Presented By:
Gregory M. Kurtzer
HPC Systems Architect
Lawrence Berkeley National Laboratory
gmkurtzer@lbl.gov

CONTAINERS IN HPC WITH

SINGULARITY

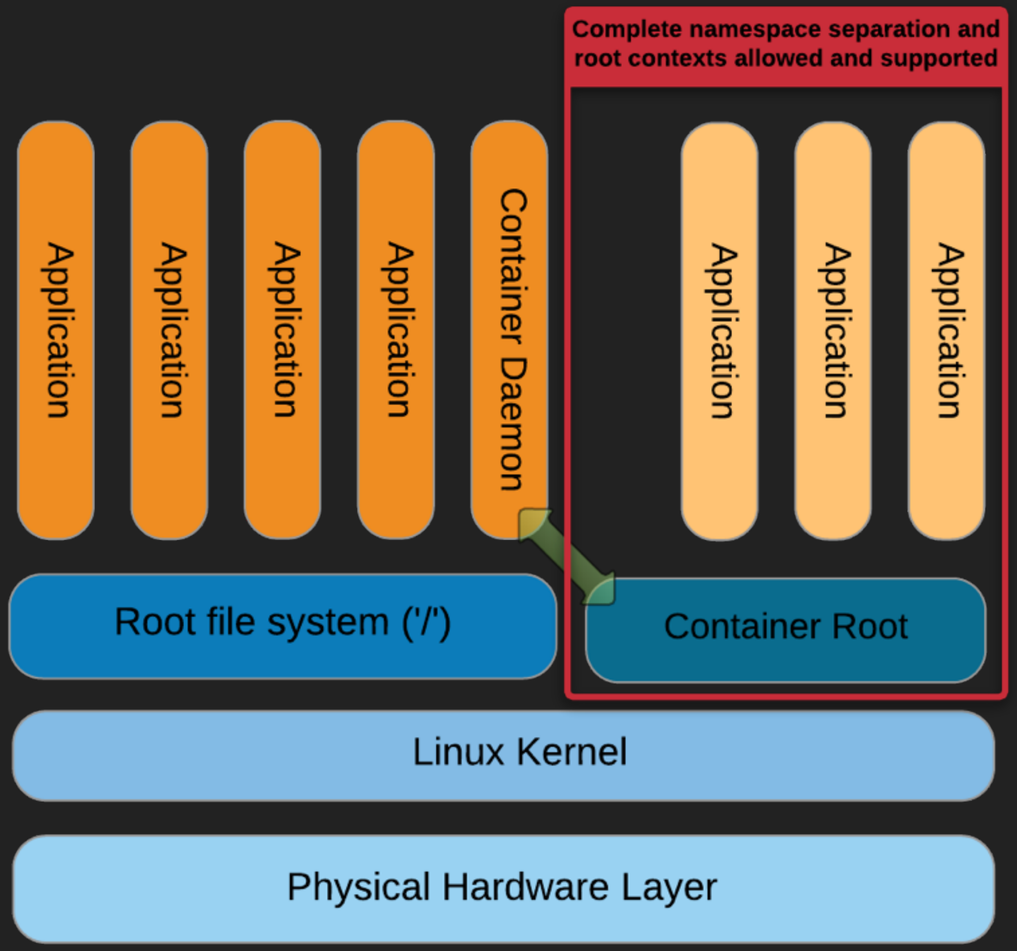
A QUICK REVIEW OF THE LANDSCAPE

- Many types of virtualization solutions in industry but we will focus on Hardware and Operating System virtualization (a.k.a. “Containers”)
- Virtualization solutions generally focus on network services, isolation and resource efficiency
- Hardware virtualization has set the stage for the current understanding and usage models of containers (service virtualization, isolation, usage model, etc.)
- People have been discussing and talking about Docker in HPC, but it has yet to be implemented… Why? (and no, NERSC’s Shifter is not Docker)

CONTAINER ARCHITECTURE

Applications which run in a container run with the same “distance” to the host kernel and hardware as natively running applications.

Generally the container daemon is a root owned daemon which will separate out all possible namespaces in order to achieve a fully emulated separation from host and other containers.

