

Containerized GEOS: Toward a Portable Climate Model

Matthew Thompson^{1,2}, Anik Mumssen², Zain Jerath²

¹Global Modeling and Assimilation Office, NASA Goddard
Space Flight Center

²Science Systems and Applications, Inc. (SSAI)

What is GEOS?

The **Goddard Earth Observing System (GEOS)** is an Earth System Model (ESM) developed by the Global Modeling and Assimilation Office (GMAO) that consists of a group of model components connected in a flexible manner in order to address questions related to different aspects of Earth Science.

GEOS used the modular architecture of the Earth System Modeling Framework (ESMF) to connect components together. This modular structure allows the code to be built with a variety of different physical models such that different radiation, moist microphysics, land physics and other schemes can be selected at run time.

GEOS itself consists of many Git repositories which uses a GMAO-developed tool, mepo, that “constructs” a source code tree that is the built using CMake and run using various shell scripts.

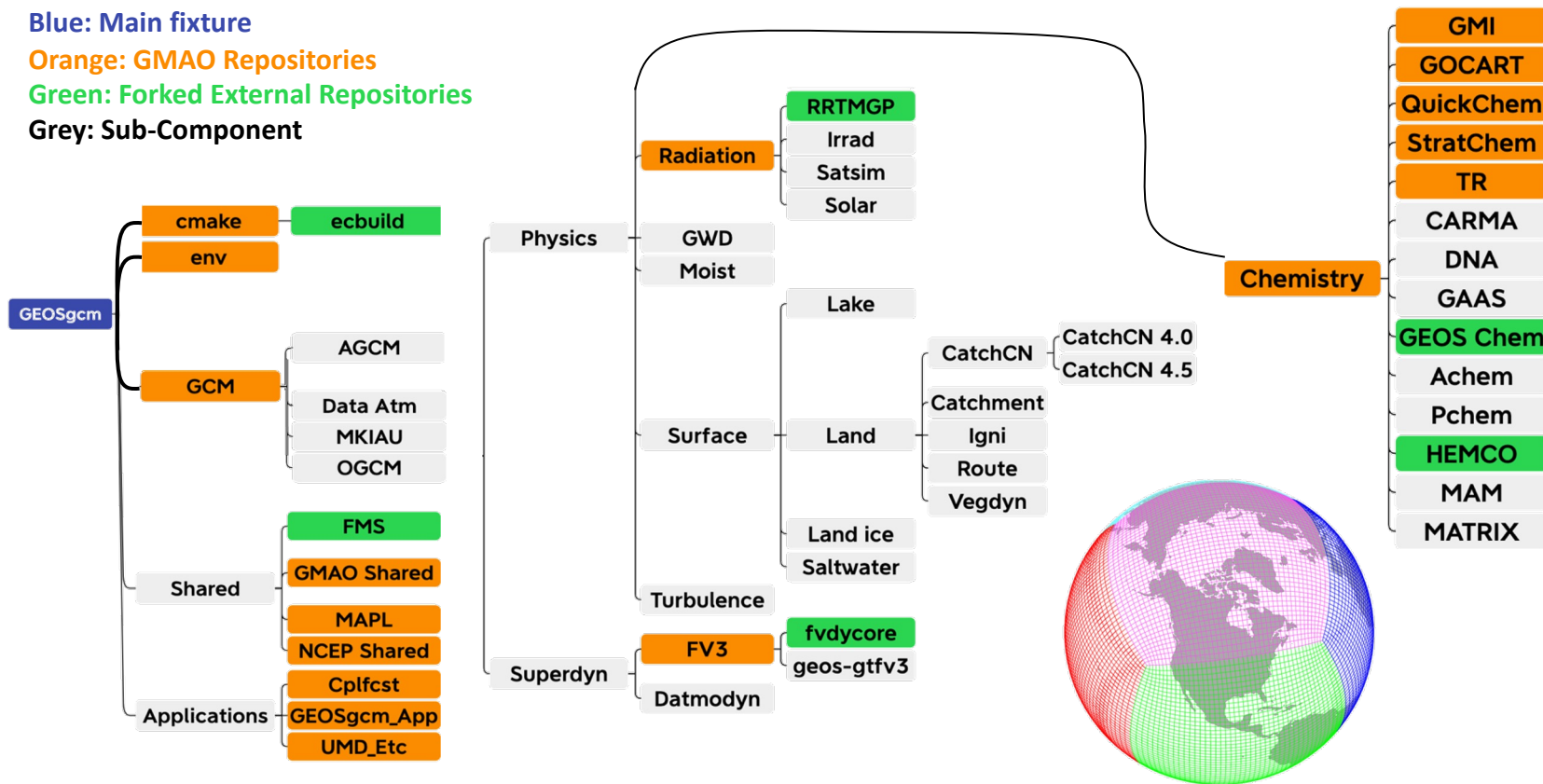
GEOSgcm

Blue: Main fixture

Orange: GMAO Repositories

Green: Forked External Repositories

Grey: Sub-Component



Thanks to Bill Putman for the figure.

Containerized GEOS: Toward a Portable Climate Model

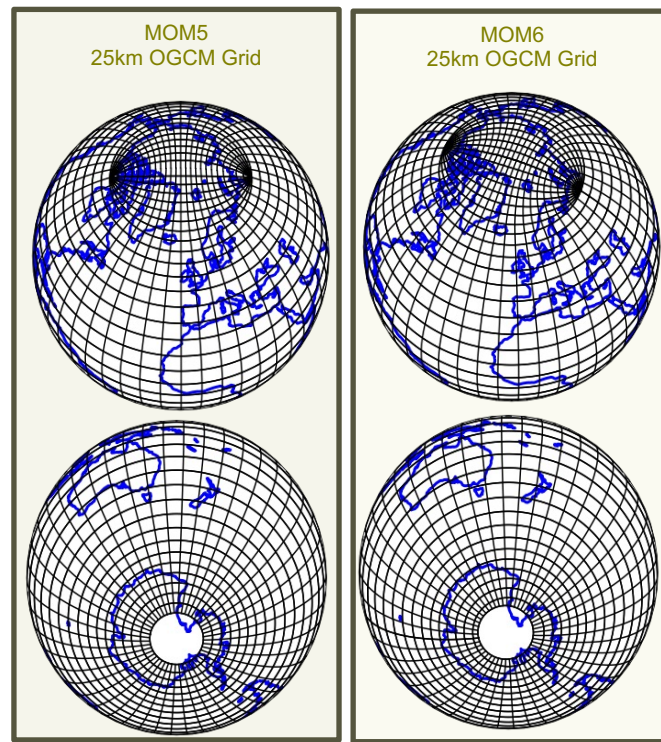
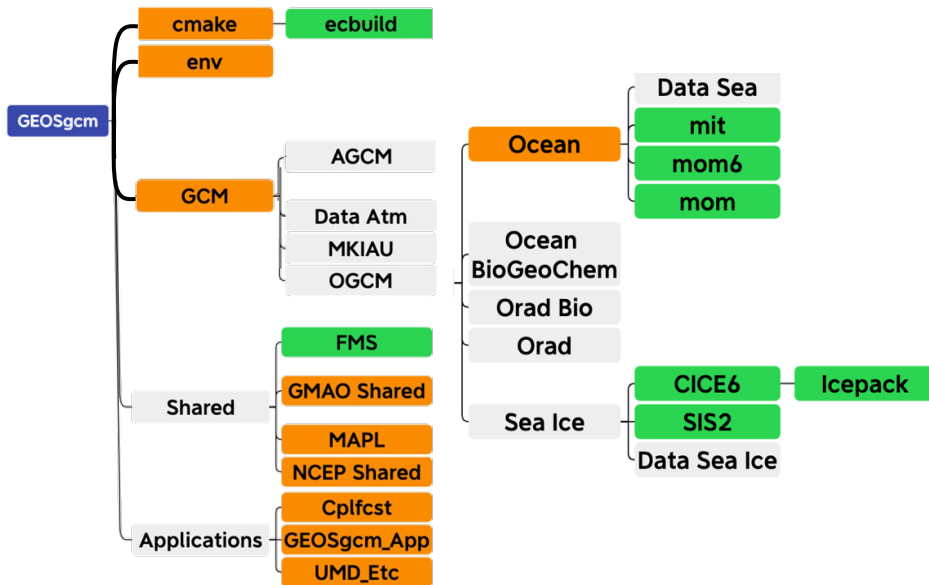
GEOSgcm

Blue: Main fixture

Orange: GMAO Repositories

Green: Forked External Repositories

Grey: Sub-Component



So Many Dependencies...

GEOS is a large earth modeling system that depends on Fortran compilers, MPI, libraries and can be quite difficult to port to new machines and operating systems. Developers often need to work with system administrators to get the require system libraries, etc. installed on a machine before runs can even begin (green in ESMA-Baselibs):

- git
- cmake
- python3
- numpy
- tcsh
- C compiler
- C++ compiler
- Fortran compiler
- MPI stack
- BLAS
- LAPACK
- MKL (for some routines)
- mepo
- zlib
- szip
- libjpeg
- curl
- HDF4
- HDF5
- netCDF-C
- netCDF-Fortran
- ESMF
- gFTL
- gFTL-shared
- pFlogger
- fArgParse
- pFUnit
- xgboost
- And any dependencies these bring in!

