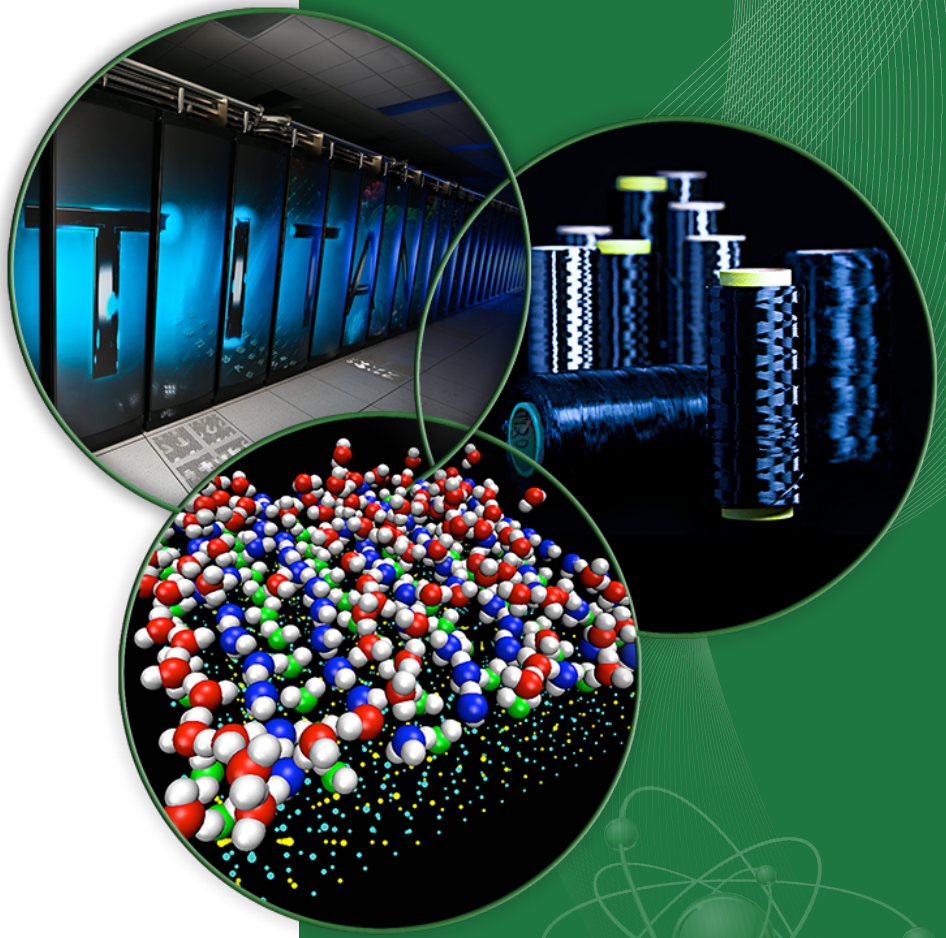


Balancing Performance and Portability with Containers in HPC: An OpenSHMEM Example

Thomas Naughton, Lawrence Sorrillo, Adam Simpson and Neena Imam

Oak Ridge National Laboratory

August 8, 2017

OpenSHMEM 2017 Workshop, Annapolis, MD, USA

Introduction

- Growing interest in methods to support user driven software customizations in an HPC environment
  - Leverage isolation mechanisms & container tools

- Reasons
  - Improve <u>productivity</u> of users
  - Improve <u>reproducibility</u> of computational experiments

Containers

- What's a Container?
  - "knobs" to tailor classic UNIX fork/exec model
  - Method for encapsulating an execution environment
  - Leverages operating system (OS) isolation mechanisms
    - Resource namespaces & control groups (cgroups)

Example: Process gets "isolated" view of running processes (PIDs), and a control group that restricts it to 50% of CPU

- Container Runtime
  - Coordinate OS mechanisms
  - Provide streamlined (simplified) access for user
  - Initialization of "container" & interfaces to attach/interact

Container Motivations

- Compute mobility
  - Build application with software stack and carry to compute
  - Customize execution environment to end-user preferences
- Reproducibility
  - Run experiments with the same software & data
  - Archive experimental setup for reuse (by self or others)
- Packaging & Distribution
  - Configure applications with possibly complex software dependencies using packages that are “best” for user
  - Benefits User and Developer <u>productivity</u>

Challenges

- Integration into HPC ecosystem
  - Tools & Container Runtimes for HPC
- Accessing high performance resources
  - Take advantage of advanced hardware capabilities
  - Typically in form of device drivers / user-level interfaces
    - Example: HPC NIC and associated communication libraries
- Methodology and best practices
  - Methods for balancing portability & performance
  - Establish “best practices” for building/assembling/using