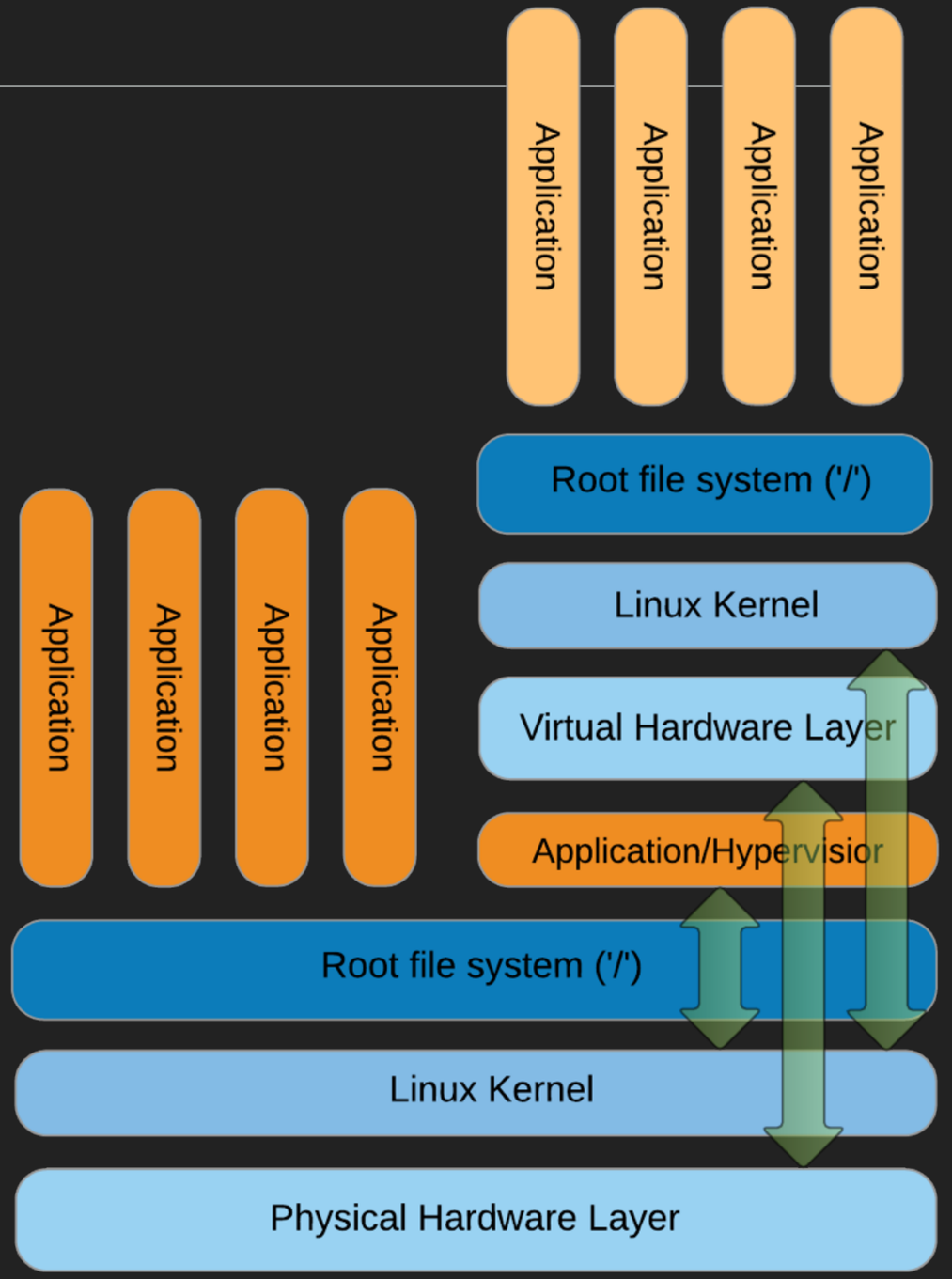


COMPARING TO VIRTUAL MACHINES

As a comparison to Virtual Machines, applications running inside a VM are not only further away from the physical hardware but the layers they must traverse are redundant.

Additionally there is a layer of emulation which contributes to a performance regression.

Containers are thus more efficient.



