John Quince Wofford III

Candidate

Computer Science

Department

This thesis is approved, and it is acceptable in quality and form for publication:

Approved by the Thesis Committee:

Patrick Bridges

Chairperson

G. Matthew Fricke

Member

Patrick Widener

Member

John Patchett

Member

Reproducible application platforms for distributed computing systems

by

John Quince Wofford III

B.S., Interdisciplinary Computing - Physics, U. of Kansas, 2017

THESIS

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Master of Science
Computer Science

The University of New Mexico

Albuquerque, New Mexico

May, 2020

Dedication

To Wes,

who taught me that a complex thing is a series of simple things
strung together with marketing.

Acknowledgments

Thanks to Patrick Bridges and the Scalable Systems Lab for shining a light on systems research to show me how fun and interesting it can be.

Thanks to John Patchett and Jim Ahrens for meeting with me at SC16 and for giving me a chance at Los Alamos National Laboratory. Thanks to John and Jim for giving me the push I needed to pursue a graduate degree. Thanks to Mike Lang, Linn Collins, Dave Rich, David Rogers, John Patchett and the Department of Energy for coming together in financial support of this effort. Thanks to the National Physical Science Consortium for their tuition assistance. Thanks to Andrew Younge for long discussions on containers and their potential in HPC. Thanks to Ben Allan, whose insight was crucial to the success of my LDMS integrations. Thanks to Josh Hursey for clearly (and repeatedly) explaining exactly how Open MPI launches distributed applications. Thanks to Ivo Jimenez, who allowed me to guide parts of his workflow management project to suit my needs. Thanks to my lab peers for their companionship: Pepper Marts, Keira Haskins, Sahba Tashakkori, Soheila Jafari, Jered Trujillo, and Zeinab Akhavan. Thanks to my committee for their attention and guidance: Patrick Widener, G. Matthew Fricke, John Patchett and Patrick Bridges. Thanks to Nick Abel, Scott Levy, and Kurt Ferreira for their thoughts and direction on the first iterations of this thesis. Thanks to Ryan Grant, Matt Dosanjh, and Whit Schonbein for their guidance on my defense presentation. Rashid Yousefi, thank you for sharing your gift for computer science with me and with anyone who needed it. So many CS students at UNM thrived because of your generosity, and I want to be sure that you know that.

I would like to thank the UNM Center for Advanced Research Computing and Texas Advanced Computing Center for providing the HPC resources used in this work.

Thanks to my Dad, for believing in my abilities and for sharing his passion for problem solving. I pride myself in providing working solutions to real problems that people *actually* have, and that drive comes from him.

Thanks to my Mom, for her good humor. Her tolerance for listening to outlandish plans brought me comfort during the various career upheavals leading up to this career, which I love. Being spontaneous and affable is a great joy, and I learned that from her. I also blame her for forgetting where I put the car keys[1].

Finally, thanks to my wife Alicia, who has learned more about computer science than she might have liked. Alicia, I deeply appreciate that you make an effort to care about what I care about and engage with my curiosity...even when I'm curious about kernel abstractions.

[1]Oh, wait, there they are.

Reproducible application platforms for distributed computing systems

by

John Quince Wofford III

B.S., Interdisciplinary Computing - Physics, U. of Kansas, 2017

M.S., Computer Science, University of New Mexico, 2025

Abstract

A scientific conclusion requires falsifiable evidence. Results from distributed systems research are often difficult to reproduce because these systems consist of multiple nodes, each running independent system software and communicating across inter-node devices. This work motivates, describes, and demonstrates a reproducible application platform for distributed computing systems based on a layered, container-based software stack. This system effectively moves all application software dependencies from the host to a portable container. Each layer represents a particular functionality of the software stack. The layers are modular and extensible so that results are not only repeatable, but they can also be built on to produce new results. This platform was developed to validate the scaling behavior of an application performance model. It was first run on one HPC system (CARC-Wheeler). It was modified slightly and then run on a second HPC system (TACC-Stampede2). The reproducible application platform gives researchers more flexibility to use and conveniently share custom software stacks, reduces the burden on system administrators to maintain libraries required for individual research efforts, and eliminates the effort required to re-use previous work in related experiments.

Contents

List of Figures

Glossary

ABI compatibility	A binary that is ABI compatible with a second binary will run in place of the second binary without modifications, provided that necessary shared libraries are present, and similar prerequisites are fulfilled.
bulk synchronous parallel	Bulk synchronous parallel (BSP) applications follow a design paradigm which consists of data distribution, asynchronous program execution, and episodic synchrony.
container	Containers are a layer of OS abstraction similar to a virtual machine, but without the flexibility and overhead of full kernel emulation. Container guests share the kernel with the host system. Linux kernel namespaces enable configurable host sharing, so that container runtimes may choose how much to share with the host at the time the container is executed.
DevOps	Short for "Development Operations". DevOps is a set of practices that combines software development (Dev) and information-technology operations (Ops). This is a school of system administration which seeks to provide software developers with the platform they need to quickly write and continuously test release software for a wide range of computing environments.

generalized extreme value theorem	Generalized extreme value (GEV) theory is concerned with statistics of summary statistics, such as distributions of maximums. This theory is used to describe 10 year flood plains and the behavior of BSP applications, among other things.
high performance computing (HPC)	A distributed computing system that is shared among users to perform complex computation. The scale of computation on HPC systems exceeds the ability of a single computer by multiple orders of magnitude through parallelism. Resources are allocated according to scheduling policies in accordance with user requests and site-specific fairness policies.
input deck	The input deck is a set of parameters that will affect a running application. Problem size, problem shape, and error tolerance are all examples of things that might be defined in an input deck.
kernel	Throughout this document, kernel refers to the core system software that coordinates a system's hardware/software interface.
namespace	Namespaces are a Linux kernel feature which allow new process ids to fork with the host operating system cloned. A cloned namespace is then decoupled from the host, such that similarity is guaranteed at the point of cloning, and not guaranteed after that point.
reproducibility and repeatability	Throughout the course of this thesis, repeatability refers to the ability to repeat a scientific result. Reproducibility refers to the entire process which delivers a repeatable result.
research artifact	A research artifact is a clearly defined experimental outcome. Research artifacts can be used as a tangible, falsifiable experiment outcome.

Glossary

Software stack	A software stack is a set of software required to run an application.

Chapter 1

Introduction

Experimental results must be repeatable in order to be validated and falsifiable. Scientific discovery can not proceed safely without these qualities. Computing systems have quickly evolving hardware and software landscapes, making experimental validation difficult. Distributed computing systems additionally suffer from added complexity due to coordination needs and potentially heterogeneous hardware/software platforms. Because of this, papers are published, systems change, and results are sometimes taken in good faith or reproduced on newer systems only with considerable effort.

Science is an iterative process. Although we rely on previous publications as a foundation for new works, there does not exist a widely used reproducibility platform for distributed computing. Such a platform would allow researchers to literally reproduce a previous work with very little work, make small changes to the platform, and produce new results without the need to fully develop their own software stack and deployment system on a targeted infrastructure.

My thesis is that a reproducible application platform for distributed computing can be achieved with container technology on high-performance computing platforms. This document describes the design, implementation, and evaluation of such a platform.

1.1 Reproducible platform goals

Through the course of this work, I identified five desirable characteristics for a reproducibility platform:

- Repeatability: Results can be demonstrated on demand by anyone from a targeted infrastructure.
- Portability: The system can be modified to run on different systems.
- Extensibility: The system is modular and can consist of multiple dependent parts.
- Re-usability: Previous work can be re-used in future work.
- Accessibility: The system is easy to use and develop.

This document presents the design and implementation of a new software system for high-performance computing systems that provides all of the desired reproducibility features presented in this section.

1.1.1 Repeatability $\in$ reproducibility

Repeatability is where reproducibility begins. While experiment validation in the form of repeatability is an important and active research area [22], reproducibility is a larger concept. In order for an experiment to be repeatable the researcher must perform the same steps to reach the same conclusion. In contrast, when an experiment is reproducible, researchers are able to use the same experiment infrastructure with slight modifications to produce results that are distinct from the original experiment. Repeatability is concerned with the ability to *repeat* a previous result. Reproducibility is concerned with the ability to *reproduce* the stages of an experiment in a way that allows extending the state of the art. Repeatability focuses on demonstrating provenance for a claim. Reproducibility extends the repeatability goal. It makes the procedures which generated those results immediately accessible for future work.

Reproducible results need context. Experiments must establish a context and control for nuisance variables. The context of a distributed computing experiment includes:

- The hardware of multiple potentially heterogenous computers
- The software running on those computers
- The devices and algorithms which allow the computers to communicate with each other
- The algorithms which govern the behavior of an application of interest
- Power delivery to the system
- and so on...

Reproducibility for distributed computing Distributed computing systems are often a shared resource, and the software available on these systems is constantly evolving. System administrators do their best to provide relevant software packages and libraries to distributed system users. Module loading systems allow system administrators to prepare loadable sets of packages. Researchers working in these shared environments build new and highly customized software against these system-provided modules, but lack the ability to share this environment easily with others. These problems are further complicated when a software system must be tested on a *different* distributed computing system, which may be running an entirely different operating system. Consequently, papers in this field are either taken in good faith, or reproduced only with considerable effort.

1.1.2 Application portability and extensibility

Reproducing distributed computing results is generally impractical due to a lack of tools. Each time a distributed computing paper is published and we seek to take advantage of its methods, a researcher must first reproduce the software stack on a system they have access to. The utility of previous work is improved when someone can immediately take that previous work to another system, use it near-verbatim, and make iterative changes toward new research goals. The scientific impact of a result will be magnified when researchers can take the *same* software system deployed to validate a publication, and deploy it on a new system, with modifications that improve the state of the art.

1.1.3 Re-usability and accessibility

Module loading systems allow software developers to select from a list of available software. These modules are curated by system administrators. Module loading systems provide a flexible method for users to control their software environment, but the availability of this common software is not guaranteed forever. Even with a common software base, reproducing software stacks between different users requires effort. This system enables users to develop their own software stacks, moving the burden of software stack maintenance from the distributed system administrator to the distributed system user. If the burden is tolerable, users gain the advantage of consistently re-usable, modular, software stacks. Users often lack the technical expertise required to manage and share a software stack with loadable modules. However, most users do have the expertise required to manage a software stack on personal computing environments. The ideal system can be *developed* on personal computers, where software stacks are managed intuitively with package managers, and system resource constraints such as time-sharing and compute cost are less important. The ideal system can be *deployed* on distributed computers. It must be easy to tweak and test these systems. The system must achieve the same flexibility as a module loading system, without the same complexity. If it is not easy to iterate on previous results, the system will not be used.

1.2 Background

The reproducible application platform brings together and extends three areas of distributed computing research: user-defined software stacks, distributed software orchestration, and distributed application performance monitoring. In Section 1.2.1 I discuss virtualization, which enables portable user defined software stacks. In Section 1.2.2, I discuss how containers safely provide user defined software stacks for shared systems. In Section 1.2.3, I discuss how modern communication libraries orchestrate jobs on HPC systems. In Section 1.2.4, I discuss performance monitoring and why it is commonly useful in computer systems research. Section 1.2.5 discusses how experiment workflow managers attempt to bring all of these components together.

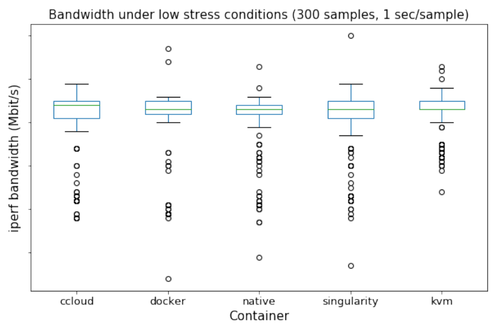
1.2.1 Virtualization

Virtualization provides software portability across platforms and flexibility to run independently from the host environment [8] [29] [37]. Often virtualization requires some compute overhead to achieve this.