Docker

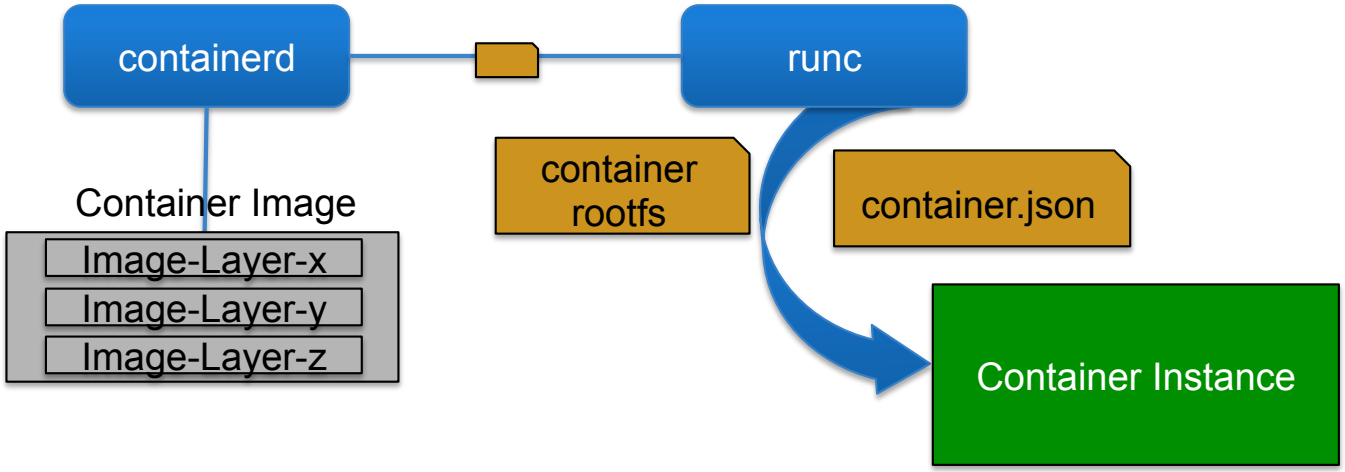
- Container technology comprised of several parts
  - *Engine*: User interface, Storage Drivers, Network overlays, Container Runtime
  - *Registry*: Image server (public & private)
  - *Swarm:* Machine and Control (daemon) overlay
  - *Compose*: Orchestration interface/specification

https://www.docker.com

- Tools & specifications
  - Same specification format for all layers (consistency)
  - Rich set of tools for image management/specification
    - e.g., DockerHub, Dockerfile, Compose, etc.

Docker Container Runtime

- runc & containerd
  - Low-level runtime for container creation/management
  - Donated as reference implementation for OCI (2015)
    - Open Container Initiative – runtime & image specs.
    - Formerly libcontainer in Docker
  - Used with containerd as of Docker-1.12



Singularity Container Runtime

- Container runtime for “compute mobility”
  - Code developed “from scratch” by Greg Kurtzer
  - Started in early 2016 (http://singularity.lbl.gov)
- Created with HPC site/use awareness
  - Expected practices for HPC resources (e.g., network, launch)
  - Expected behavior for user permissions
- Adding features to gain parity with Docker
  - Importing Docker images
  - Creating a Singularity-Hub

Docker

- Software stack
  - Several components
    - dockerd, swarm, registry, etc.
  - Rich set of tools & capabilities
- Networking
  - Supports full network isolation
  - Supports pass-through “host”
- Storage
  - Full storage abstraction
    - Example: devicemapper, AUFS

Singularity

- Software stack
  - Small code base
  - Follows more traditional HPC patterns
    - User permissions
    - Existing HPC launchers
- Networking
  - No network isolation
- Storage
  - No storage abstraction

Docker

- Images
  - Advanced read/write capabilities (layers)
    - Copy-on-Write (COW) for "container layer"
- Image creation
  - Specification file (*Dockerfile*)
- Image sharing
  - DockerHub is central location for sharing images

Singularity

- Images
  - Single image "file" (or 'rootfs' dir)
    - No COW or container layers
- Image creation
  - Specification file
  - Supports Docker image import
- Image sharing
  - SingularityHub emerging as basis to build/share images

Example: MPI Application

- MPI – Docker
  - Run SSH daemon in containers
  - Span nodes using Docker networking (“virtual networking”)
  - Fully abstracted from host
  - MPI entirely within container
- MPI – Singularity
  - Run SSH daemon on host
  - Span nodes using host network (no network isolation)
  - Support network at host/MPI-RTE layer
  - MPI split between host/container

Evaluation

- **Objective**: Evaluate viability of using <u>same</u> image on "developer" system and then on "production" system
  - Use an existing OpenSHMEM benchmark
- Container Image
  - Ubuntu 17.04
    - Recent release & not available on production system
  - Select Open MPI's implementation of OpenSHMEM
    - Directly available from Ubuntu
  - Select Graph500 as demonstration application
    - Using OpenSHMEM port