But there is Another Way: Containers!

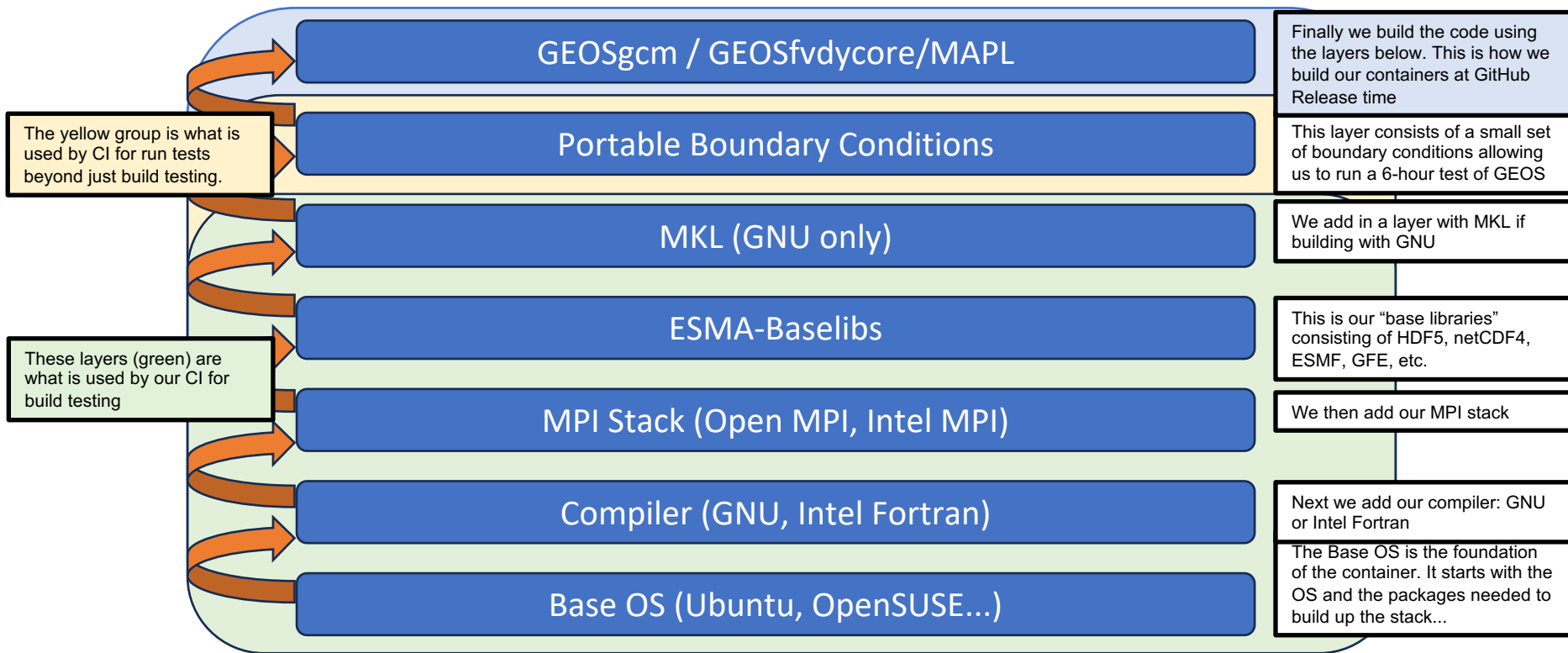
Why Containers?

Containers are a form of virtualization to make portable, versioned applications

- Portability: Containers can run on a variety of hardware and operating systems as long as they support the container runtime
- Isolation: Containers are isolated from the rest of the system. They still consume resources, but multiple containers cannot interfere with each other or native running applications.
- Consistency: Containers allow users to depend on a consistent set of libraries and applications.

For GEOS, we use multi-stage Docker builds using multiple Dockerfiles to allow for flexibility in updating different software layers. This way, we don't need to rebuild the whole stack to update

Anatomy of a GEOS Container



How We Make GEOS Containers

There are two ways we make our GEOS containers: on-demand and on-release

- On-demand containers are built by hand using Docker or the NCCS GitLab Container Builder (a behind the firewall service that allows builds as Docker is not allowed on clusters) This method is used for testing and development purposes and for our CI containers which don't include the full application, but are used to build the application when, for example, pull requests are made on our GitHub repositories.
- On-release containers are, as they are named, built when a GitHub release is made for the application by GitHub Action. At the moment, we build three different application containers on release:
 - GEOSgcm: The main GCM (global circulation model) consisting of our dynamics and physics enabling runs of atmospheric or coupled atmospheric-ocean models
 - GEOSfvdycore: A smaller application consisting solely of our cubed-sphere dynamics model. This allows users to run and develop just the dynamics using a set of test cases
 - MAPL: MAPL is GEOS' coupling layer based on ESMF. This container has been used for training purposes on cloud services

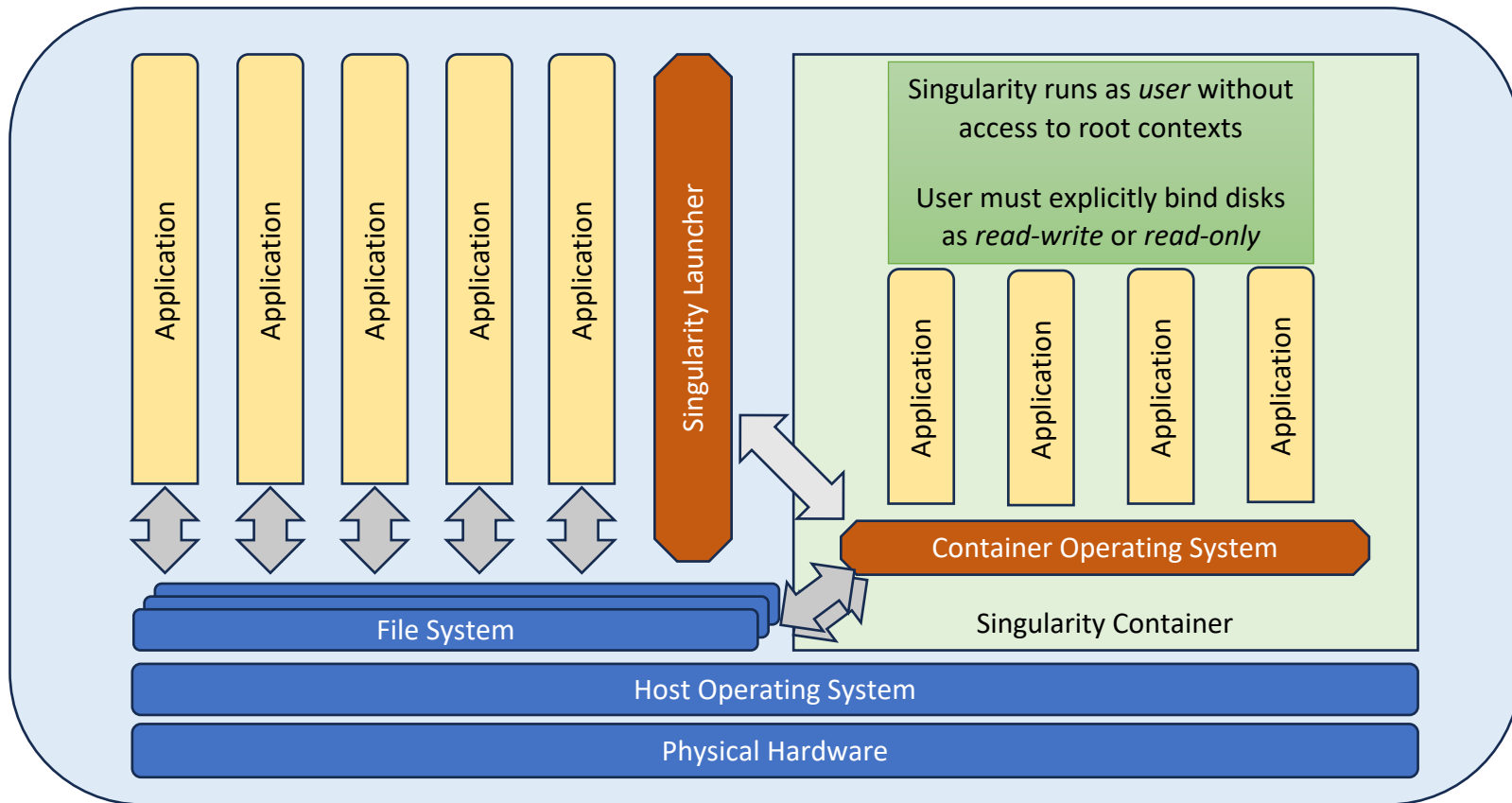
What is Singularity?

The most popular and well-know containerization runtime is Docker. Docker is a portable and runs on a vast array of different platforms. However, its use in HPC is limited because Docker requires and exposes root privileges on a system, which is a no-go on most supercomputing environments.