CONTAINER LIMITATIONS

- Architecture compatibility: always limited to CPU architecture (e.g. x86/x86_64 != ARM) and binary format (ELF) - (you could add a level of hardware virtualization)
- Portability: containers in general are portable, but there are limitations
  - Glibc / Kernel compatibility support
  - Any other kernel / user land API compatibility (e.g. OFED)
- Bitrot: containers are subject to stagnation just as any operating system is
- Performance: there “may be” a slight theoretical performance penalty with utilizing kernel namespaces
- File paths: the file system is different inside the container than outside the container

WELCOME TO DOCKER

- Docker is the most commonly used solution when considering Linux containers today
- Docker provides tools, utilities, and wrappers to easily create, maintain and distribute container images
- The primary container use case is targeted for network service virtualization (where it has gained popularity very fast)
- Computational developers love Docker because they can easily create reproducible and portable images containing their work

SO WHY NOT JUST IMPLEMENT DOCKER ON HPC RESOURCES?

DOCKER IN HPC: THE PROBLEM

- Docker emulates a virtual machine in many aspects (e.g. users can escalate to root)
- Non-authorized users having root access to any of our production networks is considered a security breech
- To mitigate this security issue, networks must be isolated for Docker access and thus will preclude access to InfiniBand high performance networks and optimized storage platforms
- People then build a virtual cluster within a cluster trying to navigate a solution
- Additional complexities arise with existing scheduling architectures and workflows which greatly complicate usage models and/or system architecture (especially with parallel MPI job execution)

LET'S RE-QUANTIFY THE NEED

- The need: *Mobility of Compute*
- Portable source code is so 1990's, now-a-days it is easier to just distribute a virtual machine or laptop hard drive image
- Ease of development: Users and developers want to work in familiar environments that they control and then run that exact code, pipeline or stack anywhere
- It must be *easily* integratable with existing complicated shared infrastructures and scheduling subsystems
- It must not create or encourage additional exploit vectors or escalation pathways which could be used to compromise the integrity of the system

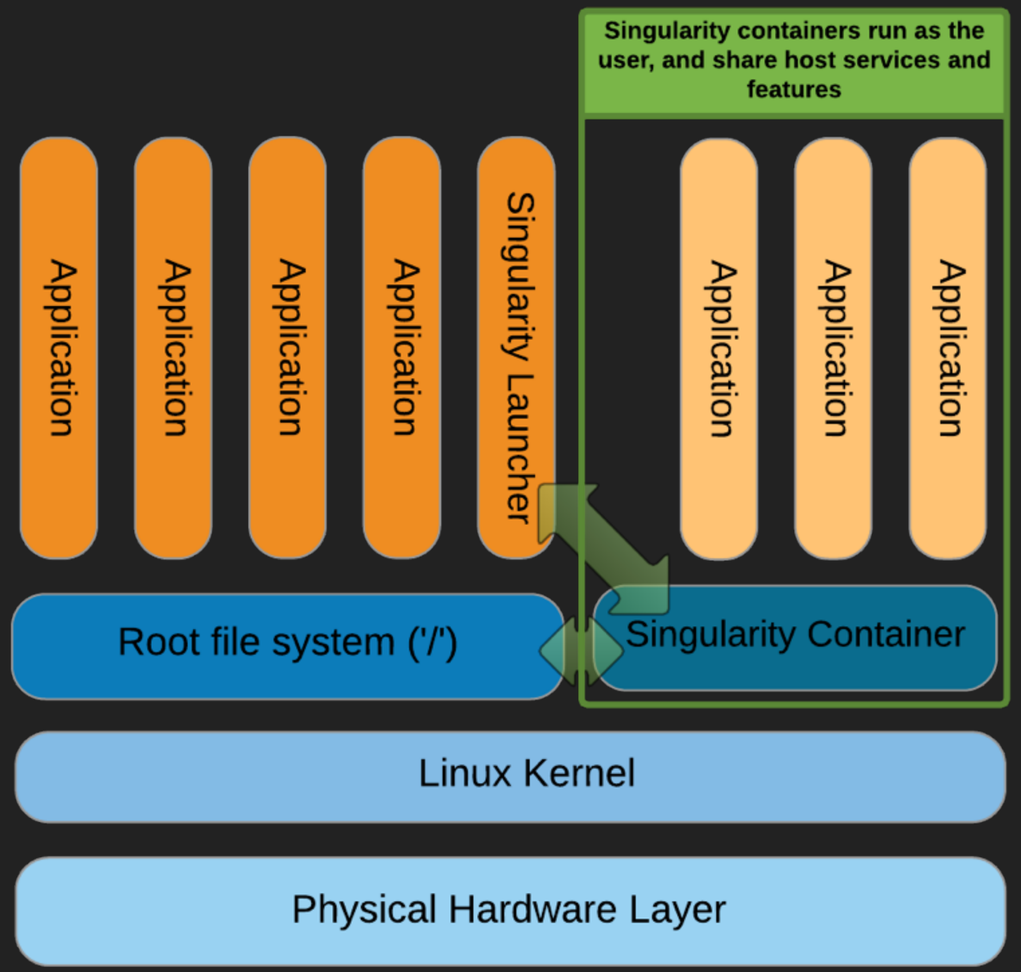
AND NOW FOR AN ENTIRELY DIFFERENT APPROACH.