Adapting Images

- Two general approaches
  a) Customize image with requisite software
  b) Dynamically load requisite software

- Pro/Con
  – Over customization may break portability
  – Loading at runtime may not be viable in all cases

Image Construction Procedure

- Create Docker image for Graph500
  - Useful for initial testing on development machine
  - Docker **not** available on production systems
- Create Singularity image for Graph500
  - Directly import Docker image, or
  - Bootstrap image from Singularity definition file
- Customize for production
  - Later we found we also had to add a few directories for bind mounts on production machine
  - Add few changes to environment variable via the “/environment” file in Singularity-2.2.1
  - Add ‘munge’ library for authentication

Cray Titan Image Customizations

- (Required) Directories for runtime bind mounts
  - /opt/cray
  - /var/spool/alps
  - /var/opt/cray
  - /lustre/atlas
  - /lustre/atlas1
  - /lustre/atlas2
- Other customizations for our tests
  - apt-get install libmunge2 munge
  - mkdir -p /ccs/home/$MY_TITAN_USERNAME
  - Edits to “/environment” file (next slide)

Singularity /environment

```
# Appended to /environment file in container image

# On Cray, extend LD_LIBRARY_PATH with host CRAY Libs
if test -n "$CRAY_LD_LIBRARY_PATH"; then
	export PATH=$PATH:/usr/local/cuda/bin
	# Add Cray specific library paths
	export LD_LIBRARY_PATH=${CRAY_LD_LIBRARY_PATH}:/opt/cray/sysutils/1.0-
1.0502.60492.1.1.gem/lib64:/opt/cray/wlm_detect/1.0-
1.0502.64649.2.2.gem/lib64:/usr/local/lib:/lib64/usr/lib/x86_64-linux-
gnu:/usr/local/cuda/lib:/usr/local/cuda/lib64:${LD_LIBRARY_PATH}
fi

# On Cray, Also add the host OMPI Libs
if test -n "$CRAY_OMPI_LD_LIBRARY_PATH"; then
	# Add OpenMPI/2.0.2 libraries
	export LD_LIBRARY_PATH=${CRAY_OMPI_LD_LIBRARY_PATH}:${LD_LIBRARY_PATH}
	# Apparently these are needed on Titan
	OMPI_MCA_mpi_leave_pinned=0
	OMPI_MCA_mpi_leave_pinned_pipeline=0
fi
```

Setup

- **Software**
  - Graph500-OSHMEM
    - D'Azevedo 2015 [1]
  - Singularity 2.2.1
  - Ubuntu 17.04
    - OpenMPI v2.0.2
      with OpenSHMEM enabled
      (Using oshmem: spml/yoda)
    - GCC 5.4.3
  - Cray Linux Environment (CLE) 5.2 (Linux 3.0.x)
- **Testbed Hardware**
  - 4 Node, 64 core testbed
  - 10GigE
- **Production Hardware**
  - *Titan @ OLCF*
    - *16 cores per node*
    - *Used 2-64 nodes in tests*
  - Gemini network

[1] *E. D'Azevedo and N. Imam, Graph 500 in OpenSHMEM, OpenSHMEM Workshop 2015,*
http://www.csm.ornl.gov/workshops/openshmem2015/documents/talk5_paper_graph500.pdf

Test Cases

1. "*Native*" – a non-container case
   - Establish baseline for Graph500 without container stuff
2. "*Singularity*" – container setup to use Host network
   - Leverage host communication libraries (Cray Gemini)
   - Inject host comm. libs into container via LD_LIBRARY_PATH
3. "*Singularity-NoHost*" – #2 minus host libs
   - Run standard container (self-contained comm. libs)
   - Not likely to have super high performance comm. libs as the containers are built on commodity machines
     - i.e., no Cray Gemini headers/libs in container

