Julia with MPI

Challenges and Best Practices

Simon Byrne
CliMA project @ Caltech

Who am I?

And what am I doing here?

- About Me:
  - Lead software engineer on the CliMA project @ Caltech
  - Building a GPU + MPI climate model in Julia
  - Contributor to the Julia language and packages since 2012
  - A primary maintainer of MPI.jl and NVTX.jl
- Why this talk?
  - Julia is great for scientific computing, but…
  - There are some pitfalls which may not be obvious

CliMA

CLIMATE MODELING ALLIANCE

Overview

1. Configuring Julia on a HPC system
2. MPI.jl
3. Profiling parallel Julia code with Nsight Systems
4. Julia + MPI pitfalls

Getting started

Configuring Julia for HPC

Installing Julia

The easy part

- Download the binaries from julialang.org
  - Don’t build from source, or use distribution-provided binaries
  - Especially don’t try to link against existing dependencies on the system
    - Julia includes quite a few patches for some of these (LLVM, libuv)
  - Building from source won’t enable any magic optimizations
    - Julia has its own compiler for that
- Plays nice with cluster module files

Some helpful environment variables

Getting the most out of your system

- JULIA_DEPOT_PATH
  - Where Julia stores packages, artifacts, precompilation caches, etc
  - Default is `~/.julia`
  - Make sure it is on a fast disk (not NFS)
  - Can specify multiple (helpful for caching artifacts across users)
- JULIA_CPU_TARGET (Julia 1.9+)
  - Julia now caches native code: specify the microarchitecture to use
  - Can specify multiple targets (helpful on a heterogeneous cluster)
    - `export JULIA_CPU_TARGET='broadwell;skylake'`

Configuring packages: the new way
Preferences.jl

- Old `Pkg.build()` scripts are now discouraged
- Preferences: per-package key-value store
  - `@load_preference(key, default_value)`
  - `@set_preferences!(key => value)`
- Stored per project (LocalPreferences.toml or Project.toml)
- Used as part of the precompilation hash
  - Specific to a given project
  - Correctly invalidate the cache on changes
  - Can be used to define constant variables

Configuring binary dependencies

Overriding your JLLs

- Julia package manager provides binary dependencies through its artifact system
  - Built using BinaryBuilder.jl cross-compilation framework
  - Windows/Mac/Linux, on x86/ARM/PPC
- Can be used to override JLL binaries
  - Need to be careful: typically you will need to override all downstream dependencies
- Inherited from LOAD_PATH
  - Modify preferences for the current session
  - Can integrate with cluster module files

```
# /config/path/Project.toml
[extras]
HDF5_jll = "0234f1f7-429e-5d53-9886-15a909be8d

[preferences.HDF5_jll]
libhdf5_path = "libhdf5"
libhdf5_hl_path = "libhdf5_hl"
```

```
export JULIA_LOAD_PATH=\
"$JULIA_LOAD_PATH:/config/path"
```

MPI.jl

Julia interface for MPI

MPI.jl basics

- Automatically determines array datatype and length
- Integration with Julia's exception handling
- Custom structs, strided arrays => generate custom datatypes
- Julia functions => generate custom reduction operators
- Covers most used MPI functions
  - Point-to-point, collectives, one-sided, I/O
  - Easy to add more: please open an issue!
- Supports GPU-aware MPI interfaces
  - Simply pass `CuArray`/`ROCArray` as a buffer
- Used by multiple Julia packages

```
# Custom datatype
struct SummaryStat
    mean::Float64
    var::Float64
    n::Float64
end

function SummaryStat(X::AbstractArray)
    m = mean(X)
    v = varm(X,m,corrected=false)
    n = length(X)
    SummaryStat(m,v,n)
end

# Custom reduction operator
# pools the mean, variance, length
# more numerically stable than sum-of-squares
function pool(S1::SummaryStat, S2::SummaryStat)
    n = S1.n + S2.n
    m = (S1.mean*S1.n + S2.mean*S2.n) / n
    v = (S1.n*(S1.var + S1.mean*(S1.mean-m)) +
         S2.n*(S2.var + S2.mean*(S2.mean-m)))/n
    SummaryStat(m,v,n)
end

# Perform reduction
summ = MPI.Reduce(SummaryStat(X), pool, comm)
```

