *Ocean Modelling*, *86*, 93–113. doi: 10.1016/j.ocemod.2014.12.004

Ramadhan, A., Wagner, G. L., Hill, C., Campin, J.-M., Churavy, V., Besard, T., ... Marshall, J. (2020). Oceananigans.jl: Fast and friendly geophysical fluid dynamics on gpus. *Journal of Open Source Software*, *5*(53), 2018. Retrieved from https://doi.org/10.21105/joss.02018 doi: 10.21105/joss.02018

Ringler, T. D., Petersen, M. R., Higdon, R. L., Jacobsen, D., Jones, P. W., & Maltrud, M. (2013). A multi-resolution approach to global ocean modeling. *Ocean Modelling*, *69*, 211–232.

Ringler, T. D., Thuburn, J., Klemp, J. B., & Skamarock, W. C. (2010). A unified approach to energy conservation and potential vorticity dynamics for arbitrarily-structured C-grids. *Journal of Computational Physics*, *229*(9), 3065–3090.

Shchepetkin, A. F., & McWilliams, J. C. (2005). The regional oceanic modeling system (ROMS): a split-explicit, free-surface, topography-following-coordinate oceanic model. *Ocean modelling*, *9*(4), 347–404.

Srinath, A. (2020, Nov). *Accelerating python on gpus with nvc++ and cython.* Retrieved from https://developer.nvidia.com/blog/accelerating-python-on-gpus-with-nvc-and-cython/

Strauss, R. R. (2023, January). *Julia Layered Shallow Water Model on Various Hardwares.* Retrieved from https://doi.org/10.5281/zenodo.7493065 doi: 10.5281/zenodo.7493065

Thuburn, J., Ringler, T. D., Skamarock, W. C., & Klemp, J. B. (2009). Numerical representation of geostrophic modes on arbitrarily structured C-grids. *Journal of Computational Physics*, *228*(22), 8321–8335.

Trott, C. R., Lebrun-Grandié, D., Arndt, D., Ciesko, J., Dang, V., Ellingwood, N., ... Wilke, J. (2022). Kokkos 3: Programming model extensions for the exascale era. *IEEE Transactions on Parallel and Distributed Systems*, *33*(4), 805-817. doi: 10.1109/TPDS.2021.3097283

Xu, S., Huang, X., Oey, L.-Y., Xu, F., Fu, H., Zhang, Y., & Yang, G. (2015, September). POM.gpu-v1.0: a GPU-based Princeton Ocean Model. *Geoscientific Model Development*, *8*(9), 2815–2827. Retrieved 2022-11-29, from https://gmd.copernicus.org/articles/8/2815/2015/ (Publisher: Copernicus GmbH) doi: 10.5194/gmd-8-2815-2015

Xu, S., Huang, X., Zhang, Y., Hu, Y., & Yang, G. (2014, June). A customized GPU acceleration of the princeton ocean model. In *2014 IEEE 25th International Conference on Application-Specific Systems, Architectures and Processors* (pp. 192–193). (ISSN: 2160-052X) doi: 10.1109/ASAP.2014.6868661

Ye, Y., Song, Z., Zhou, S., Liu, Y., Shu, Q., Wang, B., ... Wang, L. (2022, July). swNEMO_v4.0: an ocean model based on NEMO4 for the new-generation Sunway supercomputer. *Geoscientific Model Development*, *15*(14), 5739–5756. Retrieved 2022-11-30, from https://gmd.copernicus.org/articles/15/5739/2022/ (Publisher: Copernicus GmbH) doi: 10.5194/gmd-15-5739-2022

Zhao, X.-d., Liang, S.-x., Sun, Z.-c., Zhao, X.-z., Sun, J.-w., & Liu, Z.-b. (2017, August). A GPU accelerated finite volume coastal ocean model. *Journal of Hydrodynamics, Ser. B*, *29*(4), 679–690. Retrieved 2022-11-29, from https://www.sciencedirect.com/science/article/pii/S1001605816607801 doi: 10.1016/S1001-6058(16)60780-1

Julia for Geophysical Fluid Dynamics: Performance Comparisons between CPU, GPU, and Fortran-MPI

Robert R. Strauss[1], **Siddhartha Bishnu**[2], **Mark R. Petersen**[2]

[1]Center for Nonlinear Studies, Los Alamos National Laboratory, NM, 87545, USA
[2]Computational Physics and Methods Group, Los Alamos National Laboratory, NM, 87545, USA

Key Points:

- Unstructured-mesh shallow water models were created in Julia for single-core CPU, single-node GPU, and multi-core CPU clusters using MPI.
- Julia-MPI performance ranges from 2x faster to 2x slower than Fortran-MPI. Julia on GPUs is significantly faster than on CPUs.
- Julia development time is quick for prototyping, but requires more time to develop performant code; specifically, static typing is required.

Corresponding author: Mark R. Petersen, `mpetersen@lanl.gov`

Abstract

Some programming languages are easy to develop at the cost of slow execution, while others are lightning fast at run time but are much more difficult to write. Julia is a programming language that aims to be the best of both worlds—a development and production language at the same time. To test Julia's utility in scientific high-performance computing (HPC), we built an unstructured-mesh shallow water model in Julia and compared it against an established Fortran-MPI ocean model, MPAS-Ocean, as well as a Python shallow water code. Three versions of the Julia shallow water code were created, for: single-core CPU; graphics processing unit (GPU); and Message Passing Interface (MPI) CPU clusters. Comparing identical simulations revealed that our first version of the single-core CPU Julia model was 13 times faster than Python. Further Julia optimizations, including static typing and removing implicit memory allocations, provided an additional 10–20x speed-up of the single-core CPU Julia model. The GPU-accelerated Julia code is extremely fast, with a speed-up of 230-380x compared to the single-core CPU Julia code if communication with the GPU occurs every 10 time steps. Parallelized Julia-MPI performance was identical to Fortran-MPI MPAS-Ocean for low processor counts, and ranges from 2x faster to 2x slower for higher processor counts. Our experience is that Julia development is fast and convenient for prototyping, but that Julia requires further investment and expertise to be competitive with compiled codes. We provide advice on Julia code optimization for HPC systems.

Plain Language Summary

Scientists who write programs for supercomputers try to satisfy two requirements: the code should be both fast and easy to understand. These requirements are often in conflict, because fast programs use special libraries that add extra lines to the code and make it less readable. Supercomputers also change over time—for several decades, they had thousands of identical CPUs (each similar to a desktop), but in the past decade they include CPUs accelerated by graphics processing units (GPUs). This added hardware complexity results in more complex software. Here we test a relatively new programming language, Julia, which is designed to be simpler to write, but also to be fast on advanced computer architectures. We find that Julia is both convenient and fast, but there is no free lunch. Our first attempt to develop an ocean model in Julia was relatively easy, but the code was slow—it was 70 times slower than a long-standing ocean model written in Fortran. After several months of further development and experimentation, we did indeed create a Julia code that is as fast on supercomputers as the Fortran ocean model.

1 Introduction

A major concern in computer modeling is the trade-off between execution speed and code development time. In general, programs in scripting languages like Python and Matlab are faster to develop due to their simpler syntax and more relaxed typing requirements, but are limited by slower execution time. On the other end of the spectrum, we have compiled languages like C/C++ and Fortran, which have been extensively used in scientific computing for many decades. Programs in such languages are blessed with faster execution time, but are cursed with stricter and more cumbersome syntax, leading to slower development time. The Julia language strikes a balance between these two categories (Perkel, 2019). It is a compiled language with execution speed similar to C/C++ or Fortran, if carefully written with strict syntax (Lin & McIntosh-Smith, 2021; Gevorkyan et al., 2019). It is also equipped with a more convenient syntax and features, such as dynamic typing, to accelerate code development in prototyping. To this day, the majority of scientific computing models are programmed in compiled languages, which execute fast but can take months, if not years, to develop. In this paper, we investigate the feasability of writing Julia codes for computational physics simulations, since a Julia program can not only ensure high performance but also less development time in the initial stages. We develop a shallow water solver in Julia and compare its performance to an equivalent Fortran code.

An additional complication in choosing the best language is that layers of libraries have been added to C/C++ and Fortran to accommodate evolving computer architectures. For the past 25 years, computational physics codes have largely used the Message Passing Interface (MPI) to communicate between CPUs on separate nodes that do not share memory, and OpenMP to parallelize within a node using shared-memory threads. With the advent of heterogeneous nodes containing both CPUs and GPUs, scientific programmers have several new choices: writing kernels directly for GPUs in CUDA (Bleichrodt et al., 2012; Zhao et al., 2017; Xu et al., 2015); adding OpenACC pragmas for the GPUs (Jiang et al., 2019); or calling libraries such as Kokkos (Trott et al., 2022) that execute code optimized for specialized architectures on the back-end, while providing a simpler front-end interface for the domain scientist. All of these require additional expertise, and add to the length and complexity of the code base. Julia also provides an MPI library for parallelization across nodes in a cluster, and a CUDA library to parallelize over GPUs within a node. We have written shallow water codes in Julia that adopt each of these parallelization strategies.