An alternate container runtime is Singularity developed by Sylabs, Inc. Singularity was developed for HPC with the following benefits:

- Does not require root privileges and runs in user-space, limiting any security concerns (and helping save users from themselves)
- Singularity containers run as simple child processes allowing them to easily integrate with SLURM and other schedulers (no daemon necessary)
- Can be executed using a host network allowing containers to run MPI applications
- Has its own file formats but can build Dockerfiles and pull and run Docker containers

Singularity Container



Making Running Containers Easy

Setting up and running GEOS under Singularity requires extra steps. Work over two summers by SSAI interns helped automate GEOS setup and run scripts so Singularity use is transparent to users. Changes included:

- Automatic detection if our setup scripts are run inside a container by looking for the `.singularity.d` directory present only in Singularity sandboxes
- Running under Singularity requires users to bind paths on the external to the system to the container. When a Singularity build is detected, the setup script automatically constructs the necessary bind paths to allow both read-only access to boundary conditions as well as read-write access to experiment directories
- Loads the Singularity module and sets up the correct `singularity exec` commands for all executables
- Alters MPI environment settings that act differently between native and container runs

Note that users running in Native mode never see any singularity in their run scripts. This was done to not cause confusion as support was added.

Benchmarking Native v Containers

Runs made on the Cascade Lake nodes of the NCCS Discover cluster

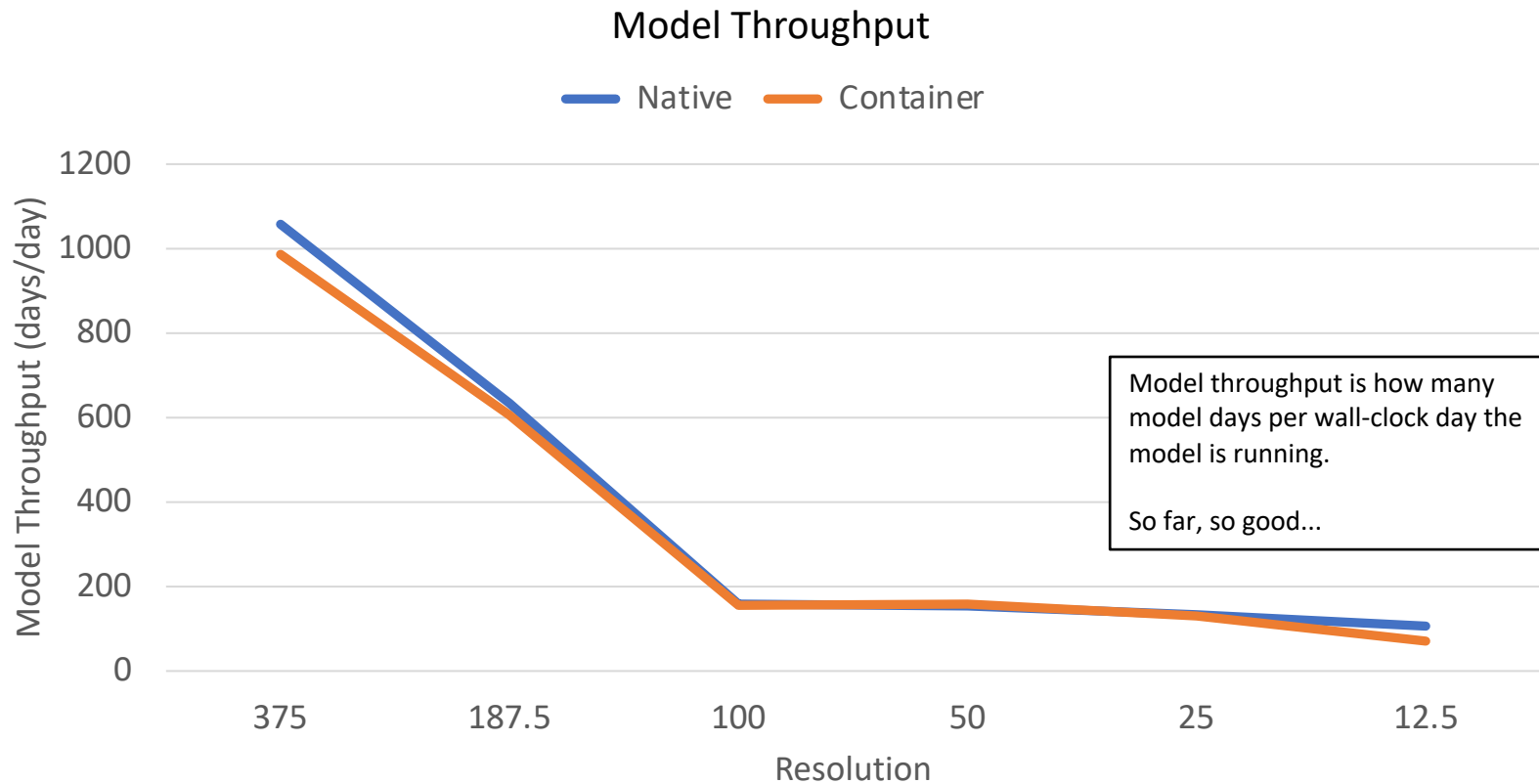
- Native OS: SLES 12.3
- Container OS: Ubuntu 20
- Compiler: Intel Fortran 2021.6.0
- MPI: Intel MPI 2021.6.0

Resolutions range from 375-km (C24, 96 processes) to 12.5-km (C720, 3456 processes)

- At resolutions of 50-km and higher, we use extra nodes to handle our output to allow for asynchronous IO. Both Native and Container used these extra nodes.
- Multiple runs were made, but resulting times were close enough that any error bar was essentially the width of the line, so they are omitted

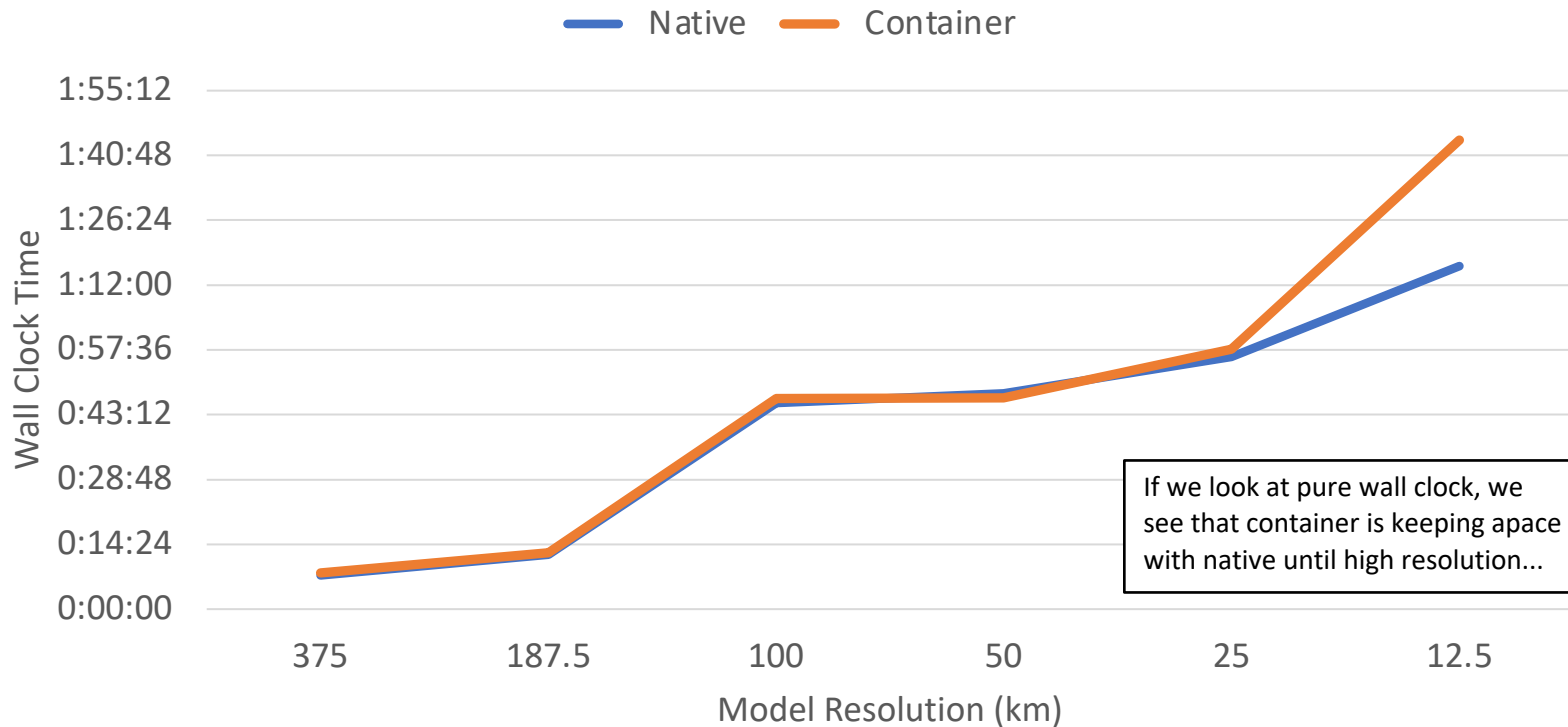
Here comes the charts! 