Configuring MPI for Julia

- By default, MPI.jl uses a JLL-built binary
  - MicrosoftMPI_jll on Windows, MPICH_jll everywhere else
  - Many downstream binaries (e.g. HDF5, ADIOS2, PETSc)
- For HPC: typically want to use the binary for that system
  - Support for specific network hardware and libraries
  - Integration with system launcher (`srun`, `aprun`)
  - Direct device-device communication (CUDA-aware)
- Problem: MPI application binary interface (ABI) is implementation-defined
  - Handles (e.g. `MPI_Comm`) can be an `int` or a pointer
  - Constants (e.g. `MPI_ANY_SOURCE`) values vary
  - May be fixed in MPI 5 🤞

MPIPreferences.jl

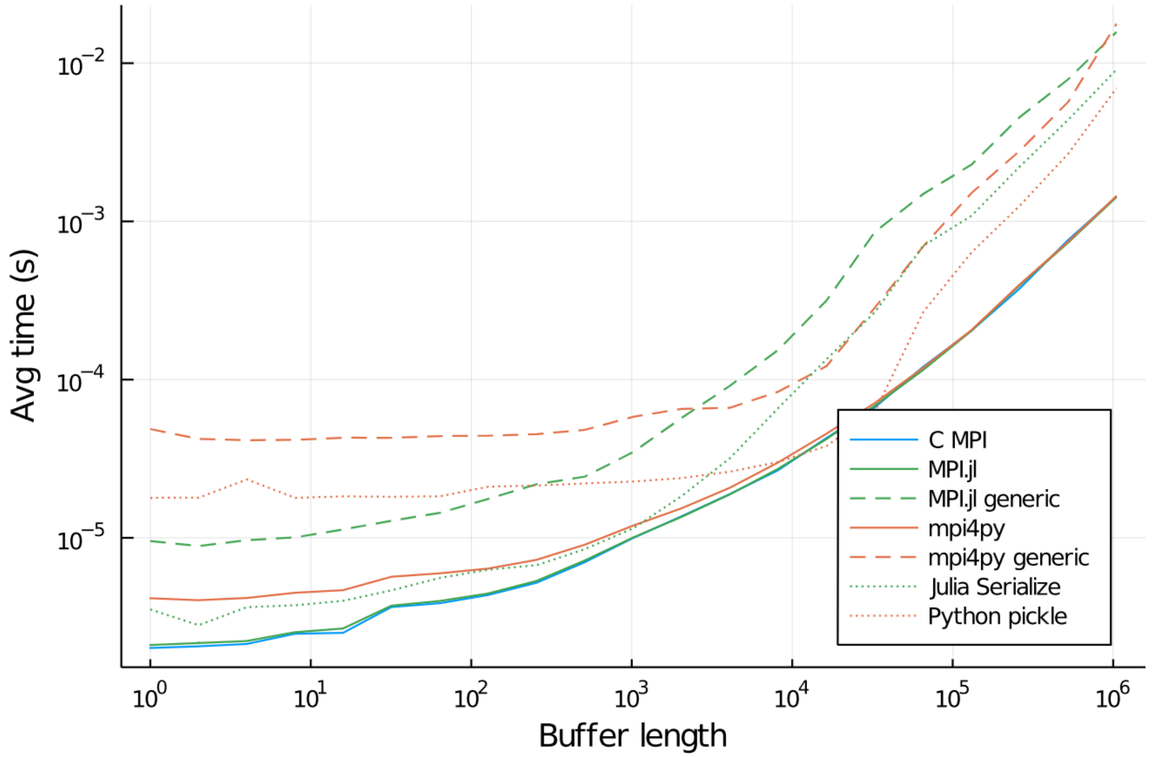
Two options

- `MPIPreferences.use_system_binary()`
  - Detects and configures ABI
  - Uses system binary directly
  - Need to override all downstream binaries
- `MPIPreferences.use_jll_binary("MPItrampoline")`
  - Uses MPItrampoline by Erik Schnetter
  - Requires building a shim (MPIwrapper) around your MPI library
  - Some operations can incur overhead (e.g. `MPI_Waitall`)
  - Downstream JLL binaries “just work”