In recent years, shallow water solvers such as Oceananigans.jl (Ramadhan et al., 2020) and ShallowWaters.jl (Klöwer et al., 2022) have been developed in Julia. These codes employ structured rectilinear meshes to discretize their domain, and are equipped with capabilities for running on GPUs to achieve high performance. Here we conduct a comparison on unstructured-mesh models, using the Fortran code MPAS-Ocean (Ringler et al., 2013) as a point of reference. MPAS-Ocean employs unstructured near-hexagonal meshes with variable resolution capability and is parallelized with MPI for running on supercomputer clusters. We developed a Julia model employing the same spatial discretization of MPAS-Ocean, and capable of running in serial mode on a single core, or in parallel mode on a supercomputer cluster or a graphics card. We discuss the subtle details of our implementations, compare the speed-ups attained, and describe the strategies employed to enhance performance.

2 Methods

2.1 Equation Set & TRiSK-Based Spatial Discretization

Our Julia model solves the shallow water equations (Cushman-Roisin & Beckers, 2011) in vector-invariant form. This is sufficiently close to the governing equations for ocean and atmospheric models to be used as a proxy to test performance with new codes and architectures. The equation set is

$$\boldsymbol{u}_t + qh\boldsymbol{u}^{\perp} = -g\nabla\eta - \nabla K, \tag{1a}$$

$$\eta_t + \nabla\cdot(h\boldsymbol{u}) = 0, \tag{1b}$$

where $\boldsymbol{u}$ is the horizontal velocity vector, $\boldsymbol{u}^{\perp} = \boldsymbol{k}\times\boldsymbol{u}$, h is the layer thickness, η is the surface elevation or sea surface height (SSH), $K = |\boldsymbol{u}|^2/2$ is the kinetic energy, and g is the acceleration due to gravity. If b represents the topographic height and H the mean depth, then $\eta = h + b - H$. Moreover, if f denotes the Coriolis parameter, and $\zeta = \boldsymbol{k}\cdot\nabla\times\boldsymbol{u}$ the relative vorticity, then the absolute vorticity, $\omega_a = \zeta + f$, and the potential vorticity, $q = \omega_a/h$. The term $qh\boldsymbol{u}^{\perp}$ is the thickness flux of the PV in the direction perpendicular to the velocity, rotated counterclockwise on the horizontal plane. Ringler et al. (2010) refer to it as the non-linear Coriolis force since it consists of the quasi-linear Coriolis force $f\boldsymbol{u}^{\perp}$ and the rotational part $\zeta\boldsymbol{u}^{\perp}$ of the non-linear advection term $\boldsymbol{u}\cdot\nabla\boldsymbol{u}$. We spatially discretize the prognostic equations in (1) using a mimetic finite volume method based on the TRiSK scheme, which was first proposed by (Thuburn et al., 2009), and then generalized by (Ringler et al., 2010). This method was chosen to horizontally discretize the primitive equations of MPAS-Ocean while invoking the hydrostatic, incompressible, and Boussinesq approximations on a staggered C-grid. Since this horizontal discretization guarantees conservation of mass, potential vorticity, and energy, it makes MPAS-Ocean a suitable candidate to simulate mesoscale eddies.

Our spatial domain is tessellated by two meshes, a regular planar hexagonal primal mesh and a regular triangular dual mesh. Each corner of the primal mesh cell coincides with a vertex of the dual mesh cell and vice versa. A line segment connecting two primal mesh cell centers is the perpendicular bisector of a line segment connecting two dual mesh cell centers and vice versa. Regarding our prognostic variables, the scalar SSH η is defined at the primal cell centers, and the normal velocity vector $\boldsymbol{u}_e$ is defined at the primal cell edges. The divergence of a two-dimensional vector quantity is defined at the positions of η, while the two-dimensional gradient of a scalar quantity is defined at the positions of $\boldsymbol{u}_e$ and oriented along its direction. The curl of a vector quantity is defined at the vertices of the primal cells. Finally, the tangential velocity $\boldsymbol{u}_e^\perp$ along a primal cell edge is computed diagnostically using a flux mapping operator from the primal to the dual mesh, which essentially takes a weighted average of the normal velocities on the edges of the cells sharing that edge. Interested readers may refer to Thuburn et al. (2009) and Ringler et al. (2010) for further details on the mesh specifications.

At each edge location $\boldsymbol{x}_e$, two unit vectors $\boldsymbol{n}_e$ and $\boldsymbol{t}_e$ are defined parallel to the line connecting the primal mesh cells, and in the perpendicular direction rotated counterclockwise on the horizontal plane, such that $\boldsymbol{t}_e = \boldsymbol{k} \times \boldsymbol{n}_e$. The discrete equivalent of the set of equations (1) is

$$(u_e)_t = F_e^\perp \widehat{q}_e - \left[\nabla (g\eta)_i + K_i\right]_e , \tag{2a}$$

$$(\eta_i)_t = - \left[\nabla \cdot F_e\right]_i , \tag{2b}$$

where $F_e = \widehat{h}_e u_e$ and $F_e^\perp$ represent the thickness fluxes per unit length in the $\boldsymbol{n}_e$ and $\boldsymbol{t}_e$ directions respectively. The layer thickness h_i, the SSH η_i, the topographic height b_i, and the kinetic energy K_i are defined at the centers $\boldsymbol{x}_i$ of the primary mesh cells, while the velocity u_e are defined at the edge points $\boldsymbol{x}_e$. The symbol $\widehat{(.)}_e$ represents an averaging of a field from its native location to $\boldsymbol{x}_e$. The discrete momentum equation (2b) is obtained by taking the dot product of (1b) with $\boldsymbol{n}_e$, which modifies the non-linear Coriolis term to

$$\begin{aligned} \boldsymbol{n}_e \cdot \widehat{q}_e \widehat{h}_e \boldsymbol{u}^\perp &= \widehat{q}_e \widehat{h}_e \boldsymbol{n}_e \cdot (\boldsymbol{k} \times \boldsymbol{u}) = \widehat{q}_e \widehat{h}_e \boldsymbol{u} \cdot (\boldsymbol{n}_e \times \boldsymbol{k}) \\ &= -\widehat{q}_e \widehat{h}_e \boldsymbol{u} \cdot \boldsymbol{t}_e = -\widehat{q}_e \widehat{h}_e u_e^\perp = -F_e^\perp \widehat{q}_e . \end{aligned} \tag{3}$$

Given the numerical solution at time level $t^n = n\Delta t$, with Δt representing the time step and $n \in \mathbb{Z}_{\geq 0}$, the Julia model first computes the time derivative or tendency terms of (2) as functions of the discrete spatial and flux-mapping operators of the TRiSK scheme. Then it advances the numerical solution to time level t^{n+1} using the forward-backward method

$$u^{n+1} = u^n + \Delta t \mathcal{F} \left(u^n, h^n\right) , \tag{4}$$

$$h^{n+1} = h^n + \Delta t \mathcal{G} \left(u^{n+1}, h^n\right) , \tag{5}$$

where $\mathcal{F}$ and $\mathcal{G}$ represent the discrete tendencies of the normal velocity and the layer thickness in functional form, and the subscripts representing the positions of these variables have been dropped for notational simplicity.

The following sections introduce the new codes that were created for this study. Three versions of the Julia code were written (Strauss, 2023): the base single-core CPU version, an altered version for GPUs with CUDA, and a multi-node CPU implementation with Julia-MPI. These were compared against existing Fortran-MPI and Python versions of shallow-water TRiSK models. All use a standard MPAS unstructured-mesh file format that specifies the geometry and topology of the mesh, and includes index variables that relate neighboring cells, edges, and vertices. All models have an inner (fastest-moving) index for the vertical coordinate and were tested with 100 vertical layers to mimic performance in a realistic ocean model.

2.2 Single-Core CPU Julia Implementation

The serial-mode implementation on a single core involves looping over every cell and edge of the mesh to (a) compute the tendencies, i.e. the right-hand side terms of the prognostic equations (2) and (b) advance their values to the next time step. The tendencies can be functions of the dependent and independent variables as well as spatial derivatives of the dependent variable. The serial version of our model is the simplest one from the perspective of transforming the numerical algorithms into code.

