Towards Modern, Accessible and Dynamic HPC Using Container-based Virtual Clusters

Joshua Higgins

A thesis submitted in partial fulfilment of the

requirements for the degree of

Doctor of Philosophy

High Performance Computing Research Group

University of Huddersfield

United Kingdom

September 2018

Copyright

i The author of this thesis (including any appendices and/or schedules to this thesis) owns any copyright in it (the "Copyright") and s/he has given The University of Huddersfield the right to use such copyright for any administrative, promotional, educational and/or teaching purposes.

ii Copies of this thesis, either in full or in extracts, may be made only in accordance with the regulations of the University Library. Details of these regulations may be obtained from the Librarian. This page must form part of any such copies made.

iii The ownership of any patents, designs, trademarks and any and all other intellectual property rights except for the Copyright (the "Intellectual Property Rights") and any reproductions of copyright works, for example graphs and tables ("Reproductions"), which may be described in this thesis, may not be owned by the author and may be owned by third parties. Such Intellectual Property Rights and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property Rights and/or Reproductions.

For Jennifer

In memory of Tommy and Ozzy

Acknowledgements

Dr Violeta Holmes, for your unending enthusiasm
and guidance in shaping this work

Shuo and Matt, without whose all night office hours
I wouldn't have survived

and my family and friends too numerous to mention,
who supported me over the past 4 years.

Abstract

In this thesis, a novel Virtual Container Cluster (VCC) framework is presented. Despite the growing popularity of container virtualisation in order to increase the flexibility of the software stack, run time environment virtualisation still poses significant portability challenges; by depending on the underlying cluster execution paradigm, a niche class of *HPC only* containers has emerged. This trend is detrimental to reusability, reproducibility, and encouraging new communities to HPC.

Traditional virtualisation techniques have a rich history within HPC, and have been demonstrated to offer much more than software flexibility. A Virtual Machine by nature requires an OS and full stack environment akin to a physical machine, and this allows it to be instantiated regardless of the underlying machine and what services it provides. This capability is essential in order to implement job forwarding and spanning - where the burden of an entire job can be transferred or shared between heterogeneous cluster systems - with a high level of confidence that the environments will be compatible. In turn, this brings improvements to global resource performance, reducing the job turnaround time and increasing cluster utilization.

The VCC is an innovative solution that combines the full stack and container virtualisation approaches. Therefore, it offers both the flexibility of containers with the improved portability, performance and scalability of the full stack approach. In order to maintain the same accessibility and lower barrier of entry as the run time environment approach, the design incorporates an autonomous configuration and contextualisation mechanism, along with a Software Defined Networking technology, to ensure the full stack container does not place an additional burden on the user. The usefulness and performance is validated through benchmarking and two case studies: virtual clusters in the classroom and inter-institutional spanning.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Fast computing is essential to modern science. It has become a fundamental tool underpinning research and innovation in an ever increasing array of domains, including the modelling of physical phenomena, fluid dynamics, molecular interactions, astronomy, genomics, game design, social media and even music technology. The dependence on research computing capabilities will continue to grow, especially as new methods are developed that require the processing of massive amounts of data. However, whilst the number of fields that rely on research computing expands, the infrastructure and technologies traditionally used to deliver it are difficult to adapt to a diverse range of users, workflows, and deployment topologies.

High Performance Computing

High Performance Computing (HPC) is typically synonymous with cluster and grid computing. However, it can be more precisely defined by 3 main characteristics, outlined in Table 1.1. Firstly, a key property of an HPC system is agressive parallelisation, allowing tasks operating on the same or different components of data to be

Parallelism	Maximise parallelism	Speedup is gained from parallelisation of tasks
	MIMD, SIMD	Clusters are composed of many nodes that are parallel machines on their own
Latency	Minimise latency	Low latency communications is required for inter- and intra-node processing elements
	Inter-node	Processor cache and memory: 1.8ns to 70ns
	Intra-node	InfiniBand and Ethernet: 1us to 25us
Scalability	Strong scaling	Constant problem size, time reduced by adding processors
	Weak scaling	Problem size is proportional to number of processors

Table 1.1: Characteristics of an HPC system

distributed across multiple processors and multiple computers. Classical architectures for parallelisation are described by Flynn's Taxonomy, although modern HPC systems rarely utilise a homogeneous form of parallelism (Sterling, Anderson, & Brodowicz, 2017). For example, distributed memory HPC systems may be classified as *Multiple Instruction Multiple Data* (MIMD), but composed of individual computers that can exploit processors with *Single Instruction Multiple Data* (SIMD) extensions, such as Advanced Vector Extensions (AVX). Secondly, these systems must provide a low latency interconnect between processing elements, local and distributed memory, in order to maximise the efficiency and speedup of parallelisation. Finally, an HPC system provides scalability - this can be exhibited as strong scaling, where the performance is improved for a constant size problem when adding more processors, or weak scaling, where the performance per processor is constant but the problem size is proportional to the number of processors, thus allowing larger problems to be solved.

An HPC system supports a range of activities, including those from the user's perspective - job submission, data management, visualisation - and those from a system

and resource management perspective - scheduling, parallel execution, and usage accounting. Underpinning these activities are layers of interconnection networks. The concurrency of processing elements enabled by this interconnection is the distinction between an HPC system and *many computers with some degree of proximity.* It enables the systems to work cooperatively in order to solve problems that typically require significantly higher processing power, memory or faster turnaround time than what can be provided by one system alone. In practice, an HPC system is composed of a hierarchy of networks that may be required to pass messages between processing elements on a scale ranging from nanometers to several hundred miles. The design of these networks is the distinguishing factor between different types of HPC systems.

At the compute node level, there are already several networks that facilitate communication between CPUs, such as QuickPath Interconnect (QPI) or HyperTransport, and between Graphic Processing Units (GPUs), accelerators and other devices, such as PCI Express and NVLink. These networks allow the implementation of shared memory access, cache coherency, and parallelisation of tasks within the boundary of a single processing element.

In a cluster, the parallelisation of tasks is performed across many processing elements. In this computing paradigm, processes belonging to the same application are distributed across many machines. This is in contrast to a web server farm, for example, where even though the task of serving clients can be parallelised by running many web servers, each is independent and there is little communication between the web servers themselves. Unlike these commodity networks and services, the interconnect between machines in a cluster must be optimised for exchanging messages and memory segments with high speed and low latency. Therefore, HPC clusters typically utilize interconnects such as InfiniBand, Omni-Path or High Speed

Ethernet which provide data rates in excess of 100Gb/s and typical latencies of less than 1μs and up to 25μs.

In a grid, several clusters are interconnected over a larger geographical distance. The network connections between clusters in a grid typically facilitate data movement, identity management and high-level scheduling in order to allow a group of clusters to be horizontally integrated and operated in a coherent fashion. These connections can be used to implement capabilities such as meta-scheduling, where jobs are forwarded between resources in a grid in order to improve the overall efficiency and throughput (Sotiriadis, Bessis, Xhafa, & Antonopoulos, 2012). For example, a job can be forwarded from a busy cluster to an idle cluster, where it would otherwise have had to wait in a queue.

The landscape of HPC systems in the UK is composed of 3 tiers, representing national, regional and local computing assets, as shown in Figure 1.1 (*EPSRC strategy for the developing landscape of Tier-2 HPC in the UK*, n.d.). Tier-1 provides the highest capability machines: ARCHER and DiRAC. Both systems can be accessed through research council funded consortia, grants and regular specific calls. ARCHER is a Cray XC30 supercomputer providing 118,080 compute cores, mainly supporting materials, climate, engineering and biosciences (*ARCHER*, n.d.). DiRAC provides compute resources for particle and nuclear physics, astrophysics and cosmology through 4 HPC systems distributed around the UK, engineered to solve distinct classes of research problems. They offer between 4116 and 20,256 cores, including nodes with Random Access Memory up to 6TB (*DiRAC*, n.d.).

In 2016, the Engineering and Physical Sciences Research Council (EPSRC) provided £20m of funding in order to create 6 Tier-2 HPC centres. The design of these systems is driven by diverse architectures, providing opportunities for ARM, Power and GPU computation in addition to traditional HPC workloads, creating a broad scope

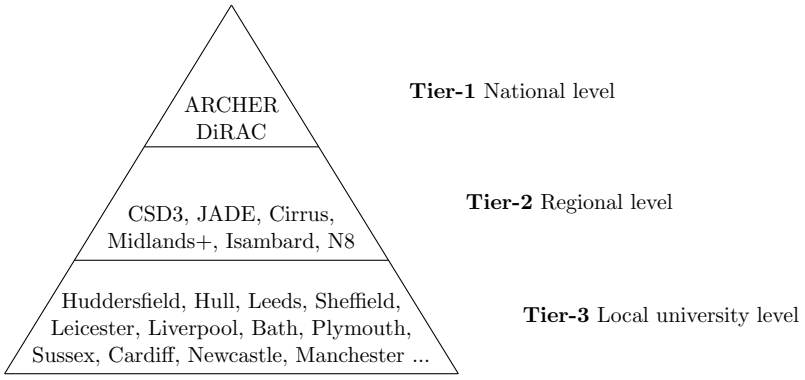


Figure 1.1: Landscape of HPC resources in the UK

in terms of user communities who can take advantage of access to these systems. Furthermore, each centre provides a mechanism for training and easy access in order to expedite the process for researchers seeking access to regional and national computing facilities.

