LA-UR-21-31104
Approved for public release; distribution is unlimited.

Title: Creating an MPAS Ocean Shallow Water Core in Julia

Author(s): Petersen, Mark Roger
Strauss, Robert Russell
Bishnu, Siddhartha

Intended for: LANL internal presentation

Issued: 2021-11-06

Creating an MPAS Ocean Shallow Water Core in Julia

Robert Strauss, Sid Bishnu, Mark Petersen
LANL Center for Nonlinear Studies
Los Alamos High School

Why Julia?

Tradeoff between execution speed and development speed:

- Development languages (e.g. Python) are easy, but slow
- Production languages (e.g. C) are hard, but fast

Julia aims to be the best of both. Was first released in 2012

I created an MPAS model in Julia to test its potential for scientific HPC.

Other Shallow Water and Ocean Models in Julia

- Klima (MIT)
- Oceananigans (MIT)
- ShallowWaters.jl (Milan K, University of Oxford)

All use a regular rectilinear mesh.

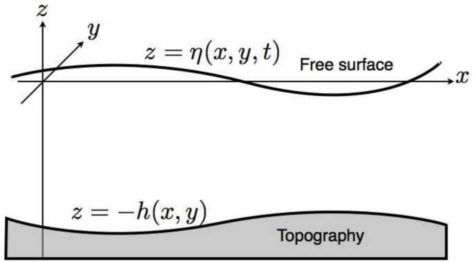
I use an unstructured TRiSK mesh in Julia, which is novel.

Equation Set & Discretization

The Shallow Water Equations

$$\frac{\partial \eta}{\partial t} + \nabla \cdot ((h+\eta)\vec{u}) = 0,$$

$$\frac{\partial \vec{u}}{\partial t} + (\vec{u} \cdot \nabla)\vec{u} + f\hat{k} \times \vec{u} = -g\nabla\eta.$$



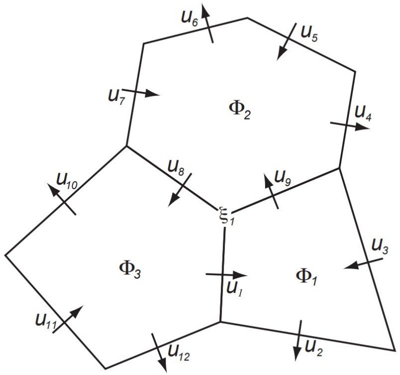
2 prognostic fields:

η - sea surface height

u&v - average water column group velocity

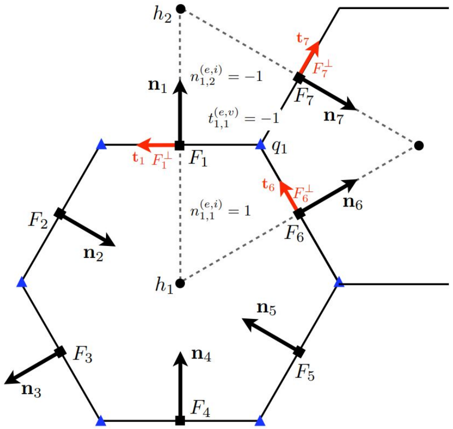
Primal & Dual Mesh (TRiSK)

- Julia version uses TRiSK discrete operators



η defined at cell centers

Normal velocity at edges



(+ Vertical layers)

Julia single-core CPU implementation

- Using a standard MPAS planar-hex mesh
- Variable names are identical to MPAS
- Code structure is similar to MPAS

```
function calculate_normal_velocity_tendency!(mpasOcean::MPAS_Ocean)
    mpasOcean.normalVelocityTendency[:] .= 0

    for iEdge in 1:mpasOcean.nEdges
        if mpasOcean.boundaryEdge[iEdge] == 0
            # gravity term: take gradient of sshCurrent across edge
            cell1Index, cell2Index = mpasOcean.cellsOnEdge[:,iEdge]

            if cell1Index != 0 && cell2Index != 0
                mpasOcean.normalVelocityTendency[iEdge] = mpasOcean.gravity * ( mpasOcean.sshCurrent[cell1Index] - mpasOcean.sshCurrent[cell2Index] ) / mpasOcean.dcEdge[iEdge]
            end
```

$$-\frac{\partial u}{\partial t} = g\frac{\partial \eta}{\partial x}$$

$$+ u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} - fv$$

(included but not shown in snippet)