Example Invocations

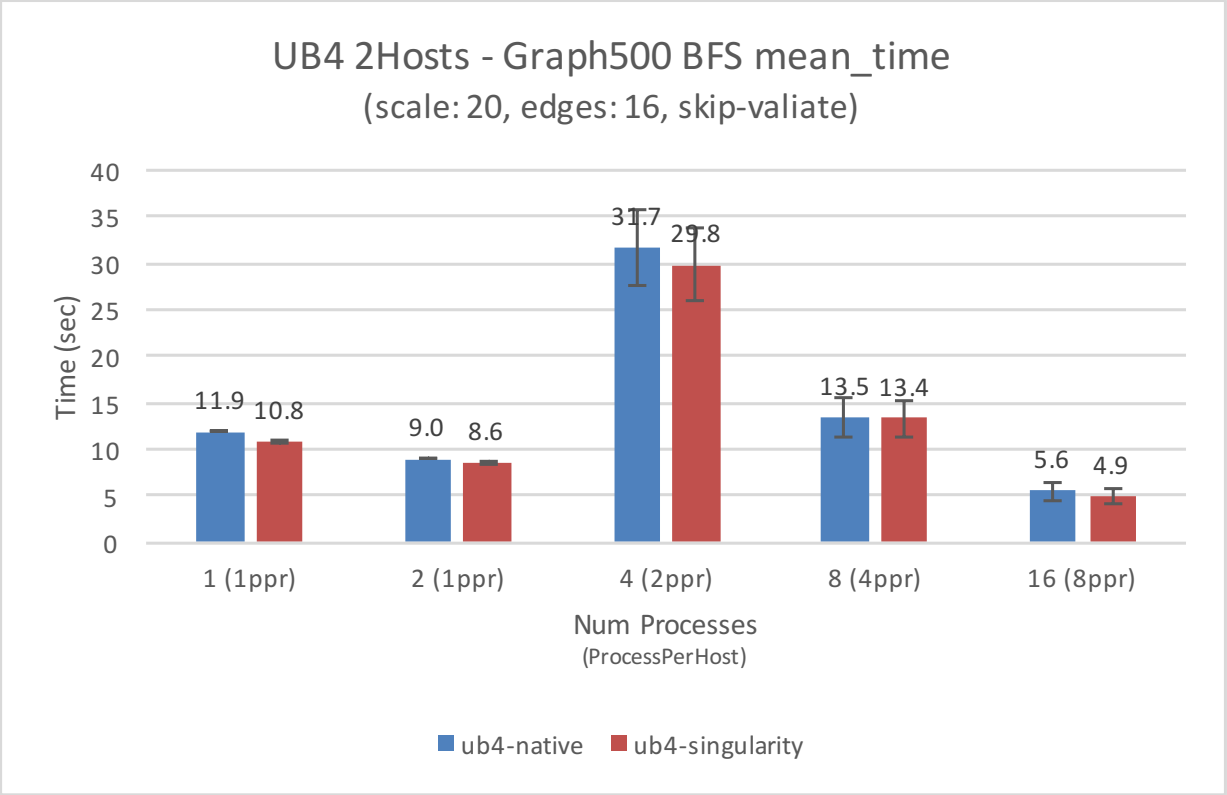
Native (no-container)

```
EXE=$PROJ_DIR/naughton/graph500/mpi/graph500_shmem_one_sided

oshrun  -np 128 --map-by ppr:2:node --bind-to core  \
   $EXE  20 16
```

Singularity (container)

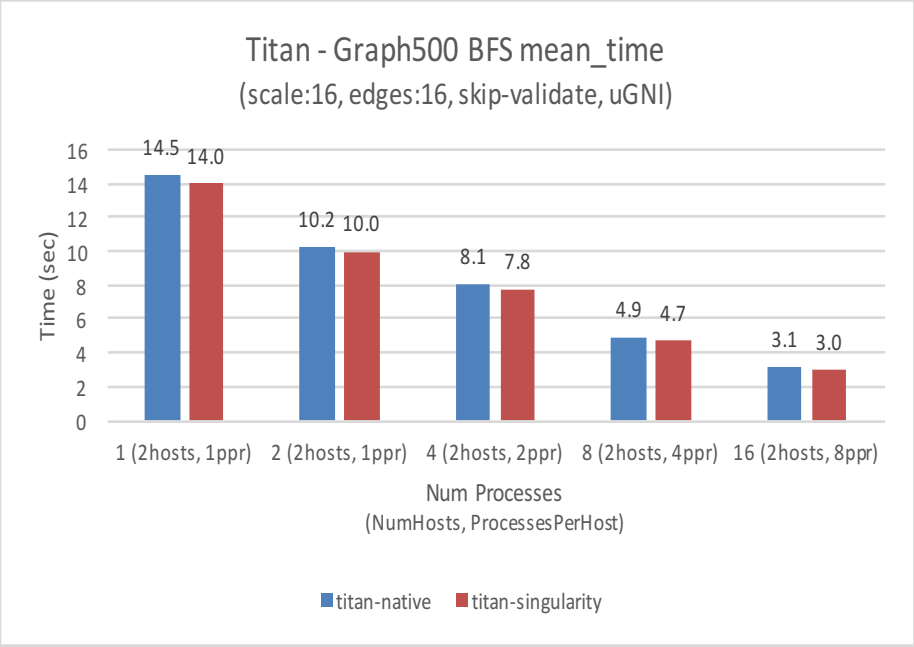
```
IMAGE_FILE=$PROJ_DIR/naughton/images-singularity/graph500-oshmem.img
EXE=/benchmarks/src/graph500-oshmem/mpi/graph500_shmem_one_sided

oshrun  -np 128 --map-by ppr:2:node --bind-to core \
   singularity exec  $IMAGE_FILE  $EXE 20 16
```