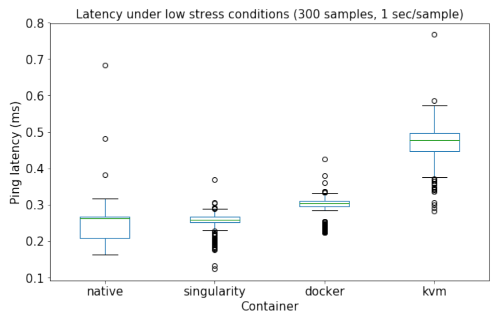
Virtualization is useful for creating portable and flexible software stacks but is complex to implement. The lowest level interface that the operating system interacts with is the kernel. That is to say that the kernel interface is the closest interface to the hardware that the operating system communicates with. The kernel is an abstraction layer which makes process management and scheduling easier for OS developers. When the system is more general, a single command may translate to multiple machine architectures. This reduces the amount of work required to interact with a system form a programming perspective. However, that improved generality increases implementation complexity for the virtualization system. This generality is also a potential source of overhead. Virtual machines implement an abstraction on top of the host kernel. Not only do virtual machines emulate the kernel itself, but they often emulate hardware which the virtual kernel runs on. This layered set of abstractions is common on a fully virtualized system.

While fully virtualized systems make portability goals more achievable, the application runtime may incur substantial overhead for some tasks. I performed some tests to independently verify the overhead of several virtualization techniques considered through this thesis work. In particular, I tested bandwidth and latency from a computer connected to the UNM gigabit network and the CARC-Wheeler headnode. I compared this baseline (native) performance with several of the virtualization techniques I considered for the reproducible application platform. Figure 1.1 shows the network overhead for Singularity, Docker, and KVM (a full virtualization technique).

Figure 1.1: Network performance demonstrates that no single metric can capture virtualization overhead



(a) Bandwidth performance is similar across virtualization technologies



(b) Latency performance differs across virtualization technologies

As this figure shows, virtualization overhead is not always noticeable for every metric. Network bandwidth is relatively unaffected by virtualization (Figure 1.1a, 1.2), but latency overhead is significant (Figure 1.1b).

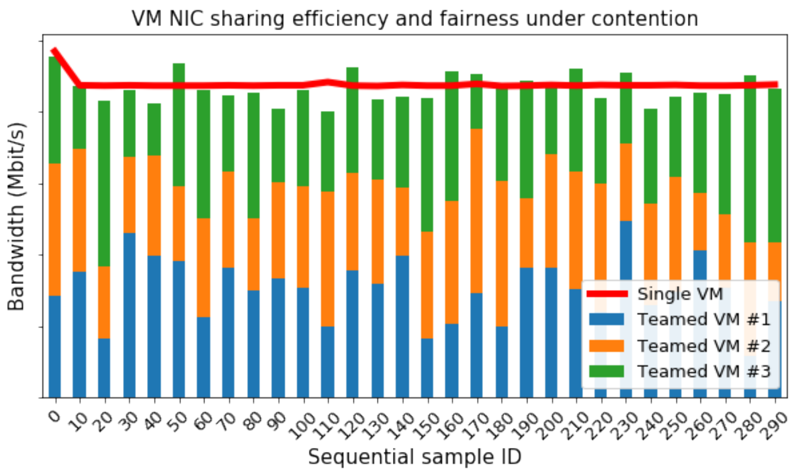
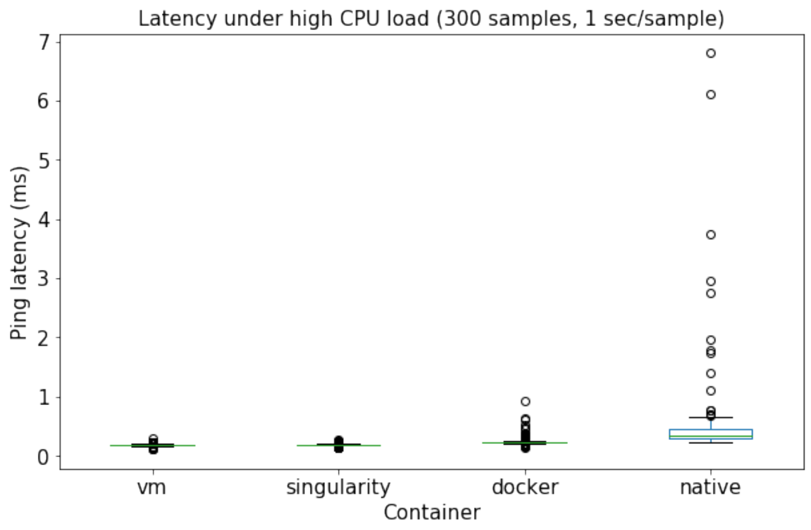


Figure 1.2: Full virtualization shares network bandwidth efficiently, even under contention, but is not a holistic performance metric

Figure 1.3: Containers may perform better under CPU load for some performance metrics



Although no studies to date have shown precisely why, containers may even improve device performance under CPU load as shown in Figure 1.3 and as noted in [14] and anecdotally elsewhere.

1.2.2 Containers

Containers are a form of virtualization and allow users to develop a software stack which is independent of the operating system that they run on without the overhead of full system virtualization. They are widely used in industry, where software developers need to test their work on multiple platforms with limited hardware availability. However, containers are most commonly used in data centers or distributed systems where software developers and administrators run trusted software with privileged access. Docker [35] is a container solution which is popular when running trusted software from privileged accounts. Docker is not popular on *high performance* distributed systems because the systems are shared, users do not have privileged accounts, and Docker may introduce some performance overhead.

Research software is inherently experimental and potentially confidential. In order to facilitate fair sharing of resources, some limits on privileged access are required. Several container runtimes exist which are suitable for untrusted workloads. HPC-friendly container runtimes include: Singularity [33], CharlieCloud [41], and Shifter [28]. Each of these solutions provide custom software stack safely. CharlieCloud uses a Linux kernel feature called "namespaces" to clone the state of the host system and make changes which only affect the running container. Shifter uses a whitelist of acceptable container images to limit risk, at the cost of total flexibility for the user. Singularity either uses namespace kernel features like CharlieCloud, or makes selective use of privileged commands using SETUID runtime binaries. In the latter case, Singularity allows the user to execute privileged commands but only as granted by Singularity. Singularity is used in this work and described in detail in Chapter 2.

Linux namespaces

Containers are a partial virtualization technique that allows custom software stacks to run on shared computers without privileged access. Running containers will *mostly* ignore the host computer's software configuration and run with a container-provided software configuration. Specifically, containers allow a user to run a custom software stack while sharing the host kernel, circumventing the need for costly kernel emulation. Containers are the partial kernel abstraction which inspired the development of Linux namespaces [16]. Namespaces are a kernel feature that allow a running program to use parts of the host operating system, while mapping over key functionality to run custom software. Namespaces are what give containers the power to run custom software without the overhead of full virtualization (virtual machines).

Full virtualization versus containerization Containers do not require kernel emulation, and are therefore said to be "partial" virtualization. Since containers do not require kernel emulation overhead, they match native software performance in many cases. Namespaces are the key container feature which allow the host kernel to operate as if it were running in isolation. Namespaces offer better performance, but are limited when compared to full virtualization. Namespaces force us to share the host kernel, while full virtualization can run any kernel independent of the host. However, the kernel changes much less frequently than other system software, so this limitation is not an overwhelming reason to choose full virtualization over containers for this work.

Containers can provide a custom software stack without overhead. Today it is common for containers to be associated with some overhead, but this is only the case when containers are orchestrated over more complex abstractions, such as emulated kernels or virtual networks [45] [48]. To clarify how we might run a container without overhead, consider the mount namespace. It's common for containers to bind paths such as `/usr/local` over the host using the mount namespace. As long as this bind mount is located on the same device where the original `/usr/local` path exists, why should we expect this use of namespaces to incur overhead?

Containers may or may not operate with overhead. The performance impacts of namespaces are not always obvious. In general, when a layer of indirection is required some compute power is required to translate from the more general to the less general. Docker, for instance, uses software IP tables to route packets to the container, much like full kernel virtualization techniques. Singularity by contrast shares the network stack with the host, so that an intermediate translation is not required. Docker's network services have some overhead, while Singularity does not (see Figure 1.1b). When designing any experiment pipeline, it's important to be aware of these sources of potential overhead. The reproducible application platform avoids network latency overhead by using the Singularity runtime.

1.2.3 Communication libraries

The defining characteristic of any distributed application is the ability to communicate across physically distinct hardware devices. Communication libraries are designed to handle this problem, but they are not currently designed with container run-times in mind. The Message Passing Interface (MPI) [47] is a standard that has been implemented in many different software libraries. MPI specifies the existence of one or more communication domains: each domain consisting of a master node, and worker nodes. The master node is responsible for establishing the communication paths for all communication domains. One such implementation is Open MPI [26], which is the mechanism I use to coordinate applications across the distributed computer system. I describe how Open MPI is used in Section 3.3.

1.2.4 Performance monitoring

Application performance predictability is a core effort in computer science, and the empirical study of application performance is often delegated to systems researchers. Performance monitoring allows a software user to measure and better understand the behavior of applications and the systems they run on. Measurement tools are required for someone to study the behavior of the host operating system, container, workload application, and system hardware. This section describes what a predictable application is, and performance monitoring tools that enable predictability. A range of performance monitoring tools is available; timers, tracing, and profiling techniques are suitable for some prediction models, and these tools are described below.

Application predictability. Application performance predictability is a challenging, yet consistent goal in computer science. Algorithm behavior is discussed in terms of math proofs and asymptotic analysis, but in order to predict how an application might perform on a real shared system, it's necessary to measure performance characteristics related to the system as whole. A distributed computing system consists not only of multiple computers, but also the infrastructure which connects them. The performance of an application depends on how these intermediate devices are being used by other users on the system. Not all of these subsystems can be controlled by the user, but some of them can be monitored and measured. The predictable application platform will measure as many of these subsystems as possible and attach them to application data.

The general strategy used to predict application performance is to run an application, measure its resource utilization and execution time, and then use this data as input to statistical or machine learning models. These models project future performance based on historical performance. Application runtime, memory usage, and disk usage are a few examples of resources we might like to model in practice. An analyst can not model and project performance without tools to measure it. There is a trade-off between thorough application monitoring and realistic run-time. The more resources a computer uses to measure the performance of an application, the less resources are available for running the application itself. I describe three styles of performance monitoring below: timers, full application tracing, and sampling. These three monitoring styles capture the trade-off between monitoring completeness and accuracy in production.

Timers: realistic performance with very little detail

Timers are the simplest performance measurement tool available. Application performance is determined by the combination of hardware and software running on a system at the time of execution. Application performance can be measured most simply with a timer. A timer can answer questions related to the relative performance of the same application over previous runs. Even when measuring a single function call, the operating system itself requires resources which will vary during the execution of that function call. The timer approach is computationally cheap and it measures performance accurately, but it fails to take into account how time is spent within that application and on the hardware systems it runs on. Timers account for system interference such as network congestion, but they do not allow an analyst to attribute performance to it.

Full application tracing: detailed information with significant performance overhead

A full application trace completely maps the execution path of an application with substantial overhead. This overhead makes application runtimes worse than we expect without monitoring. Full application traces provide a holistic view of code execution paths. Full traces suffer from exaggerated runtimes and fail to take into account the state of the system where the app is running.

Sampling: Configurable detail and overhead

Sampling is a configurable performance monitoring tool. In general, sampling refers to selecting a set of interesting observations, and taking measurements at a configurable frequency. If the sampling frequency is set high enough, sampling can be used to study every function call during the execution of a program. This extreme captures the same information as the full application trace. In practice, it's often a subset of an application that we wish to sample (just the parts we know represent potential bottlenecks). Likewise, an analysis technique may not require a sampling every function call a program makes, and a lower sampling frequency can be used in this case.