Native v Container Results

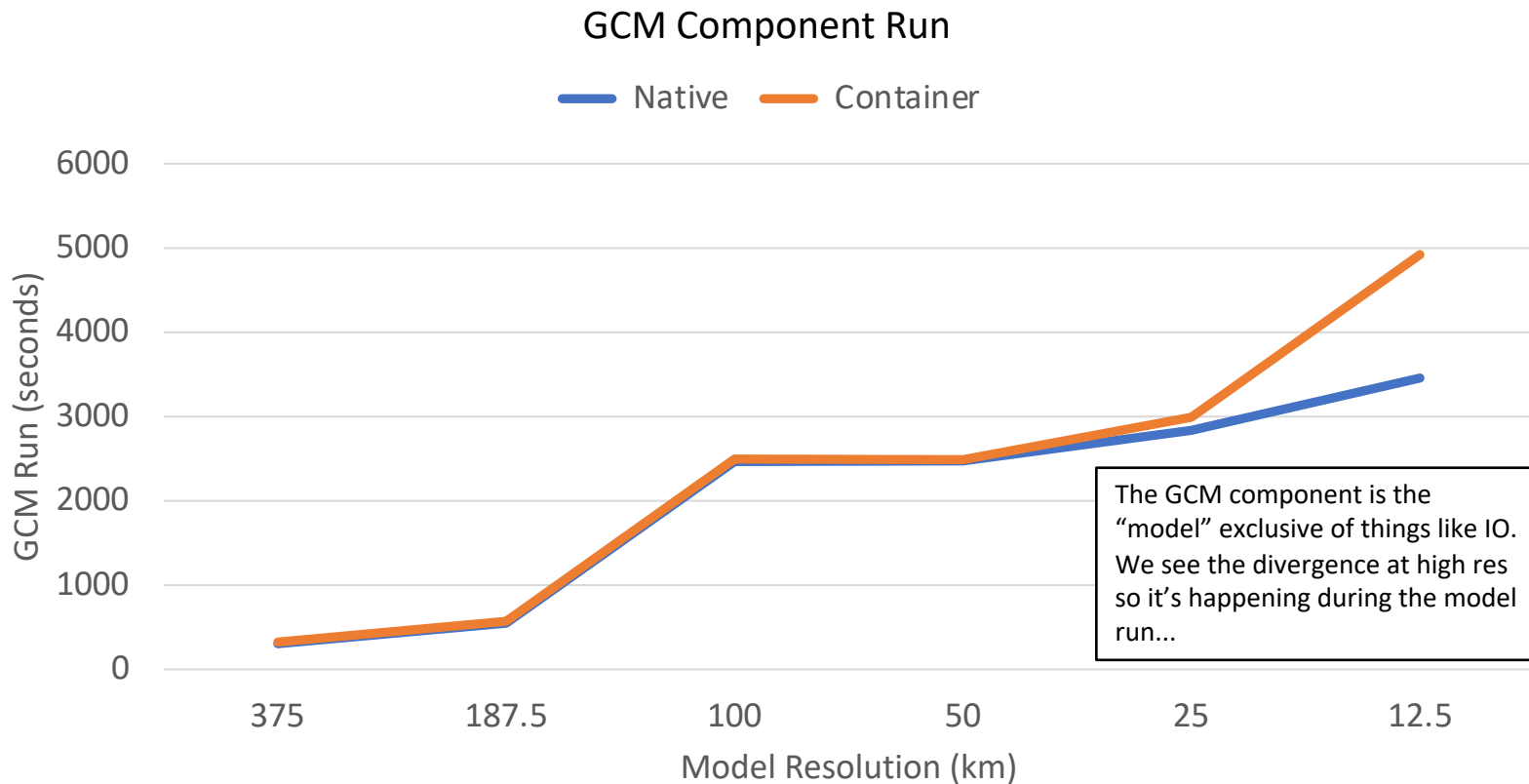


Native v Container Results

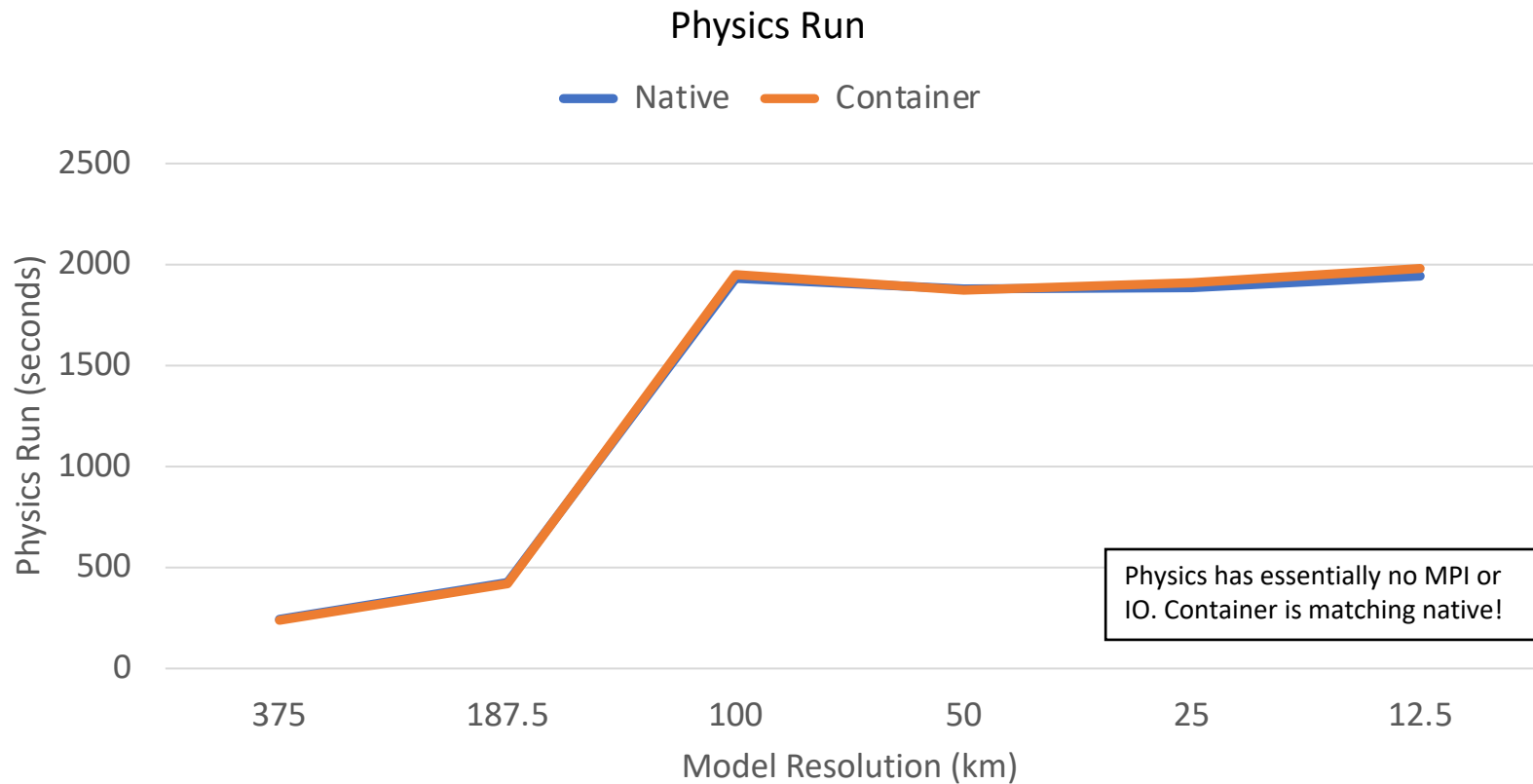
Wall Clock Time (5-Day Model Run)