Tier-3 is composed of local institution HPC systems that support a limited user base or highly specialised communities, but in general do not offer the same capability or capacity as regional or national centres. Users are encouraged to scale vertically onto higher tier systems when the demands of their projects outgrows the provision, rather than horizontal integration with other Tier-3 systems. The QueensGate Grid at the University of Huddersfield is an example of a Tier-3 system. It provides a campus grid of several clusters, supporting around 50 users in local research communities from engineering, chemistry and biology, in addition to students from taught undergraduate modules in parallel computing (Holmes & Kureshi, 2015).

There is no 4th tier in the EPSRC assessment of HPC infrastructure. However, loosley coupled workstations - often not running cluster operating systems but still running workloads in parallel - and midnight render farms, are two examples of invisible HPC-like computing capability that provide essential resources for the re-

spective communities. It is tempting to consider them as part of the Tier-3 provision, but these systems are distinguished in that they are almost always transient and composed from existing, general IT infrastructure.

Regardless of the scale of interconnection between systems, it is clear that HPC facilitates more than just aggregation of computing capacity and capability; projects such as DiRAC foster a community of knowledge exchange and collaboration between cross-discipline researchers, in addition to providing a pool of resources and expertise that support the computational requirements of the science being conducted. This community provides the ecosystem of scientific inquiry, outreach and funding that is necessary to generate and sustain the research that HPC is ultimately designed to serve.

Software Environment Challenges

In the same way that the interconnect brings disparate computing elements into a coherent domain, the software running on the HPC system must also do the same. In a cluster, the homogeneous software environment presents both the user and application with a single, unified view of the computing elements, often referred to as a Single System Image (SSI) (Buyya, Cortes, & Jin, 2001). For example, every compute node in a cluster typically runs the same Operating System (OS), the same kernel version, and the software environment that composes the userland - such as libraries, shells and compilers - will also be the same. Therefore, the application being executed can expect to find its dependencies, with consistent versioning and location, on every machine.

This poses significant challenges when porting or reusing applications on different HPC systems. Firstly, differences in the environments can lead to code which is compiled and verified on one cluster, but cannot execute as intended on another without

modification, even if they share the same processor architecture. Depending on the incompatibility, such as conflicting or missing packages, libraries and toolchains, a resolution is generally possible but often non-trivial. Furthermore, when sharing a computational experiment in order to allow others to reproduce it, the target system must logically be the same kind of cluster - providing equivalent execution and parallelisation middleware - regardless of the specific versions of individual components. If the user does not have access to a system with the correct capabilities, and does not posses the knowledge or resources in order to configure one, it may be near impossible to overcome this challenge.

Secondly, in a grid there is no guarantee that the type of homogeneity in one cluster is shared by all the others. When integrating resources at this level, common interfaces for scheduling, data transfer, authentication and authorization are used to allow a heterogeneous pool of clusters to retain some properties of an SSI. However, it does not solve the application portability problem between individual cluster members, instead, providing methods to determine which clusters are elegible to run the job based on discovered and advertised capabilities (Turilli, Santcroos, & Jha, 2018). Therefore, the portability problem limits the ability to scale and transfer jobs between these systems if they do not meet the dependencies of the application, or are unwilling to adapt.

Finally, there is a challenge in terms of training new users to become familiar with cluster software environments. In addition to understanding the capabilities, limitations and requirements of compiling software for the SSI cluster, users must also gain intimate understanding of resource management and process launching in order to effectively utilise the cluster, grid, or integrated resources at scale. Local systems can be used to an extent for this purpose, but are not likely to be representative of real world production systems. The process of experimentation and testing on a

production system must be carried out in such a way which does not put it at risk for other users, and is efficient in terms of wasting metered time on the resource.

Virtualisation as a Potential Solution

Virtualisation techniques have frequently been evaluated for HPC in order to improve the flexibility of the software environment. Virtual Machines (VM) offer a solution to this problem by allowing distinct, virtual cluster environments to be provisioned as a subset of an existing resource. In this full stack approach, the VM is analogous to a physical cluster node - containing a full OS, unique network addressing and a isolated resources from the host machine. Within a pool of integrated resources, such as a grid, a VM provides a high level of confidence that the job will be executed successfully. This compatibility between systems is essential in order to realise the full potential to improve global resource management, such as through meta-scheduling techniques, by allowing jobs to be scheduled on clusters indiscriminate of the underlying system environment. However, VMs can be cumbersome, requiring system administration skills to create and deploy, which are not immediately accessible to the average user. In addition, virtualised execution is typically accompanied by a performance penalty when compared to the native performance, due to the layers of translation introduced between the VM and the host.

Containers present a radical approach where only the run time environment of an application needs to be virtualised, rather than the full software stack. Therefore, the process of creating a container is more accessible to the average user, and it can be invoked on a resource with minimal disruption to the workflow, within the same context and security principles as other non-virtualised applications. In this way, it has gained adoption within HPC communities where traditional virtualisation techniques have failed to gain traction, in order to improve the flexibility of the

software environment at the top-most layer of the software stack.

However, the implementation of existing container solutions within the HPC environment at the University of Huddersfield established that the expectation of improved portability and flexibility is not necessarily easy to achieve. In practice, users could not transfer a container created on the campus grid to another cluster or cloud provider and execute it without first ensuring that the other system provided the necessary communication and job launching middleware. In the case of a cloud resource which provides traditional Virtual Machines as a service, this presented a high barrier of entry for the average user - essentially requiring them to configure a cluster from scratch in order to execute the container. This experience provided the motivation for the initial decisions made in forming the direction of this research and the design of a new approach.

Considering a Full Stack Container Approach

The run time environment container approach solves many limitations with virtualisation in the HPC context - it has good accessibility, performance and is convienient to create and distribute images. In the context of a single cluster, it can be used to allow the user to arbitrarily customise the environment on a per-application basis. However, there is no line that determines what constitutes the run time environment, what packages should be included, and what should not. This inconsistency means that a container may still not be portable to other systems in the same way that traditional VMs are, as it inherently relies on the lower levels of the software stack to be provided by the host system. For example, a containerized MPI application must be executed on a cluster that is already configured with appropriate process launching middleware and interfaces to support MPI execution.

This thesis considers an improvement to existing virtualisation solutions in HPC,

through the implementation of full stack virtual cluster containers, rather than just encapsulating the run time layer of the software environment. The novel Virtual Container Cluster (VCC) framework is proposed in order to address the portability and usability limitations identified in the existing full stack implementations, which make them unsuitable for general HPC deployment and pose a high barrier to entry for the average user. The framework provides deployment, configuration and interconnection of the full stack virtual environment, implemented as functions of the container itself - rather than depending on external services or brokers to provide the required capabilities. A new, self-contained discovery model is introduced, which provides dynamic configuration and contextualisation for the virtual cluster, to ensure that the user is not required to perform deployment-specific adaptations to the middleware and management layers of the stack. By coupling the application and run time environment with the required management and middleware functions, a full stack container has the potential to offer a solution with better portability, more analogous to a traditional VM, but inheriting the accessibility and performance of the container virtualisation technique.

An innovation in this new model is the application of Software Defined Networking (SDN) as the virtual cluster interconnect, to afford the same flexibility to the network environment that virtualisation grants to the software environment. Together, these capabilities have the potential to improve the portability of a container environment, regardless of the underlying system and what interfaces it provides, resulting in more opportunities to exploit resource sharing and meta-scheduling techniques - such as job forwarding and spanning - in order to improve overall workload performance.

Furthermore, the full stack container virtualisation approach enabled by the VCC allows the deployment of virtual cluster topologies that were not possible before with

traditional virtualisation techniques. Firstly, due to the lower overhead of containers, large virtual environments can be staged on a small number of machines. This offers a realistic and accessible environment for mirroring the setup of a production system for the purpose of teaching and training, without putting the real system at risk. Secondly, the innovative SDN approach allows a single logical virtual cluster, encapsulated within containers, to be deployed spanning across many geographically separated resources. The promotes horizontal integration of resources, regardless of the scale or tier of the resource, facilitating meta-scheduling like capabilities that are transparent to the middleware and run time environment. This approach has the potential to bring the community and technical capability of grid-like collaboration to user groups which would otherwise not be able to establish it with ease, without introducing a significant administrative burden, and supporting deployment on an ad-hoc or persistent basis.