1.2.5 Scientific workflow managers

Conditional workflow management is a method to organize experiment logic in the form of directed acyclic graphs (DAGs), and execute the applications that generate a reproducible result [15]. Workflow managers such as Pegasus extend these conditional workflow managers for distributed computing systems to handle mapping experiment tasks to many worker nodes [23].

Research artifacts motivate published claims. The stages of an experiment are managed by experiment workflow system. Some experiment workflow systems define "research artifacts" as experiment deliverables [32]. An experiment workflow produces a clearly defined research artifact. This research artifact is viewed as the "data of merit" of a particular experimental result. The claims of an experiment should be falsifiable using solely the research artifact as evidence. Although the research artifact may be used to demonstrate a previous result, the research artifact may also be used as evidence for future work or iterative improvements on previous work.

1.2.6 Reproducibility and repeatability

Throughout the course of this thesis, repeatability refers to the ability to repeat a scientific result. Reproducibility refers to the process which includes and delivers a repeatable result. Reproducible computing has been an active research area for the annual SuperComputing conference [9], where research artifacts have been defined by Artifact Descriptions (ADs). Artifact Evaluations (AEs) describe the process for auditing an experiment based on the ADs. Reproducibility was the subject of a 2018 SuperComputing workshop, ResCuE-HPC [5], where issues concerning auditing and validation in distributed computing were raised, resulting in some differentiation between repeatability and reproducibility, with repeatability being the more specific and reproducibility being the more general [34] [31] [40].

1.3 Contributions

This thesis describes the following contributions:

- A new system design based on layered containers. Layered containers increase the portability and reproducibility of high perfomance scientific software.
- A concrete implementation of this design using a combination of Popper, Docker, Singularity, Open MPI, and totally untrusted container runtimes.
- Examples of portable software stacks which have been constructed using this approach. Specifically we demonstrate execution of LANL's VPIC application [20] on UNM's Wheeler cluster using Open MPI and, with slight modifications, on TACC's Stampede2 cluster using MVAPICH. Notably, this is one of the few instances where an Open MPI-based workload has been executed on a TACC system.
- A demonstration that the described system can be used to reproduce results such as those previously described by Mondragon [39].

1.4 Thesis outline

This chapter has established motivation for a reproducible application platform. It provided requisite background to understand the material that follows. Finally, it listed the set of contributions which are to be described in the remainder of this document. Chapter 2 discusses related works, and areas where this work extends the state of the art. Chapter 3 describes the design and implementation of my reproducible application platform. Chapter 4 describes the evaluation of this system as a reproducible application platform. Chapter 5 offers concluding thoughts, and future work.

Chapter 2

Related works

2.1 Introduction

My reproducible application platform is novel for its use of layered image containers. The layered image container discretizes functionality into modular components. System performance monitoring is provided by one of these layers. An experiment workflow manager coordinates these containers across the distributed computer system. This chapter discusses my novel work and how it relates to existing container solutions, performance monitoring, and experiment workflow managers.

Section 2.2 discusses containers at length, and how containers are used in my reproducible application platform. Section 2.2.1 explains how Docker image layering is used in industry, and how image layering can be a useful tool in research software development. Experiment workflow managers are described in Section 2.4. Section 2.3 discusses how performance monitoring is done on large HPC systems, and how incorporating these tools into a container image stack makes them easier to use.

2.2 Containers

My reproducible software platform uses containers to provide custom software stacks on distributed computing platforms, without privileged container runtimes. This work uses design patterns from industry and applies them to a shared computing context, where application workloads are not allowed to have root access. The shared distributed computing context is different than the typical distributed computing context. Containers run in a privileged mode in industry, they do not run in a privileged mode in my system. Containers are not necessarily coupled with an experiment model, but they are coupled with an experiment model in my work. When containers communicate, my containers communicate over high speed network cards and are capable of executing tasks over these networks. This section describes example container technologies, and the features which I have ported to the distributed computing context with my reproducible application platform.

Example container systems. Docker [35] is a container solution which manages reproducible software stacks for DevOps use-cases, and we utilize some of the same tools. Shifter [28], Singularity [33], and CharlieCloud [41] are three container runtimes that can launch Docker containers without granting privileged access to unauthorized users. Shifter only allows approved images to launch, and ultimately launches these containers with privileged access. Singularity achieves "rootless" container launch using selective setuid functionality, or through a truly unprivileged user namespace launch. CharlieCloud launches with the user namespace, exclusively. This work uses Docker for reproducible platform development, and Singularity for reproducible platform deployment. The way these development modes interact is discussed in Section 3.2.4 and visually represented in Figure 3.5.

2.2.1 Container image layering

The primary novel contribution of my reproducible application platform is the layered container image design. The vocabulary associated with container technology suffers from overloaded terms. To better describe the container image layering approach, this section describes terms which are sometimes misunderstood.

Container environment

In general, the word "*container*" is used to refer to a running container environment. The container environment shares the host's kernel. The running container utilizes namespaces to provide overlays on the host system such that the entire software stack can operate independently of the host while still utilizing the host kernel for process management and scheduling. The container environment is the context from which a reproducible application workload is called.

Container runtime

The container environment is launched by a *container runtime*. The container runtime could be Docker, Singularity, CharlieCloud, or something else. The container runtime is responsible for launching a process within the context of the container environment. Container runtimes have many optional launch parameters. Container runtimes can bind filesystem resources into the guest. Container runtimes typically clone the `mount` namespace by default, but any other namespaces can be cloned on execution as well.

Container image

The term "*container image*" is a term popularized by Docker. Docker uses the "*container image*" concept to describe a container build and execution plan. Container images consist of binary blobs. Each blob represents a deviation from a base image. A common base image is the ISO format OS distribution package, unpacked on some available filesystem. If we unpack a standard Linux distribution (say, Debian), and use `chroot` on the directory, this is much like a container base image with fewer security guarantees. Going forward, if I use the word *image* without context, it should be interpreted to mean *Docker container image*.

Once a base image layer is established, a new image layer can be built on top of it. In this example I will demonstrate how image layering works. The purpose of this new image layer is to create a script in the container's view of `/usr/bin`.

```
wget http://debian.org/my_dist.iso
unpack my_dist.iso /tmp
chroot /tmp
echo "#!/bin/bash; echo hello world;" >/usr/bin/my_script
chmod +x /usr/bin/my_script
exit
```

The operations performed between `chroot` and `exit` would be captured in an image blob. This blob could be committed back to a container image. Subsequent image layers will be generated in the same way.

A container image is distinct from the container. Containers are manifestations of a container runtime executing a container image to produce a container. Images are not containers, but a container runtime can launch an image. All processes launched from the context of a container runtime are said to be "containerized".

Dockerfiles

Docker popularized a scripting paradigm with their "Dockerfile". A Dockerfile consists of a list of commands to apply to a base image. If we were to create a Dockerfile from the above `chroot` example, it might look something like this:

```
FROM: debian/latest
RUN echo "#!/bin/bash; echo hello world;" >/usr/bin/my_script
RUN chmod +x /usr/bin/my_script
```

There are three parts to this simple pseudo-Dockerfile:

1. `FROM` indicates the image we base our new image off of.
2. `RUN echo...` is the first command that will be saved in the form of an image layer.
3. `RUN chmod +x...` is the second command that will be saved in the form of an image layer.

In this case, the base image is `debian/latest`. This is meant to represent the latest Debian release, as deployed from some pre-packaged distribution ISO file. The `RUN` commands are simply commands to be issued as the image is built. The image will save each `RUN` command as a separate blob in the layered container image. It is possible to run a Docker image interactively, run from the command line to apply changes to the running container and commit back to the image. Unfortunately, image layer blobs do not contain the specific commands issued. The blobs can be replayed over the image, but command history is not clear unless something like a Dockerfile is used. Hence, this system encourages the use of Dockerfiles for all software stack build operations.

Using Docker as a development platform has ease-of-use benefits, and Docker images are compatible with the HPC-friendly Singularity and Charliecloud container runtimes. If someone develops from a computer with a Docker daemon installed and running, they can test new Docker builds without re-building the entire stack from scratch. Docker will detect which image layers have changed, and only replay commands from that image layer and dependent image layers. This reduces redundant build procedures, and greatly expedites container development. Further recommendations for developing HPC applications using the container paradigm are discussed in Section 3.2.4.

Container image stack

The reproducibility benefits of the container image stack come from the ability to build image plans on top of previous image plans using the `FROM` keyword described in the preceding section. This feature makes Docker images composable. Dockerfiles are typically kept in a git repository, along with any dependency files a program may require. The Dockerfile itself is used to `COPY` files into the image recipe in the same style as the `RUN` command. There are two files which commonly appear in Dockerfile definitions:

```
entrypoint.sh
commands.sh
```

The entrypoint file establishes environment variables relevant to the container. The `commands.sh` file is where the workload application is called. Typically, `commands.sh` is called from `entrypoint.sh` so that the application workload is called from the context of the container rather than from the context of the host.

The layering techniques described in this section are commonly used by system administrators to deliver development environments to software engineers, but their applications to research software on distributed computers is novel to my work toward building a reproducible application platform.

2.3 System and application sampling

This work integrates and packages HPC performance monitoring tools as a part of a reproducible framework. Performance monitoring is necessary for performance prediction, and performance predictability is a common goal in systems research. I demonstrate that deploying a performance monitoring system can be simple to use with my Docker image layering technique.

A wide range of performance monitoring tools have been developed for HPC including LDMS [10], Tau [43], HPCToolkit [44], Paradyn [36], and Caliper [19]. Each of these systems have strengths and weaknesses, and each could be appropriate for inclusion in the system described by this thesis.

In my system, the lightweight distributed metric service (LDMS) provides system and application performance sampling. LDMS has access to any system metric which is mounted to the filesystem, and can be configured to sample at variable frequencies. LDMS can also be configured to write files to temporary storage, and read these back into the sampling infrastructure at the LDMS configuration's designated sampling frequency. This capability allows LDMS to wrap application function calls, write them to a fast filesystem, and report them in the sampled, time-order metric sets provided by LDMS.

LDMS is typically maintained by system administrators, and access to LDMS data must be granted on a per-user basis. In my system, LDMS is integrated as a Docker image layer. Users have total control over system performance sampling, and sysadmins don't need to manage access to metrics, or spend time maintaining these performance monitoring systems.

This work extends performance monitoring solutions like LDMS [10] by providing a means to package monitoring functionality in a software stack which is already demonstrated on a target infrastructure. Real world systems vary widely in their hardware and software configurations, so placing a pre-configured system like LDMS with a working set of default configurations is a time-saving feature for both sysadmins and for experiment pipeline developers.

2.4 Experiment workflow managers

This work extends the state of the art in experiment workflow management by combining the experiment workflow manager with a layered container image launch, and validating that a research artifact is output which conforms to the artifact description.

2.4.1 Research artifacts

A research artifact is the key deliverable for this reproducible platform. An experiment workflow is not falsifiable without a research artifact. The cTuning Foundation, and particularly their cKnowledge Project [25], began as an effort to evaluate compiler optimizations in a reproducible way. Through practical experience they found that reproducibility was difficult, and developed a set of research artifact criteria called the Extended Artifact Description Guide [24]. This is relevant to my work, because I've used the concept of research artifacts in the validation stage of my experiment pipeline. This work focuses on generating the research artifact. The reproducibilty platform specifies the processes leading up to the generation of a research artifact.