Julia GPU version

CUDA getting index from thread replaces for loop, otherwise identical

```julia
function calculate_normal_velocity_tendency_cuda_kernel!(nEdges,
                                                       normalVelocityTendency,
                                                       normalVelocity,
                                                       ssh,
                                                       cellsOnEdge,
                                                       nEdgesOnEdge,
                                                       edgesOnEdge,
                                                       weightsOnEdge,
                                                       fEdge,
                                                       dcEdge,
                                                       gravity)

    iEdge = (CUDA.blockIdx().x - 1) * CUDA.blockDim().x + CUDA.threadIdx().x
    if iEdge <= nEdges

        # gravity term: take gradient of ssh across edge
        cell1 = cellsOnEdge[1,iEdge]
        cell2 = cellsOnEdge[2,iEdge]

        if cell1 != 0 && cell2 != 0
            normalVelocityTendency[iEdge] = gravity * ( ssh[cellsOnEdge[1,iEdge]] - ssh[cellsOnEdge[2,iEdge]] ) / dcEdge[iEdge]
        end
```

```julia
    for iEdge in 1:mpasOcean.nEdges
```

The same gravity term calculation, but written as a GPU kernel

$$-\frac{\partial u}{\partial t} = g\frac{\partial \eta}{\partial x} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} - fv$$

```julia
function calculate_normal_velocity_tendency_cuda!(mpasOcean::MPAS_Ocean)
    CUDA.@cuda blocks=cld(mpasOcean.nEdges, 1024) threads=1024 maxregs=64 calculate_normal_velocity_tendency_cuda_kernel!(
                                                           mpasOcean.nEdges,
                                                           mpasOcean.normalVelocityTendency,
                                                           mpasOcean.normalVelocityCurrent,
                                                           mpasOcean.sshCurrent,
                                                           mpasOcean.cellsOnEdge,
                                                           mpasOcean.nEdgesOnEdge,
                                                           mpasOcean.edgesOnEdge,
                                                           mpasOcean.weightsOnEdge,
                                                           mpasOcean.fEdge,
                                                           mpasOcean.dcEdge,
                                                           mpasOcean.gravity)
end
```

CUDA runs our kernel function for every edge/cell each on its own thread

Julia MPI (multi-core CPU) version

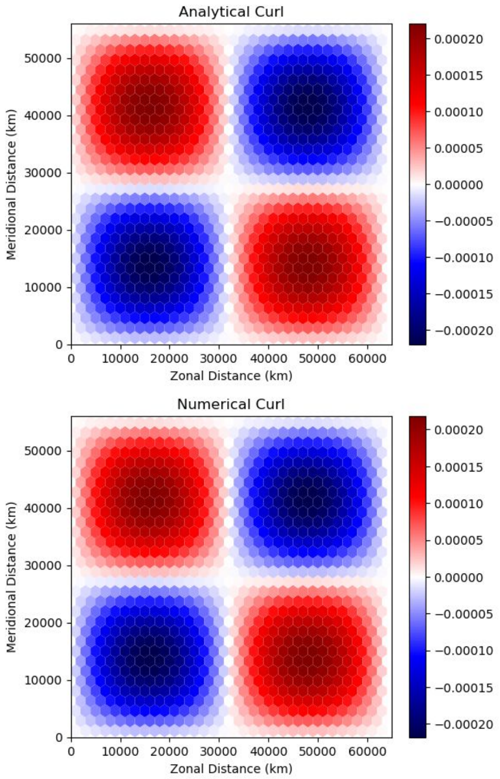
- MPI libraries are available for Julia
- Implemented domain decomposition and halo updates, like in MPAS

```julia
for f in 1:nFrames
    # simulate until the halo areas are all invalid and need to be updated
    for h in 1:halowidth
        forward_backward_step!(mpasOcean)
    end

    ### request cells in my halo from chunks with those cells
    halobufferssh = [] # temporarily stores new halo ssh
    halobuffernv = [] # temporarily stores new halo normal velocity
    recreqs = []
    for (srcchunk, localcells) in cellsFromChunk[rank+1]
        newhalossh = Array{eltype(mpasOcean.sshCurrent)}(undef, length(localcells))
        append!(halobufferssh, [newhalossh])
        reqssh = MPI.Irecv!(newhalossh, srcchunk-1, 0, comm) # tag 0 for ssh
        append!(recreqs, [reqssh])

        localedges = collect(Set(mpasOcean.edgesOnCell[:,localcells]))
        newhalonv = Array{eltype(mpasOcean.normalVelocityCurrent)}(undef, length(localedges))
        append!(halobuffernv, [newhalonv])
        reqnv = MPI.Irecv!(newhalonv, srcchunk-1, 1, comm) # tag 1 for norm vel
        append!(recreqs, [reqnv])
    end

    MPI.Barrier(comm)
```