In order to highlight differences in formulation, we provide a Julia code example for the single tendency term from (2) for the SSH gradient $-g\nabla\eta$, which is discretized as $-\left[g\nabla\eta_i\right]_e$. We then add a vertical index k to mimic the performance of a multi-layer ocean model, but each layer is trivially redundant. In a full ocean model this term would be the pressure gradient, and would involve the computation of pressure as a function of depth and density. For the single-core CPU version, the Julia function computing the SSH gradient is

Listing 1: Julia example for serial CPU

```julia
velocity_tendencies!(sshGradient, ssh, ...)

function velocity_tendencies!(sshGradient, ssh, ...)
    for iEdge in 1:nEdges
        cell1 = cellsOnEdge[1,iEdge]
        cell2 = cellsOnEdge[2,iEdge]
        for k in 1:nVertLevels
            sshGradient[k,iEdge] = - gravity / dcEdge[iEdge]
                * ( ssh[k,cell2] - ssh[k,cell1] )
        end
    end
end
```

Here `cellsOnEdge` is an array of index variables describing the mesh that points to the cells neighboring an edge, and `dcEdge` represents the distance between the centers of adjacent cells sharing the edge on which the normal velocity tendency is computed. In the actual code all the tendency terms are computed within this function, but here we only show the ssh gradient as a brief sample.

2.3 SIMD GPU Julia Implementation

GPUs are very powerful tools for SIMD (Same Instruction Multiple Data) computations: they have thousands of independent threads, which can execute the same operation at the same time with different input values. Since we numerically solve the same prognostic equation for (a) the SSH at every cell center $\boldsymbol{x}_i$, and (b) the normal velocity at every edge $\boldsymbol{x}_e$ of the mesh, a GPU is a logical tool to employ for our computations. By placing subsets of cells and edges on different threads of the GPU, we can perform the tendency computations, and advance the prognostic variables at once in parallel rather than looping over every cell and edge, which would scale in wall-clock time according to the size of the mesh.

We wrote CUDA kernels for an Nvidia GPU using the CUDA.jl library for computing the tendencies and advancing the prognostic variables to the next time step. The code for the single-core implementation can be converted to CUDA with surprising ease by removing the `for` loop over the cells and edges, and instead performing the underlying computation on a single cell or edge:

Listing 2: Julia example for GPU with CUDA

```julia
CUDA.@cuda blocks=cld(nEdges, 1024) threads=1024 maxregs=64
    velocity_tendencies_cuda!(sshGradient, ssh, ...)

```

```julia
function velocity_tendencies_cuda!(sshGradient, ssh, ...)
    iEdge = (CUDA.blockIdx().x - 1) * CUDA.blockDim().x
        + CUDA.threadIdx().x
    cell1 = cellsOnEdge[1,iEdge]
    cell2 = cellsOnEdge[2,iEdge]
    for k in 1:nVertLevels
        sshGradient[k,iEdge] = - gravity / dcEdge[iEdge]
            * ( ssh[k,cell2] - ssh[k,cell1] )
    end
end
```

Each cell and edge of the mesh will be designated to a different thread on the GPU. The computation for a single cell or edge will run on a single thread, and a CUDA method will be used to map the index of the thread to the index of the cell (i) or edge (e), at which the prognostic variable is being updated. To execute this method over all threads on the GPU, we use a CUDA macro to call our kernel and designate the number of threads to use, which should be equal to the number of cells or edges in the mesh. Note that the inner computation of `pressureGradient` is identical for the CPU and CUDA kernal codes.

2.4 CPU/MPI Julia Implementation

Rather than iterating through every cell or edge of the mesh, we may parallelize the simulation with multiple processors by assigning to each processor a portion of the mesh, a process called domain decomposition. However, the computations of some spatial operators may require information from the outermost cells of the adjacent processors. So, we need the neighboring processors to communicate these pieces of information with each other. To ensure an efficient communication, we include an extra ring or "halo" of cells around the boundary of the region assigned to each processor, which overlaps with the region assigned to adjacent processors. We do not compute the tendencies of the prognostic variables in the halo region of a processor. In fact, we cannot perform this operation without information in an additional ring of halo cells, which is not assigned to the processor under consideration. So, we obtain the updated values of the prognostic variables in the halo region by communication with adjacent processors, which contain these halo cells in their interior, and update the prognostic variables in them.

A number of crucial modifications are necessary to implement this parallelization scheme. For instance, the simulation methods are amended so that each process (rank) only performs computations for the set of cells or edges assigned to it. We use the MPI communication channel (comm) to receive the updated values of the prognostic variables in the halo region of a processor from adjacent processors which advance these variables. Similarly, we send the updated values of the prognostic variables along the outermost region of the above-mentioned processor to adjacent processors, for which these variables belong in the halo regions. For the TRiSK-based spatial discretization and the forward-backward time-stepping method, the halo region consists of only one layer (one halo ring) of cells.

Listing 3: Julia example for CPU with MPI

```julia
# each process executes the following, receiving a different value
# on each rank:
comm = MPI.COMM_WORLD
rank = MPI.Comm_rank(comm)

myCells = cells_for_rank(mesh_file, rank, partition_file)
myEdges, myHaloEdges = edges_on_cells(myCells)

velocity_tendencies!(myEdges, sshGradient, ssh, ...)
update_halo_edges!(sshGradient, myHalodEdges, rank, comm)
```

```
function velocity_tendencies!(myEdges, sshGradient, ssh, ...)
    for iEdge in myEdges
        cell1 = cellsOnEdge[1,iEdge]
        cell2 = cellsOnEdge[2,iEdge]
        for k in 1:nVertLevels
            sshGradient[k,iEdge] = - gravity / dcEdge[iEdge]
                * ( ssh[k,cell2] - ssh[k,cell1] )
        end
    end
end

function update_halo_edges!(data, edgesInMyHalo, rank, comm)
    for neighborRank in find_neighbors(rank, comm)
        MPI.Irecv!(data[edgesInMyHalo,:], neighborRank, 0, comm)
        edgesToNeighbor = find_halo_overlap(rank, neighbor, comm)
        MPI.Isend(data[edgesToNeighbor,:], neighborRank, 0, comm)
    end
end
```

Here `myCells` and `myEdges` are the lists of cells and edges in the local domain, owned by the rank running this code, plus its halo.

2.5 CPU/MPI Fortran Implementation

The baseline comparison code for this study is the Model for Prediction Across Scales (MPAS-Ocean) (Ringler et al., 2013; Petersen et al., 2015), which is written in Fortran with MPI communication commands. It is the ocean component of the Energy Exascale Earth System Model (E3SM) (Golaz et al., 2019; Petersen et al., 2019), the climate model developed by the US Department of Energy. In this study, the code is reduced from a full ocean model solving the primitive equations to simply solving for velocity and thickness (1). Thus the majority of the code is disabled, including the tracer equation, vertical advection and diffusion, the equation of state, and all parameterizations. In order to match the Julia simulations, we employ a forward-backward time-stepping scheme, exchange one-cell-wide halos after each time step, compute 100 layers in the vertical array dimension, and use the identical Cartesian hexagon-mesh domains (Petersen et al., 2022).

MPAS-Ocean is an excellent comparison case for Julia because it is a well-developed code base that uses Fortran and MPI, which have been standard for computational physics codes since the late 1990s. The highest resolution simulations in past studies used over three million horizontal mesh cells and 80 vertical layers, scale well to tens of thousands of processors (Ringler et al., 2013) and have been used for detailed climate simulations (Caldwell et al., 2019). MPAS-Ocean includes OpenMP for within-node memory access, and is currently adding OpenACC for GPU computations, but these were not used for this comparison to Julia-MPI on a CPU cluster.

2.6 CPU Python Implementation

In addition to MPAS-Ocean, we compare the performance of the Julia shallow water code against an object-oriented Python code Bishnu (2022). The Python code solves the rotating shallow water system of equations using two types of spatial discretizations: the TRiSK-based mimetic finite volume method used in MPAS-Ocean, and a discontinuous Galerkin Spectral Element Method (DGSEM). The code offers a number of standard predictor-corrector and multistep time-stepping methods, including those analyzed for ocean modeling in Shchepetkin and McWilliams (2005).

The Julia shallow water code was first written by translating this Python code into Julia syntax. While the Julia code was expanded for parallelization and performance, the Python code was further developed to serve as a platform for conducting a verification suite of shallow water test cases for the barotropic solver of ocean models. Each of these test cases in the Python code verifies the implementation of a subset of terms in the prognostic momentum and continuity equations, e.g. the linear pressure gradient term, the linear constant or variable-coefficient Coriolis and bathymetry terms, and the non-linear advection terms. Bishnu et al. (2022) and Bishnu (2021) provide detailed discussions on these test cases along with specifics of the numerical implementation, the time evolution of the numerical error for both spatial discretizations and a subset of the time-stepping methods, and results of convergence studies with refinement in both space and time, only in space, and only in time. Out of all of these test cases, only the linear coastal Kelvin wave and inertia-gravity wave test cases were implemented in the Julia code for the current study.