SINGULARITY

- Container technologies can be utilized to support “Mobility of Compute” and making workflows and applications portable
- Container images facilitates modification, distribution, and execution (e.g. OS X packages)
- With no direct means to obtain root/superuser within the container, there are no additional security risks thus separation between the container and the host can be blurred (makes efficient use of existing HPC resources, workflows and parallel MPI jobs)
- We don’t have to virtualize everything, we can pick and choose on a case by case basis and thus blur the line between container and host
- No architectural or workflow changes are necessary to run Singularity containers

SINGULARITY: ARCHITECTURE

The “Singularity Launcher” is run by the user, loads the Singularity container and executes applications as the user.



SINGULARITY: BOOTSTRAP DEFINITION

- The process of building a Singularity container is simple and makes use of standard included distribution tools (e.g. Apt and YUM)
- Two steps to build a container from scratch (bootstrapping):
  - Create the image
  - Bootstrap the container using a definition file
- note: These two steps maybe merged so that the image details can be part of the bootstrap definition

SINGULARITY: BUILDING THE CONTAINER

- Singularity utilizes a definition file to bootstrap a new container
- You can include packages, binaries, scripts, libraries, data, and configurations into the definition file
- Bootstrap command installs a minimal version of the distribution
- Containers can be much smaller then a bootable operating system
- The output of the build process is a Singularity container image which can be modified with changes persisted within the image in real time

SINGULARITY: PERMISSIONS, ACCESS AND PRIVILEGE

- User contexts are always maintained when the container is launched (if it is launched by a user, it will always be that user) with no escalation pathway within the container! Thus….

If you want to be root inside of the container,

you must first be root outside of the container

SINGULARITY: LAUNCHING A CONTAINER

- When the Singularity container is launched, the namespaces are created and processes will be contained within the image
- By default the container image is accessed as read only
- Even when the container is mounted as read-write, root still maybe required to make changes (just like on any Linux system)
- All expected IO (this includes pipes, program arguments, stdout, stdin, stderr and X11) will all be passed directly through the container
- Home directory and other writable directories are automatically shared between the container and the host

SINGULARITY: WORKFLOWS AND PIPELINES

- Using the RunScript bootstrap directive, you can have a container execute a custom pipeline or workflow
- You can easily obtain the arguments via the RunScript using standard shell syntax
- The runscript exists within the container at '/singularity'
- Programs within the container can access data that exists within the container or outside of the container
- Singularity containers are a portable and lightweight means to distribute these workflows

SINGULARITY: IMAGE MANAGEMENT

- Singularity does not separate the user Linux namespace
- To make changes that require root inside the container requires that you have root access outside the container
- You will also need to use the -w/—writable option to make the container writable (without this, not even root can make changes)
- Once inside the container, use the system as you would normally