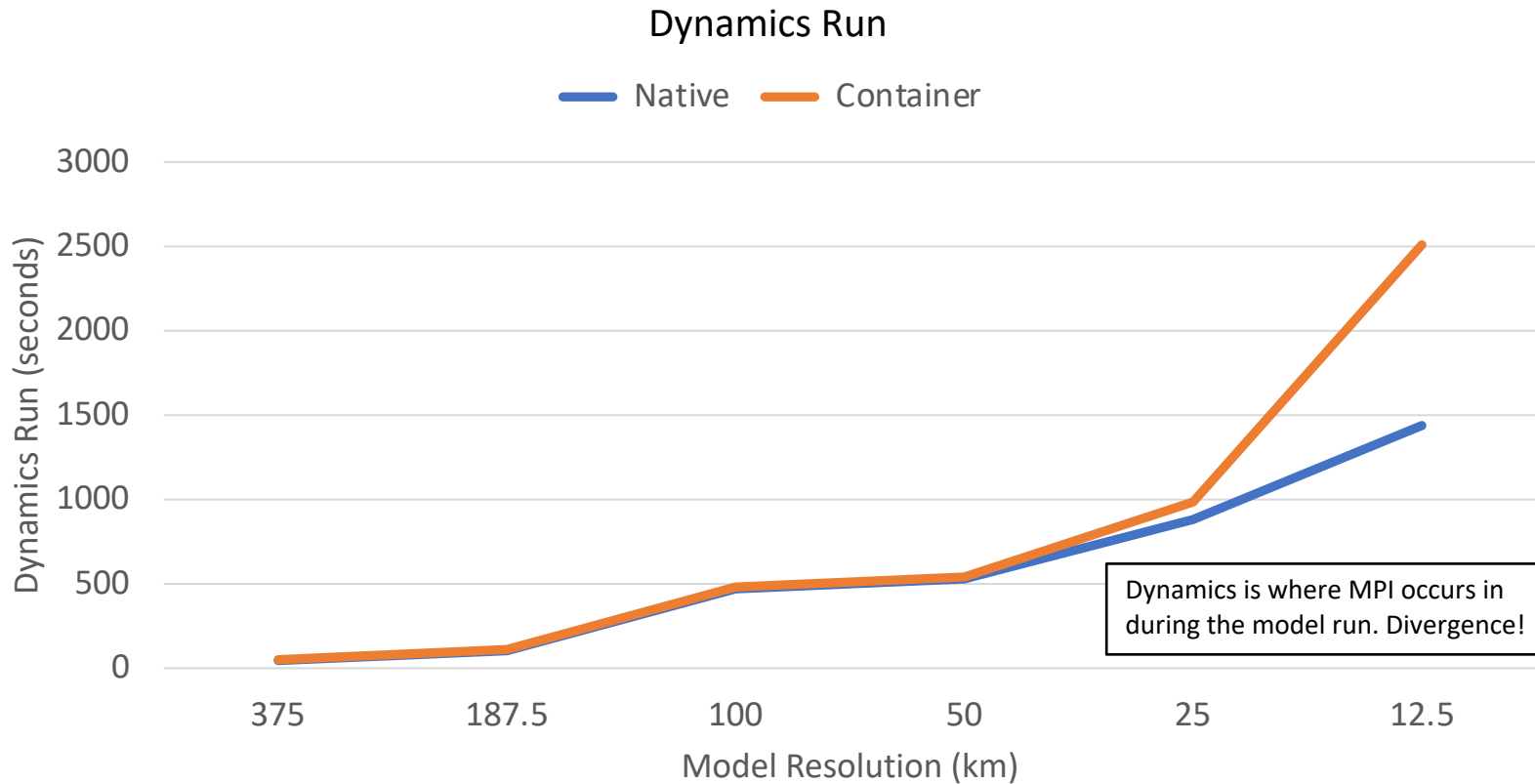
Native v Container Results



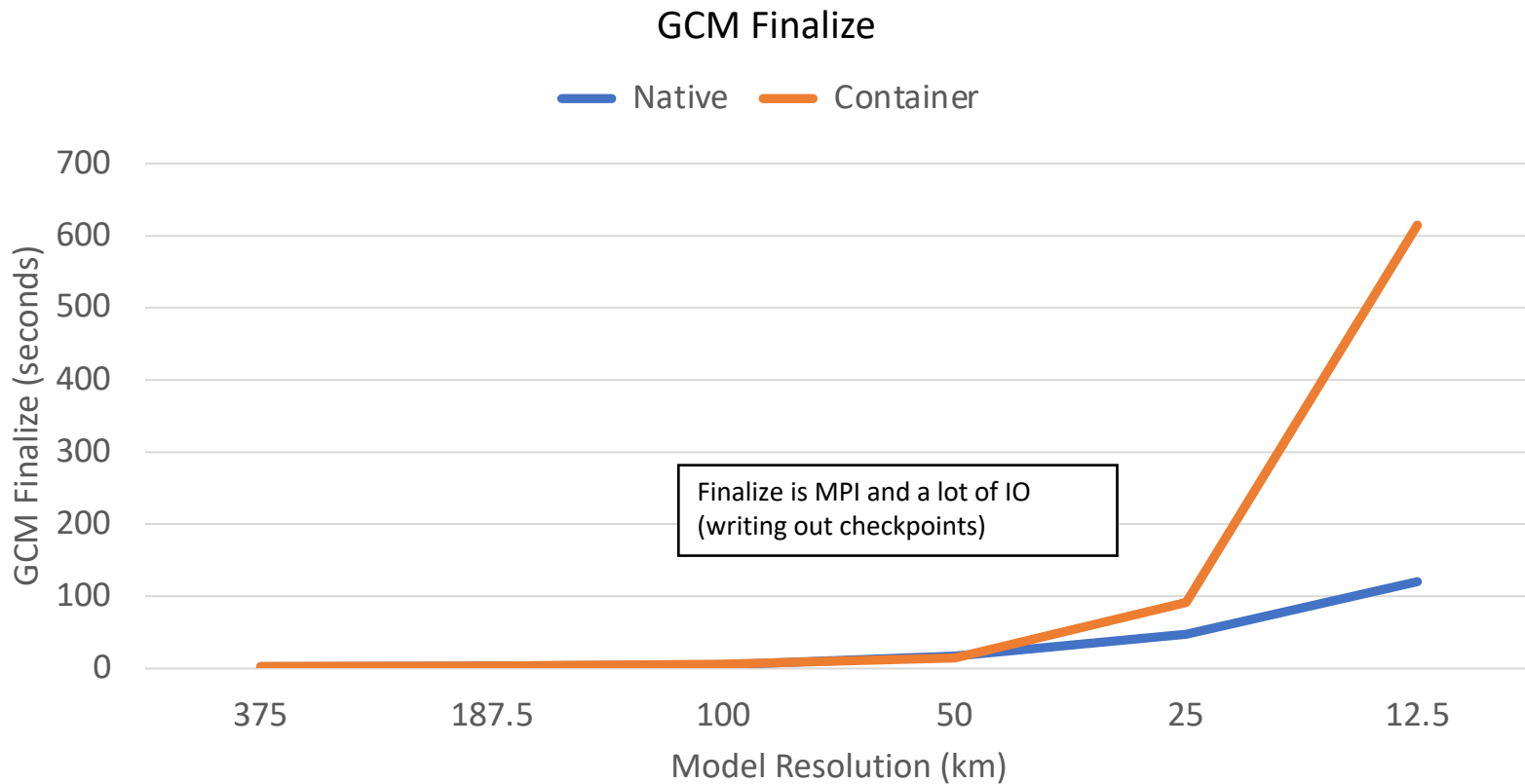
Native v Container Results



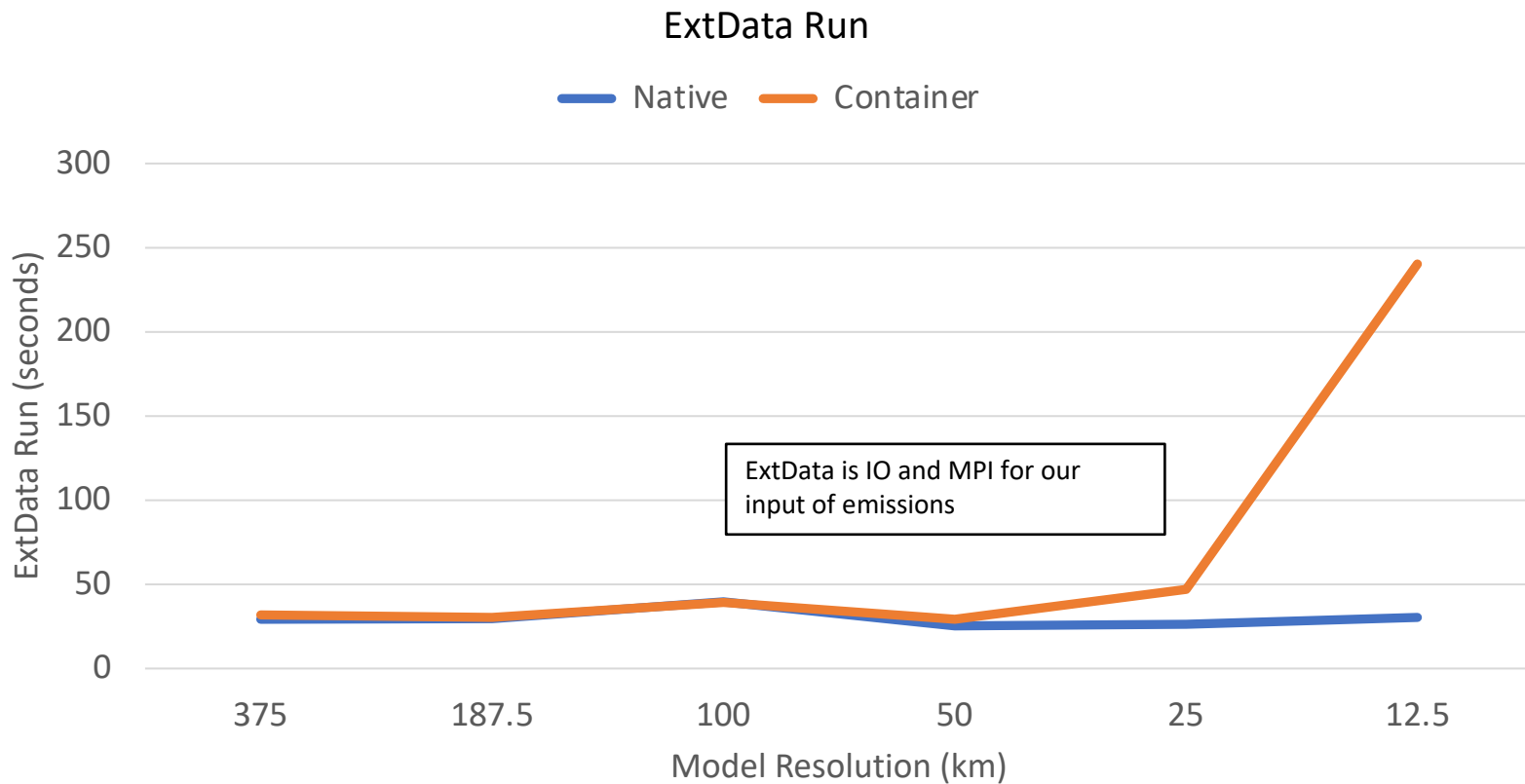
Native v Container Results



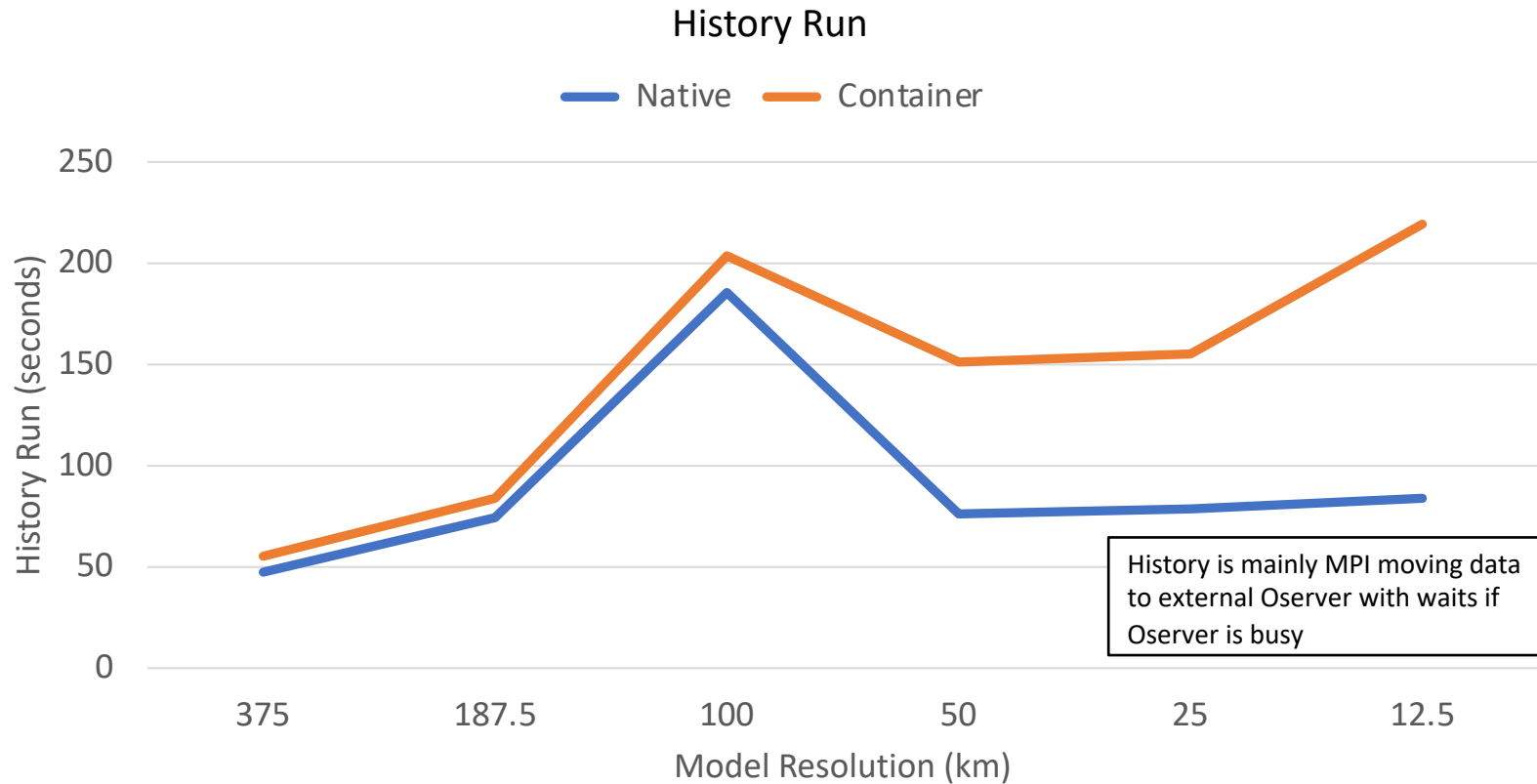
Native v Container Results



Native v Container Results



Native v Container Results



Native v Container Takeaways

The Good!