While not used in this study, a number of libraries exist to accelerate Python for various architectures. These include Numba and PyCuda for GPUs, mpi4py for CPU clusters, and Cython for single-CPU acceleration. Numba (Lam et al., 2015) is an open-sourced Anaconda-sponsored NumPy-aware optimizing compiler, which translates Python functions to fast machine code at runtime using the remarkable industry-standard LLVM compiler library. PyCUDA (Klöckner et al., 2012), written in C++ (the base layer) and Python, provides access to Nvidia's CUDA parallel computation API from Python. Mpi4py (Dalcín et al., 2005, 2008), provides Python bindings for the Message Passing Interface (MPI) standard. As an alternative, one can 'cythonize' an existing Python code by providing static type declarations and class attributes, that can then be translated to C++/C code and to C-Extensions for Python. Cython is an optimising static compiler for both the Python programming language and the extended Cython programming language. It is designed to offer C-like performance with code mostly written in Python with additional C-inspired syntax. The rotating shallow water Python code Bishnu (2022) is currently undergoing cythonization. Cythonized codes can further be accelerated on GPUs using Nvidia's HPC C++ compiler, and the C++ Standard Parallelism (stdpar) for GPUs (Srinath, 2020). However, the extent of additional modifications and enhancements required to bring GPU-accelerated C++ algorithms to the Python ecosystem may not always be a reasonable investment of time. As we will see in later sections, a serial Julia code, which already achieves the performance of a fast compiled language, does not require extensive modifications to be parallelized on GPUs or multiple cores, and is therefore more convenient than python for high-performance scientific computing applications.

3 Results

3.1 Model Verification

Each serial and parallel implementation of the shallow water model described in the previous section was verified for accuracy with convergence tests against exact solutions. We obtained the expected second-order convergence of the various TRiSK-based spatial operators on a uniform planar hexagonal MPAS-Ocean mesh. The operators included the gradient, the divergence, the curl, and the flux-mapping operator used to interpolate the tangential velocities from the normal velocities (Figure 1). The formulation of these operators is shown in Figure 3 of Ringler et al. (2010). Once the operator tests were complete, the linearized shallow water equations were verified against exact solutions for the coastal Kelvin wave and inertia-gravity wave cases, as described in Bishnu et al. (2022) and Bishnu (2021). With refinement in both space and time, we observe the expected first-order convergence of the numerical solution (Figure 1), spatially discretized with the second-order TRiSK scheme, and advanced with the first-order forward-backward time-stepping method (Bishnu, 2021).

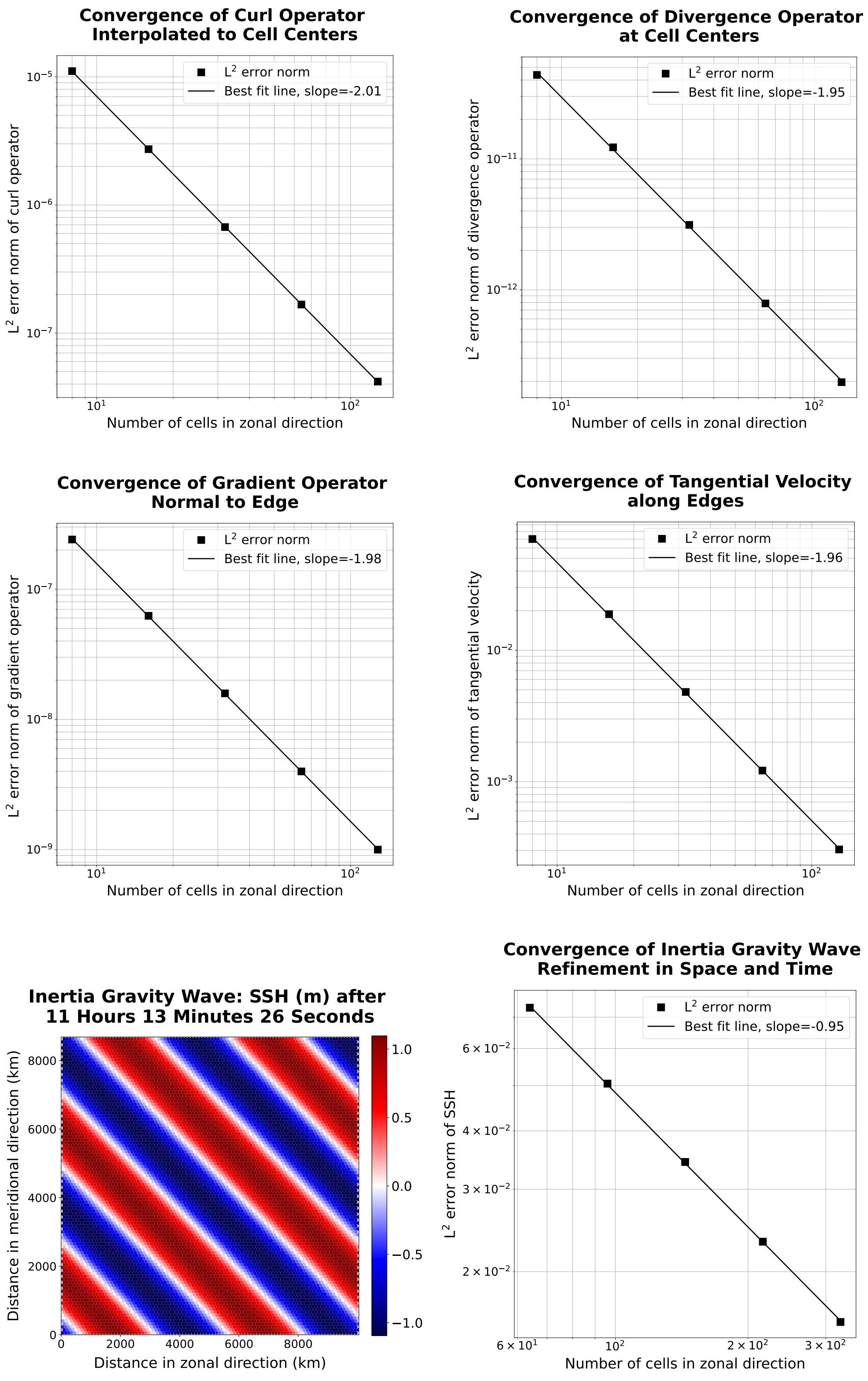


Figure 1: The first two rows show convergence plots of the TRiSK-based spatial operators for the newly-developed Julia code. Tests were run with both CPU and GPU implementations, and identical results were obtained. The slope of -2 indicates the expected second-order convergence. The third row shows a snapshot of the inertia-gravity wave test case, and the convergence plot of the numerical solution with refinement in both space and time.

3.2 Acceleration of Julia with GPU Hardware

The Julia serial CPU version of the shallow water model was compared against the Julia CUDA library GPU version and the reference Python CPU code (Table 1 and Figure 2). Tests were conducted on the Darwin cluster at Los Alamos National Laboratory, using a single node equipped with Intel Cascade Lake CPUs (Gold 6248 with a clock rate of 2.5 GHz and 27.5M Cache) and the Nvidia Quadro RTX 8000 "Turing" GPU architecture (4608 CUDA cores, 16.3 TFLOPS peak single precision performance, 48 GB GPU memory, and GPU memory bandwidth of 672 GB/s). All performance tests described in this and the following sections used the coastal Kelvin wave test case on a planar hexagon mesh with the linear shallow water equations and 100 vertical layers. Samples are averaged over ten trials. All codes use double-precision (8 byte) real numbers, and performance tests do not include the time for initialization, input/output, or generating plots.

In our first version of the Julia single-core CPU code, we did not take any special steps for code optimization, and it was already 13 times faster than Python. Julia and Python both have dynamic typing, but Julia has the ability to go much faster since it also supports concrete typing. Julia is compiled, but hides it cleverly by compiling on the fly based on what datatypes are provided at run time. It supports a hierarchical abstract typing system, allowing for semi-specified types, such as "Any", which all types extend and is the default if no type is specified (thus acting like python), or "AbstractArray", which can be occupied at run time with any Array-like data.

After the initial Julia development, further effort was put into optimization, which led to a 10–20 times speed-up for the CPU-serial code. The changes included optimizing for memory management by tracking down and reducing unnecessary allocations that contributed significantly to the run time, as well as making all types and subtypes concrete rather than abstract, to minimize on-the-fly compilation. These improvements are explained in more detail in section 4.

We found the CUDA GPU implementation to be *significantly* faster than the single-core implementation. Because the memory transfer between the CPU and GPU takes many orders of magnitude longer than the actual on-GPU computations, we split them out in Table 1 and Figure 2. The memory transfers require between 0.015s and 0.68s and scale with the array size, while the GPU computations alone are extraordinarily fast, at 0.00027s for the 512x512 resolution case, and do not scale with resolution. This shows the power of GPUs, where computations alone can run over 40,000 times faster on the GPU than the CPU, but this speed-up is substantially diminished by the memory transfer time. Still, codes that are designed with a small memory footprint and limited memory transfer can greatly benefit from GPU computations. Strategically reducing array precision to 4-byte or even 2-byte reals for certain variables allows higher-resolution domains to fit on GPUs (Ye et al., 2022; Klöwer et al., 2022). In addition, single-precision floating point numbers (CUDA `Float32` data type) calculations may execute significantly faster than `Float64` (*Introduction to CUDA*, 2022). We did not leverage `Float32` in this work, but it shows that GPU simulations could run even faster than the results shown here.