MPI.jl tips

How to use it efficiently

- Use uppercase functions (`MPI.Send`/`MPI.Recv`) where possible
  - Requires both sender & receiver to know datatype & length
  - Minimal overhead vs C
- Lowercase functions (`MPI.send`/`MPI.recv`) serialize & deserialize
  - More flexible (e.g. strings, mutable objects), but slower
- Pre-allocate output arrays
  - Use `MPI.Reduce!` instead of `MPI.Reduce`
- Define `MPI.Buffer(array)` wrapper outside loop/function
  - Important when using custom datatypes (e.g. `SubArray`s)
- Learn how to make your launcher output ranks to individual files
  - Interleaved error stacks are difficult!
- Profile, profile, profile



Parallel profiling of Julia code

Nsight Systems + NVTX.jl

- **Nsight Systems** is an instrumenting profiler from Nvidia
  - Profiler supports Linux & Windows
  - Viewer for Linux, Windows and MacOS
  - Supports multithreading and MPI
  - Does not require a GPU (can be used for CPU-only code)
- **NVTX.jl** package for instrumenting Julia code
  - Annotate Julia code with macros which the profiler can then access
  - Options to instrument the Julia garbage collector
  - Safe to include as a package dependency (becomes a no-op if not using the profiler)

NVTX.jl basics

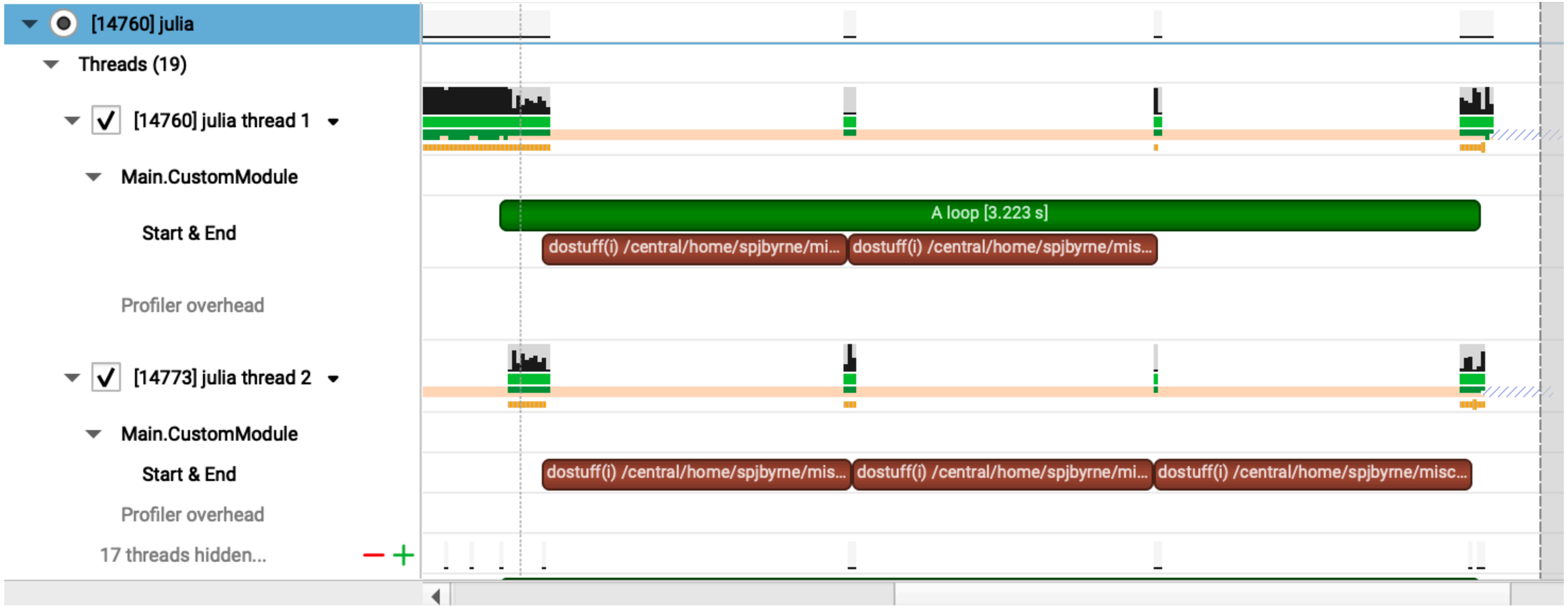
Annotating Julia code

- Basic macros
  - `NVTX.@mark`: Instruments an instantaneous event
  - `NVTX.@range expr`: Instruments a code block with a range
  - `NVTX.@annotate function … end`: Instruments a function definition with a range
- Can attach metadata to each of these
  - `domain` (default: current Julia module)
  - `message` (default: [function name] file:lineno)
  - `color`: (default: generate unique)
  - `payload`: a single integer or float value
  - `category`: enum value

```
module CustomModule

using NVTX

NVTX.@annotate function dostuff(i)
    sleep(1)
end

NVTX.@range "A loop" begin
    Threads.@threads for i = 1:5
        dostuff(i)
    end
end

end
```