2.4.2 Experiment workflow managers

Experiment workflow management is a persistent and ongoing challenge for distributed computing systems, as demonstrated by the 14th annual workshop on Workflows in Support of Large-Scale Science (WORKS19). Re-usability, extensibility, repeatability, and performance monitoring capabilities are all common goals in the work on experiment workflow management. Tschueter et al. coupled system performance data with various stages of an application run [46]. This is similar to my work, but different because I utilize containers to inextricably link the performance monitoring tools and the application. Mitchell et al. discuss emerging features in workflow management systems [38], and they identified that modern workflow management systems typically possess either data-driven workflows and task-driven workflows. This work incorporates a data-driven workflow in the validation stage of the experiment pipeline, where the experiment will fail if the pipeline does not produce a research artifact. This workflow is task-driven because all of the steps leading up to the repeatable research artifact are clearly defined and tracked in version control systems.

2.4.3 Container-specific workflow managers

Containers are currently used in workflow management systems for HPC, as demonstrated by the popularity of container workshops such as the CANOPIE-HPC [2] workshop at the 2019 Supercomputing Conference and the Software Sustainability Institute's Containers for Docker Containers for Reproducibility in Reproducible Research (C4RR) [3] workshop. While these workshops describe how to use Singularity to launch Docker containers safely in the HPC context, most of the publications describe how to launch containers using the host communication library (bind method), or by matching the container communication library with the host's to facilitate distributed application launch (hybrid model) [4]. This work uses a technique that isolates the communication library within the container. The two-phase container runtime launch featured in this work enables the container to provide its own MPI. This technique is not widely known, and it is based off of personal discussions with Joshua Hursey at IBM and Scott McMillan at NVIDIA.

2.4.4 Machine learning workflow manager

Experiment workflow management is crucial in machine learning contexts, because model efficacy is necessarily determined experimentally. There are a wide range of tools being developed in this space. MLFlow [49] is representative of the general approach taken.

MLFlow is a machine learning framework which carefully designs inputs and outputs so that machine learning results can be cataloged for later analysis. This work is similar to MLFlow because MLFlow clearly defines end-to-end experiment stages. This work is different because MLFlow does not consider the software stack as part of their experiment workflow system. The software stack and its properties are the key contribution of this work.

2.4.5 Lightweight experiment workflow manager

While the experiment workflow system is crucial to the success of this reproducible application platform, the needs of this system are simple. A lightweight orchestration framework is sufficient to coordinate software dependency resolution, host/container coordination, and batch job scheduling. Popper [32] achieves all needs in a convenient Python CLI. Popper is used in this work to coordinate the coarse stages of the experiment. Popper enables reproducible conditional workflow execution, but it does not include a method to manage modular software stacks with layered image containers. A Popper experiment can repeat previous work, and it could re-use previous work with effort. The image layering technique I demonstrate is a separate and novel contribution from Popper.

Chapter 3

Reproducible Application Platform

3.1 Introduction

In this chapter I present an original reproducible application platform design, and an implementation of that design. The layered container image is novel and a primary contribution of this work. Layered container images move all application dependencies from the host to the container. The implementation of this design consists of popular tools used in new ways to demonstrate new was to built scientific workflows. The implementation demonstrates functionality with simulated HPC workloads and actual HPC workloads.

3.2 Platform design

The reproducible application platform consists of two major parts:

1. The experiment pipeline and container orchestration system; and

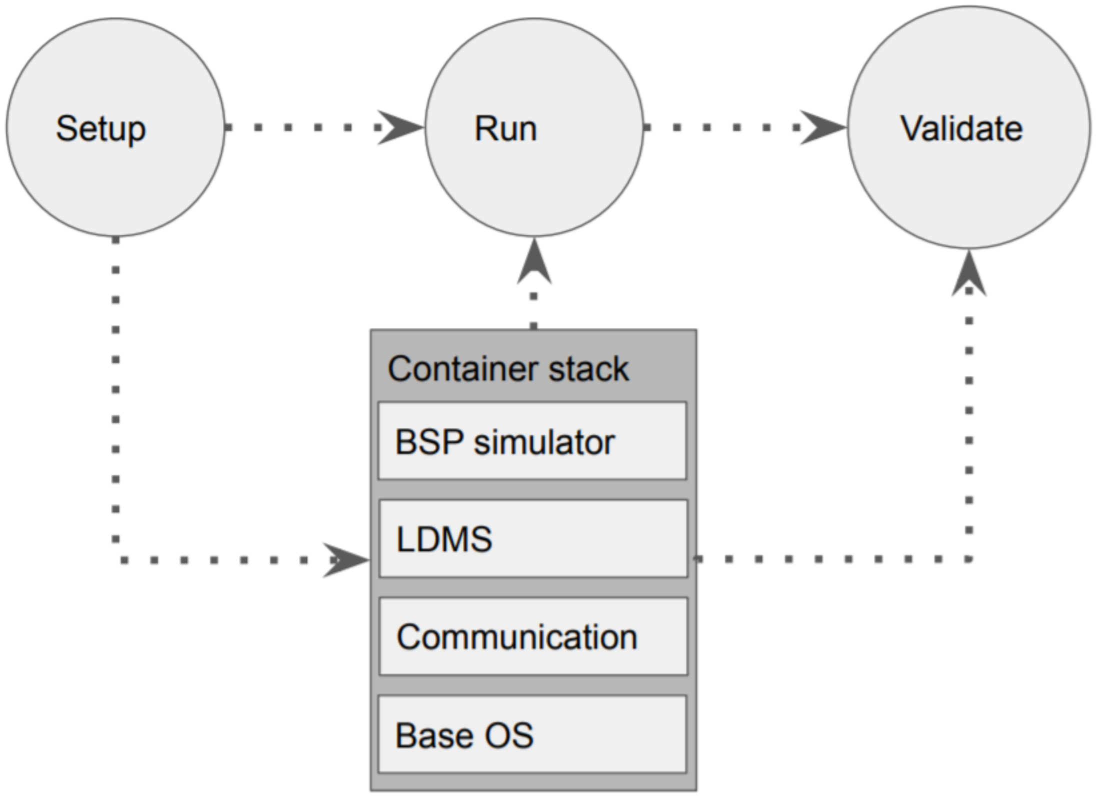


Figure 3.1: The experiment pipeline can be viewed as a DAG

2. The container image stack

3.2.1 The experiment pipeline and container orchestration system

The experiment pipeline is a tool to organize and deploy containers, coordinate sharing between host and container, and synchronize container deployments across nodes. The container orchestration system is invoked as part of an experiment pipeline.

Organize and deploy software stacks

The experiment pipeline is responsible for selecting and deploying software stacks. The reproducible software stack is defined independently of host configuration. The experiment pipeline selects a software stack by referencing a particular, cloud hosted container. The pipeline is tracked using version control. The logical separation between pipeline and container can be seen in Figure 3.1, where the nodes represented as circles are considered separately from the node represented as a rectangle. The nodes represented as circles (experiment pipeline nodes) depend on the node represented as a rectangle (container image stack), but the container image stack can operate independently. The ability for the container to operate independently is an important development feature, as described in the section on accessibility, Section 3.2.4. Containerized software stacks are different than the experiment pipeline, and their design must be considered independently of the pipeline. Each container image is managed by its own version control. The most fundamental container definition is a version control repository, which contains a recipe for how the container is built. Building the container takes time, however, so the definition file referenced by version control will also have a pre-built image that may be downloaded and executed to avoid that build time. The version controlled recipe file provides provenance, and the pre-built container offers a fast deployment option if those assurances aren't required.

Coordinate sharing between host and container

This system uses containers to isolate software resources so that the containerized application references containerized dependencies, ignoring the host configuration almost entirely. The two things a container can not isolate are the kernel and the host's job scheduler. Host kernel sharing is an important performance feature. The implications of host kernel sharing are discussed in Section 2.1. The host's job scheduler can't be isolated because this is the component that actually grants the user permission to run on the individual nodes of the distributed system.

The experiment pipeline is responsible for launching the container image stack across many nodes. The experiment pipeline is responsible for supplying application input parameters, and specific output directories that may be unique to each user or each experiment run. To handle this requirement, the experiment requires an interface between the host and container which is well defined. The purpose of this interface is to define which environment variables from the experiment pipeline (host), should be shared with the deployed image stack (container).

Synchronize container deployments across nodes

The previous section discusses how the experiment pipeline and a single deployed container image stack operate together. This section describes how containers are deployed across nodes and synchronized. Containers are downloaded from some public and trusted source in a binary format. The container binary is unpacked into a directory somewhere on a host's mounted filesystem. It is possible to run a container directly from the binary, without unpacking. However, deploying the container binary rather than running the binary directly has four advantages:

1. Deployed containers are easier to work on during development phases. This is discussed more in Section 3.2.4.
2. Totally unprivileged container launch (no SUID) requires a deployed container image.
3. Binary container launch consumes memory unnecessarily. When launching a container from a binary, the entire binary is loaded into system RAM during

runtime, for each container launch on that node. If multiple processes need to use the container during an experiment run, then deploying the image stack is more memory efficient because the container files are deployed on a read-only filesystem mount point. Multiple containers can launch from the same deployed container. Running binaries directly will load all container image files into memory, one per instance.

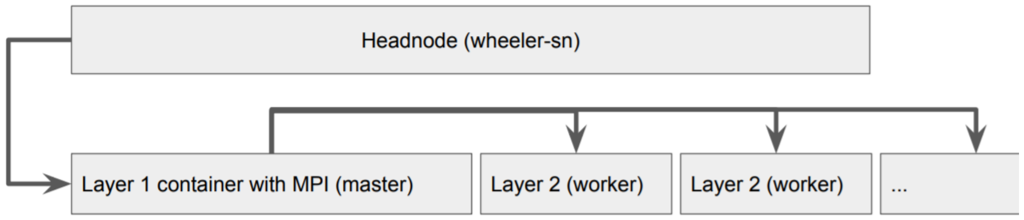
4. In order to coordinate the host with the container, the container files must be edited at runtime to account for the host configuration. This is crucial for container orchestration, as discussed later in Section 3.3.1.

This section has described how containers are synchronized across a computing environment. Distributed application launch is discussed in the Section 3.2.2.

3.2.2 The distributed container launch

One of the design goals of this system is to encapsulate the container environment, so that the only host software dependency is the container runtime itself. In this case, I use Popper to manage our experiment pipelines. Popper is a Python tool with a CLI that can be invoked with a simple *popper run experiment* command. This is technically a dependency, though we could achieve similar results through bash scripting alone. Popper is discussed more in Section 1.1.1, Section 3.3, and Chapter 2. In order to achieve my host isolation goal, a communication library is used to interface with the host scheduler and establish communication paths between nodes that will run the distributed application. See Section 1.2.3 for background on communication libraries. I use a layered container runtime launch strategy to isolate the container from the host. The container runtime launch occurs in two phases: the L1 container launch and the L2 container launch. This technique *deploys the same*

Figure 3.2: A typical launch of L1 and L2 on an HPC system



container, twice. This method does not require two separate, deployed container environments, but it does require two separate container runtime launches.

The L1 container

The layer 1 container (L1) is launched on a master node using the host-provided container runtime. Consequently, the container environment must contain a container runtime which is application binary interface compatible with the host system.

The L2 container

The layer 2 container (L2) is launched from the container environment. This requires that the container has a container runtime somewhere in it's software stack. To repeat the point made regarding L1, the *container's* container runtime must be ABI compatible with the host's container runtime. A communication library from the deployed container image stack is launched in L2 to establish connections to worker processes. The host's communication libraries are effectively circumvented.

3.2.3 Container image stacks

As described in Section 2.2.1, there are two files which commonly appear in Dockerfile definitions, and I've added a `env.sh` for this system's design:

```
entrypoint.sh # Container environment definition
commands.sh # Workload run script
env.sh # New with the reproducible
       # application platform for
       # distributed systems.
```

The purpose of `env.sh` is to selectively communicate host environment information to the container. After the container image is deployed, this host/environment sharing script must be copied somewhere into the deployed image stack. Finally, `env.sh` must be referenced in the container environment file, `entrypoint.sh`, so that the container runtime will fail immediately if host/container environment sharing is undefined. This design places a strict requirement for host/environment sharing, which is a core requirement for any isolated running environment operating on a distributed system.