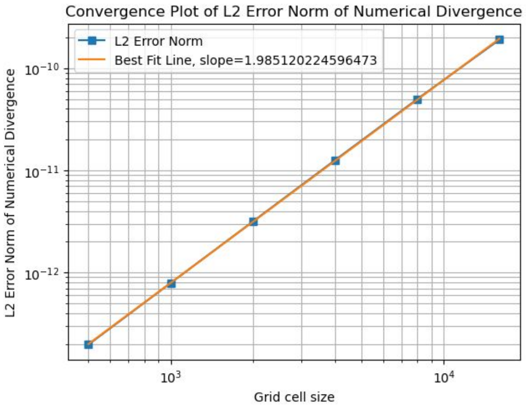
Time stepping

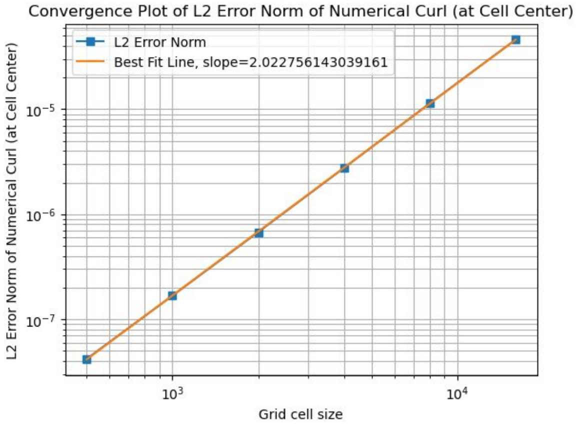
MPI Communication for Halo update

Unit Tests of TRiSK Discrete Operators



- Gradient, Divergence, Curl, Flux Mapping (primal to dual)
- GPU and CPU versions produce nearly identical results



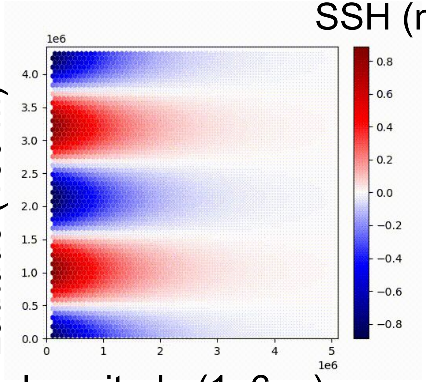


Exact solution test cases & Convergence (CPU & GPU)

Kelvin wave

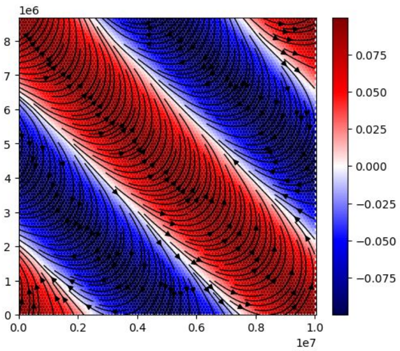
SSH (m)

Latitude (1e6 m)

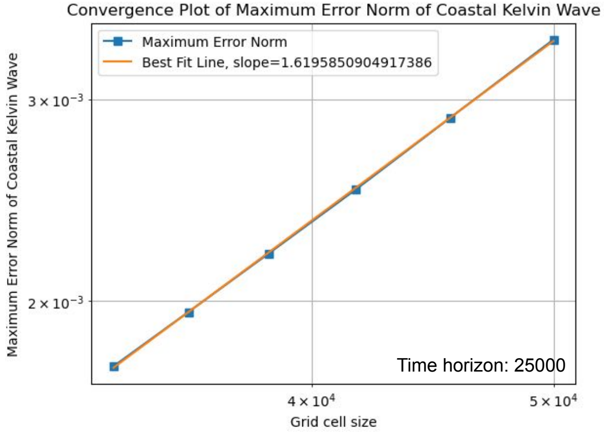


Longitude (1e6 m)