1.1 Aim and Objectives

The aim of this thesis is to develop a scalable, container-based full stack cluster virtualisation solution, which will faciliate improvements in usability, administration, resource management and software environment flexibility of HPC resources using modern virtualisation techniques.

In order to accomplish this aim, the following objectives are identified:

1. Design and develop a novel, unified virtual cluster model which satisfies the requirements of a general purpose solution synthesised from the literature.

2. Devise a novel mechanism in order to facilitate dynamic reconfiguration of the cluster environment. The configuration mechanism must resolve the portability limitations identified in previous work that rely on external infrastructure.

3. Evaluate the performance of the virtual cluster model demonstrating improvements to resource utilisation, job throughput and turnaround time, through deployment on the campus grid at the University of Huddersfield.

4. Demonstrate the scalability and portability of the solution through deployment of the virtual cluster model, facilitating horizontal integration between Tier-3 HPC resources, and identifying the feasibility to achieve the same improvements to global resource performance as in the campus grid.

5. Evaluate the usability through deployment as a tool for teaching and training in a classroom setting, quantifying any difference in accessibility and barriers to entry by comparison with the OSCAR middleware.

6. Assess whether the opportunities of virtualisation are applicable to scenarios and novel topologies that cannot be created with existing solutions, including geographically distributed, multi-fabric spanned, and nested virtual clusters.

1.2 Methodology

It is essential that the design of the full stack container virtualisation solution is cognisant of the existing work. Based on the extensive literature review into related and existing work, the need for a new container-based, full stack virtualisation solution for HPC is identified. The definition of a taxonomy and analysis of the features provided by existing implementations will be used as a framework to ensure that the designed solution addresses the gap in knowledge and is relevant to the aim of the research.

A hypothesis can be made that the implementation of full stack environment within a container, as opposed to a VM, has a predictable outcome; firstly, that the opportunities to improve portability and global resource performance offered by the

full stack approach is transferable to the container virtualisation technique, and secondly, that containers offer an additional gain in accessibility and a lower performance overhead. However, the VCC introduces an innovative integration of cluster software environment and interconnect virtualisation, which is not adequately considered by the literature. Therefore, a system to evaluate the proposed solution will also need to be devised, and the design of this method is of critical importance.

Software engineering is a field closely tied with Information Systems and Mathematics, resulting in machines or theories that can be evaluated with measurement or proofs. However, the software aspect of engineering typically requires consideration of human interaction, often borrowing research methods from psychology and sociology (Hanenberg, 2010). Case study is an empirical method that can serve many research purposes, such as exploratory and explanatory, conducted within a real world context (Runeson & Höst, 2009). Case studies may also combine complementary methods of research, especially within Information Systems and Computer Science, such as survey methods (Gable, 1994; Runeson & Höst, 2009). Therefore, a mixed method approach has been chosen in order to comprehensively evaluate the solution and its proposition.

Firstly, a controlled experiment will be conducted in order to benchmark the performance of the virtual cluster model in a variety of contexts, inter-cluster bandwidth and latency configurations. The results will be used to determine the performance characteristics and feasibility of the model before observing the deployment on a real system.

Secondly, representative case studies will be utilised in order to empirically evaluate the performance of the model within different deployment scenarios, representing scale of deployments from campus grid to geographically distributed Tier-3 resources.

Finally, the teaching case study will evaluate the usability of the VCC through deployment in a classroom teaching environment, using survey and skills audit techniques, throughout the course of the Parallel Computer Architectures modules at the University of Huddersfield.

A rationale and justification of the methodology and design of each case study is presented in the respective chapters, outlined in Table 1.2. The enhanced portability aspects of the VCC model encourages collaboration and reuse of virtual cluster environments, regardless of the underlying infrastructure that the user possesses in order to execute them. Thus, the methods and experiments used in this research are instilled with this philosophy. In order to ensure that they are reproducible by others, the code and experimental configurations will be released online[1] as open source under the MIT License (*The MIT license*, 2006).

1.3 Outline

The remainder of this thesis is generally ordered chronologically, in the order that the respective research activities were undertaken. Chapter 2 reviews the literature surrounding virtualisation in HPC, assessing the related work in performance evaluation and application in cluster environments. At the end of Chapter 2, the gap in knowledge and solution in order to address it is synthesized.

Chapter 3 describes the novel design of the Virtual Container Cluster (VCC) framework and the implementation of the services that provide the deployment, configuration and contextualisation capabilities.

Chapter 4 outlines the process of applying the VCC framework in order to construct a virtual cluster. The container built in this chapter is used as the foundation for

[1]Repository accessible at `https://github.com/hpchud`

Controlled experiment	**Execution performance** • HPL benchmark • Ethernet and Infiniband	Parallel performance on local cluster
	Latency scalability • HPL benchmark • 0.1 to 100ms latency	Parallel performance on spanned cluster
Case Study	**Campus grid** • Feasibility using historic data • Spanned job queue wall time	Resource management and administration
	Inter-institution grid • HPL benchmark • Non-academic network	Novel topologies Tier-3 integration
	Classroom • Usability survey • Workflow comparison	Usability

Table 1.2: Summary of method and associated activities

the evaluation carried out in the performance evaluation and case studies.

A performance evaluation of the VCC is carried out in Chapter 5, followed by the case studies for geographically distributed spanning, and teaching in the classroom in Chapters 6 and 7 respectively.

Finally, the conclusions, insight and impact gained during the course of this work is considered in Chapter 8, in addition to directions for future work in order to extend and improve this area of research.

Chapter 2

Related Work

Recent developments in the delivery and management of computing environments, including the rapid adoption of cloud based systems, suggest that the landscape of High Performance Computing is changing in favour of virtualisation technologies; there is a growing demand for highly flexible computational resources that are reconfigurable to meet the needs of different communities.

However, virtualisation techniques have continuously been evaluated for the purpose of research and execution of scientific workloads since their introduction in the CP/CMS operating system for the IBM System/360 (Creasy, 1981).

Whilst there is a perceived performance cost of modern Hypervisors and Virtual Machine Monitors managing aspects of the code execution, the case for adopting them within HPC is frequently restated (Youseff, Wolski, Gorda, & Krintz, 2006; Emeneker & Stanzione, 2006; Birkenheuer et al., 2012).

In this chapter, work addressing the contemporary performance analysis of virtualisation in HPC is considered. In Section 2.2, the application of these techniques in order to virtualise cluster software stacks is examined, followed by container vir-

tualisation solutions within HPC in Section 2.3. Finally, a critical analysis of the literature in context is presented in Section 2.4. The limitations identified in the existing work are outlined, and a solution is synthesized in order to address the gap in knowledge.

2.1 Virtualisation Techniques: Past and Present

Pioneered by IBM in the CP/CMS, and later VM-family of operating systems, classical (full) virtualisation uses a Virtual Machine Monitor (VMM) or Hypervisor in order to provide emulated hardware on which a separate guest OS executes (Creasy, 1981). Typically, the emulated hardware will provide interfaces for storage, networking and video devices. The advantage of this approach is that, as far as the guest OS is concerned, the interfaces provided by the VMM are indistinguishable from a non-virtualised system (Rodríguez-Haro et al., 2012). Therefore, the guest OS and software does not need to be modified in order to run under virtualisation. The main disadvantage of this approach is that, depending on the extent of device emulation required, the performance overhead can be significant. VMware, VirtualBox (Li, 2010) and KVM (Kivity, Kamay, Laor, Lublin, & Liguori, 2007) are solutions that implement classical virtualisation techniques on modern x86 platforms.

Paravirtualisation avoids machine emulation by requiring modifications to the guest OS. The guest OS must utilise *Hypercalls* provided by the VMM within its kernel and driver code (Rodríguez-Haro et al., 2012). Hypercalls are a special set of instructions which replace the privileged instructions in the host machines Instruction Set Architecture (ISA). This allows the VMM to partition the resources that are accessed via these privileged instructions between guests. Userspace applications do not need to be modified and these instructions are passed through directly. This

technique is considered a lightweight approach as no machine emulation takes place, appearing more suited for high performance execution due to the lower overhead. However, the main disadvantage of paravirtualisation is that it requires modifications to the guest OS. Xen (Barham et al., 2003) is a popular implementation of paravirtualisation and features heavily in the HPC context.

Rather than faithfully recreating hardware interfaces as in a Virtual Machine, OS-level virtualisation techniques focus on virtualisation at the application level, where the OS kernel is responsible for providing multiple isolated user-space instances (Fink, 2014). This evolution of virtualisation architectures, from providing optimised paths through the hypervisor to the hardware, to removing the hypervisor entirely, is shown in Figure 2.1. The advantage of OS-level virtualisation is that the overhead is very low, typically no greater than the overhead of launching regular processes within the OS. However, using this approach the guest applications must share the same kernel, and thus, the same hardware type and OS family. For example, it is not possible to virtualise a Windows guest within a Linux host using OS-level virtualisation techniques alone. Implementations of OS-level virtualisation include OpenVZ (Kolyshkin, 2006), Linux-VServer (Soltesz, Pötzl, Fiuczynski, Bavier, & Peterson, 2007) and Docker (Merkel, 2014; Fink, 2014).