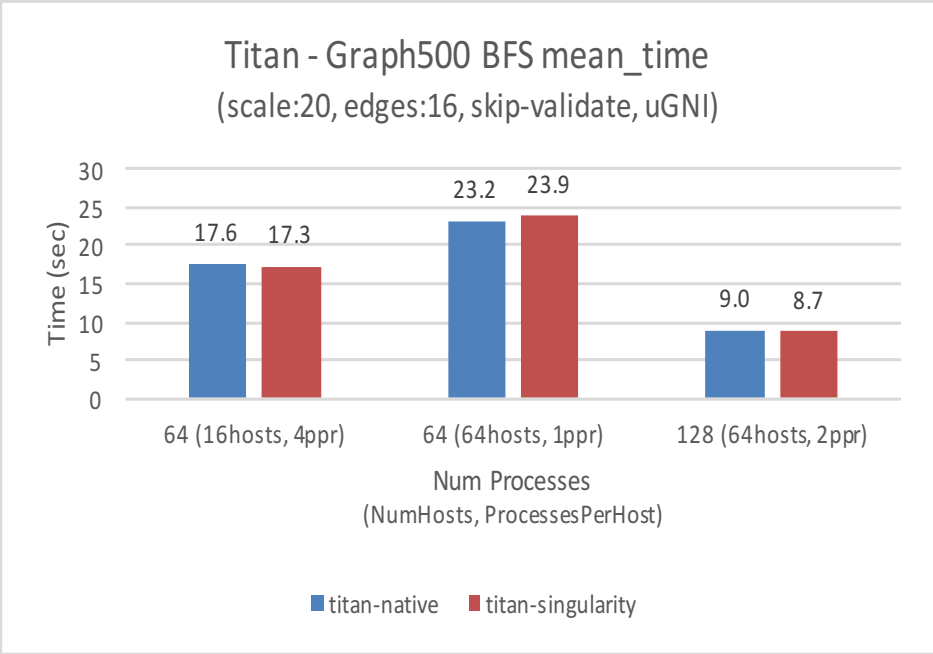
Development Cluster – Graph500 BFS



- Roughly same performance
  - Native (no-container) & Singularity
  - 2 hosts @ scale=20