3.2.4 Development paradigms

Deployed, layered image stacks make development tasks easy in several ways:

Convenience and cost efficiency

Developing software stacks on distributed computing systems can cost money. Containers allow us to do most development work on a local computer, where compute time is not charged to a cost account. Working locally also means that a connection to the distributed computer isn't required to make progress on a project.

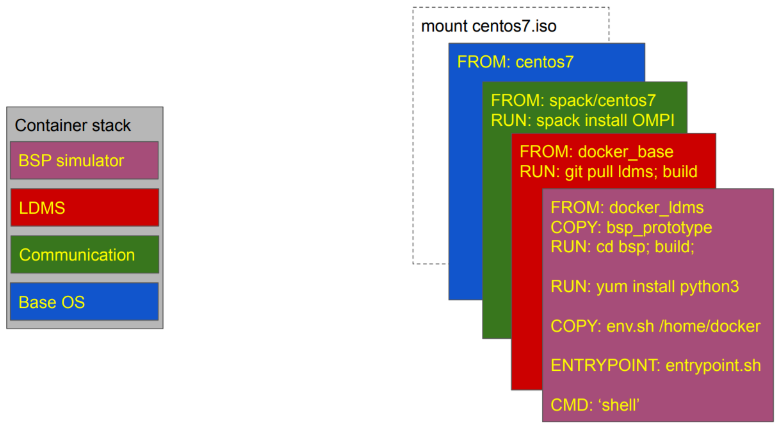


Figure 3.3: Image stacks build functionality iteratively, reducing time to deployment

Re-usability

The layered image approach discretizes the functionality of a software stack. Each layer defines its own interface to utilize functionality built into that layer. The required interface consists of: a container environment script, a host/container sharing script, and a command script. The command script demonstrates useful functionality at that layer.

Development modes expedite project delivery

Container development can happen iteratively and quickly, without needing a full software stack re-deployed for minor changes. I've identified two development modes which have been useful in expediting my work in building and testing reproducible experiment pipelines for distributed computers (see Figure 3.5):

1. **Local development mode:** Occurs on a local computer.
2. **Deployed development mode:** Occurs on the distributed computer, at the filesystem mount where a container is deployed.

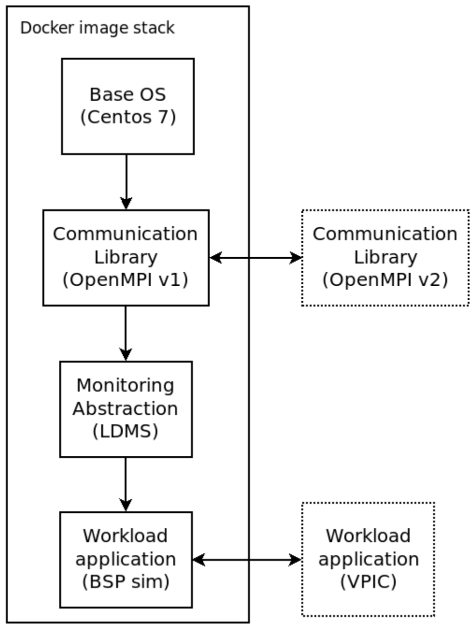

Docker image stack
Base OS (Centos 7)
Communication Library (OpenMPI v1)
Communication Library (OpenMPI v2)
Monitoring Abstraction (LDMS)
Workload application (BSP sim)
Workload application (VPIC)

Figure 3.4: Image stacks enable efficient experiment re-usability through modularity

Local development mode is useful when a connection to the target infrastructure is unavailable or when compute time is expensive. Local development happens on a local computer using a container editor (Docker). Deployed development mode is useful when troubleshooting a particular application, and greatly reduces the time to make and test changes, as seen in Figure 3.6. Deployed development takes place on the target infrastructure, where containerized scripts can be edited and re-run, either with the help of the experiment pipeline or without. The container image can execute in a standalone mode for quick tests. Without deployed development mode, even a simple `container run "echo hello world"` command would require that the entire, potentially hour-long process depicted in Figure 3.6 take place.

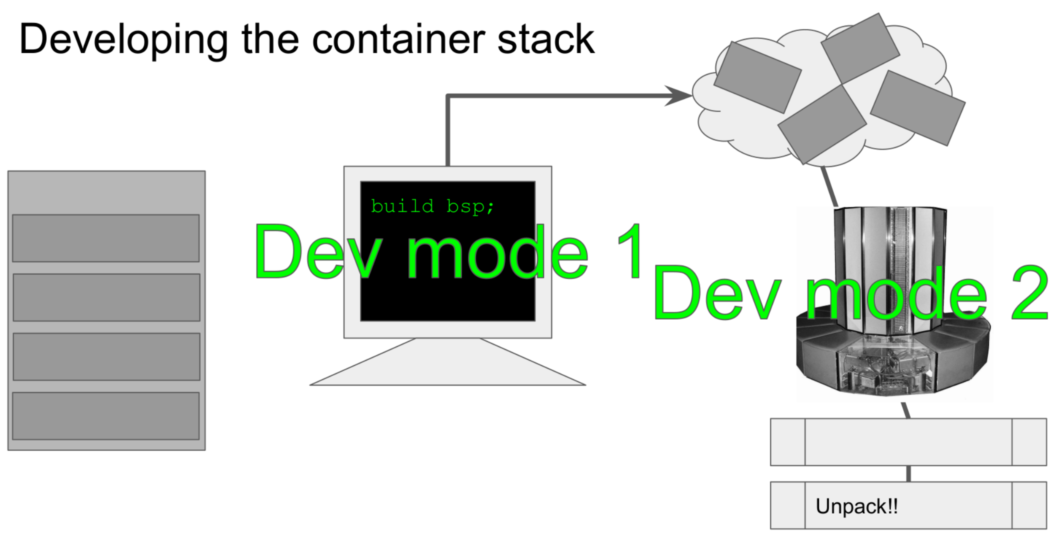


Figure 3.5: There are two development modes: local development, and deployed container development.

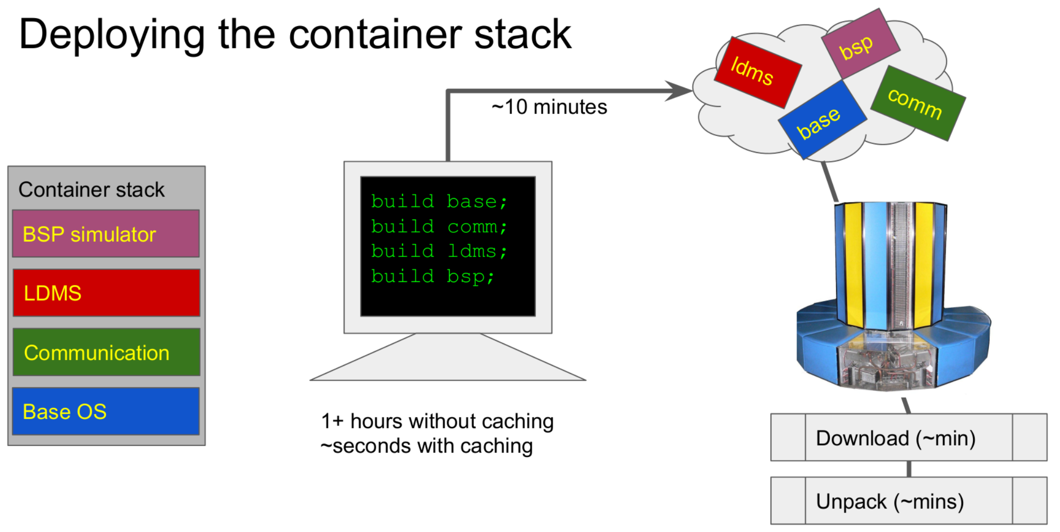


Figure 3.6: Local images are pushed to a cloud provider, downloaded to a distributed computing environment, and then deployed to a host filesystem during experiment pipeline execution.

3.2.5 Research artifacts

As discussed in Section 1.1.1, reproducibility is a larger concept than repeatability. The reproducible pipeline must produce evidence, but because it's goals are more general than repeatability, another metric of success is required. In this design, every reproducible application platform is required to output a research artifact. The research artifact is data which has been sufficiently processed to enable validation. The research artifact does not necessarily validate a particular result, but a result should be falsifiable based on the research artifact alone.

3.3 Platform implementation

The previous section describes attributes I identified as important for reproducible application platforms on distributed computers. This section discusses the details of an implementation of these design principles. Figure 3.1 illustrates the high-level structure of the platform described in this section.

3.3.1 Experiment pipeline

In this section I will discuss a particular experiment pipeline, and how the pipeline framework generalizes to other experimental needs. I am using a python tool, Popper [32], to manage the experiment pipeline. Popper uses a Github workflow syntax configuration file called `main.workflow`. This file is JSON format, and defines a directed acyclic graph (DAG). The DAG represents experiment logic. Experiment stages may be defined with parallel dependencies, but a sequential experiment pipeline was sufficient for this work.

My pipeline consists of three sequential stages: setup, run, and validate. These stages are discussed in detail below:

setup

The purpose of the setup stage is to define an input deck for the application being studied and to synchronize a container with the host.

Define input parameters. The first step an experiment pipeline user must perform is to edit the host/container environment sharing script, `env.sh`. A pipeline user is someone who understands the high-level function of the software stack but lacks knowledge of its detailed implementation. A pipeline user might want to change experiment parameters to study output research artifacts, for instance. The `env.sh` script that a pipeline user can modify is found in the root of the experiment pipeline directory. In `env.sh`, we specify paths to the container runtime, paths to output directories and application workload input parameters (such as how many nodes and processors to use for a parallel application run). After the host/container `env.sh` script is defined, the pipeline can be executed, beginning with the setup stage.

Pull and deploy a container image. The setup stage begins by pulling a valid container image from Dockerhub, using Singularity. The container downloads as a SquashFS binary file. The container is then unpacked in the container image directory specified by `env.sh`.

Communicate host environment to deployed container. Once the container is downloaded and unpacked, `env.sh` is copied *from the experiment pipeline to the container itself.* In an effort to safeguard a container against launching without knowledge of its host environment, the first thing the container does in its entrypoint file is attempt to call `env.sh`. If `env.sh` does not exist, the container will fail before any real work is done. This ensures that some information is shared between the container and the system it is running on. Once the container is unpacked and the environment shared, the setup stage is complete. For a refresher on entrypoint files, see Related Works Section 2.2.1.

A note on more thorough provenance. Although the pipeline saves time when pulling an image directly from Dockerhub, full provenance is achieved by building from the Dockerfile image recipes hosted on Github. As described in Section 3.2.1, container images are defined in version control separately from the experiment pipeline. In this implementation, a GitHub repository exists for each container. There is a branch for each container which is named to match the experiment pipeline it is paired with (`carc-wheeler` and `tacc-stampede2`).

run

The run stage is responsible for running the application workload. The run stage makes requests to a job scheduler. This pipeline runs on Wheeler with the PBS/Torque scheduler and it runs on Stampede2 with the SLURM scheduler. Git version control allows a pipeline user to swap between a `carc-wheeler` and a `tacc-stampede2` branch, which makes the appropriate changes to the run script. Run parameters which affect job launch and runtime behavior were set in the `env.sh` file, described in the setup section.

validate

The validation stage is generally used to say something about the hypothesis an experiment is designed to test. Research artifacts are used to formalize this need without strictly enforcing any particular result. A research artifact is used as evidence toward some experimental goal. In this pipeline, I wanted to couple system performance data with application data. I use a Pandas dataframe to achieve this, which is essentially an array with self-describing metadata attached. During the validation stage of the pipleline, a python script takes the JSON format application output, and merges it with CSV format LDMS data. The application state is coupled with the system state, so we can investigate the state of the system at a particular stage of the application's execution.