2.1.1 Xen era

The performance of a Xen-based HPC cluster has been evaluated in terms of two critical subsystems: computation and communication (Youseff et al., 2006). Two well established benchmarks are used: High Performance Linpack (HPL) (Dongarra, Luszczek, & Petitet, 2003) and the `Com` benchmark from the LLNL Presta Stress Benchmark (Carnes, 2002), respectively. It is demonstrated that the CPU execution overhead is statistically insignificant compared to execution within the host system;

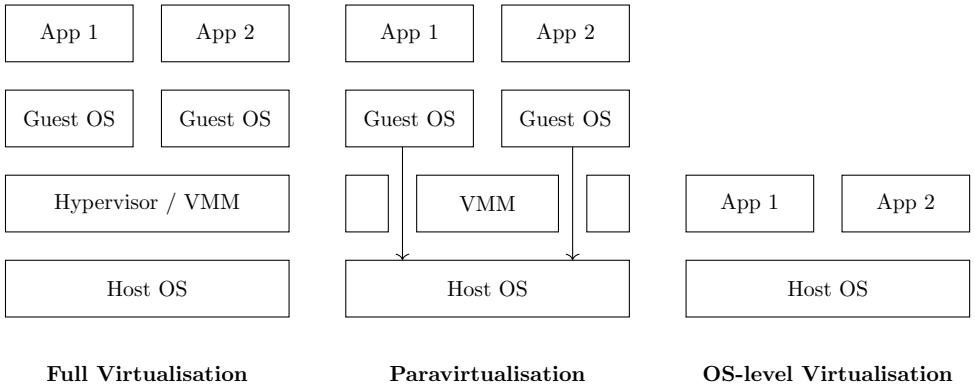


Figure 2.1: Comparison of Virtualisation Techniques

the overall performance of Linpack running in the Xen environment is approximately 2% lower than execution on the native system. In terms of communication, the results are varied depending on the communication patterns. The `Com` benchmark records bandwidth and latency of communication between pairs of processors in a ping-pong fashion, over a range of message sizes. Overall, the Xen communication bandwidth is lower due to the guest OS requiring an exchange of packets with the host OS for each send and receive. This is further degraded by small packet sizes since a large number of such exchanges take place. For large message sizes, the bandwidth is comparable to the native system. In terms of latency, the results are counterintuitive. Within a single node, the communication latency within Xen outperforms the native execution due to page-flipping optimisations. As shown in Figure 2.2, the VM can exchange an empty page with the VMM for one which contains the packet to be received, rather than copying the data into the VM's domain. As the number of communicating processes increases beyond a single node, this optimisation becomes ineffective and the latency becomes equivalent to the native system.

From this study, Youseff et al. conclude that paravirtualisation poses no significant overhead compared to traditional OS configurations used in HPC, except where an

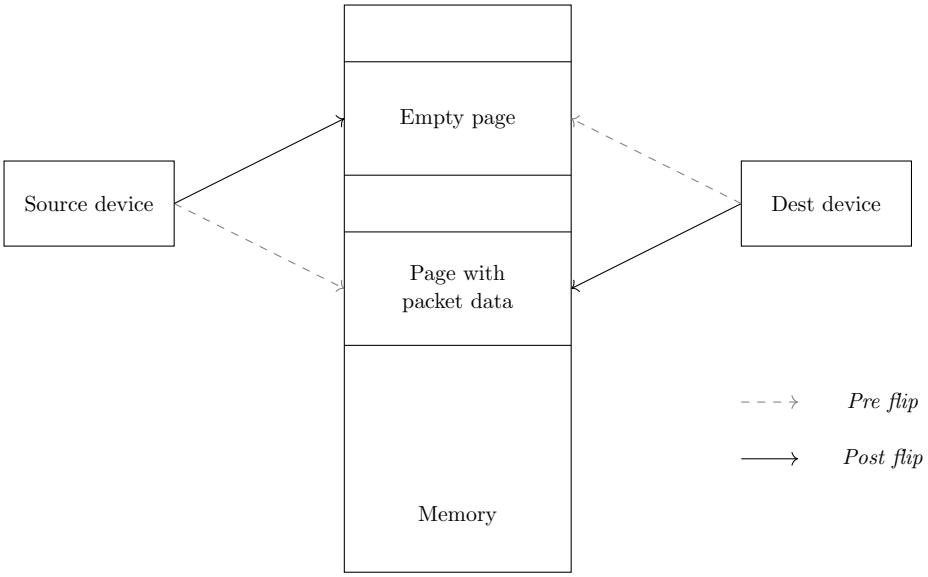


Figure 2.2: Receive Page Flipping in Xen

application exercises a specific subsystem or a combined subsystems across many cluster resources. It is suggested that virtualisation should be reconsidered based on the empirical evaluation presented. It also cites several advantages, such as more flexible system maintenance and customisation, as reasons to consider a virtualised approach.

In Emeneker and Stanzione (2006), the performance penalty of Xen is quantified using the HPC Challenge benchmark (Luszczek et al., 2006), which incorporates HPL along with other benchmarks in order to test I/O and memory bandwidth subsystems. An important distinction is made between single node execution, where the CPU execution overhead is as low as 1%, and multi-node parallel execution, where the overhead is up to 15% slower as the processor count increases on the HPL benchmark. The results demonstrate an average latency increase under Xen by 1.6x and a bandwidth overhead of approximately 40%. As expected, benchmarks that are CPU and memory bound vary less than 5% from, and in some cases outperforming

the native execution.

This work acknowledges that a performance overhead is inevitable, but that the potential advantages in terms of administration and resource management gained will outweigh the cost if it is low enough. The network latency and bandwidth are identified as critical to running parallel applications, and it was expected to gain close to native networking performance from Xen, and thus, similar MPI parallel performance. However, from the results, it is clear that this is not always the case, and must be appreciated depending on the profile and communication patterns of the application being tested.

Despite this, the authors identify many advantages to virtualisation in the context of this experiment. For example, the ability to make changes to a host without expensive downtime allowed various kernels to be tested and changes to be propagated to the nodes in the cluster without rebooting. They also suggest that virtualisation can be used to solve challenges in terms of cluster administration, such as resolving conflicting package or library requirements, improving the reliability of the software environment and effectively partitioning resources.

Huang, Liu, Abali, and Panda (2006) recognise that the requirement of Xen to intervene on every send and receive from a guest OS is the key challenge to reduce the critical network virtualisation overhead for HPC applications. A framework is presented that combines a specialised OS image for an application with a VMM bypass driver, also proposed by the authors. The VMM bypass allows access to the networking device to be negotiated through the VMM, with subsequent communication operations being performed directly from the user process on the device. This removes Xen from the critical path of communication and allows the virtualised application to achieve native I/O performance. HPL is used to evaluate the application performance, along with the NAS Parallel Benchmarks (NPB) (Bailey et al., 1991).

For communication intensive benchmarks, the results demonstrate a reduction in the performance overhead from 17% to 4%. This work concludes that the costs of virtualisation can be reduced through optimisation of specific subsystems, such as communication, in order to provide desirable features such as ease of management, customisation and isolation with low performance degradation. However, this solution can only be implemented where a dedicated interconnect that already provides a native user-space programming interface, such as InfiniBand, is available.

Comparisons between Xen and classical virtualisation hypervisors are introduced by Walters, Chaudhary, Cha, Guercio Jr, and Gallo (2008) and Nussbaum, Anhalt, Mornard, and Gelas (2009). In these studies, Xen is compared with VMware and Kernel Virtual Machine (KVM) respectively. In terms of CPU execution performance, VMware exhibits performance up to 10% slower than the native execution on the NPB (Walters et al., 2008). The communication performance shows the greatest overheads, with a 2x increase in latency observed within Xen, and over 3x increase within VMware. The KVM performance overhead is equally significant, with a 4x increase in latency and 30% slower than the native execution on the HPL benchmark (Nussbaum et al., 2009). In general, these studies suggest that paravirtualisation is still the preferred technology for HPC applications due to the lower communication overhead than with classical virtualisation. In addition, any remaining overhead with paravirtualisation can be eliminated with VMM bypass techniques such as those proposed by Huang et al. and Liu, Huang, Abali, and Panda (2006).

2.1.2 Classical era

Whilst previously demonstrated to introduce the greatest overhead, the performance of classical virtualisation techniques are shown to surpass paravirtualisation in the context of HPC applications in a study by Younge et al. (2011). The aim of this

study is to evaluate whether virtualisation within HPC is viable and to select the best virtualisation technology for this task. The motivation differs from previous work, as the authors describe running HPC environments within a cloud deployment and the increasing prevalence of cloud computing within academic settings. However, the experimental setup for the study is within a local HPC resource, rather than in the cloud, thus a valid comparison can be drawn.