Summing the GPU memory transfer and compute for the 10 timestep performance test, the GPUs were 229 to 386 times faster than the single CPU (Table 2). This compares to published studies of ocean models that show a speed-up from CPU to GPU ranging from 5–50 (Bleichrodt et al., 2012; Zhao et al., 2017; Xu et al., 2014), and a speed-up of up to 1556x for a GPU/CUDA Based Parallel Weather and Research Forecast Model (WRF) (Mielikainen et al., 2012). Note that our speed-up factor could be increased substantially by transferring data from the GPU to CPU less frequently. For a low-resolution ocean model with 30-minute time steps, the speed-ups in Table 2 correspond to collecting data every 10 time-steps, which is 5 hours of model time. One could instead collect data for analysis every 100 time-steps (∼2 days), and that would result in a GPU speed-up of 2290 to 3860, because the compute time is negligible compared to the memory transfer. On the other hand, if model communication is required frequently for surface data forcing or coupling with atmospheric and sea ice components, the speed-up is drastically reduced. For example, if memory must be transferred between the CPU and GPU every time step, the speed-ups range from 23—39. The point is that GPU performance is wholly dependant on the GPU communication frequency.

	128x128	**256x256**	**512x512**
Python, CPU	3.08E+03	1.31E+04	4.96E+04
Julia, CPU-serial (unoptimized)	2.25E+02	8.64E+02	3.86E+03
Julia, CPU-serial (optimized)	1.12E+01	7.43E+01	3.33E+02
Julia, GPU, total	4.90E−02	2.03E−01	8.64E−01
transfer to GPU	2.98E−02	1.16E−01	4.58E−01
compute on GPU	2.51E−04	2.67E−04	2.67E−04
transfer back to CPU	1.53E−02	9.54E−02	6.84E−01

Table 1: Wall clock duration (seconds) of performing ten timesteps with 100 layers on an Intel Cascade Lake CPU or an NVidia Turing GPU.

	128x128	**256x256**	**512x512**
Python, CPU	274	177	149
Julia, CPU-serial (unoptimized)	20	12	12
Julia, CPU-serial (optimized)	1	1	1
Julia, GPU	**229**	**366**	**386**

Table 2: Speed-up (bold) or slow-down (non-bold) factor compared to the optimized CPU-serial Julia version at the same resolution. GPU speed-ups are based on transferring arrays between GPU and CPU every ten time steps.

GPU threads are grouped into threadblocks (or just "blocks") for efficiency. While calling the kernel function, we must specify the number of blocks and number of threads per block (the "block size"), as shown in listing 2. Within the kernel, we obtain the index of the block and thread, multiply the block index by the block size, and add the thread index to compute a global index. There is a maximum possible block size, but we can choose any smaller value to execute the kernel with. The block size does have an effect on how quickly the kernel runs, so we benchmarked the evaluation time of the same kernel run with different block sizes, as shown in Figure 3. Smaller block sizes run faster on the GPUs by 15%. This is interesting to note, but GPU compute time is so small compared to the memory transfer time that thread tuning has little impact on the overall simulation time.

3.3 Julia-MPI versus Fortran-MPI

Julia and Fortran codes were compared on multi-node CPU clusters, where both used MPI for communication between processors. Comparisons were made with domains of 128, 256, and 512-squared grid cells solving the shallow water equations. All timing tests were conducted for 10 time steps and repeated 12 times on each processor count, spanning 2 to 2048 processors by powers of two. The vertical dimension included 100 layers to mimic ocean model arrays and provide sufficient computational work on each processor. Separate timers report on computational work versus MPI communication within the time-stepping routine. The i/o, initialization, and finalization time is excluded.

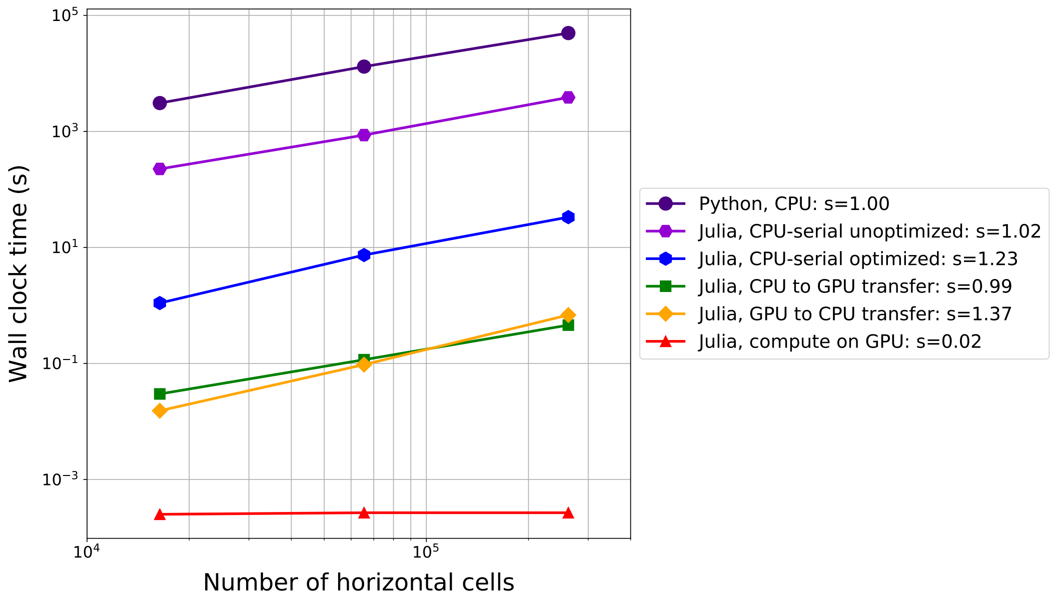


Figure 2: Timing data from Table 1, comparing ten timesteps of the Kelvin Wave test case on an Intel Cascade Lake CPU or an NVidia Turing GPU. The log-log slope, shown as s in the legend, is 1.0 for perfect scaling.

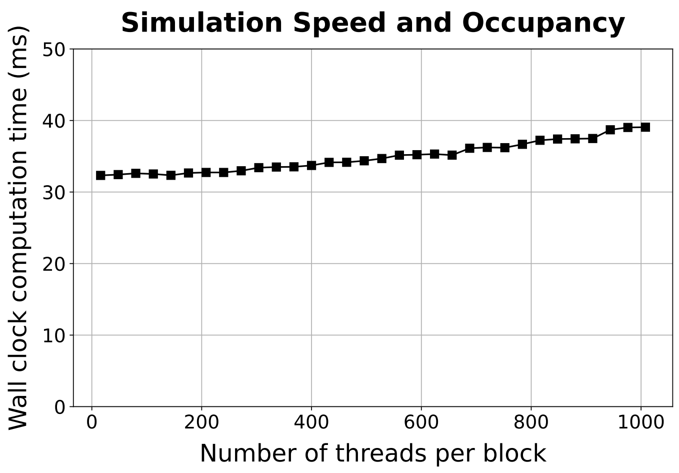


Figure 3: The same kernel was executed with the same data but different block sizes and the average execution time over 1000 runs was recorded. Fewer threads per block results in faster execution times on the GPUs.

Simulations were conducted on Cori-Haswell at the National Energy Research Scientific Computing Center (NERSC). Cori-Haswell consists of 2,388 nodes in 14 cabinets, using Intel Xeon Processor E5-2698 v3 with a clock rate of 2.3 GHz. Each processor has 32 physical threads per node and two hyper-threads per core, with 128 GB of memory per node. The interconnect is a Cray Aries with Dragonfly topology and > 45 TB/s global peak bisection bandwidth. The Julia-MPI and Fortran-MPI tests were both run with up to 32 ranks per node.

The scaling plots in Figure 4 show that the Julia-MPI and Fortran-MPI models have identical performance at two cores; Julia-MPI is faster by up to a factor of two for mid-range core counts; and Fortran-MPI is 2x faster than Julia-MPI at higher ranges, depending on the resolution. For both languages, computation scales well with processor count, while communication does not, and communication progressively requires a much larger fraction of time at higher processor counts (Figure 5). Once computations are optimized, communication, which is fixed by the interconnect speed, will remain a bottleneck regardless of the language. At the lowest resolution of 128x128, there is insufficient work beginning at 512 processors (which corresponds to 32 grid-cells per processor), and timing is dominated by communication, resulting in poor scaling above 512 processors. Communication times in Julia are much more variable than in Fortran across samples and processor counts, as shown in the right column of Figure 4. When measuring computation time without communication (Figure 4, right column), Julia-MPI scales nearly perfectly, while Fortran-MPI computational time drops off from perfect scaling at 8 and 16 cores. This produces the Julia times that are 2x faster for the total times for mid-range processor counts of 16 and higher. Overall, Julia performance on CPU clusters is extremely competitive with Fortran. Once the high-level codes have been optimized, the "winner" between Julia and Fortan will likely depend on the details of the MPI libraries and hardware.