In order to make sure these dataframes can be generated, additional software was added to the container image stack for analysis. Although it's common to expect some flavor of Python to be available on systems, I decided to add these dependencies to the container so that the pipeline is well encapsulated against potential versioning issues that might prevent dataframe generation, causing the entire pipeline to fail at the last stage.

3.3.2 Layers of the container image

This section describes particular layers of the container image used in this work.

Base image layer The image container stack was tested on systems where Centos7 was the host operating system. The base image layer for the container stack is Centos7. Centos is an open source Linux distribution that is notable for its use on many clusters. The base image distribution was chosen to match the host operating system.

Communication layer The first image layer developed specifically for this work is called `docker_base`. The image recipe is hosted on Github and may be downloaded from https://github.com/unm-carc/docker_base. In this git repo, and all associated git repos, there is a branch associated with each target infrastructure. The relevant branches are `carc-wheeler` and `tacc-stampede2`. The image recipe includes a Dockerfile, and Spack environment files. Spack [27] is a tool designed to build software from source trees on demand. This allows Spack users to have greater control over build parameters for software dependencies. Spack will, by default, build all source dependencies from scratch which can take a long time. Spack gives greater control over packages at the expense of build time. We may not always need packages to be built from a specific source tree version, or a Spack package maybe broken or unavailable. To tell Spack to rely on the host to provide certain system packages (such as openssl), we define a `packages.yaml` file. So long as the package exists in the host operating system before Spack launches, Spack will simply rely on the host versions of these dependencies rather than trying to build them from scratch. The Spack environment is a set of software packages and their build options. You can formally define a spack environment in a `spack.yaml` file. This file is where we select build parameters for the communication library that the image stack will use, OpenMPI version 3.1.4. There is a minor difference between the `carc-wheeler` branch and the `tacc-stampede2` branch. The Stampede2 branch builds OpenMPI with the Slurm plugin, which is required for communicating with the scheduler on that system. Similar functionality is provided on Wheeler by passing the `hostfile` around to all nodes.

Monitoring layer The Lightweight Distributed Metric Service (LDMS) provides a practical means to measure the time between collective intervals during the execution of a bulk synchronous parallel program. LDMS works in the client/server paradigm, with system *samplers* acting as clients, and one or more system *aggregators* working as the server. Samplers are configured to "watch" specific files mounted on the filesystem at a given interval, such as `/proc/meminfo`. It is possible to sample inside or outside of the container. The host has a view into the container, but the container does not have a view of the host unless specifically granted by container runtime parameters. LDMS will transform the data within these files, `/proc/meminfo` for example, to conform to a regular schema. The LDMS aggregators are also configured with an independent sampling rate and set of metrics to sample. The aggregator pulls data from the samplers at that interval, and either writes the data to the filesystem or submits it to a queueing system (such as RabbitMQ [42]). This system uses a single aggregator node, pulling from every sampler in 2 second intervals, and writing data to CSV in a pipeline-configured logging directory. LDMS samplers are configured to read data in 1 second intervals from:

- `/proc/meminfo`
- `/proc/stat`
- `/proc/interrupts`
- Special counter files located in `/dev/shm`

In his dissertation and in a subsequent publication [30], Izadpanah extends LDMS to include MPI sampler functionality. He builds this into the framework by generating wrappers for MPI function calls. When an LDMS sampler starts up, it begins writing to files in `/dev/shm`. Every sampling interval, the aggregator reads these counter files in the same way as it does `/proc/meminfo`.

LDMS as a pod service Container ochestration frameworks like Kubernetes [18] and Podman [7] use the term pod-service to describe services that must run as separate PIDs alongside a workload application. LDMS runs as a daemon on each sampler node, and the aggregator runs as a separate service on the master node. These daemons must be coordinated with the workload. The aggregator and each sampler are launched as a standalone container process before the workload runs. Samplers are killed after the application workload completes, and the aggregator is finally killed by the last stage of the experiment pipeline.

Convenient scripts for launching these pod services are referenced in the entrypoint.sh file at this image layer. The entrypoint file also, critically, defines the `LD_PRELOAD_LIBRARY` environment variable which is used to wrap and count MPI calls.

Workload application layer The workload application is a bulk synchronous parallel application simulator. A bulk synchronous parallel application is any program that can be broken down into periods of parallelism with cycles of synchronization. Mondragon et al. show that under certain conditions, application run-time can be predicted at scales previously untested [39].

The performance model If the time between communication collectives can be represented as an IID random variable, then this is a special class of BSP program. In this special case, the time between collectives can be determined by the maximum run-time of participating worker nodes. According to Generalized Extreme Value theory (GEV), we don't need to know the distribution of the IID random variable between collectives in order to model the maximum of these samples. So long as the random variable has some distribution, we can use GEV to predict maximums, given sufficient sample size.

The stages of the simulator program include:

1. Start the application on the master node, using MPI
2. Worker nodes launch. All nodes generate a sample from the same random variable. The RV's available are: Gaussian, exponential, Pareto, uniform, constant.
3. Each worker node sleeps for the time generated by step 2.
4. A cycle completes with an MPI barrier collective. We measure the sleep time of each worker node, including the sleep time of the maximum node.

3.3.3 Distributed application launch mechanism

The application has been tested in the past outside of this experimental pipeline, and outside of the container context. Making the app work in this context is original work. The application is launched via the `commands.sh` file, from the L2 container context. Output paths are specified in the `env.sh` script, which establish shared paths between host and container. The `LD_PRELOAD_PATH` from the LDMS `entrypoint.sh` file is carried over from the LDMS layer. Finally, the communication library must be considered to successfully execute a multijob launch.

Launching containers in parallel with OpenMPI The mechanisms use to launch an MPI workload are dependent on the MPI implementation. In one popular implementation, OpenMPI, when `mpirun` is called, a connection is established to the participating worker nodes, and a service daemon (`orted`) is launched along with a list of OpenMPI derived arguments. These arguments establish what hardware is used for communication, among other launch configuration details. Ordinarily, the host system will search system paths for `orted`. Since our container provides its own communication library, relying on system `orted` would break our distributed container environment.

OpenMPI provides a shim that allows us to launch a script rather than the `orted` available in the host system path. The environment variable that must be set is `OMPI_MCA_orted_launch_agent`. This environment variable is set to refer to a script in `commands.sh`, just before the L2 `mpirun` command.

When `mpirun` is invoked, the launching process is already in the contianer environment, so no further action is required on the MPI master node. However, before OpenMPI sets up communication on the worker nodes, `orted` must launch from the container. The script that does this, in this case, is called `ompi_launch.sh`:

```
module load singularity

singularity run bsp_prototype orted $@
```

Note that the L1 container launches with by loading the host module directly, in a Bash script. All the worker nodes launch using communication library mechanisms, which load a host provided singularity module and then start a service daemon from the context of the container. This is the reason why it's important to match container runtimes between the host and the container.

When singularity invokes the `run` command, `entrypoint.sh` is invoked. To reiterate, the entrypoint script is used to set up the environment for the container. The name of the container image here is `bsp_prototype`. The next argument, `orted`, falls through a case statement in the entrypoint script until it matches orted, and then launches `orted` from the container context.

That is to say, the OMPI launch script quickly transitions form the L1 context to the L2 context, with crucial `orted` command line parameters passed along with the `$@` call in the entrypoint.

3.3.4 Re-using and extending the container image stack for new work

Each layer of the container image stack is described by a set of scripts: a Dockerfile, a container environment definition file, a host/container environment definition file, and a command file.

Dockerfile

The Dockerfile begins with a FROM statement. This is the layer that serves as the base of a new image. It consists of scripts which modify the software stack, and concludes with defining an ENTRYPOINT and CMD. These refer to a container environment definition file and a command file, respectively. When re-using a layer of the image stack, the Dockerfile should be edited to suit the needs of your new layer. If you're extending the image stack, a new Dockerfile should be created using a previous layer as a template.

The entrypoint file

The container definition file (typically `entrypoint.sh`) defines environment variables that pertain to the container. This could include `PATH` environment setting, or other global variable that are important to your application. When re-using a layer of the image container the entrypoint file can be modified to suit your needs. When extending a layer of the image file, you should consider the container environment definition file in the layer immediately beneath the new layer. Make sure the new entrypoint file persists global variables that ensure the functionality of the software stack beneath the new layer, and add new variables for your layer as needed.

The env file

The `env.sh` file is used to define the elements of the host configuration that must persist to the container. This includes output directories and application input parameters. The rules for modifying and extending the verb—env.sh— file are the same as the entrypoint file.

The command file

The command file, typically `commands.sh` defines useful functionality at a given layer. This functionality may be as simple as a call to "bash" so that the container can be used interactively, or it may be a script that does something more complex. You can define your command file without consideration for previous image layers, because the command file is responsible for executing the tasks that your image stack is designed to provide. It does not matter what previous layers have done. The only command file that matters is the one on the top of the stack.

Chapter 4

Platform Evaluation

4.1 Introduction

In this Chapter, I evaluate the implementation of my reproducible application platform. Section 1.1.1 describes how *repeatability* is a more specific goal than *reproducibility*. Repeatability is important to validate results and previous work. Reproducibility is a broader goal to reproduce previously published works while re-using that work and eliminating the need for re-implementation. Consequently, evaluating this reproducible application platform will be considered in five parts: repeatability, portability, accessibility, re-usability, and extensibility.

4.2 Evaluating the reproducible application platform

The evaluation criteria for this reproducible application platform are:

- **Repeatability:** Results can be demonstrated on demand.
- **Portability:** The system can be modified to run on different systems.
- **Accessibility:** The system is easy to use and develop.
- **Re-usability:** Previous work can be re-used in future work.
- **Extensibility:** The system is modular and can consist of multiple dependent parts.

The system will be evaluated based on these goals.

4.2.1 Repeatability

This platform guarantees repeatability for a specific infrastructure. For the first experiment pipeline, if results are previously published, pipeline outcomes may not agree with previous research outcomes if the state of the system and experiment design was not documented. However, in the future, this pipeline will guarantee that a user can run the same experiment on the same system to generate a research artifact. Repeatability is not necessarily confirmation of the research outcome. It is confirmation that a repeatable process exists to produce a tangible research artifact. The research artifact is evidence that may be interpreted and discussed regarding published claims.

Research artifacts

A research artifact is a summary that may involve figures, statistics, or other data products which serve as evidence of a research conclusion. While distributed application performance will differ based on the hardware and system state over multiple runs, a research artifact is always produced. The research artifact is designed to be a link between the hypothesis and an evidence-based conclusion.

This pipeline has been run by multiple users on CARC's Wheeler system. In all cases, a Pandas dataframe is output. The Pandas Dataframe is the research artifact. Based on these evaluation criteria, the reproducible application platform meets this repeatability design goal.

4.2.2 Portability

Portability refers to the ability to take an experiment, modify it slightly or not at all, and run it on a new system. A portable experiment may yield interesting new data with very little work by the researcher. This section discusses how my design serves the portability goals outlined in the introduction. It describes safety concerns particular to shared distributed computers, and how Singularity mitigates that risk. Finally, it explains how portability was demonstrated in my implementation.

Docker and Singularity

Docker enables local container development, and Singularity deploys and runs the container on a distributed computer. Docker's image development platform allows a pipeline developer to work on a local computer which can be very helpful when troubleshooting build parameters and small-scale run-time functions.

Singularity as an unprivileged container runtime

Some shared infrastratructures, such as HPC systems, do not allow users to have access to privileged commands (root access). Local development using Docker image layering does utilize a privileged account, and this design assumes the user has a privileged account locally. Once the image is built and subsequently hosted, however, Singularity pulls those images from a cloud-based host, deploys to a filesystem may then run applications in an unprivileged mode using the User namespace described in Section 2.1.