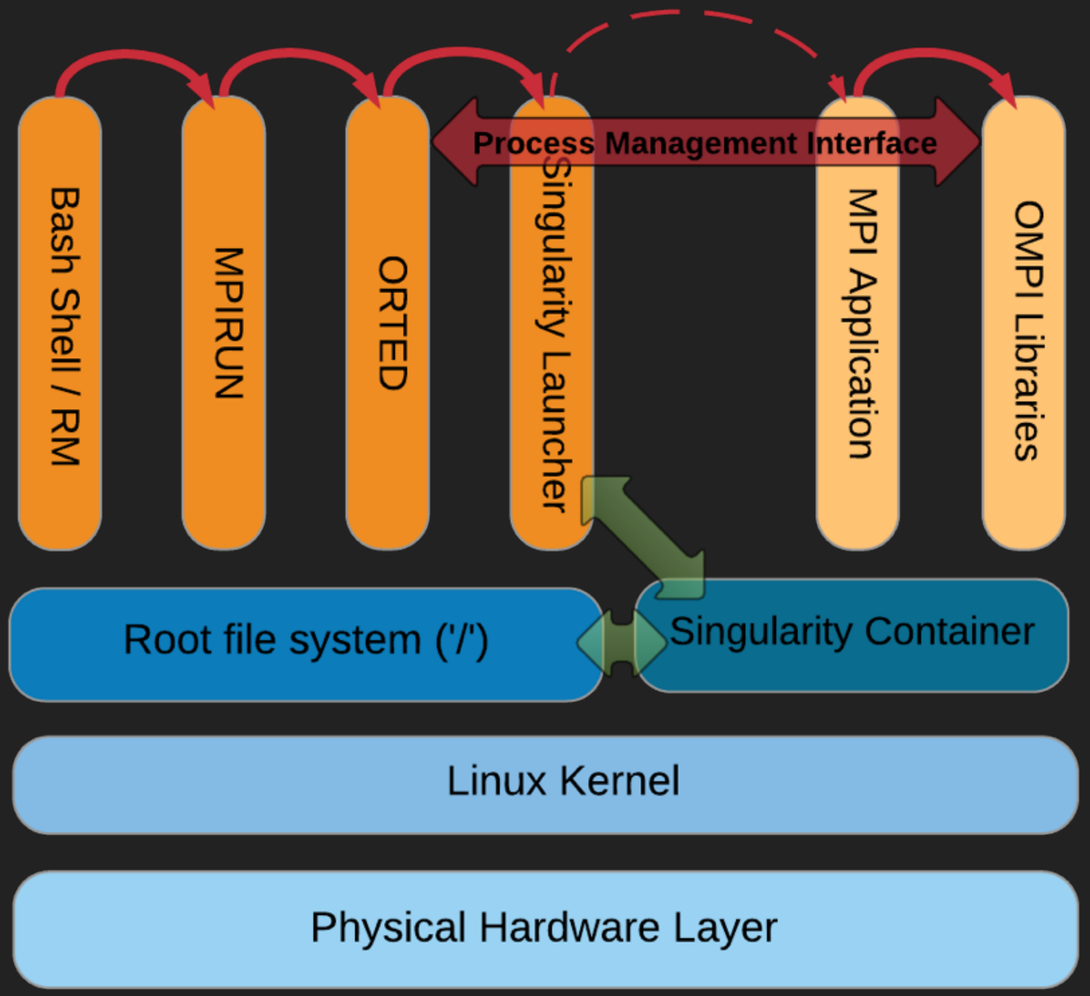
HPC – SINGULARITY AND OPEN MPI

SINGULARITY AND OPEN MPI

- Open MPI utilizes a parent/children relationship and communication pathway
- The parent MPI process (Orted) fork/execs the children processes
- Communication from parent to children occurs over the Process Management Interface (PMI)
- Singularity becomes the interface to exec the children
- Communication between the inside and outside of the container occurs over the PMI interface
- This architecture allows for optimal use of the hosts interconnect and storage
- So far there has been very little perceived performance penalty (but theoretically there maybe some due to Linux namespace overhead)