Simple example

```
nsys profile --trace=nvtx julia example.jl
```

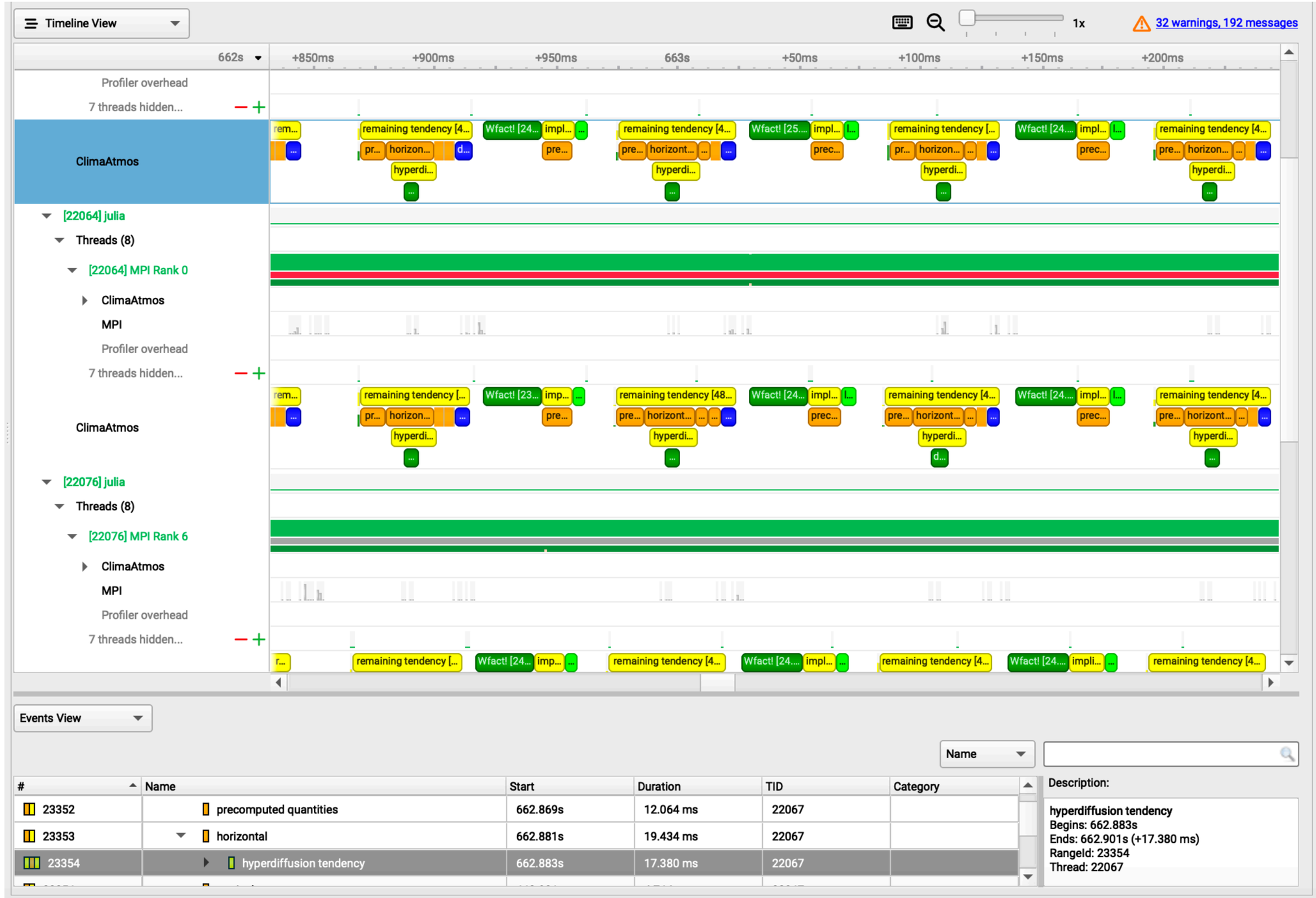


- Can also be invoked from the Julia REPL

Nsight + MPI

Profiling parallel Julia code

- Instruments common MPI operations using MPI profiling interface
- Can be used in two ways
  - profiler - launcher - program
    - `nsys profile <nsys-args> mpiexec <mpi-args> julia …`
    - Generates single profile
    - Only works if all processes are on the same node
  - launcher - profiler - program
    - `mpiexec <mpi-args> nsys profile <nsys-args> julia …`
    - Generates a profile per rank
    - Can be opened as a “multi-profile view”
- Profiler itself has overhead
  - Recommend allocating additional CPU core per profiler instance + launch with core binding


Timeline View
1x
32 warnings, 192 messages
662s
+850ms
+900ms
+950ms
663s
+50ms
+100ms
+150ms
+200ms
Profiler overhead
7 threads hidden...
ClimaAtmos
[22064] julia
Threads (8)
[22064] MPI Rank 0
ClimaAtmos
MPI
Profiler overhead
7 threads hidden...
ClimaAtmos
[22076] julia
Threads (8)
[22076] MPI Rank 6
ClimaAtmos
MPI
Profiler overhead
7 threads hidden...
Events View
Name
Name Start Duration TID Category
23352 precomputed quantities 662.869s 12.064 ms 22067