Portable communication systems for distributed computers

The container image stack coordinates its own launch using a 2-phase, layered container runtime launch. The communication layer of the image stack is job scheduler aware, through build parameters chosen in the communication layer Dockerfile. MPI implementation built into its communication layer. Since the container encapsulates the communication libraries needed to run a distributed application, the dependencies for the distributed application are fully isolated. The only dependency and requirement is that the host container runtime matches a container runtime present within the container. This encapsulation provides application communication portability across platforms, despite the communication libraries a host system might support.

Evaluating portability of the implementation

I demonstrate portability by compiling and executing a simulated BSP workload through an experiment pipeline deployed on both CARC-Wheeler and TACC-Stampede2. The container was first successfully interfaced with Torque/PBS scheduler on the Wheeler cluster. I successfully ported the container to Stampede2, which uses the Slurm scheduler. This port was achieved with a simple change to the Open MPI build stage. I changed the Dockerfile of the communication layer to port the image stack from Wheeler to Stampede2. After changing this file, I rebuilt the entire image stack and deployed on Stampede2. The approximate time to convert

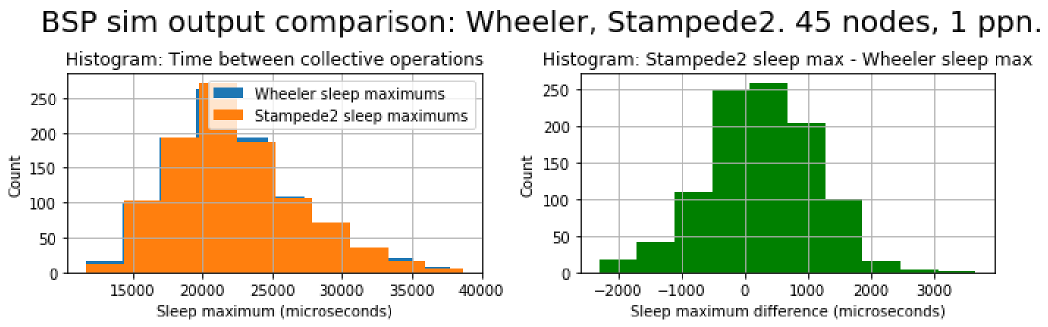


Figure 4.1: The Bulk Synchronous Parallel (BSP) application performs similarly on Wheeler and Stampede2. Intervals between collectives follow similar distributions on both systems.

this experiment from Wheeler to TACC was 15 minutes. The build procedure was unattended and finished in about 2 hours. This success is proof that the system meets my portability goals. Notably, the container used Open MPI as the communication library on TACC, which demonstrates that this system can operate without being constrained by communication libraries on the host (MVAPICH2).

I collected samples at various scales with the same experiment pipelines, and validated that research artifacts (Pandas dataframes) were produced. I performed a set of experiments on both Wheeler and Stampede2 at the following scales: 1, 10, 20, 30, 45. The application outputs on both platforms are compared in Figure 4.2.

4.2.3 Accessibility

Accessibility refers to ease-of-use. The solution should be easy to develop, and it should be easy to use. This approach makes development and usage easier by clearly defining experiment stages and stage interfaces. Usage is made easier by utilizing tools which are either self contained, or automate dependency resolutions.

Clearly defined workflows

Experiment workflows can be represented as directed acyclic graphs (DAGs). These workflows are executed with a lightweight orchestration framework, Popper. A single `popper run` command will execute all stages of the DAG (see Figure 3.1). The setup stage of the DAG satisfies dependencies, the run stage runs the workload application, and the validation stage prepares the research artifact.

The DAG abstraction makes experiment design tasks clear and therefore easier to implement. Popper makes experiment execution easy by issuing two simple `git clone` and `popper run` commands.

Freedom to work locally

A user begins developing the container image on a local computer with Docker. Docker provides an interactive command like which is very similar to the Linux command line. Saving the state of a Docker image is accomplished through issuing the `docker commit` command. These Docker images can be edited on the local computer, such that an active connection to the remote resource is not required for much of experiment pipeline development process.

Flexibility to work remotely, offline

A user may edit the deployed container on the distributed system, as appropriate. The user then deploys a Docker container image onto the filesystem with a `singularity pull` or `scp` command. The deployed container image is now mounted on the filesystem, where all files are visible. The experiment pipeline will execute them the next time it runs.

Together, the experiment pipeline DAG makes the coarse grain experiment logic easy to understand for an **experiment workflow user** and the multiple development modes make the pipeline easy to use for **experiment workflow developers**.

4.2.4 Re-usability

Re-usability is the ability to take an existing work and modify it for new purposes. I consider each added feature in the software stack a potentially re-usability feature. The design is modular, so that any layer can be swapped out for a different layer while leaving intact the layers above and below it. As shown in the Section 2.2.1, this modularity is achieved through the use of Docker image layers and git-tracked Dockerfile definitions. This platform achieves re-usability, as demonstrated when I successfully ported an experiment pipeline from the Torque/PBS scheduler (CARC-Wheeler) to a SLURM scheduler (TACC-Stampede2). The change was a simple Open MPI build option added to the image stack at the communication layer.

Later, I replaced the application workload, a BSP simulator, with a real parallel application, VPIC [21] [1]. In doing so, I demonstrated that all the benefits of the underlying image stack were available to the new application. LDMS sampled data for VPIC, just as it had with the BSP simulator. It took less than 15 minutes to implement this change.

Evaluating the re-usability of the implementation

In order to swap communication layer of the image from a Torque/PBS-aware build of Open MPI to a Slurm-aware build of Open MPI, I modified the Dockerfile call to build Open MPI. I included the `--with-slurm` build parameter.

In order to swap the application layer of the image from the BSP simulator app to the VPIC app, I copied the Dockerfile from the BSP app, and removed the lines associated with building that BSP app. I added a line to clone the VPIC application into the container image. VPIC is unusual in that it builds the application with its input deck at runtime, so the app was not built from source in the Dockerfile definition. I also had to persist the `entrypoint.sh` file for the VPIC layer, because this file contains references to the `LD_PRELOAD` library that allows LDMS to count Open MPI function calls. I defined a new `commands.sh` file which calls Bash. I executed the container stack interactively on two nodes, and ran one of the sample VPIC input decks on those two nodes. I validated that the program terminated successfully, generated expected outputs. I also validated that LDMS was sampling the system and MPI calls over the execution of that sample input deck.

By swapping the communication layer, I demonstrate that higher levels of the image stack are re-usable. Swapping the workload application layer demonstrates that the lower levels of an image stack are re-usable.

4.2.5 Extensibility

Extensibility is related to re-usability, but refers to the ability to build added functionality into a software stack while leveraging previous work. Original image stack prototypes ran the workload application itself. Later, I added a system sampling layer with LDMS built-in. I extended the software stack with new functionality, demonstrating that this solution satisfies the extensibility design requirement.

Evaluating the extensibility of the the implementation

After extending the image stack to include LDMS, I ran the experiment pipeline to produce a research artifact. I show an example of coupled application and system

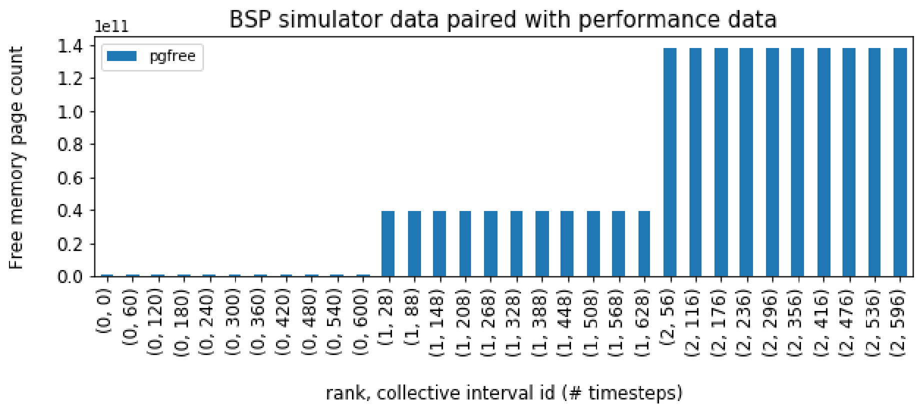


Figure 4.2: Coupled system and application data shows that the implementation's research artifact to explore the state of a system at a particular stage of an application run.

performance in Figure 4.2. The application state is reflected by the rank and interval labels on the x axis. The system's state is reflected by the `pgfree` metric on the y axis. The `pgfree` metric is just one of 897 other available performance and application metrics in the research artifact.

4.3 Reproducing a previous work

This section demonstrates that the platform fulfills these design goals by reproducing a published result. In this case, I implemented a software stack and experiment pipeline to evaluate claims based on an application performance predictability model.

4.3.1 The BSP prediction model

The model is valid for all bulk synchronous parallel (BSP) applications. The model takes time between collective intervals as inputs, and predicts application run-time at previously untested scales. To evaluate this model, a BSP simulator was produced by Mondragon et al. [39]. The BSP simulator draws a series of random numbers from a configurable random variable distribution, and runs in parallel on as many nodes as there are random numbers. Each node is instructed to sleep and then an MPI Barrier is called. The time when the `MPI_Barrier` call returns is recorded, and a new iteration of the work simulator begins. This goes on for as many iterations as the user specifies.

4.3.2 Collecting application outputs

The BSP simulator was developed explicitly for the purpose of evaluating the application run-time prediction model, and so the time between barriers is measured and written to an output file in JSON format. The simulator application outputs include: the random variable distribution from which sleep times are drawn, the shape parameters for that distribution, the number of times a collective interval should be performed, the name of the node performing a barrier call, the requested time between intervals, and the actual time between intervals.

4.3.3 Enabling general collective interval measurement with performance monitoring tools

In real applications, however, the time between barriers would more likely need to be inferred somehow. The monitoring software in this work, LDMS, counts the calls to all Open MPI calls at a configurable interval on the order of seconds. LDMS jointly collects system state and application state information. The validation stage of the pipeline couples the state of the simulator program to the period between interval start and stop tags. The BSP simulator establishes a ground truth for the collective interval, and communication library data from LDMS can be used to approximate the collective intervals. The analysis required to estimate those collective intervals with Open MPI counters is future work, and described in Section 5.2.1.

Chapter 5

Conclusion

The goal of my thesis-work was to build a reproducible application platform for distributed computing that allows platform users to validate and re-use previous work to build new work. In this chapter, I present a summary of my contributions and explore future work.

5.1 Summary

In evaluating my thesis, I considered what attributes a useful reproducible application platform would have and designed a reproducible application platform with those attributes. I implemented the reproducible application platform based on those design goals. I introduced a novel layered container approach that makes scientific results easier to repeat, and easier to re-purpose for future work. Finally, I evaluated the reproducible application platform based on my design goals.

5.2 Future work

There are many future directions that will extend this reproducibility platform, some which are direct extensions of the current work and others could more broadly improve the way HPC systems are managed.

5.2.1 Using communication library calls as a proxy for application behavior

In this work, I measured application outputs, system state information, and Open MPI function counters and sent them all to the research artifact. One analysis goal I did not get to was using the Open MPI call counters to estimate time between collective intervals. A naive approach would watch the calls to `MPI_Barrier` and add the sampling intervals every time the counter is incremented. A more sophisticated approach would also look at calls like `All_Gather` and blocking receives.

5.2.2 HPC job scheduler with layered container base images

Modern job schedulers run on HPC systems that run a particular operating system. Software is installed and managed by system administrators on a base image. Base image software may be supplemented with module loading systems. The module loading system is also managed by system administrators. In the future, it is possible for HPC centers to run a bare-bones host operating system and build a base software stack as a layered image container. Users of the system can download this container image to their local computer, swap layers, add layers, and finally upload the image to a remote host. This model greatly simplifies sharing software stacks with collaborators. This will greatly reduce the burden on system administrators, and allow users more control over their computing environment.