A comparison is made between KVM, VirtualBox and Xen hypervisors using the HPC Challenge benchmark, focusing on the HPL, Fast Fourier Transform (FFT) and ping-pong communication performance results. In addition, the SPEC (Dixit, 1991) benchmarks are evaluated in order to gain insight on shared memory multi-processor (SMP) performance. In terms of communication bandwidth, the benchmarking results show that Xen performs close to native speeds whilst KVM and VirtualBox incur significant overheads. However, the opposite is observed when considering communication latency: KVM and VirtualBox achieve near-native latency, whilst Xen incurs extremely high overheads. In terms of execution, each hypervisor is comparable when considering single node Star/Single FFT and SPEC performance. In addition, each hypervisor performs similarly on the HPL benchmark, albeit with a considerable overhead compared to native performance. However, the parallel execution performance of Xen in MPI FFT is considerably lower. This result is significant as, unlike a synthetic benchmark, the FFT represents a common real-world application usage. As demonstrated by Walters et al. (2008) and others, the latency has a significant impact on parallel execution. Therefore, in real-world usage, it makes sense that KVM and VirtualBox will outperform Xen as their networking capabilities become more optimised.

Overall, this study suggests that the KVM hypervisor is the best technology for the HPC community. Even though KVM does not exhibit the lowest overhead in

all respects, it is shown that the latency has a greater impact on parallel execution performance. The performance results from Xen and VirtualBox have an extremely high variance; In some cases, they achieve acceptable performance, but the unpredictable nature makes them unsuitable for building a system that provides lasting quality of service for general HPC deployments.

In a comparison between VMware, KVM and VirtualBox carried out by Luszczek et al. (2012), the authors provide insight in the general performance behaviours of virtualisation, rather than providing a recommendation as to which is best suited for high performance execution. The paper outlines challenges to measuring accurate performance data within a virtual environment. Firstly, a significant disparity is demonstrated between the measured wall clock and CPU time. Whilst this effect is observed even without virtualisation, the behaviour is more inconsistent within a virtual environment - over a range of problem sizes, there can be nearly an order of magnitude difference. Secondly, it is shown that state accumulation within the hypervisor or VMM can lead to inconsistent results when running the same code repeatedly, such as when benchmarking a system until satisfactory results are obtained. To illustrate this, the authors ran the HPL benchmark twice using the same configuration of problem sizes, with the first run being in ascending order and the second run being in descending order. In the small problem sizes, a difference in performance of up to 50% was observed in both VirtualBox and VMware. Without virtualisation, there is no noticeable difference in the results regardless of the order in which they were executed.

However, they outline a limitation in that previous benchmarking methods do not adequately consider how to mitigate the issues identified when measuring the performance data. Therefore, Luszczek et al. present their own evaluation of HPL and MPIRandomAccess performance. They demonstrate that the HPL performance is

good with relatively low variability across VMware, VirtualBox and KVM, with overheads of approximately 20%, 10% and 5% respectively. However, on the MPI-RandomAccess benchmark, which produces significant demand on the memory subsystem, the virtualisation overhead can be as high as 70%. A high variability is observed across these measurements, indicating that accumulated state within the hypervisor could be negatively affecting the accuracy of results.

Birkenheuer et al. (2012) reiterate the case for implementing virtualisation within HPC. They identify the usefulness of the high level of abstraction that has been developed in order to provide multi-tenant cloud facilities over the Internet and relate this to the scientific context. Virtualisation would allow providers of HPC resources to focus on delivering infrastructure whilst allowing users to provide the software, rather than maintaining both the hardware and a global software stack. This work recognises the lack of successful implementations and provides an assessment of the opportunities and challenges, in a similar vein to Figueiredo, Dinda, and Fortes (2003). A framework for combining virtualisation and HPC is presented in order to address these challenges, such as the placement of VMs on physical resources, and the scheduling overhead of provisioning a VM in order to execute a job. In contrast to previous work, the proposed virtualisation framework is evaluated using a real application rather than tools designed for the purpose of benchmarking. This has the advantage of relating the results directly to the domain that is of interest. However, it does not illustrate the general performance of the model given the high variability observed by other work. The results demonstrate that the performance and scaling behaviour of the molecular dynamics application Gromacs (Hess, Kutzner, Van Der Spoel, & Lindahl, 2008) incurs a performance overhead of 15-30% in the VMware virtualised environment, depending on the interconnect that was used for communication. This overhead is attributed to the network communication, as the results demonstrate that the virtualised CPU performance is equivalent to the native CPU

performance when running on a single node. However, the authors suggest that this performance loss is acceptable due to the opportunities presented by virtualisation, such as VM checkpointing and migration, customisable software environments, and the ability to dynamically reconfigure properties of the cluster during runtime.

Work by Younge et al. (2011) has shown that whilst classical virtualisation hypervisors achieve low communication latency, the bandwidth overhead may be up to 50%, as in the case of KVM. This problem is addressed by Musleh, Pai, Walters, Younge, and Crago (2014) using a similar methodology that was employed to improve the communication performance in Xen: VMM bypass. The solution exploits the coming of age of *Single Root I/O Virtualisation* (SR-IOV) in InfiniBand Host Channel Adapters, where the hardware provides a set of virtual functions to allow resource sharing virtualisation to be performed on the device itself, effectively bypassing the VMM or hypervisor in order to achieve bare metal performance (Kutch, 2011). A performance evaluation is presented using the IBVerbs benchmark and the NAS Parallel Benchmarks. It shows that careful tuning of the network driver can realise a reduction in communication overhead by 15-30% when using SR-IOV.

2.1.3 Container era

An IBM Research Report on the performance evaluation of VMs and Docker containers demonstrates that "In general, Docker equals or exceeds KVM performance in every case [we] tested" (Felter, Ferreira, Rajamony, & Rubio, 2014). The results show that while both methods do not introduce a significant overhead for CPU and memory operations, communication and latency sensitive applications perform significantly better within the container environment. This is as expected, as the previous benchmarks in Section 2.1.2 detail the overhead of classical virtualisation in terms of communication bandwidth and latency. Without VMM bypass, opera-

tions within the VM typically must traverse many layers before being serviced by the hardware, adding overhead.

In particular, this study compares the HPL performance of the native, Docker and KVM execution. The performance within the container environment is equal to the native environment, whilst the KVM execution incurs a 17% overhead. It is suggested that the KVM hypervisor's abstraction of the hardware and processor topology does not allow runtime tuning and optimisation to take place, such as those provided by modern linear algebra libraries. This holds great significance for scientific applications that are more likely to be heavily optimised than regular workloads. With additional tuning effort, the performance overhead within KVM can be reduced from 17% to 2%. However, this study only considers the single-node performance of HPL and not the multi-node distributed execution that is usually employed within HPC systems. Therefore, whilst it suggests that containers could achieve comparable parallel performance to the native system, it does not demonstrate it. Finally, the authors point out that whilst KVM can provide good performance, configuration and tuning are difficult, presenting a high barrier to entry. On the other hand, a container provides the combination of good performance with ease of use and convenient workflows for deployment and configuration, even though in some use cases they may not be faster than classical virtualisation.

As the performance characteristics of container virtualisation are reproduced and accepted within the scientific community, the attention is turned to how viable it is to deploy container technologies within the often constrained software environments of HPC clusters in order to realise these benefits.

In order to integrate Docker support into the Cray XC system, Jacobsen and Canon (2015) found that the requirement to run an always-on administrative daemon on each node, to manage execution of the containers, poses several deployment issues.

For example, Docker requires exclusive use of local storage on a node in order to store images and the working state. Since all virtualised processes are running under the same kernel, it can also pollute the process table on the compute node and introduce a risk that processes may not be cleaned properly. In addition, an administrative daemon that offers user interaction may become an attack vector in order to undermine the security of the system.

However, they recognised that the Docker workflow, by allowing users to easily define, share and instantiate customised software environments, is essential in order to meet the needs of their scientific community while reducing the management burden, especially for new data-intensive research. Therefore, the *Shifter* tool is proposed in order to bridge the gap between Docker and the interfaces already provided by the Cray system that are used to customise the software environment. Shifter allows the user to create, upload and download images using the same methods that the standard Docker runtime uses. When instantiating a container, the filesystem of the Docker image is converted into a format suitable for deployment to the Cray system. This allows the container to utilise the parallel file system for storage, rather than a local disk. Benchmarking of this implementation shows that the Shifter approach performs close to the native performance of the best storage configuration (Jacobsen & Canon, 2015). However, this work does not demonstrate how well the system scales in terms of execution performance, as the benchmark focuses only on disk performance.