Inertia-gravity wave



Close to second-order convergence between numerical and exact solution

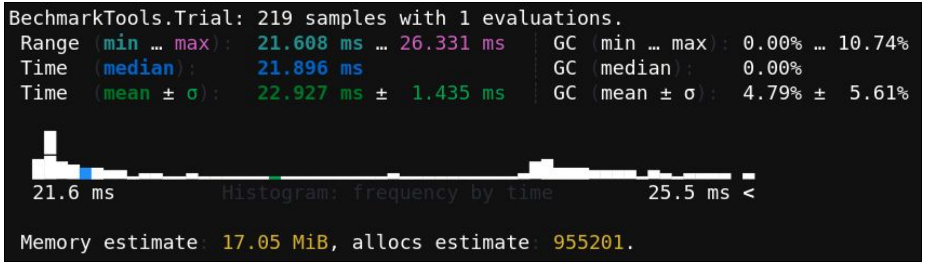


- Verification against exact solutions for two test cases
- Julia can produce visualization like python (pull in python libraries)

CPU versus GPU performance comparison

- Tested on personal NVIDIA GTX 1080 GPU
  - 2560 NVIDIA CUDA Cores
  - one GPU thread per MPAS edge
- Test domain is 100x100 cells.
- Timing with Benchmarks.jl package

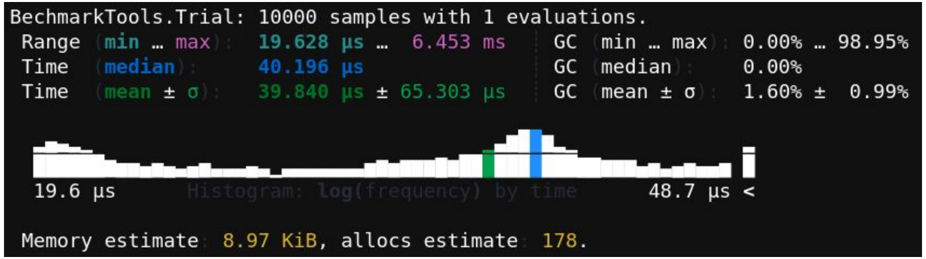
20 Streaming multiprocessors,
2048 threads per streaming multiprocessor



```
BechmarkTools.Trial: 219 samples with 1 evaluations.
 Range (min … max):  21.608 ms … 26.331 ms  ┊ GC (min … max): 0.00% … 10.74%
 Time  (median):     21.896 ms              ┊ GC (median):    0.00%
 Time  (mean ± σ):   22.927 ms ±  1.435 ms  ┊ GC (mean ± σ):  4.79% ±  5.61%

 21.6 ms         Histogram: frequency by time        25.5 ms <

 Memory estimate: 17.05 MiB, allocs estimate: 955201.
```

Julia-CPU: 22.9ms per timestep

40x times faster than Python-CPU version
(using MPAS-Python code from Sid Bishnu)



```
BechmarkTools.Trial: 10000 samples with 1 evaluations.
 Range (min … max):  19.628 μs …  6.453 ms  ┊ GC (min … max): 0.00% … 98.95%
 Time  (median):     40.196 μs              ┊ GC (median):    0.00%
 Time  (mean ± σ):   39.840 μs ± 65.303 μs  ┊ GC (mean ± σ):  1.60% ±  0.99%

 19.6 μs      Histogram: log(frequency) by time      48.7 μs <

 Memory estimate: 8.97 KiB, allocs estimate: 178.
```

Julia-GPU: 0.04ms per timestep

500x faster on the GPU!

Comparison of Julia-MPI to Fortran-MPI on CPUs

- We are currently benchmarking Julia and Fortran MPAS on supercomputers.
- Our early rough benchmarks put Fortran strongly in the lead, almost 70x faster than Julia
- However, the Fortran MPAS Ocean has been highly optimized, and we just started optimizing Julia-MPI MPAS, like core-count to thread-count.
- These are also early results, nonlinear scaling may effect this as we test with higher resolutions and add to the Julia code.
    - Similar projects have found comparable speeds between Julia-MPI and Fortran or C with MPI

Conclusion

- Julia was fast to develop
  - 3 months for MPAS shallow water Julia CPU, GPU, and multi-core versions
- Easy to switch from from CPU to GPU version
  - Drop in CUDA lines instead of for loop, add kernel wrapper
- Julia does require some time to learn - not quite as easy as Python
  - Julia has dynamic typing like python
  - Can create prototypes very fast with this feature
  - For performance, we end up typing everything anyway
- Julia delivers excellent performance
  - 40x faster than Python on single CPU
  - 500x speed-up from CPU to GPU
  - In current testing, Fortran-MPI is much faster (70x) than Julia-MPI, but this is preliminary
- This project shows that julia could be useful for computational physics, and deserves further investigation.