Porting the NOAA global weather forecast system (GFS) to the cloud

Daniel Abdi, Chris Harrop, Jebb Stewart, Mark Govett*

CIRES, NOAA*

Abstract

We ported the NOAA Global Forecast System (GFS) that is often run on-premise to different Cloud Computing Platforms (CCPs) such as AWS and GCP. First, the application is containerized using Docker and Singularity for ease of portability and reproducibilty. We demonstrate that using containers does not affect performance significantly as compared to bare-metal runs. Singularity does not pose the security risks of Docker and also offers better integration with job schedulers such as SLURM. One downside we found regarding portability of multi-node runs is the need to match the MPI library on the host with that inside the container, including drivers used for InfiniBand. GFS uses a workflow management system called rocoto that pauses additional complications when used along with containers. The scripts that rocoto calls contain slurm commands, however, our containers do not have SLURM inside them. We solve this problem by writing the SLURM commands into a file, and then a daemon process executes the commands on the host and writes back the output to a file.

Introduction

Porting high-performance scientific applications such as GFS to the cloud often pose major challenges such as requirement of a tightly coupled system with high-speed interconnect and homogeneous compute-units, high storage requirements to store the outputs of the model, secure and efficient network for data ingress and egress, etc.

Containerization

a)Bundling an app with everything it needs to run everywhere exactly the same way. "Build once, run everywhere".
b)Docker is industry-standard. Singularity is preferred on HPC systems because Docker gives escalated privileges to all users of the container
c)Use singularity for deployment and Docker for development.

Motivation and Goals

a) Run GFS end-to-end (pre-processing, forecast, post processing) on the cloud

b) Extend rocoto to support various cloud computing platforms (CCPs)

c) Provide containers (singularity/docker) for GFS and its components for easy deployment to any CCP

d) Evaluate performance of GFS on single and mulit-node runs.

e) Evaluate feasibility of emulating HPC environment in CC e.g. ParallelCluster from AWS

Methodology

1) Currently, the GFS code-base can only be compiled with proprietary Intel compilers, so we faced the prospect of making it compile with free GNU compilers that took a significant portion of our time. Eventually though the GNU build turned out to be 3x slower so the Intel build must be used for production.

2) Testing out various MPI libraries (IntelMPI, MPICH, MVAPICH, OpenMPI). Having this ability is important because running docker/singularity containers on multi-nodes require that the host and container MPI libraries match

3) GFS has many support libraries (around 10 NCEPlibs, ESMF, NETCDF, CRTM, GEMPAK etc). There is no "module load" to save us from compiling these libraries ourselves

4) Multi-node runs with Singularity as

mpirun -np 264 singularity exec fv3_latest.sif /opt/fv3_1.exe

5) Workflow is managed with rocoto which generates a script for a batch job which in-turn invoke SLURM jobs that we replaced with singularity exec calls as in (4). The script that rocoto calls have SLURM commands in them, however, we do not have SLURM installed inside our containers. To solve this problem, we write the commands to a file, which are then read by a daemon process on the host, that executes the SLURM commands on the host and writes back the results. This approach needs minimal modifications to the code as show below, where we call pre- and post- job scripts to initialize and kill the daemon process on the host.

```
{{
#! /bin/sh
#SBATCH --job-name=c48_gfspost001_00
#SBATCH --account=gsd-hpcs
#SBATCH --qos=batch
#SBATCH --nodes=1-1
#SBATCH --tasks-per-node=12
#SBATCH --cpus-per-task=1
#SBATCH -t 06:00:00
#SBATCH -o /scratch2/BMC/gsd-
hpcs/NCEPDEV/global/noscrub/Daniel.Abdi/fv3gfs/comrot/c48/logs/20180
82700/gfspost001.log
#SBATCH --export=ALL
.....
$GFS_DAEMON_RUN; $GFS_SING_CMD /opt/global-
workflow/jobs/rocoto/post.sh; $GFS_DAEMON_KILL
                                        ....
                                        }}
```

The SLURM commands inside the job scripts are wrapped with code that writes commands to a file, and then wait for execution of the job to finish. This way we have the job scheduler SLURM and workflow manager rocoto installed on the host, while the rest of the code resides inside a container.

Results

1) Performance comparison with bare-metal runs

C96 problem, using 6-mpi ranks, and 3-hrs forecast: almost no cost associated with using containers

Bare metal:	78 seconds
Within Docker:	83 seconds
Within Singularity:	77 seconds
Host-mpi singularity:	78 seconds

2) Multi-node runs Hera (on-premise) HPC system.

C384 problem, using 264 mpi ranks (216 compute + 48 IO) with 2 threads/core

Bare-metal:	500 sec
INTEL container:	520 sec
GCC container:	824 sec
GCC container no OFED drivers:	2700 sec

3) Google Cloud Platform run

C768 24-hour run completed in about 470s (7.8 min) meeting the 8-min per forecast day requirement.
C2-standard-60 VMs for all compute and I/O nodes.