4 Optimization Tips for Julia Developers

Julia serves the dual purpose of a prototyping language as well as a production language. Not only can we construct quick-to-write but slow-performing code (although still significantly faster than other development languages, as we saw with comparison to python) to demonstrate an idea, we can also spend a bit more time to carefully construct an optimized code to achieve performance on par with Fortran. Julia's ability to act as a prototyping language can be attributed to one of its key features: dynamic typing. Just like Python, variables may be initialized without defining their types. However, Julia is also endowed with a static typing feature, even though it is optional. If the variable types are statically defined in a concrete fashion, performance is greatly improved. Julia activates its dynamic typing feature with an "Any" type which could be any type at run time. So, Julia must compile parts of the code on the fly (*Eval of Julia code*, 2016). A method involving an "Any" type is compiled at run time for whatever type is actually provided during execution (called just-in-time compiling). The implication is that without static typing, performance will greatly suffer from compilation during run time. Additionally, with concrete types, the Julia compiler may optimize the code much further than if it is compiled for an unknown type.

When first creating the MPAS shallow water core in Julia, we did not specify the array types, and let Julia assign them the "Any" type:

```julia
struct MPAS_Ocean
    layerThickness
    normalVelocity
    ...
end
```

However, by concretely defining these variables to be floating point arrays, we gain a substantial performance boost:

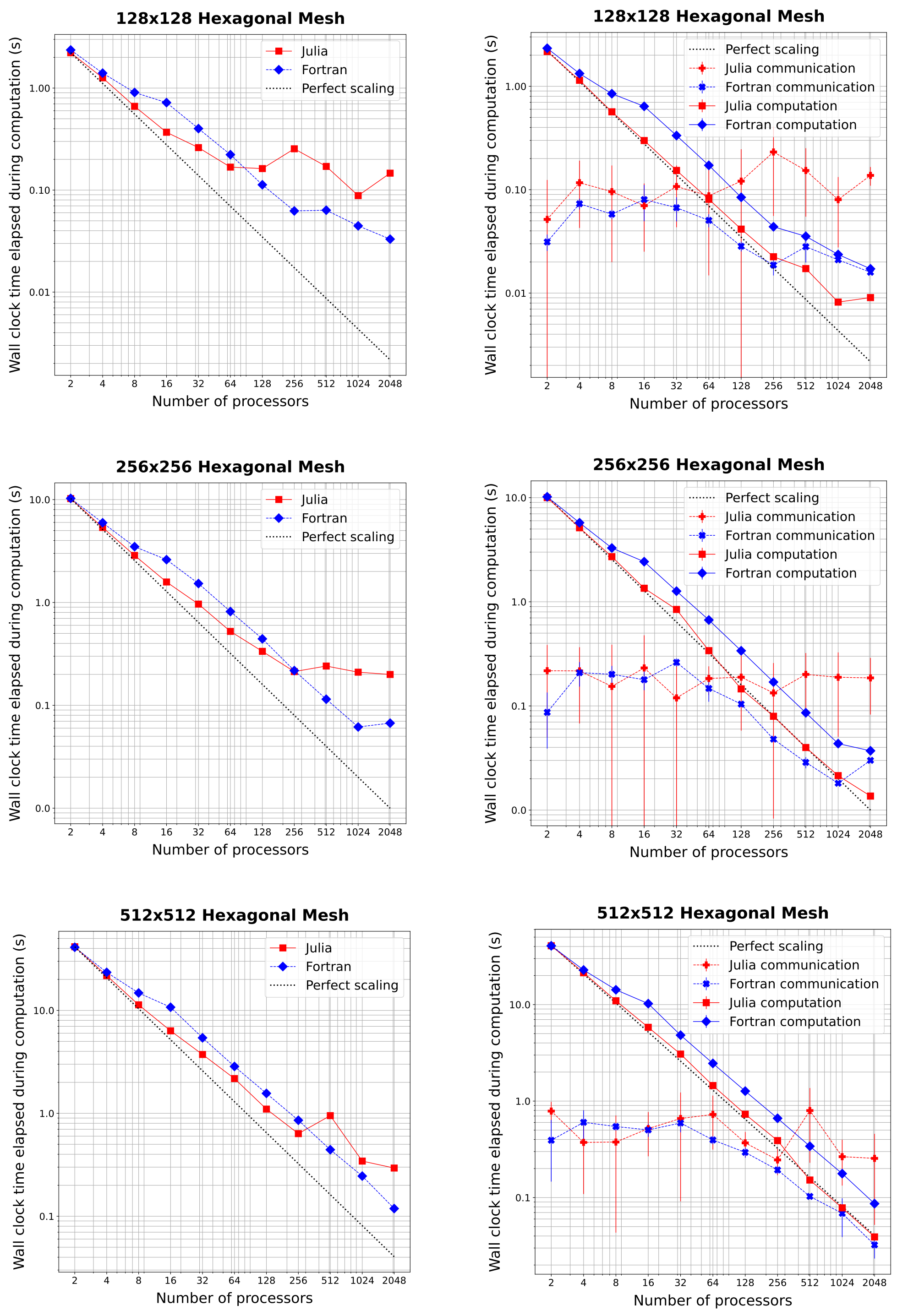
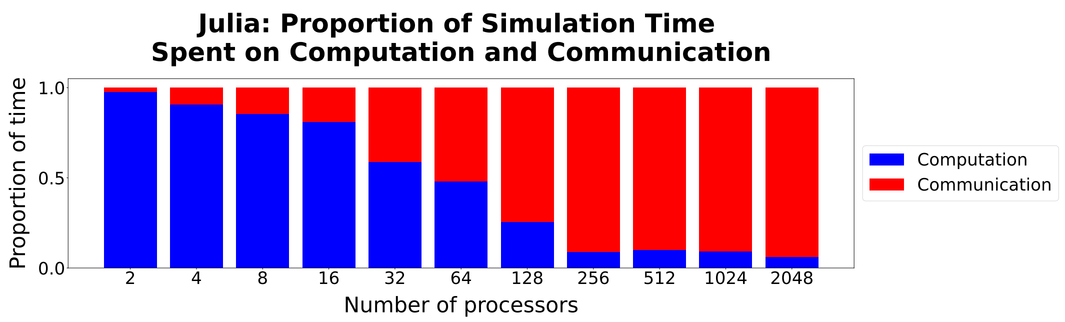


Figure 4: Wall clock time versus the number of processors to simulate 10 steps of the coastal Kelvin wave test with 100 layers. Left column shows total time without i/o; right column splits MPI communication and computation. Vertical lines display the standard deviation of communication times.



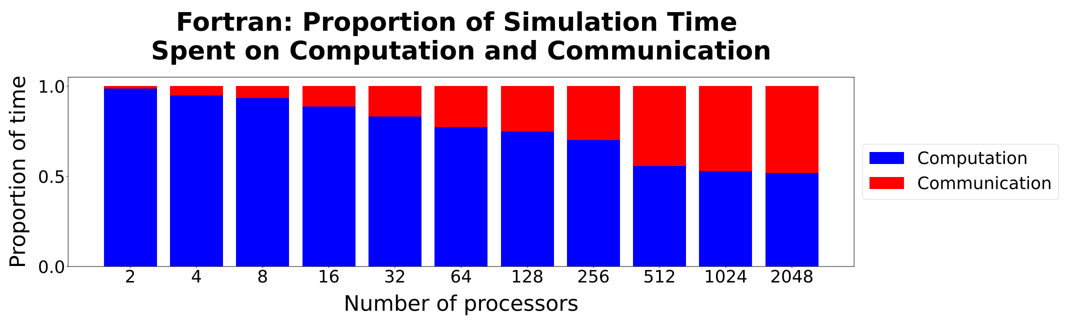


Figure 5: Comparison of the proportion of time spent in computation (blue) versus communication (red) in Julia-MPI (top) and Fortran-MPI (bottom) on the 128x128 hexagonal mesh. The relative time spent in communication increases dramatically at high processor counts.

```
struct MPAS_Ocean
    layerThickness::Array{Float64}
    normalVelocity::Array{Float64}
end
```

When parallelizing for the graphics card, a different array type is used that is suited for GPUs. We tried defining an abstract array type that encompasses both the CPU and GPU data types, so that `CUDA.CuArray`s and regular `Array`s could be used interchangeably, allowing the model to be run on the GPU or CPU at will. We also used an abstract type specification on the contents of these arrays `F <: Float`, meaning any type extending the abstract floating point type can be used at runtime.

```
struct MPAS_Ocean
    layerThickness::AbstractArray{F <: Float}
    normalVelocity::AbstractArray{F <: Float}
end
```

This approach seems like it should be performant, since the types are defined before run time. However, abstract types, like an Any type, slow down execution since at run time they may actually be a different type that extends the abstract type (`CUDA.CuArray` or `Array`), meaning the compiler is doing just-in-time compiling. Similarly, specifying an inexact element type (`F <: Float`) rather than a concrete type (`Float64`) is very inefficient.