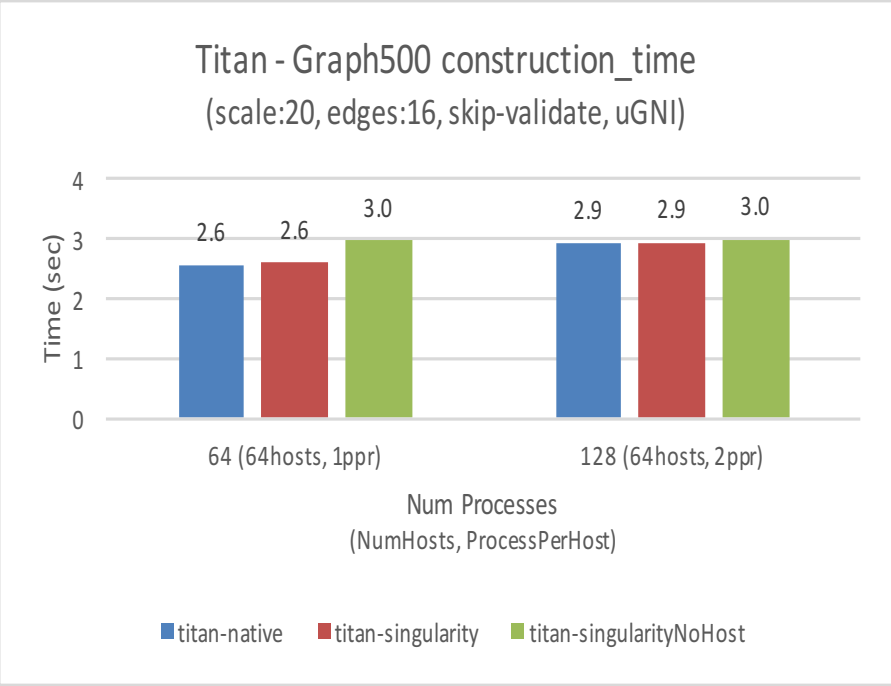
Titan – Graph500 BFS

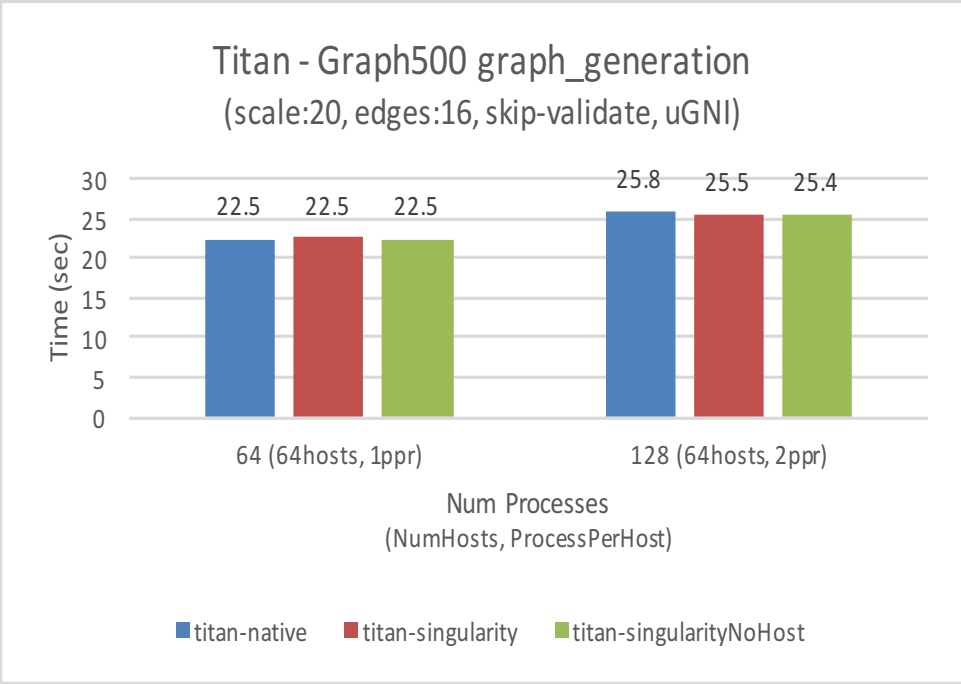




- Roughly same performance
  - Native (no-container) & Singularity
  - Left: 2 hosts @ scale=16 with uGNI BTL
  - Right: 16 hosts & 64 hosts @ scale=20 with uGNI BTL