5.2.3 Pantheon: reproducibility with simpler tools

Not all reproducibility efforts are containerized. Pantheon [6] is a new project at Los Alamos National Laboratory with many of the same reproducibility goals mentioned here, but the pipelines are achieved through BASH scripting and Git version control alone. The focus of Pantheon is to create reproducible experiment platforms with a small set of simple tools. Pantheon grew out of the Cinema Database [11] project, and has been designed and implemented in parallel to this work. Pantheon experiment pipelines always produce a research artifact in the form of a Cinema Database [11]. Pantheon currently operates by automatically building software into loadable modules, rather than by providing pre-built software stacks through containers. Pantheon is part of the Exascale Computing Project (ECP), and work will continue over the duration of that project.

5.2.4 Volunteer computing

With the success of projects liked Folding@Home [17] and SETI@Home [13], anyone with a computer can and may choose to volunteer spare cycles to solve specific scientific problems. Projects like the Berkeley Open Infrastructure for Network Computing (BOINC) [12], are building out the communication required for large scale distributed computing science. Using something like the layered container approach with BOINC may allow consumer devices to run arbitrary applications rather than applications which are developed in one-off software development efforts. The consequence of this would enable generic HPC research on a volunteer system. A scheduler in the cloud could coordinate these applications across a volunteer network, enabling "citizen scientists" to contributed to generic HPC research tasks.

5.2.5 Containers over virtual machines

Another potential direction for flexible software stacks could take advantage of emergent full virtualization technologies with container technology overlaid. As mentioned in the Section 1.2.2, not all virtualization overheads are problematic. Increasingly, applications are bottlenecked at accelerators (GPU's, NICs) rather than at the processor. In these cases, the overhead required to run an emulated kernel is less important. Features like SR-IOV can safely "pass through" PCI bus devices to the virtualized guest, providing near-native device performance for the guest. In this configuration, multiple virtual machines could be running multiple kernels, and users can still define their own software stacks using the layared container approach. In this design, sysadmins would manage a set of virtual machines and a base container image that runs on any of them.

References

[1] lanl/vpic: Vector Particle-In-Cell (VPIC) Project. `https://github.com/lanl/vpic`, 2015.

[2] CANOPIE HPC Workshop - Containers and New Orchestration Paradigms for Isolated Environments in HPC. `https://www.canopie-hpc.org/`, 2017.

[3] Docker Containers for Reproducible Research. `https://www.software.ac.uk/c4rr`, 2017.

[4] Singularity and MPI Applications. `https://sylabs.io/guides/3.4/user-guide/mpi.html`, 2017.

[5] ResCuE-HPC 2018 Program. `https://rescue-hpc.org/program2018.html`, 2018 (accessed April 20, 2020).

[6] Pantheon Science — Reproducible Workflows for Extreme Scale Science. `https://pantheonscience.github.io/`, 2019.

[7] Podman. `podman.io`, 2019.

[8] VMWare - Official Site. `https://www.vmware.com/`, 2019.

[9] Reproducibility Initiative - SC19. `https://sc19.supercomputing.org/submit/reproducibility-initiative/`, 2019 (accessed April 20, 2020).

[10] Agelastos, A., Allan, B., Brandt, J., Cassella, P., Enos, J., Fullop, J., Gentile, A., Monk, S., Naksinehaboon, N., Ogden, J., Rajan, M., Showerman, M., Stevenson, J., Taerat, N., and Tucker, T. The Lightweight Distributed Metric Service: A Scalable Infrastructure for Continuous Monitoring of Large Scale Computing Systems and Applications. In *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Nov 2014), pp. 154–165.

[11] Ahrens, J., Jourdain, S., OLeary, P., Patchett, J., Rogers, D. H., and Petersen, M. An image-based approach to extreme scale in situ visualization and analysis. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2014), IEEE, pp. 424–434.

[12] Anderson, D. P. BOINC: A system for public-resource computing and storage. In *Fifth IEEE/ACM international workshop on grid computing* (2004), IEEE, pp. 4–10.

[13] Anderson, D. P., Cobb, J., Korpela, E., Lebofsky, M., and Werthimer, D. SETI@ home: an experiment in public-resource computing. *Communications of the ACM 45*, 11 (2002), 56–61.

[14] Arango, C., Dernat, R., and Sanabria, J. Performance evaluation of container-based virtualization for high performance computing environments. *arXiv preprint arXiv:1709.10140* (2017).

[15] Bahsi, E. M., Ceyhan, E., and Kosar, T. Conditional workflow management: A survey and analysis. *Scientific Programming 15*, 4 (2007), 283–297.

[16] Banga, G., Druschel, P., and Mogul, J. C. Resource containers: A new facility for resource management in server systems. In *OSDI* (1999), vol. 99, pp. 45–58.

[17] Beberg, A. L., Ensign, D. L., Jayachandran, G., Khaliq, S., and Pande, V. S. Folding@ home: Lessons from eight years of volunteer distributed computing. In *2009 IEEE International Symposium on Parallel & Distributed Processing* (2009), IEEE, pp. 1–8.

[18] Bernstein, D. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing 1*, 3 (2014), 81–84.

[19] Boehme, D., Gamblin, T., Beckingsale, D., Bremer, P.-T., Gimenez, A., LeGendre, M., Pearce, O., and Schulz, M. Caliper: performance introspection for HPC software stacks. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2016), IEEE, pp. 550–560.

[20] Bowers, K. J., Albright, B., Yin, L., Bergen, B., and Kwan, T. Ultrahigh performance three-dimensional electromagnetic relativistic kinetic plasma simulation. *Physics of Plasmas 15*, 5 (2008), 055703.

[21] Bowers, K. J., Albright, B. J., Bergen, B., Yin, L., Barker, K. J., and Kerbyson, D. J. 0.374 Pflop/s trillion-particle kinetic modeling of laser plasma interaction on Roadrunner. In *SC'08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing* (2008), IEEE, pp. 1–11.

[22] Collberg, C., and Proebsting, T. A. Repeatability in computer systems research. *Communications of the ACM 59*, 3 (2016), 62–69.

[23] Deelman, E., Blythe, J., Gil, Y., Kesselman, C., Mehta, G., Patil, S., Su, M.-H., Vahi, K., and Livny, M. Pegasus: Mapping scientific workflows onto the grid. In *European Across Grids Conference* (2004), Springer, pp. 11–20.

[24] Fursin, G., Childers, B., Heroux, M., and Taufer, M. Extended Submission Guide: artifact appendix. `https://ctuning.org/ae/submission_extra.html`, 2015.

[25] Fursin, G., Lokhmotov, A., and Plowman, E. Collective knowledge: towards r&d sustainability. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)* (2016), IEEE, pp. 864–869.

[26] Gabriel, E., Fagg, G. E., Bosilca, G., Angskun, T., Dongarra, J. J., Squyres, J. M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., et al. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting* (2004), Springer, pp. 97–104.

[27] Gamblin, T., LeGendre, M., Collette, M. R., Lee, G. L., Moody, A., de Supinski, B. R., and Futral, S. The Spack package manager: bringing order to HPC software chaos. In *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2015), IEEE, pp. 1–12.

[28] Gerhardt, L., Bhimji, W., Fasel, M., Porter, J., Mustafa, M., Jacobsen, D., Tsulaia, V., and Canon, S. Shifter: Containers for hpc. In *J. Phys. Conf. Ser.* (2017), vol. 898, p. 082021.

[29] Habib, I. Virtualization with kvm. *Linux Journal 2008*, 166 (2008), 8.

[30] Izadpanah, R., Allan, B. A., Dechev, D., and Brandt, J. Production Application Performance Data Streaming for System Monitoring. *ACM Trans. Model. Perform. Eval. Comput. Syst. 4*, 2 (Apr. 2019), 8:1–8:25.

[31] Jimenez, I., and Maltzahn, C. Spotting Black Swans With Ease: The Case for a Practical Reproducibility Platform, 2018.

[32] Jimenez, I., Sevilla, M., Watkins, N., Maltzahn, C., Lofstead, J., Mohror, K., Arpaci-Dusseau, A., and Arpaci-Dusseau, R. The popper convention: Making reproducible systems evaluation practical. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)* (2017), IEEE, pp. 1561–1570.

[33] Kurtzer, G. M., Sochat, V., and Bauer, M. W. Singularity: Scientific containers for mobility of compute. *PloS one 12*, 5 (2017).

[34] Mercier, M., Faure, A., and Richard, O. Considering the Development Workflow to Achieve Reproducibility with Variation.

[35] Merkel, D. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux J. 2014*, 239 (Mar. 2014).

[36] Miller, B. P., Callaghan, M. D., Cargille, J. M., Hollingsworth, J. K., Irvin, R. B., Karavanic, K. L., Kunchithapadam, K., and Newhall, T. The Paradyn parallel performance measurement tool. *Computer 28*, 11 (1995), 37–46.

[37] Miller, K., and Pegah, M. Virtualization: virtually at the desktop. In *Proceedings of the 35th annual ACM SIGUCCS fall conference* (2007), pp. 255–260.

[38] Mitchell, R., Pottier, L., Jacobs, S., da Silva, R. F., Rynge, M., Vahi, K., and Deelman, E. Exploration of Workflow Management Systems Emerging Features from Users Perspectives. In *2019 IEEE International Conference on Big Data (Big Data)* (2019), IEEE, pp. 4537–4544.

[39] Mondragon, O. H., Bridges, P. G., Levy, S., Ferreira, K. B., and Widener, P. Understanding performance interference in next-generation HPC systems. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2016), IEEE, pp. 384–395.

[40] Oliveira, L., Wilkinson, D., Mosse, D., and Childers, B. R. Supporting thorough artifact evaluation with occam.

[41] Priedhorsky, R., and Randles, T. Charliecloud: Unprivileged containers for user-defined software stacks in hpc. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2017), pp. 1–10.

[42] Rostanski, M., Grochla, K., and Seman, A. Evaluation of highly available and fault-tolerant middleware clustered architectures using RabbitMQ. In *2014 federated conference on computer science and information systems* (2014), IEEE, pp. 879–884.

[43] Shende, S. S., and Malony, A. D. The TAU parallel performance system. *The International Journal of High Performance Computing Applications 20*, 2 (2006), 287–311.

[44] Tallent, N., Mellor-Crummey, J., Adhianto, L., Fagan, M., and Krentel, M. HPCToolkit: performance tools for scientific computing. In *Journal of Physics: Conference Series* (2008), vol. 125, IOP Publishing, p. 012088.

[45] Torrez, A., Randles, T., and Priedhorsky, R. HPC container runtimes have minimal or no performance impact. In *2019 IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)* (2019), IEEE, pp. 37–42.

[46] Tschueter, R., Herold, C., Williams, W., Knespel, M., and Weber, M. A Top-Down Performance Analysis Methodology for Workflows: Tracking Performance Issues from Overview to Individual Operations. In *2019 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS)* (2019), IEEE, pp. 21–30.

[47] Walker, D. W., and Dongarra, J. J. MPI: a standard message passing interface. *Supercomputer 12* (1996), 56–68.

[48] Younge, A. J., Pedretti, K., Grant, R. E., and Brightwell, R. A tale of two systems: Using containers to deploy HPC applications on supercomputers and clouds. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)* (2017), IEEE, pp. 74–81.

[49] Zaharia, M., Chen, A., Davidson, A., Ghodsi, A., Hong, S. A., Konwinski, A., Murching, S., Nykodym, T., Ogilvie, P., Parkhe, M., et al. Accelerating the Machine Learning Lifecycle with MLflow. *IEEE Data Eng. Bull. 41*, 4 (2018), 39–45.