Singularity is a container runtime, similar to Docker and Shifter, that has emerged as a popular solution that targets HPC as the primary use case (G. Kurtzer, 2016). The motivation for the development of a new container runtime is to remove the requirement of running an administrative daemon on each compute node to manage container execution, due to problems such as those previously identified by Jacobsen

and Canon (2015). Despite the broad acceptance of container virtualisation in industry, this requirement is presented as the reasoning behind the lack of adoption by the scientific community.

In order to fulfil this aim, Singularity does not allow user contextual changes and must execute all processes as the calling user. Singularity emulates the ease of use of the Docker workflows by providing analogous, albeit incompatible, services for defining and sharing containers. The execution of Singularity containers always follow the same security principles as the underlying system. In addition, whilst Docker uses a Copy-On-Write (COW) filesystem in order to optimise the data transfer and reuse, Singularity uses a single *blob* based filesystem image. This has the advantage of being easily supported by parallel filesystems and existing tools such as `scp, gridftp, nfs` and etc, at the cost of potentially increasing the amount of data being transferred. This cost may be significant, for example, when an image must be transferred to every node within a cluster. Performance analysis of Singularity containers shows that, as is the case with Docker, the performance is comparable to native execution (Le & Paz, 2017). This is expected as both implementations use the same features of the Linux kernel in order to implement the OS-level virtualisation and neither require the use of emulated hardware or a VMM.

It is argued that this approach mitigates a fundamental security problem with Docker that allows unprivileged users on a system to easily exploit privilege escalation attacks. However, in a review of the security issues surrounding container virtualisation by Higgins, Holmes, and Venters (2016), this was found only to be true where Docker is installed without performing the recommended configuration for deployment on a multi-tenant system. For example, in the default configuration, the administrative root account within the container is equivalent to the root account outside of the container. Therefore, the owner of a container may gain complete con-

trol over any resources, such as shared filesystems, that are available to it. Whilst this may be acceptable when utilising containers as an administrative tool, clearly, it is not acceptable to allow normal users this capability. Configured appropriately, Docker can offer an equivalent security model as the other container solutions, which does not introduce a greater attack surface and restricts the most common privilege escalation vulnerabilities from within the container Higgins et al. (2016).

Each virtualisation technique reviewed in this section, and the results of the corresponding performance analysis in HPC, are illustrated in Table 2.1. It is clear that there is extreme variation: significantly different results are observed among the tests even though they use a common set of benchmarking tools.

2.2 Virtualised Clusters

2.2.1 In a Cluster Fabric

Dynamic Virtual Clustering (DVC) is a seminal work that considers the creation of virtualised clusters within a fabric of one or more directly interconnected clusters (Emeneker, Jackson, Butikofer, & Stanzione, 2006). The DVC utilises Xen paravirtualisation in order to deploy virtual cluster nodes which are made available to the underlying scheduler. This architecture is outlined in Figure 2.3. The functionality is used in order to enable 3 fundamental capabilities:

Job forwarding

Job forwarding allows the execution of an entire job to be transferred to a different cluster.

Reference	Hypervisor	Benchmark	Overhead
Figueiredo et al. (2003)	VMware	SPEC	4-10%
Emeneker and Stanzione (2006)	Xen	HPL	5-20%
Youseff et al. (2006)	Xen	HPL	2%
Huang et al. (2006)	Xen	HPL	1-11%
Tikotekar et al. (2009)	Xen	HPL	12-18%
Walters et al. (2008)	Xen	NAS	1-400%
	OpenVZ	NAS	1-25%
	VMware	NAS	1-750%
Nussbaum et al. (2009)	Xen	HPL	-5-20%
Luszczek et al. (2012)	VMware	HPL	15-20%
	VMware	RandomAccess	2-30%
	KVM	HPL	1-10%
	KVM	RandomAccess	50-70%
Younge et al. (2011)	Xen	HPL	4-60%
	Xen	PingPong	-25-500%
	VirtualBox	HPL	30%
	VirtualBox	PingPong	-25-1%
	KVM	HPL	30%
	KVM	PingPong	1%
Birkenheuer et al. (2012)	VMware	Gromacs	15-30%
Musleh et al. (2014)	KVM	NAS	20-250%
	KVM	IB-Verbs	7-15%
Felter et al. (2014)	KVM	HPL	2-17%
	Docker	HPL	0%
Higgins, Holmes, and Venters (2015)	KVM	HPL	7-50%
	Docker	HPL	1%

Table 2.1: Summary of works addressing virtualisation performance in HPC context

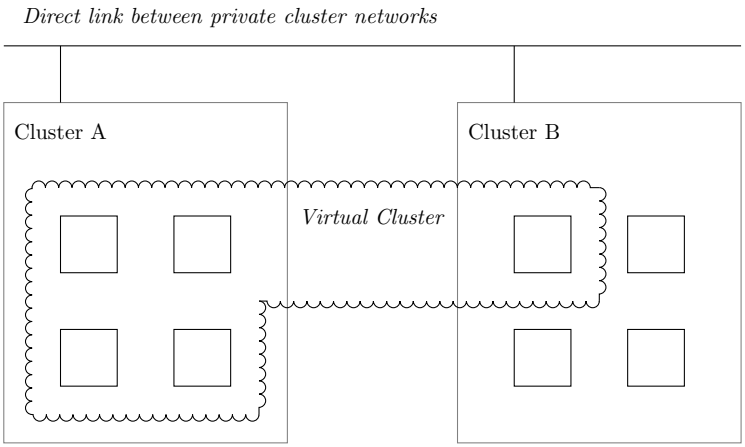


Figure 2.3: Architecture of a Dynamic Virtual Cluster

Job spanning

Job spanning allows processors from multiple clusters to be combined in order to execute a single job.

Software environment compatibility

Virtualisation allows the software environment to be customised on a per-job basis, offering a solution to the problem of maintaining a broad array of software packages on an HPC resource. The ability to create a consistent software environment for the job between multiple clusters is also a prerequisite for forwarding and spanning.

It is suggested that these capabilities offer an opportunity to improve resource performance: utilisation and job throughput of several clusters in close proximity can be increased by exploiting the combined idle capacity (Emeneker et al., 2006). For example, if a job is waiting in the queue on cluster A, but the combined idle capacity of clusters A+B meets the requirement of the job, a virtual cluster can be

provisioned that spans the idle nodes within both clusters. The queued job can then be forwarded to this virtual cluster and started immediately. However, the authors do not discuss how the inter-cluster interconnect is implemented to enable this, beyond the requirement that the nodes within one cluster must be able to communicate directly with the nodes in every other cluster. The typical architecture of an HPC system does not scale geographically, characterised by the use of a private, low latency network for inter-node communication. In order to challenge this approach, the feasibility of creating the physical connections, in addition to the effect on security and performance, must be addressed.

The implementation of a DVC is presented based on the Moab resource manager (Emeneker et al., 2006). The resource manager is modified in order to provide integration for deploying Xen-based VMs, static assignment of IP addresses to each VM, and the ability to lease virtual nodes to another cluster. However, these design decisions introduce significant limitations: Firstly, each cluster wishing to participate in a DVC must use the same resource manager. Therefore, a significant part of the environment between the clusters must already be homogenous. This limits the prospect of deploying a DVC between established clusters, given a similar diversity in resource managers and middlewares as with user communities in HPC and their respective applications. Secondly, a DVC requires deployment-specific information to be embedded in advance, such as static IP addresses and hostnames. Services such as forward and reverse DNS resolution must also be available for all physical and virtual nodes. Therefore, it is not possible to dynamically instantiate or scale a DVC beyond the initial deployment it was designed for. Finally, the user must manually create a VM image that contains an OS installation, middleware, libraries and required packages. Therefore, whilst a DVC can be deployed in response to a job submission, it is unlikely that a typical user will hold the required system administration knowledge in order to customise it.

A method in order to evaluate the potential impact of a DVC on resource performance is presented by Emeneker and Stanzione (2007). A workload profile composed of 60% parallel and 40% serial jobs is defined based on historical workload data, using instances of the HPL benchmark to create the distribution of jobs. Two clusters are used for deployment of the DVC, interconnected via a single Gigabit Ethernet link. The authors demonstrate that, despite the limitations of virtualised execution, workload performance was improved; The ability to start individual jobs earlier in time by forwarding or spanning offsets the performance cost, decreasing the turnaround time, and thus, increasing the throughput of the workload. Overall, the DVC realised a 33% reduction in turnaround time and a 57% increase in throughput. However, it is uncertain if this result will translate to other HPC environments and applications - the experimental configuration was small scale, using 12 nodes per cluster and 1 processing core per node. This does not represent a high demand on the inter-cluster interconnect, reflected in only an approximate 5% increase in the run time of a spanned job, despite the obvious networking inefficiency.