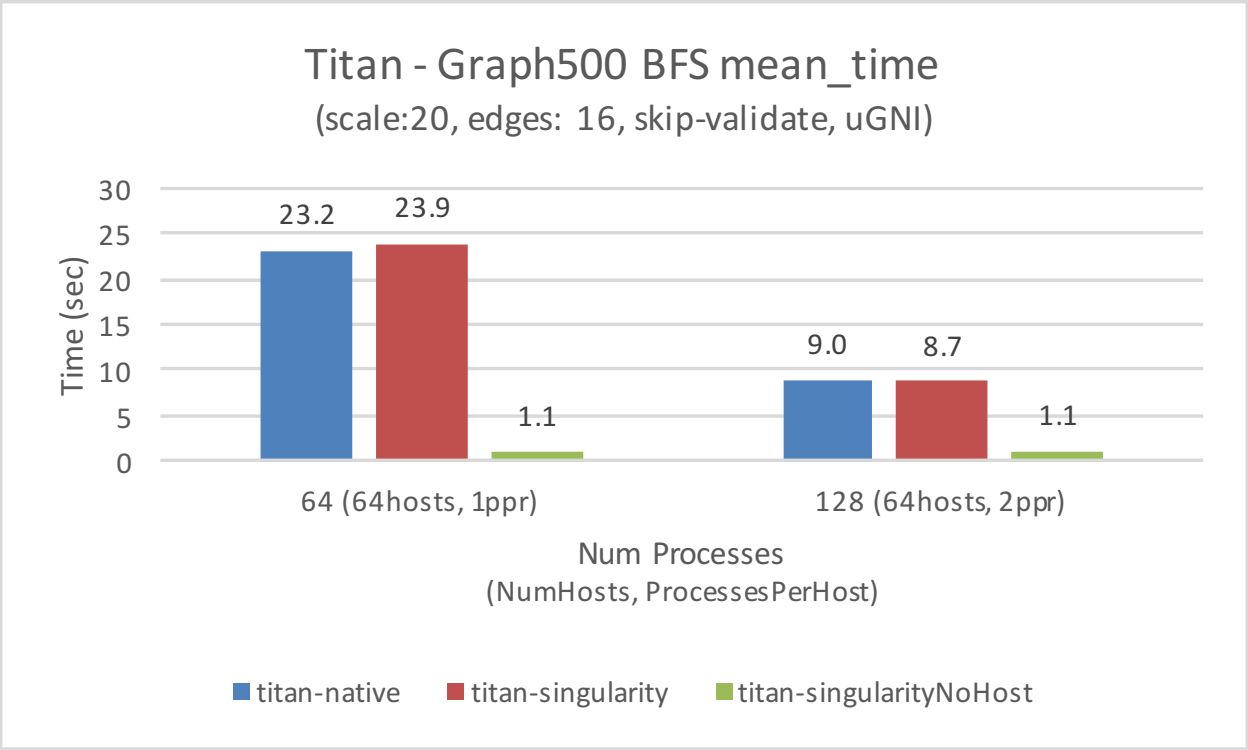
Titan – Graph Construction & Generation





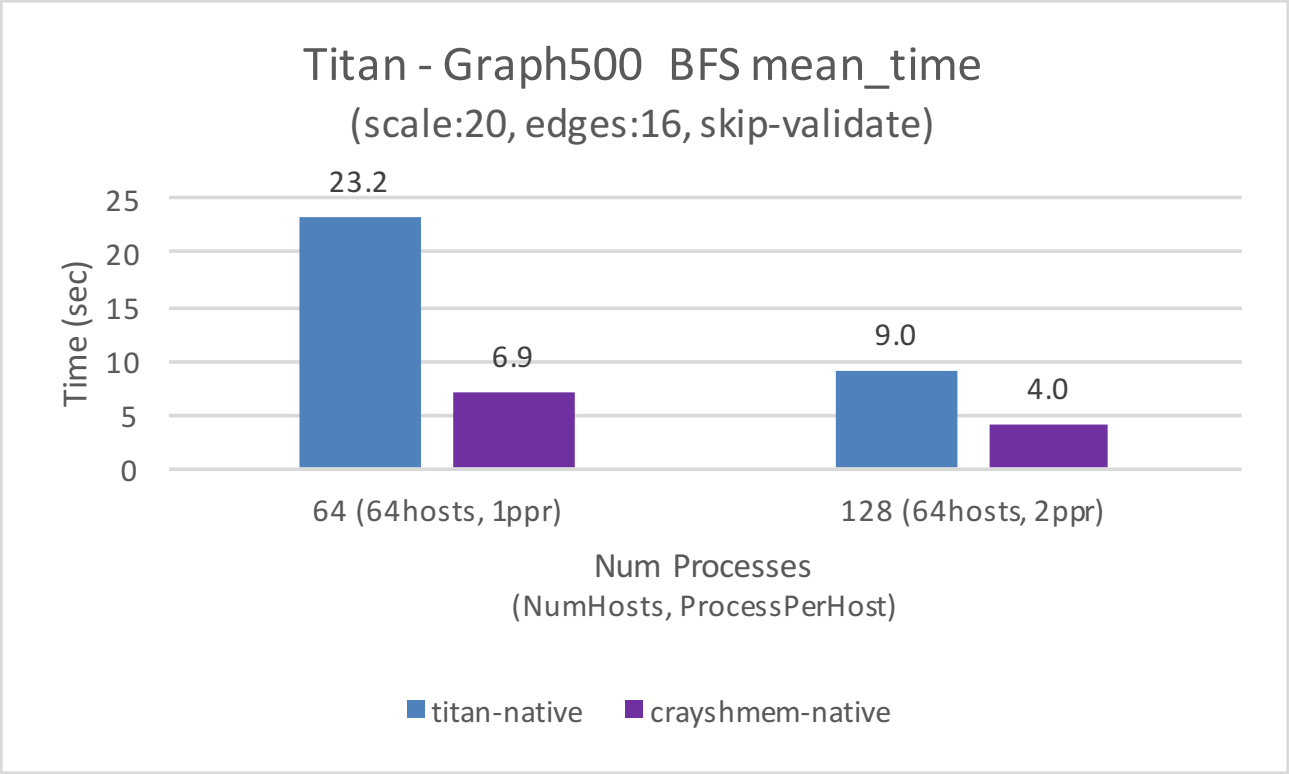
- Graph construction & generation times roughly same
  - Native (no-container), Singularity, and Singularity-noHost
  - 64 hosts @ scale=20 with uGNI BTL