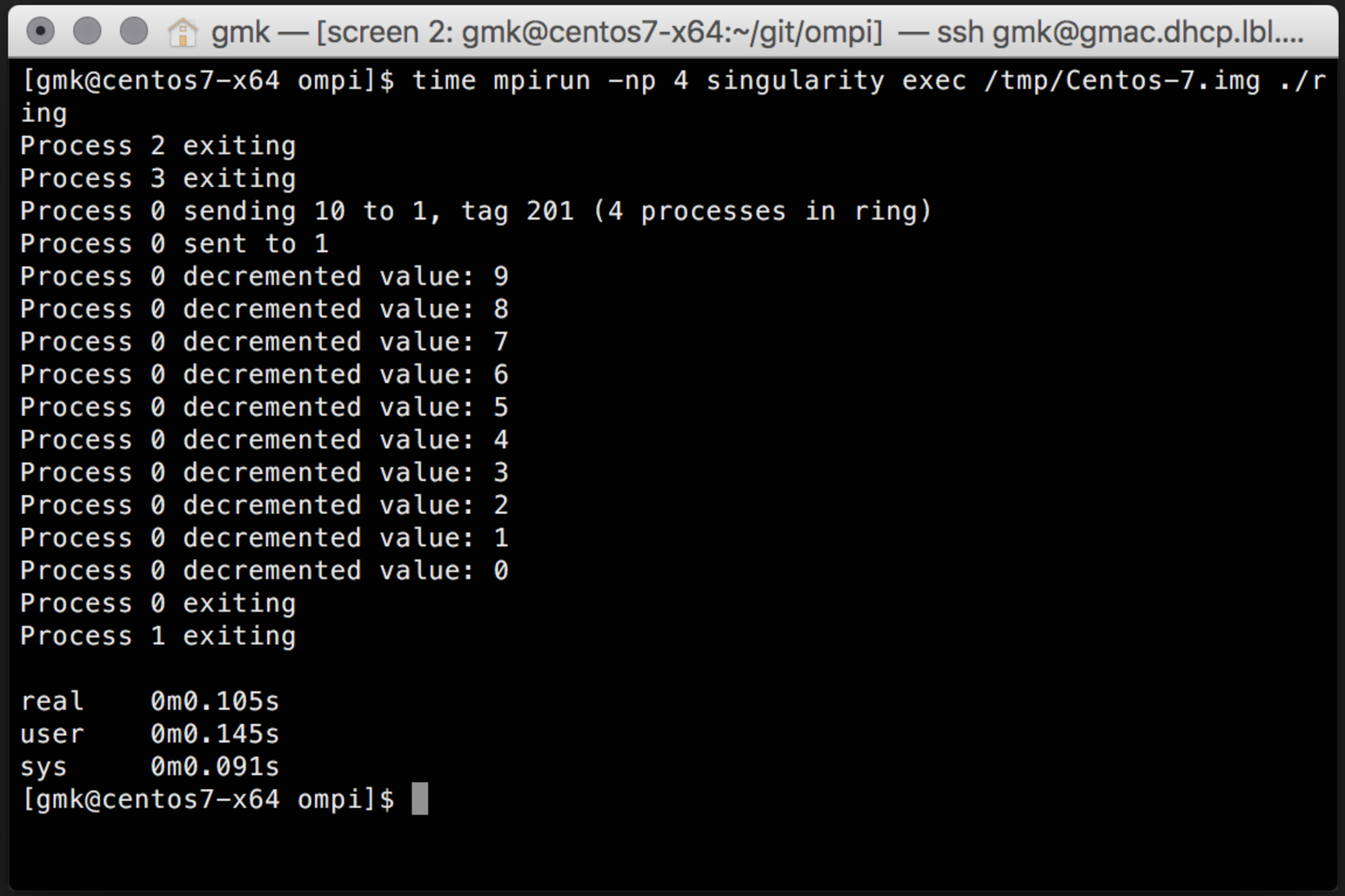
SINGULARITY AND OPEN MPI

- Starting with your Bash Shell or resource manager…
- MPI run gets executed which forks an orted process
- Orted launches Singularity which starts the container process
- The MPI application within the container is linked to the Open MPI runtime libraries within the container
- The Open MPI runtime libraries then connect and communicate back to the Orted process via a universal PMI