23353 horizontal 662.881s 19.434 ms 22067
23354 hyperdiffusion tendency 662.883s 17.380 ms 22067
Description:
hyperdiffusion tendency
Begins: 662.883s
Ends: 662.901s (+17.380 ms)
RangeId: 23354
Thread: 22067

More on profiling

- Alternatives:
  - Score-P/ScoreP.jl (https://github.com/JuliaPerf/ScoreP.jl) by Carsten Bauer
  - Intel VTune/IntelITT.jl (https://github.com/JuliaPerf/IntelITT.jl) by Valentin Churavy
- Recent effort at instrumenting the Julia runtime
  - Compilation, task switching, etc.
  - Requires a custom build of Julia
  - Currently supports Tracy & Intel ITT
- Future work
  - Develop a common instrumentation API

Julia + MPI Pitfalls

My Julia HPC gripes

1. MPI library quirks

Every MPI implementation is weird in its own way

- MPI libraries do lots of weird things
  - Intercept signals, malloc, dynamic libraries
  - GPU-aware interfaces make everything worse
- Julia is good at triggering these issues
  - `ccall` uses `dlopen`, multithreaded runtime
- Can often be worked around (e.g. with environment variables)
  - See https://juliaparallel.org/MPI.jl/stable/knownissues/
- MPI launchers have poor support for interactivity
  - No parallel Julia REPL 😦