Titan – Graph500 BFS mean_time



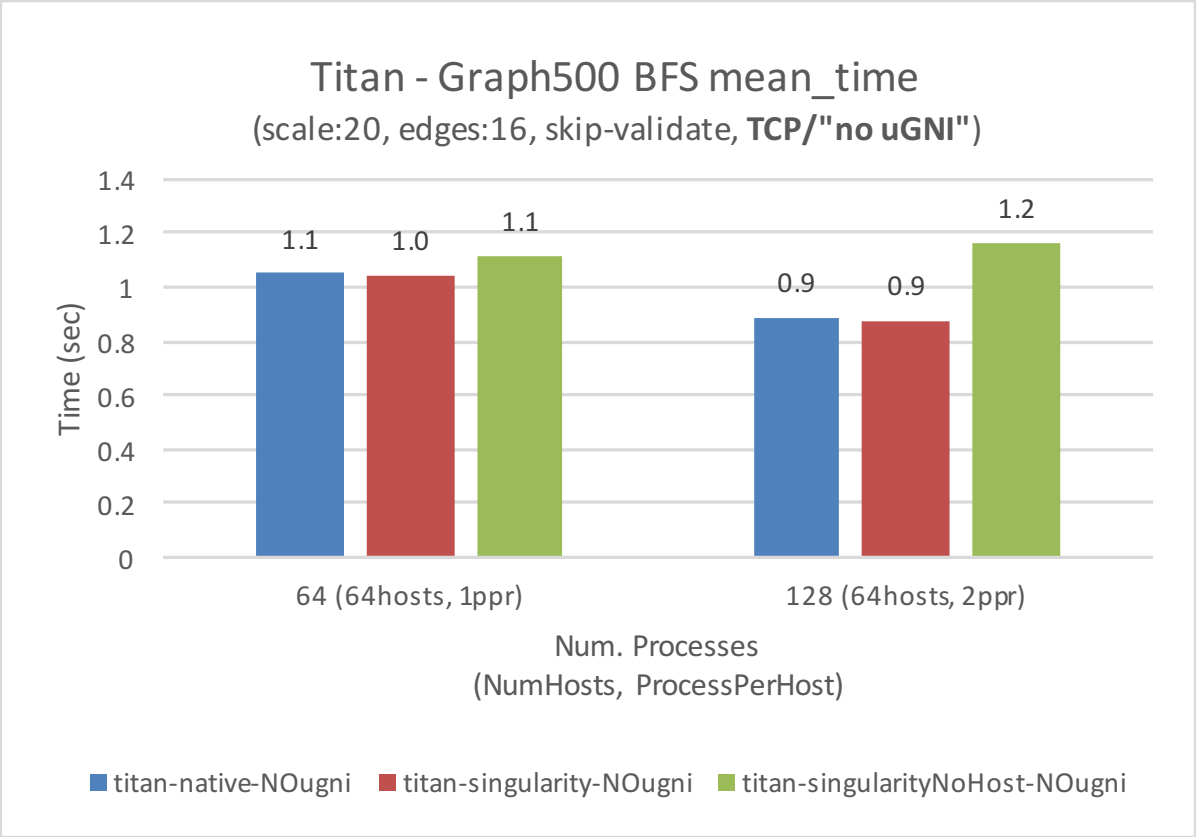
- Much worse performance when using host library (uGNI)
  - Native (no-container), Singularity (with uGNI), Singularity-noHost
  - 64 hosts @ scale=20 with uGNI BTL

Titan – Native comparison



- OMPI-oshmem uGNI (spml/yoda) worse than CraySHMEM
  - All native (no containers)
  - Suggests something wrong with our OMPI-oshmem uGNI setup ???

Titan – Graph500 BFS mean_time



- Roughly same when disable uGNI BTL
  - Native (no uGNI), Singularity (no uGNI), Singularity-noHost
  - 64 hosts @ scale=20 without uGNI BTL

Observations (1)

- Graph500
  - Native & Singularity had roughly same performance
    - Both uGNI & TCP BTLs
  - Singularity-NoHost had better performance in our tests due to problems with OMPI-2.0.2's OSHMEM with uGNI

- Open MPI's (v2.0.2) OpenSHMEM
  - Good: Maybe the only OSHMEM included in a Linux distro
  - Good: General TCP BTL was stable for testing and showed decent performance in our Graph500 tests at scale=20 on 64 nodes
  - Bad: Cray Gemini (uGNI) BTL is not stable with OSHMEM interface (using MCA spml/yoda)

Observations (2)

- Singularity in production
  - Performance was consistent between native & singularity
  - Required to customize image with dirs for bind mounts
  - Inconvenient to push full image for all edits to the image
    - Example: change '/environment' file require full re-upload
    - Note: Singularity-2.3 may have improved this, e.g., not need 'sudo' for "copy" and can set env via 'SINGULARITYENV_xxx'.
  - User-defined bind mounts disabled on older CLE kernel
  - Not able to use Cray SHMEM for container case
    - Note: Can not create image on system, and defeats purpose of portable container (not run on devel cluster)

Future Work

- Run with revised OpenSHMEM configuration
  - Use different component in Open MPI's SPML framework
    - Disable Yoda (spml/yoda)
    - Enable UCX (spml/ucx)
  - Determine root cause of unexpected performance with uGNI

- Scale-up tests on production system
  - Perform larger node/core count MPI and OSHMEM tests on Titan

Summary

- Containers in HPC
  - Motivations
    - Productivity of users
    - Reproducibility of experiments
  - Overview of two key container runtimes
    - Singularity & Docker
- Evaluation on Titan
  - Graph500 benchmark to investigate performance of OpenSHMEM application with Singularity based containers on production machine
  - Identified basic set of image edits for use on *Titan*
  - Consistent performance between Native and Singularity
    - Note: Identified unexpected slowness (unexplained) when using uGNI BTL

Acknowledgements

This work was supported by the United States Department of Defense (DoD) and used resources of the Computational Research and Development Programs at Oak Ridge National Laboratory.

Questions?