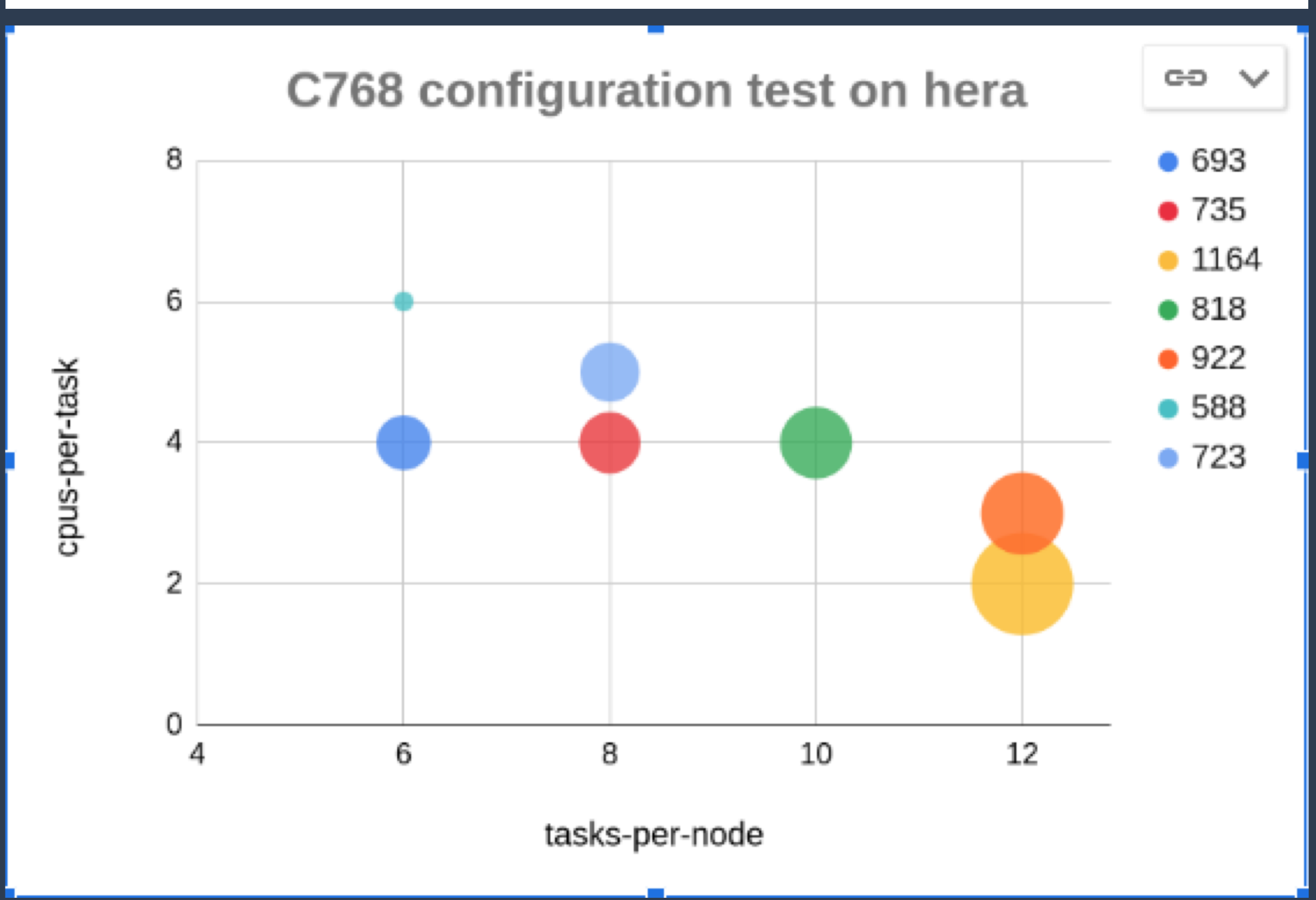
4) AWS run using ParallelCluster

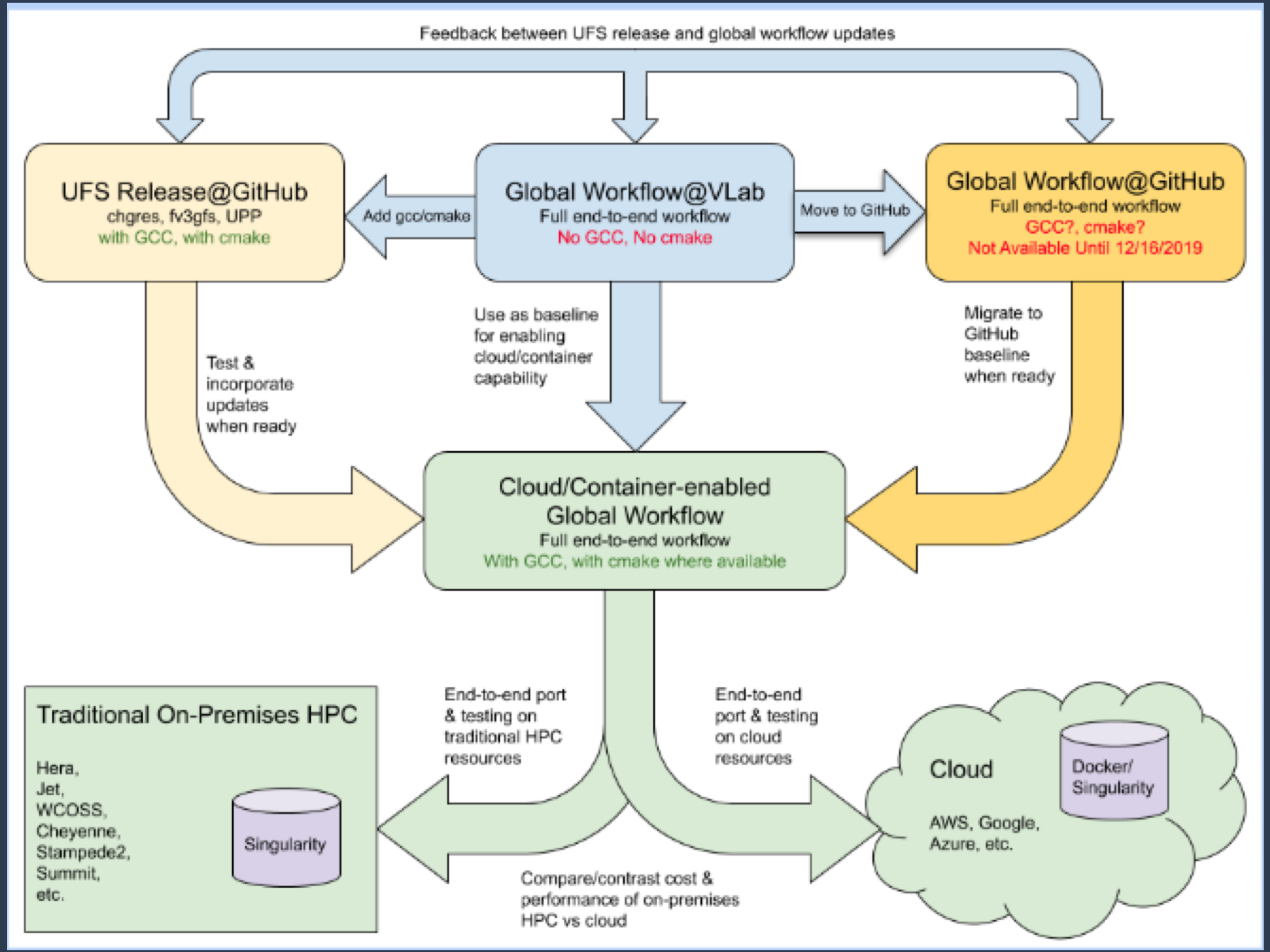
Single node results using C5n.18xlarge instances similar to that on Hera using Skylake processor.
C384 24-hour forecast in about 25 min, which is not that bad given GCC is 2-3x slower

5) Workflow management with rocoto

Workflow management using containers required only minimal modifications.

```
$ rocotostat -v 10 -w c48.xml -d c48.db
CYCLE             TASK            STATE           TIME
201808270000      gfsfv3ic          -               -
201808270000      gfsfcst         SUCCEEDED      19
201808270000      gfspost001      SUCCEEDED      1110
```




Feedback between UFS release and global workflow updates
UFS Release@GitHub
chgres, fv3gfs, UPP
with GCC, with cmake
Add gcc/cmake
Global Workflow@VLab
Full end-to-end workflow
No GCC, No cmake
Move to GitHub
Global Workflow@GitHub
Full end-to-end workflow
GCC?, cmake?
Not Available Until 12/16/2019
Test & incorporate updates when ready
Use as baseline for enabling cloud/container capability
Migrate to GitHub baseline when ready
Cloud/Container-enabled Global Workflow
Full end-to-end workflow
With GCC, with cmake where available
Traditional On-Premises HPC
Hera, Jet, WCOSS, Cheyenne, Stampede2, Summit, etc.
Singularity
End-to-end port & testing on traditional HPC resources
End-to-end port & testing on cloud resources
Compare/contrast cost & performance of on-premises HPC vs cloud
Cloud
AWS, Google, Azure, etc.
Docker/ Singularity

Conclusion

1) We have ported the Global Forecast System to run on different Cloud Computing Platforms

2) Using containers (Singularity and Docker) offers great portability and reproducibility of application. We demonstrated that performance using containers is not significantly impacted, however, there are other challenges associated with using containers such as:

a) multi-node runs with different MPI libraries
b) integration with existing workflow management tools
c) security issues due to escalated privileges of docker

3) Integration of containers with workflow management and job schedulers needed some tricks to get working. This problem is due the fact that that environment within and outside the container are completely isolated, hence, unless the code, workflow manager and job scheduler are all within or outside the container, there is going to be problems.

4) Some CCPs provide environments that emulate HPC environments with a few clicks of buttons (e.g. ParallelCluster). This coupled with use of containers for the HPC app offers a quick and reproducible environment for deploying HPC apps to the cloud.

References

1) The Global Forecast System
https://www.ncdc.noaa.gov/data-access/model-data/model-datasets/global-forcast-system-gfs
2) Rocoto workflow manager
https://github.com/christopherwharrop/rocoto