2. Shared file system woes

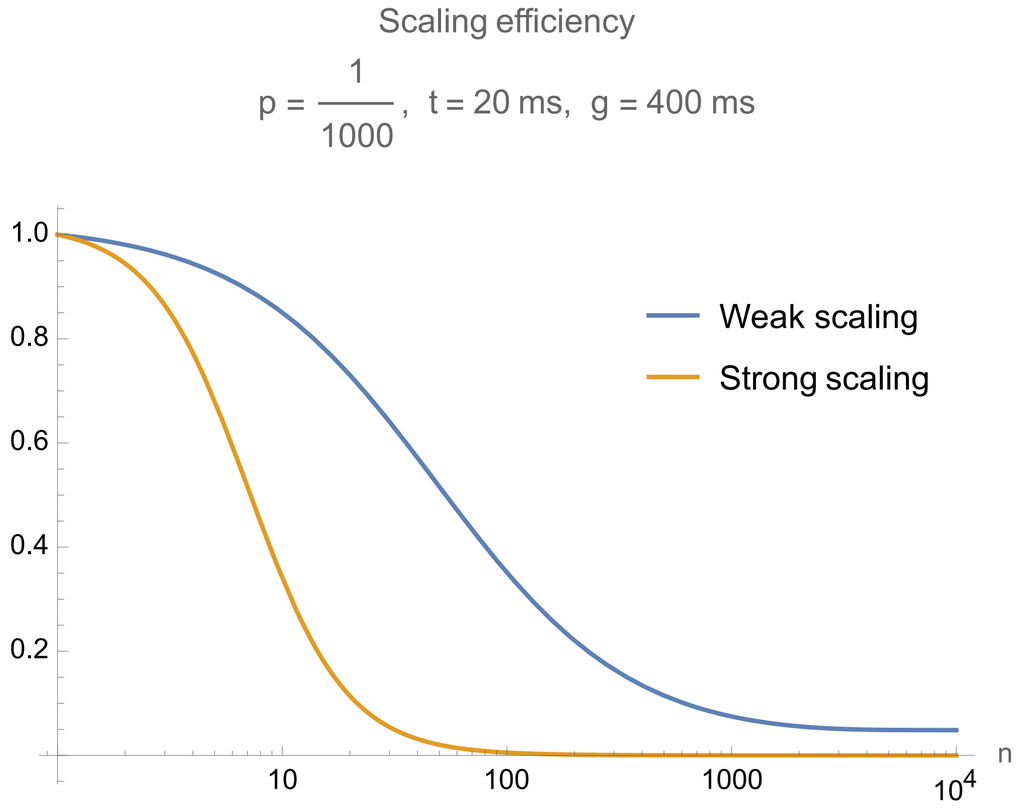
- **Performance issues**
  - User home directories are mounted via NFS: set your depot to be on a fast disk
  - DDOS-ing your parallel file system metadata server: system images, containers, scratch partitions
- **Incompatibilities**
  - Typically don't allow memory mapping (mmap): JLD2.jl, use `jldopen(… ; iotype=IOStream)`
- **Race conditions**: multiple processes/multiple jobs trying to write at the same time
  - e.g. Pkg operations, precompilation, artifact downloading
  - Recommend `Pkg.instantiate()` on a singleton process
  - Use a clean depot on top (see DepotCompactor.jl)
- **Disk usage**: on systems with low limits
  - Use a shared depot of Julia artifacts

3. Compilation complaints

- JIT for HPC is extremely powerful
  - Benefits: generating specialized code for your particular problem (especially for GPU)
  - Downside: compiling specialized code for your particular problem
- Much recent work compiler work on reducing latency
  - Time-to-first-X (TTFX)
- Critical to avoid method invalidations
  - SnoopCompile.jl and related tools
- Native code caching (system images, pkgimages on Julia 1.9)
  - Complicated on a heterogeneous cluster
  - Caching for GPU kernels a work-in-progress