Instead, two separate structures should be defined concretely when running on GPUs versus CPUs:

```
struct MPAS_Ocean_CUDA
```

```
    layerThickness::CUDA.CuArray{Float64,2}
    normalVelocity::CUDA.CuArray{Float64,2}
end

struct MPAS_Ocean
    layerThickness::Array{Float64,2}
    normalVelocity::Array{Float64,2}
end
```

Now the array types are concrete, element types are concrete (`Float64`), and the number of dimensions is specified (`Float64,2`). This code no longer has the advantageous feature of being able to switch between running on the CPU and GPU on the fly. However, the execution speed is massively improved. We found that making this change from abstract to concrete array types sped up computation by a factor of 34x.

The key in optimizing Julia code, we found, was reducing allocations. Memory allocation significantly slows down execution. And it is not always obvious what seemingly innocent actions may allocate memory. For example, simply reading a pair of values from an array with two columns:

```
cell1Index, cell2Index = cellsOnEdge[:,iEdge]
```

can allocate significant memory. In one test, this one line (executed repeatedly throughout the simulation) allocated 408 KiB. This is because the line is really creating a tuple, not directly reading each column into the two scalar variables. If we separate this into two lines to enforce only using scalars and not allocating tuples or arrays:

```
cell1Index = cellsOnEdge[1,iEdge]
cell2Index = cellsOnEdge[2,iEdge]
```

then this cuts allocations to zero—making this line almost instantaneous, and dropping the time spent on the whole tendency calculation from 198 μs to 99 μs. That means this line alone was responsible for about 50% of the computation time, when it could be rewritten to take no time at all.

There are likely many inconspicuous lines like this lurking in one's Julia code, slowing it down substantially. Additionally, even one overlooked field which is not concretely typed may significantly slow execution. Luckily, Julia is equipped with a tool to quickly locate such memory-hoarding lines. This tool is called `@code_warntype`. Prefixing a function call with it will print out a color-coded list breaking each line down to individual memory operations:

```
@code_warntype calculate_normal_velocity_tendency!(mpas)
```

It helpfully highlights inexact types and memory allocations with red, pointing a user right to the lines and fields that need to be optimized. This feature alone makes Julia very powerful for high-performance applications, significantly speeding up development time to optimize a model's performance.

Another very helpful tool when optimizing Julia code is `--track-allocations`, a command line option that can be added to any Julia execution as follows:

```
$ julia --track-allocations=user ./anyJuliascript.jl
```

A new file is created at `./anyJuliascript.jl.XXX.mem` (where `XXX` is some unique number). This file contains each line of the script prefixed by the number of memory allocations created by that line, giving a line-by-line breakdown of where allocations occur.

5 Conclusion

As new programming languages and libraries become available, it is important for model developers to learn new techniques and evaluate them against their current methods. This is particularly true as computing architectures continue to evolve, and long-standing languages such as C++ and Fortran require additional libraries to remain competitive on new supercomputers.

In this work, we created three implementations of a shallow water model in Julia in order to compare ease of development and performance to standard Fortran and Python implementations. The three Julia codes were designed for single-CPU, GPU-enhanced single CPU, and parallelized multi-core CPU architectures. Julia-MPI speeds were identical to Fortran-MPI at low core counts, 2x faster for mid-range, and 2x slower at higher core counts. Julia-MPI exhibited better scaling than Fortran-MPI for computation-only times, and more variability for communication times.

The most surprising result of this study was the speed of computations on the GPUs—a speed-up of 40,000 to over 100,000 times compared to the CPU. Of course, this comes with the caveat that memory transfer between CPU and GPU can take thousands of times longer than the computation, up to 0.5s at our highest resolution. So the key is to transfer memory to and from the GPU as little as possible, which is a well-known practice. If one can fit the full resolution of a computational physics domain within the memory of a single graphics card and sample results rarely, GPUs offer extraordinary speed-ups. For climate models, a single low-resolution component may well fit into GPU memory if the developers are careful with their memory footprint. The difficulty is that including ocean, atmosphere, land, and sea ice components requires the use of multiple nodes, and inter-node communication will keep the model slow, regardless of the GPU speed. Higher-resolution domains will need many nodes for each component and present the same problem.

The shallow water equations are simple enough for rapid development and verification, yet contain the salient features of any ocean model: intensive computation of the tendency terms, a time-stepping routine, and for the parallel version, interleaved halo communication of the partition boundary. Indeed, this layout, and the lessons learned here, apply to almost all computational physics codes.

This work specifically tests unstructured horizontal meshes, as opposed to structured quadralateral grids. Unstructured meshes refer to a neighbor's index using additional pointer arrays, so require an extra memory access for horizontal stencils. In structured grids, the physical neighbors are also neighbors in array space ($i + 1, j + 1$, etc), which leads to more contiguous memory access patterns that are easier for compilers to optimize. Our results show that unstructured meshes do not present any significant challenge in either Fortran or Julia. The use of a structured vertical index in the inner-most position and testing with 100 layers provides sufficient contiguous memory access for cache locality.

In the end, we were impressed by our experience with Julia. It did fulfill the promise of fast and convenient prototyping, with the ability to eventually run at high speeds on multiple high performance architectures—after some effort and lessons learned by the developers. The Julia libraries for MPI and CUDA were powerful and convenient. E3SM does not have plans to develop model components with Julia, but this study provides a useful comparison to our C++ and Fortran codes as we move towards heterogeneous, exascale computers.

Open Research

Three code repositories were used for the performance comparisons in this study. These are publicly available on both GitHub and Zenodo:

1. Julia Shallow Water code for serial CPU, CUDA-GPU, and MPI-parallelized CPU

GitHub: https://github.com/robertstrauss/MPAS_Ocean_Julia
Zenodo: https://doi.org/10.5281/zenodo.7493065

2. Python Rotating Shallow Water Verification Suite
   GitHub: https://github.com/siddharthabishnu/Rotating_Shallow_Water_Verification_Suite.git. This study used the specific code version https://github.com/siddharthabishnu/Rotating_Shallow_Water_Verification_Suite/tree/v1.0.1
   Zenodo: https://doi.org/10.5281/zenodo.7425628
3. Fortran-MPI MPAS Shallow Water code with Coastal Kelvin wave initial condition (Petersen et al., 2022)
   GitHub: https://github.com/MPAS-Dev/MPAS-Model. This study used the specific code version https://github.com/mark-petersen/MPAS-Model/releases/tag/SW_julia_comparison_V1.0.
   Zenodo: https://doi.org/10.5281/zenodo.7439134

The planar hexagonal MPAS-Ocean meshes used in this study for the numerical simulations and convergence tests of the coastal Kelvin wave and the inertia-gravity wave can be obtained from the Zenodo release of the Python Rotating Shallow Water Verification Suite Meshes at https://doi.org/10.5281/zenodo.7419817.

Acknowledgments

RRS gratefully acknowledges the support of the U.S. Department of Energy (DOE) through the Los Alamos National Laboratory (LANL) LDRD Program and the Center for Nonlinear Studies for this work. SB was supported by Scientific Discovery through Advanced Computing (SciDAC) projects LEAP (Launching an Exascale ACME Prototype) and CANGA (Coupling Approaches for Next Generation Architectures) under the DOE Office of Science, Office of Biological and Environmental Research (BER). MRP was supported by the Energy Exascale Earth System Model (E3SM) project, also funded by the DOE BER.

This research used computational resources provided by: the Darwin testbed at LANL, which is funded by the Computational Systems and Software Environments subprogram of LANL's Advanced Simulation and Computing program (NNSA/DOE); the LANL Institutional Computing Program, which is supported by the DOE National Nuclear Security Administration under Contract No. 89233218CNA000001; and the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the DOE under Contract No. DE-AC02-05CH11231.

References

Bishnu, S. (2021). *Time-Stepping Methods for Partial Differential Equations and Ocean Models* (Doctoral dissertation, Florida State University). doi: 10.5281/zenodo.7439539

Bishnu, S. (2022, December). *Rotating shallow water verification suite.* Zenodo. Retrieved from https://doi.org/10.5281/zenodo.7425628 doi: 10.5281/zenodo.7425628

Bishnu, S., Petersen, M., Quaife, B., & Schoonover, J. (2022, dec). *Verification suite of test cases for the barotropic solver of ocean models.* Authorea. Retrieved from https://doi.org/10.22541/essoar.167100170.03833124/v1 doi: 10.22541/essoar.167100170.03833124/v1

Bleichrodt, F., Bisseling, R. H., & Dijkstra, H. A. (2012, January). Accelerating a barotropic ocean model using a GPU. *Ocean Modelling*, *41*, 16–21.

Retrieved 2022-11-29, from https://www.sciencedirect.com/science/article/pii/S1463500311001661 doi: 10.1016/j.ocemod.2011.10.001

Caldwell, P. M., Mametjanov, A., Tang, Q., Van Roekel, L. P., Golaz, J. C., Lin, W., ... Zhou, T. (2019). The DOE E3SM coupled model version 1: Description and results at high resolution. *Journal of Advances in Modeling Earth Systems*, *11*(12), 4095–4146. doi: 10.1029/2019MS001870

Cushman-Roisin, B., & Beckers, J.-M. (2011). *Introduction to geophysical fluid dynamics: physical and numerical aspects*. Academic press.