OSCAR is a middleware that automates the deployment of physical HPC clusters (Des Ligneris, Scott, Naughton, & Gorsuch, 2003). It provides a mature platform for managing parallel and distributed computing resources, considering both the deployment and configuration of a cluster. The main functionality of OSCAR is implemented as a wizard, guiding the user through the process of selecting software packages, generating images for deployment, node discovery and installation, and finally, validating that the cluster is operational. OSCAR-V is an extension of this middleware in order to support Virtual Machines (VMs) as a deployment target (Vallée, Naughton, & Scott, 2007). It enables OSCAR to manage both the deployment of the physical and virtual nodes, including the installation of a cluster environment within them. This approach allows several distinct cluster environments to be deployed on top of a single physical cluster, rather than requiring the virtual

nodes to be integrated into the physical cluster as in a DVC (Emeneker et al., 2006). To improve portability, OSCAR-V introduces V3M, a C++ library providing a high-level interface which abstracts the underlying virtualisation technology. Therefore, the VMs created by OSCAR-V can be deployed on any virtualisation technology supported by V3M, rather than tied to a specific hypervisor or VMM.

In order to take advantage of the OSCAR-V deployment and configuration framework, the underlying physical cluster must be built with OSCAR. This means that while the virtual cluster could be reused between OSCAR managed clusters, they cannot be deployed into other execution fabrics which do not provide the V3M and OSCAR interfaces required to configure the VMs at run time. While not specifically stated by the authors, the images built for VM deployment in OSCAR-V are not distinguished in any way from the images built for the physical nodes. This is a notable advantage, as it does not introduce virtualisation-specific modifications into core components of the cluster software stack, such as the resource manager, and benefits from the user-friendly wizard for creating the VM image.

However, OSCAR is a tool primarily aimed at the initial configuration of a cluster, and does not allow dynamic instantiation of virtual clusters, such as at job submission time. Furthermore, the OSCAR middleware is not currently maintained or updated for modern Operating Systems: the latest version supports CentOS 5 which is no longer supported (*CentOS Linux 5 EOL*, 2017), and the official website was last updated in 2005 (*OSCAR Homepage*, 2005).

The Virtual System Environment (VSE) provides an alternative mechanism for integrating virtualisation into a cluster fabric (Vallée et al., 2008). This work aims to solve the problem of adapting or porting software to meet the execution requirements of different cluster fabrics. It suggests that, since the science is performed within the application, the OS and runtime environment should be adapted to meet the

needs of the application, rather than adapting the application to meet the needs of the resource. Therefore, the VSE addresses the creation of a tailored virtual cluster environment and its deployment on a target resource.

In a VSE, the VM image is defined by the combination of *package sets*. For example, the user can provide a package set containing the required software for their computational requirements. The system administrator can provide a package set with the resource manager and other software required to meet the local policy of the system on which the VSE will be deployed. The combination of package sets defines the VSE and is stored in an XML file. This is used to generate a *golden image*, which contains an installation of the package sets and defers configuration until deployment time, allowing the same image to be reused between different deployment configurations. The VSE is integrated into the OSCAR-V middleware (Vallée et al., 2007), which provides the package sets, configuration and deployment capabilities. This also ensures that the VMs built using the VSE methodology conform to the cluster architecture and include the required software stack in order to implement a distributed or parallel system.

A clear advantage of this model is in the abstraction of the virtual environment definition, allowing the user to compose a VSE by simply specifying the required package sets and the image is generated automatically. However, this abstraction is supported by a custom binary packaging format that documents software dependencies as well as configuration required to support deployment in a cluster context. This introduces an additional burden in order to build these packages and maintain them in the long term, whereas a typical VM image can be customised using standard distribution or upstream provided packages. Therefore, this undermines the advantage of delegating the environment customisation tasks to the user: whilst they most likely hold the best knowledge on how to configure their software, they

are unlikely to hold the required system administration knowledge in order to create, modify and maintain package sets. The authors attempt to address this limitation by suggesting that the responsibility for maintaining the package sets would be on the application developer, not the user. However, a user wishing to customise either the package or how it is configured would still face a high barrier of entry.

2.2.2 In a Grid Fabric

The Virtual Workspace (VW) is proposed as a top-down approach to integrating virtualisation within the grid architecture (Keahey, Foster, Freeman, Zhang, & Galron, 2005). It is based on an abstraction of a "transient, dynamically created execution environment" that provides the required services or supporting infrastructure for an application to run on a resource (Keahey, Doering, & Foster, 2004). The interactions required in order to configure the VW are implemented as grid services and a reference implementation is presented based on the Globus Toolkit. It does not specifically define a virtualisation technology, rather, a set of criteria that it must provide. These include providing protection between user resources, enforcement of policies such as CPU time, the ability to configure and preserve state such as environment variables and the ability to support a diverse range of codes. It is suggested that using classical and paravirtualisation techniques to implement a workspace offers the most flexibility (Keahey et al., 2005). The *VW Manager* service is responsible for the implementation details of how to launch the virtual environment on the target resource. The *VW Factory* provides the interface to the grid that allows authorised users to request a VW. However, once created the VW is atomic and does not typically interact with other workspaces or services, apart from those required by the application being run within the workspace. Therefore, it cannot be used on it's own in order to provision a virtualised cluster environment.

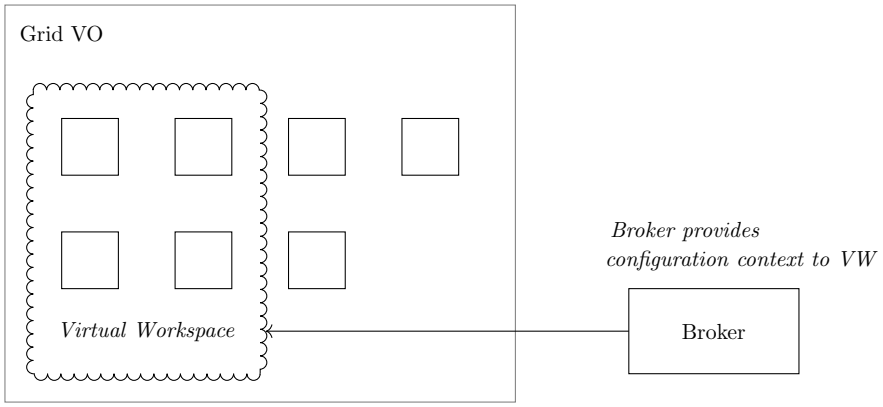


Figure 2.4: Architecture of a Virtual Workspace in the Grid

Extensions to the Virtual Workspace by Foster et al. (2006) and X. Zhang, Keahey, Foster, and Freeman (2005) provide the ability to provision a linked set of workspaces as shown in Figure 2.4. Metadata describing the set, such as hostnames and number of nodes, is provided in an XML file, in addition to the configurations for the individual workspaces. For example, a set of VWs sharing a common image can be deployed to a resource that represent worker nodes of a cluster. This set can be deployed on any grid resource that provides VW services, without requiring knowledge of how the underlying resource is configured. The primary limitation in this approach is the requirement for the user to supply a VM image, inheriting the usability issues associated with generating this. However, it provides basic facilities to separate configuration from the VM image through the XML file, specifying scripts to be executed at VM boot time. This improves the portability of the image, as it allows adaptations based on the deployment context to be performed separately from the software environment customisation using a scripting language of choice. For example, scripts could be used to install additional packages, certificates, or configure a client application, but must still be defined statically.

Image propagation is identified as a limitation that is likely to affect the scalability

- the deployment time of a VW is dominated by copying the VM image to the target resources, and increases with the number of nodes in the set. To improve the scalability, where more than one workspace uses the same image, a solution is proposed where the image is transferred once to the resource across the grid, and then cloned to the respective nodes using a local interconnect at the start of execution (X. Zhang et al., 2005). Once the files for a workspace are staged, the overheads of launching, executing and destroying are minimal.

A proof-of-concept deployment of VWs is presented in Keahey, Freeman, Lauret, and Olson (2007) for STAR, a High Energy and Nuclear Physics application. Due to the fact that the configuration must be defined statically, a head node is manually deployed and an image for the VW is created that contains the hardcoded address of the head node. The user may invoke instances of the image on a pool of dynamic worker nodes, deployed across systems in the grid by the VW services as required. Even though the experiment was successful, an additional limitation in terms of image management is identified, suggesting that the lack of methods for easily procuring images and demonstrating their provenance will prevent resource providers from offering VM hosting capabilities, for fear of attracting low quality or harmful codes. The authors predict that this will prevent users from taking advantage of virtualisation for their workloads. One could argue that all customisation can be performed by scripts and the VW only needs to provide a minimal set of trusted images for this purpose. However, this would require the user to perform more significant customisations within the XML file, requiring a higher degree of system administration expertise in order to define. Furthermore, the customisations will be applied everytime the VM is booted, potentially extending the provisioning time of the virtual cluster.