4. MPI + GC

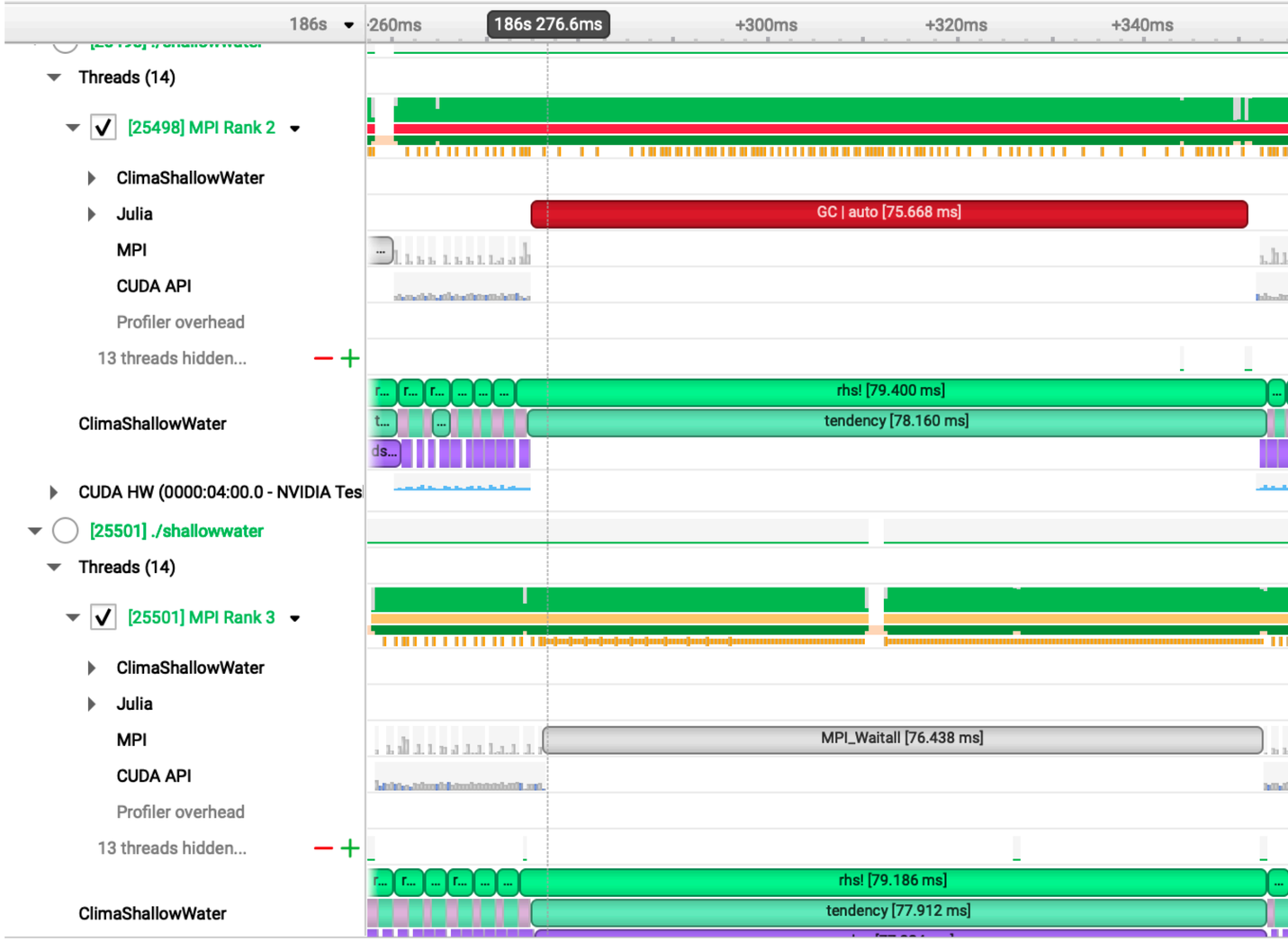
- All other processes stuck waiting on a process calling the GC
  - More MPI ranks => higher probability of a rank invoking GC
- Consider an iteration of compute + all-to-all communication
  - t = time per iteration without GC (assume perfect scaling)
  - g = time per GC pause
  - p = probability of GC invocation per iteration on each processor
    - Assuming each processor independent
  - n = number of processes
- Average time per iteration
  - Single process: $t + gp$
  - Weak scaling regime: $t + g(1 - (1 - p)^n)$
  - Strong scaling regime: $\frac{t}{n} + g(1 - (1 - p)^n)$



Identifying the problem

Profiling the Julia Garbage Collector

- Load NVTX.jl, set `JULIA_NVTX_CALLBACKS=gc`
  - Activates a hook into the Julia garbage collector
- Look for ranks blocking on MPI operations



What can be done?

1. Reduce memory allocations wherever possible
   - A good idea anyway!
     - See Julia docs on memory allocation profiling
   - Not always possible to eliminate entirely: many small memory allocations
     - e.g. CUDA kernel launches, saving output, log printing
2. Disable the garbage collector: `GC.enable(false)`
   - Feasible once allocations are sufficiently small
3. Synchronize calling GC on all processes
   - Manually call `GC.gc()` at set intervals
   - Requires tuning heuristics for a particular workload

Summary

Summary

- Julia is a very powerful language for scientific computing
  - Fast, expressive, multithreading, cross-platform, excellent GPU support
  - Easy-to-use MPI support
- But you need to develop a good understanding of the runtime
- You can only optimize what you can measure
  - Profiling is essential
- Julia HPC has a great community
  - Slack
  - Discourse
  - JuliaHPC monthly call