Dalcín, L., Paz, R., & Storti, M. (2005). Mpi for python. *Journal of Parallel and Distributed Computing*, *65*(9), 1108–1115.

Dalcín, L., Paz, R., Storti, M., & D'Elía, J. (2008). Mpi for python: Performance improvements and mpi-2 extensions. *Journal of Parallel and Distributed Computing*, *68*(5), 655–662.

Eval of Julia code. (2016). Retrieved 2022-10-10, from https://docs.julialang.org/en/v1/devdocs/eval/#

Gevorkyan, M. N., Demidova, A. V., Korolkova, A. V., & Kulyabov, D. S. (2019, April). Statistically significant performance testing of Julia scientific programming language. *Journal of Physics: Conference Series*, *1205*, 012017. Retrieved from https://iopscience.iop.org/article/10.1088/1742-6596/1205/1/012017 doi: 10.1088/1742-6596/1205/1/012017

Golaz, J.-C., Caldwell, P. M., Van Roekel, L. P., Petersen, M. R., Tang, Q., Wolfe, J. D., ... Zhu, Q. (2019). The DOE E3SM Coupled Model Version 1: Overview and Evaluation at Standard Resolution. *Journal of Advances in Modeling Earth Systems*, *11*(7), 2089–2129. doi: 10.1029/2018MS001603

Introduction to CUDA. (2022). Retrieved 2022-12-13, from https://cuda.juliagpu.org/stable/tutorials/introduction/#A-simple-example-on-the-CPU

Jiang, J., Lin, P., Wang, J., Liu, H., Chi, X., Hao, H., ... Zhang, L. (2019). Porting LASG/ IAP Climate System Ocean Model to Gpus Using OpenAcc. *IEEE Access*, *7*, 154490–154501. (Conference Name: IEEE Access) doi: 10.1109/ACCESS.2019.2932443

Klöckner, A., Pinto, N., Lee, Y., Catanzaro, B., Ivanov, P., & Fasih, A. (2012). PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation. *Parallel Computing*, *38*(3), 157–174. doi: 10.1016/j.parco.2011.09.001

Klöwer, M., Hatfield, S., Croci, M., Düben, P. D., & Palmer, T. N. (2022). Fluid simulations accelerated with 16 bits: Approaching 4x speedup on a64fx by squeezing shallowwaters.jl into float16. *Journal of Advances in Modeling Earth Systems*, *14*(2), e2021MS002684. Retrieved from https://agupubs.onlinelibrary.wiley.com/doi/abs/10.1029/2021MS002684 (e2021MS002684 2021MS002684) doi: https://doi.org/10.1029/2021MS002684

Lam, S. K., Pitrou, A., & Seibert, S. (2015). Numba: A llvm-based python jit compiler. In *Proceedings of the second workshop on the llvm compiler infrastructure in hpc* (pp. 1–6).

Lin, W.-C., & McIntosh-Smith, S. (2021, November). Comparing Julia to Performance Portable Parallel Programming Models for HPC. In *2021 International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)* (pp. 94–105). St. Louis, MO, USA: IEEE. Retrieved from https://ieeexplore.ieee.org/document/9652798/ doi: 10.1109/PMBS54543.2021.00016

Mielikainen, J., Huang, B., Huang, H.-L. A., & Goldberg, M. D. (2012, August). Improved GPU/CUDA Based Parallel Weather and Research Forecast (WRF) Single Moment 5-Class (WSM5) Cloud Microphysics. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, *5*(4), 1256–1265. (Conference Name: IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing) doi: 10.1109/JSTARS.2012.2188780

Perkel, J. M. (2019, August). Julia: come for the syntax, stay for the speed. *Nature*, *572*(7767), 141–142. Retrieved from http://www.nature.com/articles/d41586-019-02310-3 doi: 10.1038/d41586-019-02310-3

Petersen, M. R., Asay-Davis, X. S., Berres, A. S., Chen, Q., Feige, N., Hoffman, M. J., … Woodring, J. L. (2019). An Evaluation of the Ocean and Sea Ice Climate of E3SM Using MPAS and Interannual CORE-II Forcing. *Journal of Advances in Modeling Earth Systems*, *11*(5), 1438–1458. doi: 10.1029/2018MS001373

Petersen, M. R., Bishnu, S., & Strauss, R. R. (2022, December). *Mpas-ocean shallow water performance test case.* Zenodo. Retrieved from https://doi.org/10.5281/zenodo.7439134 doi: 10.5281/zenodo.7439134

Petersen, M. R., Jacobsen, D. W., Ringler, T. D., Hecht, M. W., & Maltrud, M. E. (2015, February). Evaluation of the arbitrary Lagrangian–Eulerian vertical coordinate method in the MPAS-Ocean model. *Ocean Modelling*, *86*, 93–113. doi: 10.1016/j.ocemod.2014.12.004

Ramadhan, A., Wagner, G. L., Hill, C., Campin, J.-M., Churavy, V., Besard, T., … Marshall, J. (2020). Oceananigans.jl: Fast and friendly geophysical fluid dynamics on gpus. *Journal of Open Source Software*, *5*(53), 2018. Retrieved from https://doi.org/10.21105/joss.02018 doi: 10.21105/joss.02018

Ringler, T. D., Petersen, M. R., Higdon, R. L., Jacobsen, D., Jones, P. W., & Maltrud, M. (2013). A multi-resolution approach to global ocean modeling. *Ocean Modelling*, *69*, 211–232.

Ringler, T. D., Thuburn, J., Klemp, J. B., & Skamarock, W. C. (2010). A unified approach to energy conservation and potential vorticity dynamics for arbitrarily-structured C-grids. *Journal of Computational Physics*, *229*(9), 3065–3090.

Shchepetkin, A. F., & McWilliams, J. C. (2005). The regional oceanic modeling system (ROMS): a split-explicit, free-surface, topography-following-coordinate oceanic model. *Ocean modelling*, *9*(4), 347–404.

Srinath, A. (2020, Nov). *Accelerating python on gpus with nvc++ and cython.* Retrieved from https://developer.nvidia.com/blog/accelerating-python-on-gpus-with-nvc-and-cython/

Strauss, R. R. (2023, January). *Julia Layered Shallow Water Model on Various Hardwares.* Retrieved from https://doi.org/10.5281/zenodo.7493065 doi: 10.5281/zenodo.7493065

Thuburn, J., Ringler, T. D., Skamarock, W. C., & Klemp, J. B. (2009). Numerical representation of geostrophic modes on arbitrarily structured C-grids. *Journal of Computational Physics*, *228*(22), 8321–8335.

Trott, C. R., Lebrun-Grandié, D., Arndt, D., Ciesko, J., Dang, V., Ellingwood, N., … Wilke, J. (2022). Kokkos 3: Programming model extensions for the exascale era. *IEEE Transactions on Parallel and Distributed Systems*, *33*(4), 805-817. doi: 10.1109/TPDS.2021.3097283

Xu, S., Huang, X., Oey, L.-Y., Xu, F., Fu, H., Zhang, Y., & Yang, G. (2015, September). POM.gpu-v1.0: a GPU-based Princeton Ocean Model. *Geoscientific Model Development*, *8*(9), 2815–2827. Retrieved 2022-11-29, from https://gmd.copernicus.org/articles/8/2815/2015/ (Publisher: Copernicus GmbH) doi: 10.5194/gmd-8-2815-2015

Xu, S., Huang, X., Zhang, Y., Hu, Y., & Yang, G. (2014, June). A customized GPU acceleration of the princeton ocean model. In *2014 IEEE 25th International Conference on Application-Specific Systems, Architectures and Processors* (pp. 192–193). (ISSN: 2160-052X) doi: 10.1109/ASAP.2014.6868661

Ye, Y., Song, Z., Zhou, S., Liu, Y., Shu, Q., Wang, B., … Wang, L. (2022, July). swNEMO_v4.0: an ocean model based on NEMO4 for the new-generation Sunway supercomputer. *Geoscientific Model Development*, *15*(14), 5739–5756. Retrieved 2022-11-30, from https://gmd.copernicus.org/articles/15/5739/2022/ (Publisher: Copernicus GmbH) doi: 10.5194/gmd-15-5739-2022

Zhao, X.-d., Liang, S.-x., Sun, Z.-c., Zhao, X.-z., Sun, J.-w., & Liu, Z.-b. (2017, August). A GPU accelerated finite volume coastal ocean model. *Journal of Hydrodynamics, Ser. B*, *29*(4), 679–690. Retrieved 2022-11-29, from https://www.sciencedirect.com/science/article/pii/S1001605816607801 doi: 10.1016/S1001-6058(16)60780-1