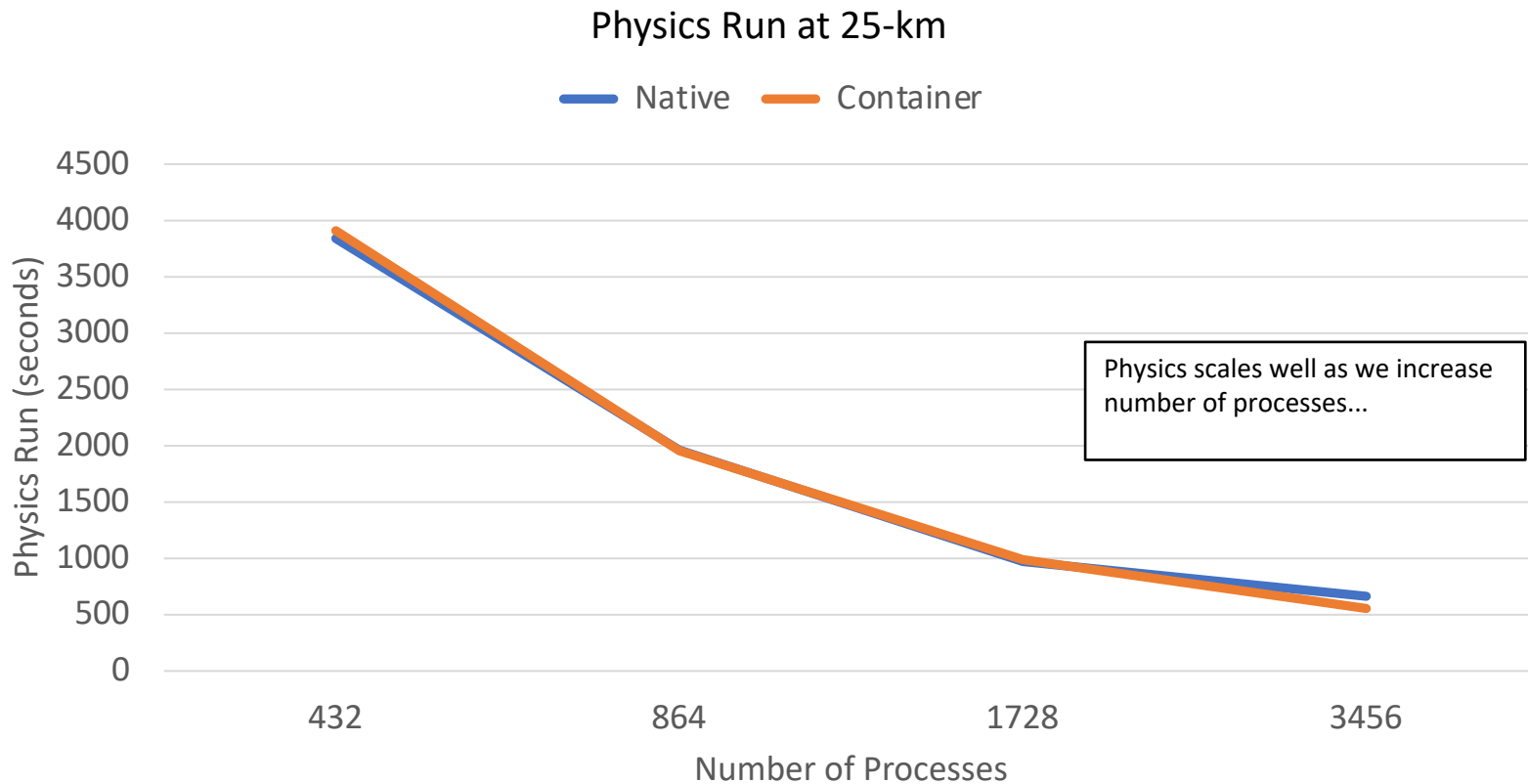
- The containerized model runs and is nearly as performant as the Native model
- Testing shows the results are **zero-diff** even though we are running under a different OS! (In the past, OS upgrades often triggered differences.)
- We have also demonstrated running GEOSgcm at very-low-res using the baked in Portable Boundary Conditions in the Docker image for portability

The Bad!