SINGULARITY: THE ACTUAL OUTPUT

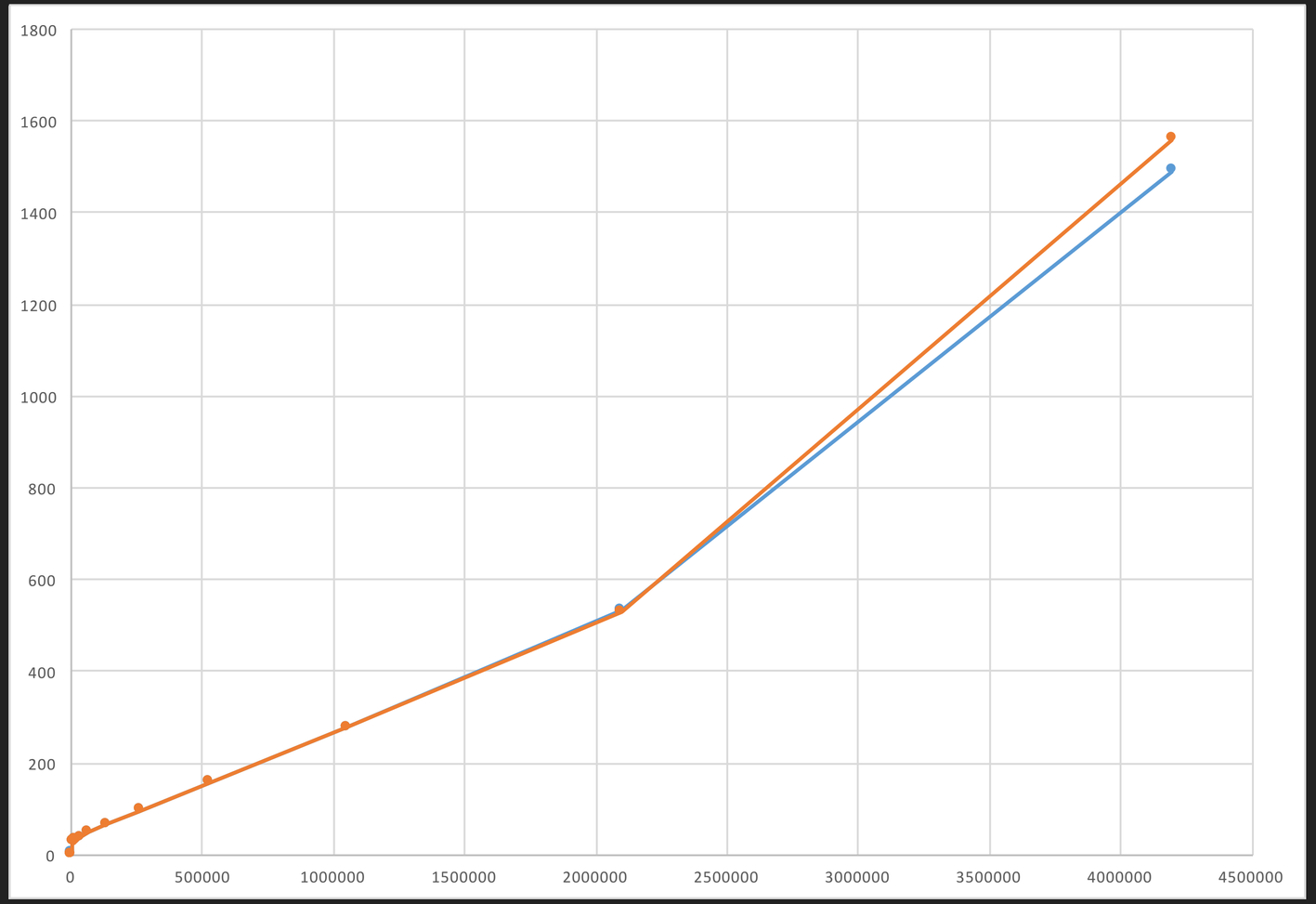
Launching `mpirun` on the host runs the Orted process in the background and it then forks and exec's the given Singularity command. Singularity then exec's `./ring` which loads and communicates back to the parent Orted process on the host via PMI.



SHARED MEMORY LATENCY IN OPEN MPI

This is an example shared memory latency graph comparing the same application in Singularity container (orange) and running directly on the host.

You can see both are performing closely and only diverge on very large messages.



Presented By:
Gregory M. Kurtzer
HPC Systems Architect
Lawrence Berkeley National Laboratory
gmkurtzer@lbl.gov

CONTAINERS IN HPC WITH

SINGULARITY