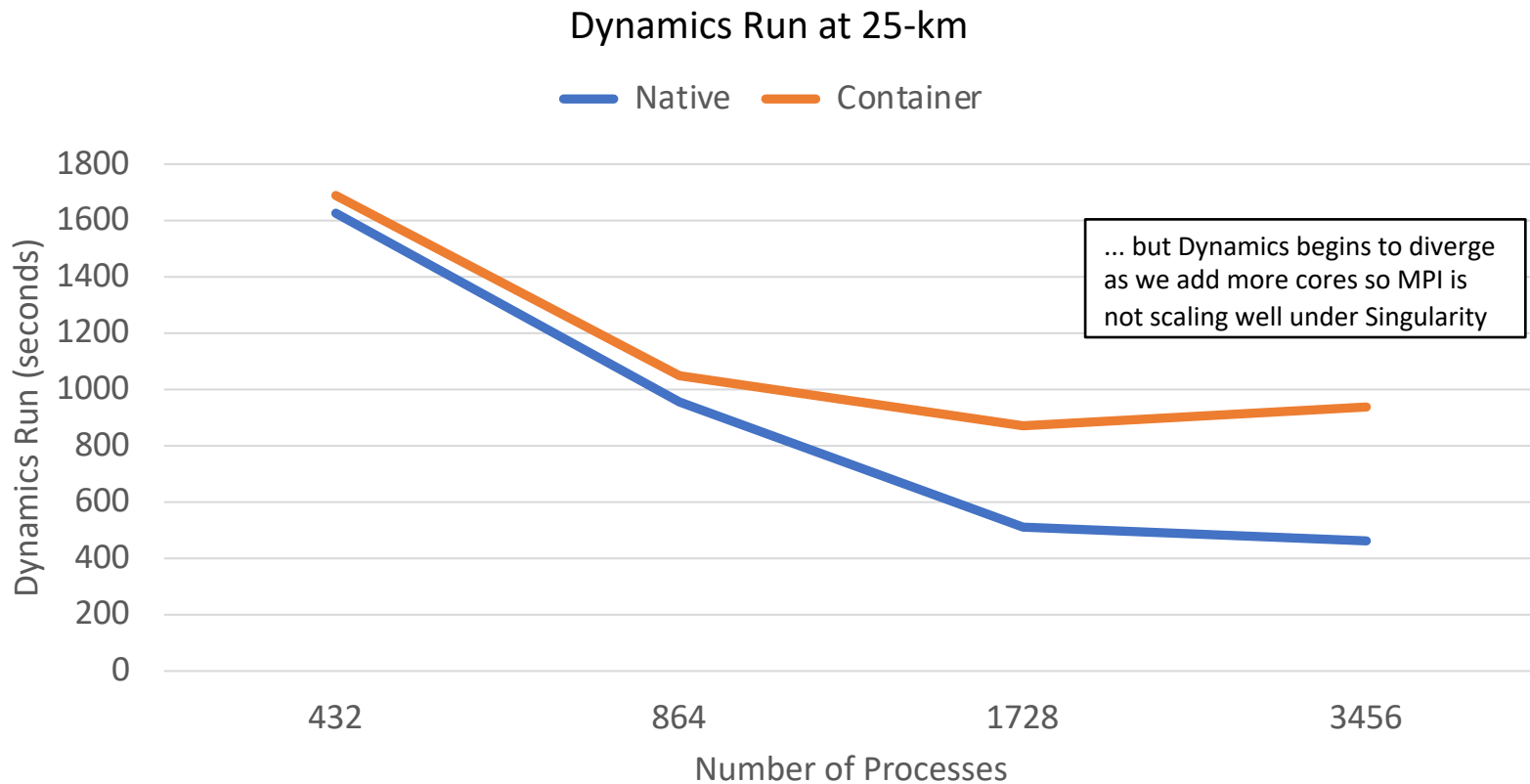
- There seems to be some sort of divergence at high-resolution. Seems like it is related to MPI and I/O.

But with higher resolution comes more processes...what if we try keeping model size the same but change processor count. Here comes more charts! 

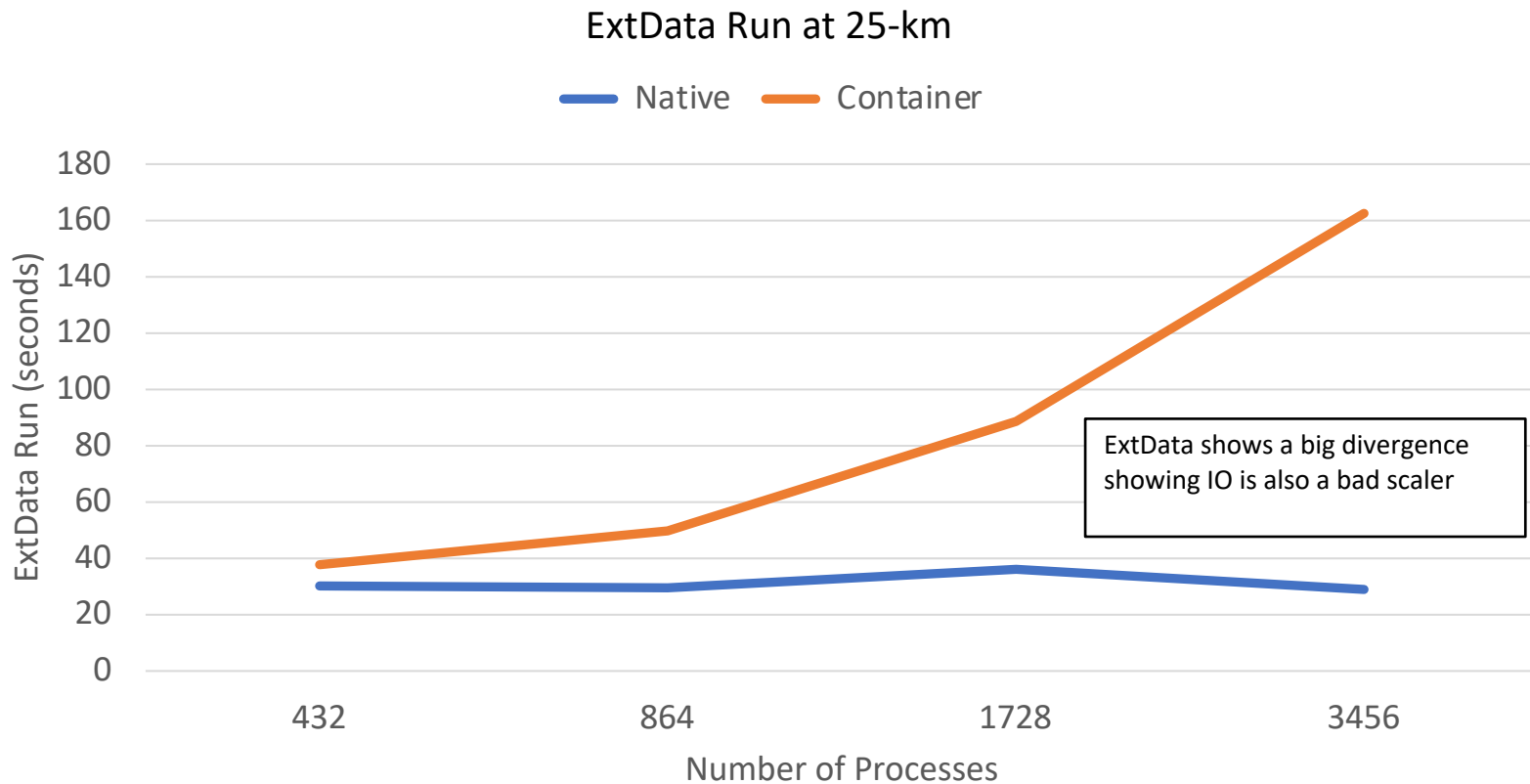
Native v Container 25-km Scaling



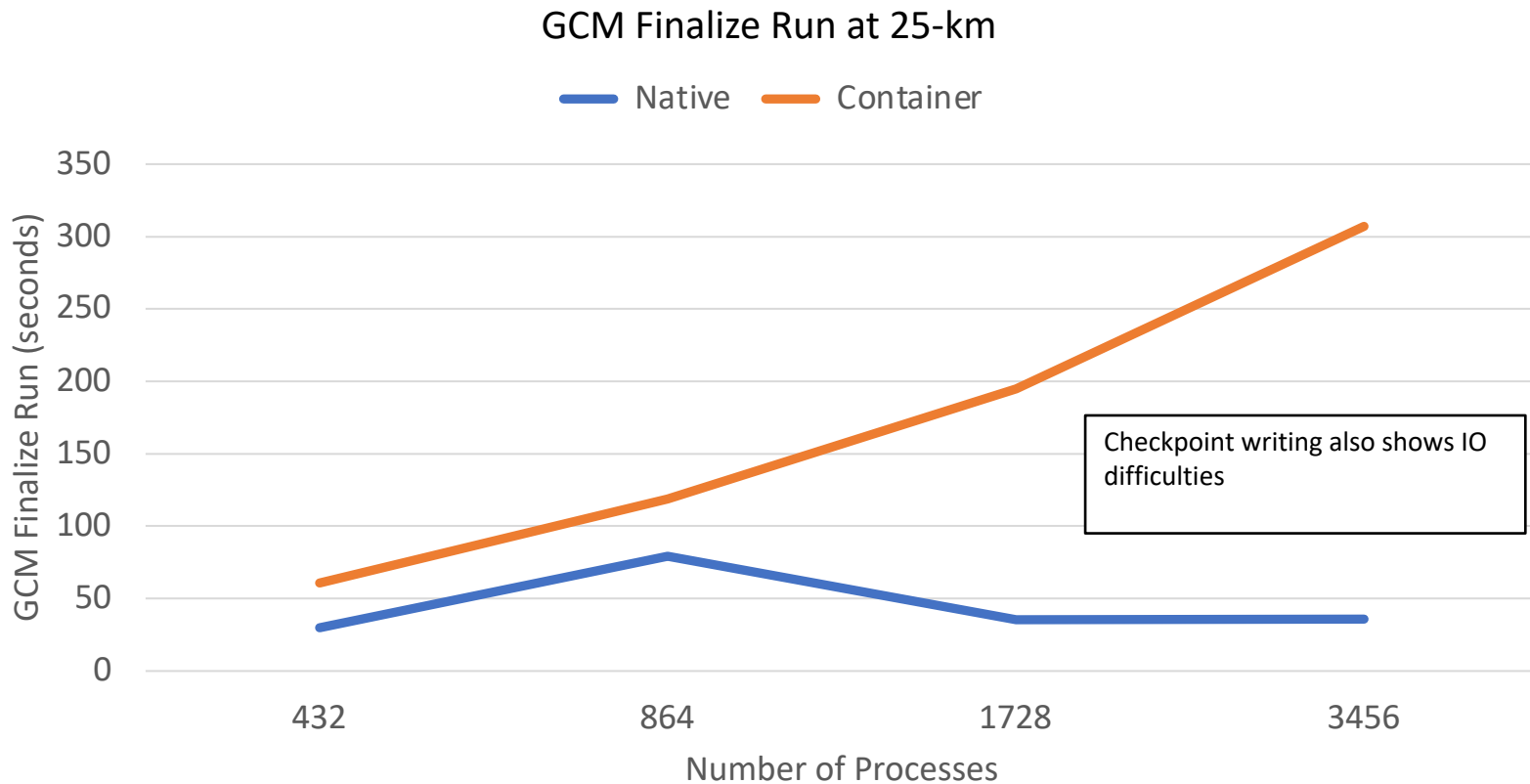
Native v Container 25-km Scaling



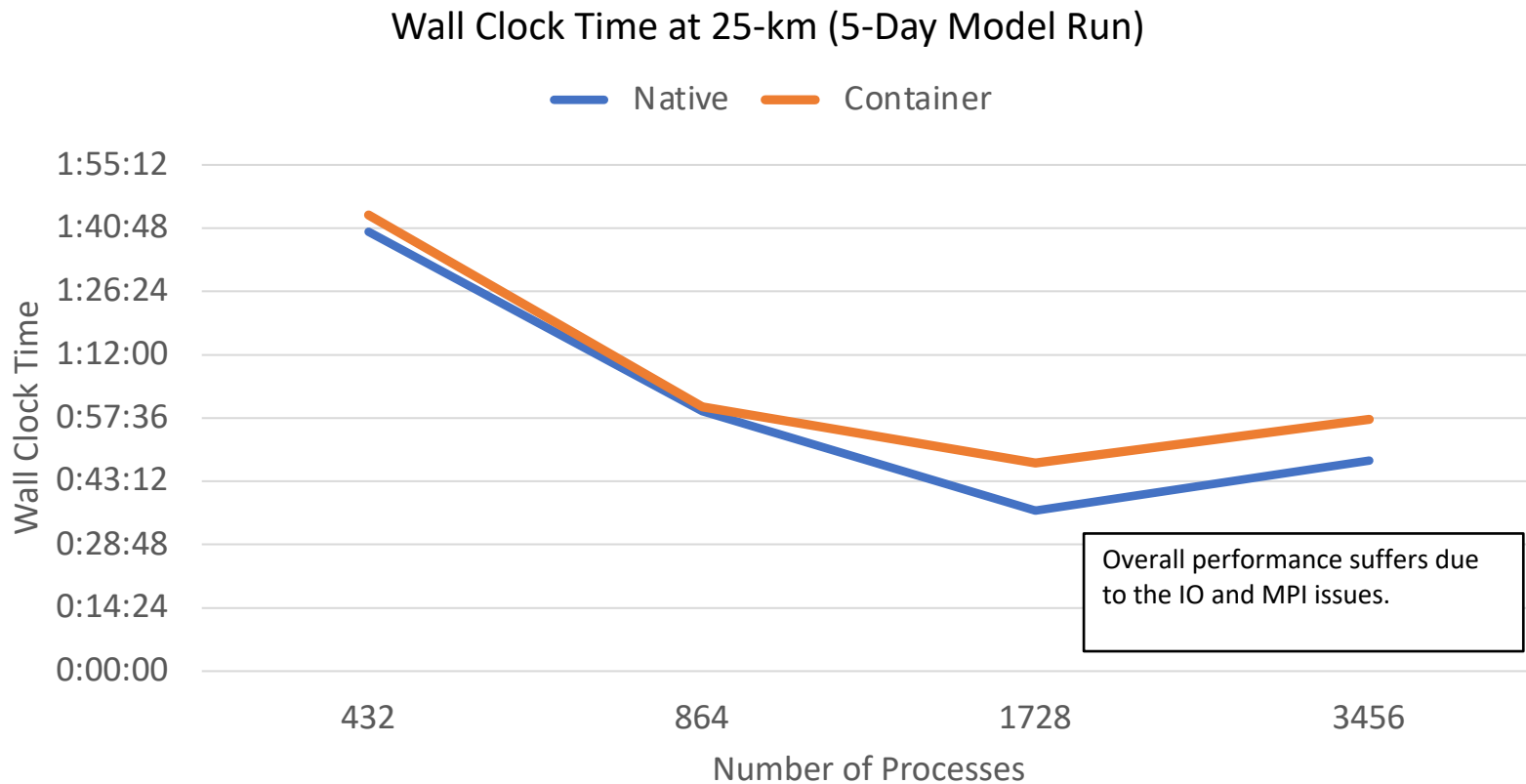
Native v Container 25-km Scaling



Native v Container 25-km Scaling



Native v Container 25-km Scaling



Moving Forward: Issues and Challenges

Large Runs on Non-NCCS Machines

- We can run large jobs at NCCS because we can use Singularity path binding to point to our complete boundary conditions
- This is potentially doable at NAS (being explored now) as all BCs are mirrored there
- On cloud providers like AWS or Azure, data ingress and storage is a significant cost: How do we address this?

Runs on Non-NASA Machines

- We have demonstrated running GEOSgcm at very-low-res using the baked in Portable Boundary Conditions in the Docker image

Scaling at Large-Core Count

- Need to investigate why IO and MPI fall off as core count increases
 - Issues with Singularity and MPI not “seeing” the same thing as Native?
 - Issues with GPFS as we scale? The container probably does not know about GPFS

Moving Forward: Issues and Challenges

Lustre

- These runs were done on GPFS, but NAS and most cloud services use Lustre. What will that show

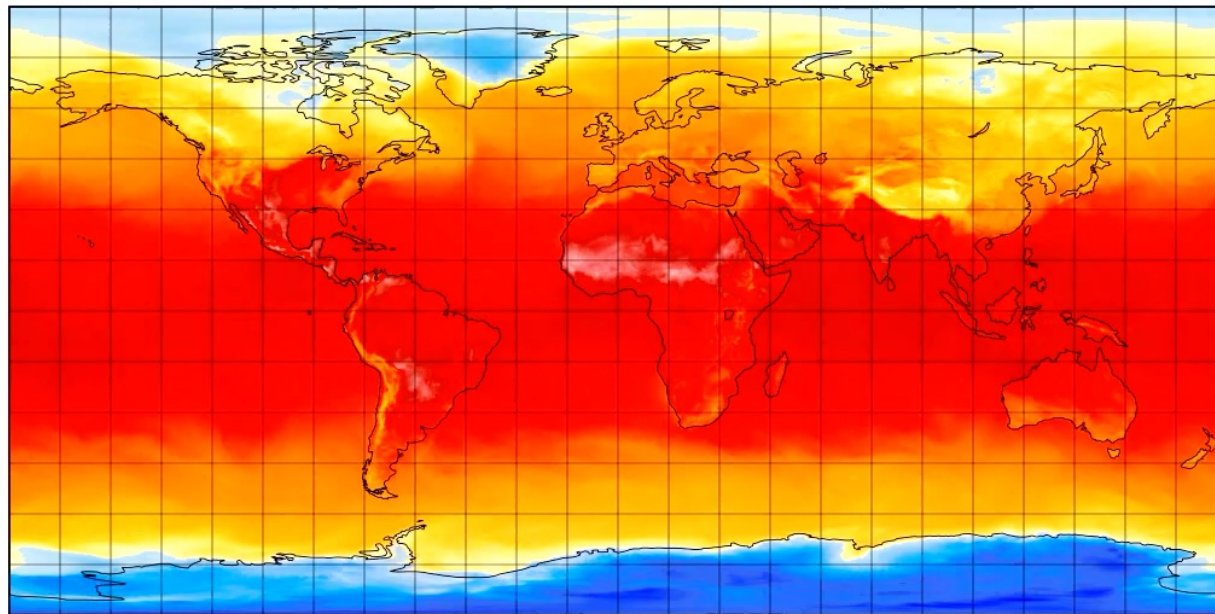
Open MPI

- Tests to duplicate the Intel-Intel MPI runs previously show fail because Open MPI is not as flexible. The internal Open MPI needs to be built exactly as the system Open MPI (pmix, Infiniband, etc.)
- This could be done...but then would that work on non-Infiniband systems?
 - Maybe this isn't a big issue as all cloud systems are Infiniband
 - We will experiment with using internal pmix, etc.
- Important as currently on our new AMD Milan system, we use Open MPI for Intel (as Intel MPI + Milan has led to bad behavior at high-res)

Run of Containerized GEOS at 25-km

30-day Run of Containerized GEOS

Time: 2015-04-14 21:30:00



2-meter air temperature (°F)



Data Min = -82.6, Max = 100.4, Mean = 56.9