Contextualisation aims to more clearly define the configuration required for a Vir-

tual Workspace in order to adapt it to a deployment context (Keahey & Freeman, 2008). This work recognises that in order to integrate a set of virtualized resources together as a logical cluster, a dynamic provisioning method is required in addition to other configuration tasks. Contextualisation provides a clear separation between creating a reusable *appliance*, deploying an instance of the appliance, and the subsequent contextual configuration. The usefulness of this separation is in supporting the division of labor between a provider, responsible for maintaining the virtualised software environment package, and a deployer, responsible for mapping it to an available resource with the required contextual configuration. This does not allow users unlimited control over what environment they wish to define, allowing them to select from prebuilt appliances - on top of which they can supply runtime customisations such as input data - rather than requiring the environment to be built from scratch. The deployment mechanism performs both the contextual configuration and installing the user-supplied customisations. This reduces the amount of system administration knowledge required by the user in order to operate a virtual environment.

An example is demonstrated where the configuration of an NFS server and client, running in separate VWs, is performed based on IP address allocations obtained from a grid resource manager (Keahey & Freeman, 2008). Scripts are executed depending on the role of the workspace - for the server, it executes a script that parses a file describing NFS clients and generates the appropriate configuration. This is implemented on top of a simple dependency graph that describes the relationship between workspaces. A *context broker* runs as an external service, which provides a method of querying contextual information, such as hostnames and IP addresses, from a grid resource manager.

Contextualisation offers a clear advantage over previous configuration schemes in

that it allows the definition of dependencies between the instances within a virtual cluster. This enables dynamic instantiation without knowledge of the infrastructure in advance, and allows reuse of common components in different contexts. However, from the end user perspective, it is relatively inflexible - it requires the user to choose from predefined appliances, in which only a limited set of parameters can be customised. It also places the burden of maintaining the appliance on the administrator or software provider.

The Virtual Workspace and contextualisation models are complementary - the latter can be used to address the configurability limitations of the former. However, they are designed to work within a grid infrastructure. This presents a high barrier of entry in order to implement: context broker and deployer services must be compatible with the desired execution fabric in order to support workspaces and appliances. Therefore, the hosting system must either implement the grid software stack, or provide its own VW-compatible interfaces. In some execution fabrics, there may not be a central source of contextual information that can be queried. This does not immediately make them portable outside of grid infrastructures.

2.2.3 In a Cloud Fabric

StarCluster is a toolkit that facilitates dynamic creation of virtual clusters on the Amazon EC2 service (*StarCluster*, n.d.). Originally designed to support the computational requirements of university courses, where the resource may only be required for a short duration throughout the year, the toolkit provides automated provisioning of a cluster software environment in the cloud, as shown in Figure 2.5.

Virtual cluster nodes can be added or removed on demand, with the appropriate reconfiguration of the resource manager, Open Grid Scheduler (*Open Grid Scheduler/Grid Engine*, n.d.), taking place automatically. This process is conceptually

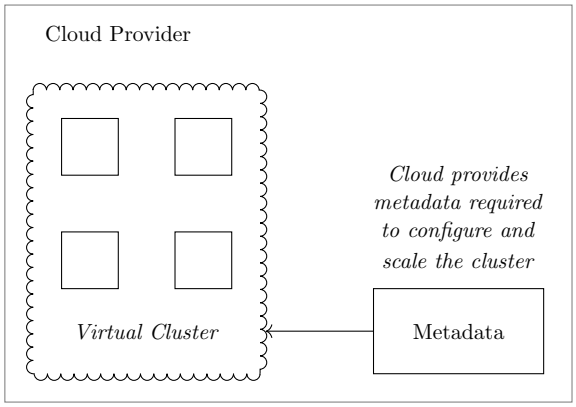


Figure 2.5: Architecture of StarCluster on Amazon EC2

similar to that of Keahey and Freeman (2008); the EC2 service is providing contextually relevant metadata, such as IP addresses, in order to perform dynamic configuration. This configuration includes creation of VM instances, hostnames and IP addresses, Public Key Infrastructure (PKI) and a shared filesystem among the virtual nodes.

An load balancer is utilised in order to utilize the elasticity of the cloud deployment method. It is a separate service that continuously polls the job queue to determine when more nodes are required, allowing the virtual cluster to be automatically scaled proportionally to the queue length, thus optimising the hourly usage for a workload profile that is difficult to predict in advance. This implementation demonstrates that a flexible and dynamic solution can be created without directly modifying core components of the cluster software stack. The service performs the required task (such as adding a new virtual node) in an automated manner, but using the same interactions which would be performed manually by the administrator. However, it is dependent on the Amazon EC2 API and does not offer support for local resources or other cloud providers.

Whilst StarCluster has desirable features, it is aimed at a relatively narrow use case

which is reflected in the capabilities for customisation. A minimal default image is provided, and the user can select from a range of plugins in order to deploy common cluster applications, such as MPI, Hadoop, MySQL and Python (*StarCluster Plugin Documentation*, 2011). However, it is clear that the configured VM image is not designed to be a reused between invocations of a virtual cluster. This approach allows the barrier of entry to be lowered, providing the configuration prescribed by the plugins is acceptable. Any other customisations rapidly increase this barrier, as they must be performed manually in the same way as a physical cluster, or defined as a StarCluster plugin.

Dynamic Torque is a solution that enables dynamic scaling of a local cluster running the Torque resource manager into an OpenStack cloud (S. Zhang, Boland, Coddington, & Sevior, 2014). It facilitates the provisioning of VMs that represent virtual cluster nodes within the cloud, integrated into the resource manager to appear alongside existing physical nodes in the scheduling queue. Dynamic Torque enables *cloud bursting*, where additional virtual nodes can be temporarily provisioned as an extension of the local cluster in order to process a backlog of jobs. Similarly to StarCluster, the integration approach has been to create an external service that runs parallel to the resource manager, performing the communication with the OpenStack API in order to automate common deployment tasks. However, it suffers from similar limitations: it requires the virtual cluster to be deployed on an OpenStack cloud, and does not consider the creation and customisation of the VM image used for the worker nodes - only the methodology for dynamically scaling them. The administrator must supply an image configured in the same way as the physical worker nodes in the local cluster.

Unlike a virtual cluster that is nested within an existing cluster fabric, a standalone cluster configured in a cloud fabric poses the same application portability challenge

as a physical cluster; They are largely equivalent from a configuration perspective, still requiring a homogeneous environment. Even though the cloud enables a multitude of distinct clusters to be created, each with it's own unique software environment, it is clear that the low level abstraction of the VM does not lend itself to easy and efficient customisation by an end user.

2.3 Containers

2.3.1 Run Time Environment

In Chamberlain and Schommer (2014), a method of creating Docker containers in order to encapsulate scientific software is presented. The motivation for this work is to improve the reproducibility of a computational experiment. They identify that any code used to run the experiment is only a single part of a larger software environment, which would need to be replicated in order to run the code. Therefore, they propose that containers can be used to encapsulate this environment, and present a workflow where containers are used as both the method of development and execution.

The container is based on a minimal Linux installation, available from the Docker Hub online repository (*Overview of Docker Hub*, n.d.), and must include an installation of all dependencies and libraries required for the execution of the code. This container is used as a base to extend two more containers that are customised for development or execution. The development container will include tools such as compilers and debuggers, whilst it is not necessary to distribute these with the final runnable code. In the context of a workflow, the container is invoked in place of tools that would otherwise be installed locally.

The authors advocate the use of containers for runtime environment virtualisation: unlike a VM which requires a separate OS instance to be booted within, the container is executed seamlessly within the host OS as though it was any other executable, but within the environment defined by the container. The advantage of this approach is that it offers an accessible path to reproducibility, with minimal disruption to the user. However, the main contribution is a method for defining the environment of the container, and in this regard it does not consider where to draw the line between what is reasonable to expect the host system to provide and what the container must provide.

Boettiger (2015) provides a more detailed analysis of the opportunities and benefits, demonstrating a similar workflow using R, a programming language designed for statistical computing and data analysis (*The R Project for Statistical Computing*, n.d.). They argue that virtualisation at the runtime level allows the researcher to use familiar tools, such as text editors and web browsers, whilst ensuring that the code execution occurs in the container - where the environment remains consistent and can easily be deployed to other systems without having to adopt different workflows. A set of best practices are proposed to ensure that the advantages of virtualisation are realised in the context of reproducibility. This includes setting up the environment of a project within containers from the outset, and using Dockerfiles to define the configuration of the software dependencies, environment variables, and other necessary configuration required in order to execute the application. However, as in Chamberlain and Schommer (2014), it does not adequately consider how the container will be executed in context on a resource, and the extent to which software dependencies and libraries should be included in the container, such as those that provide interactions with the host for process launching, networking and storage. Therefore, whilst this methodology might be appropriate for simple or sequential workflows, it does not suggest that it can be applied to distributed parallel applica-

tions that are typical of HPC workflows.

Singularity has popularised *executable containers* for HPC that addresses some of these issues, where the container provides the runtime environment to support a single application (G. Kurtzer, 2016). In most cases, the container can be utilised as though it were any other executable on the system, and the behaviour is consistent. The transparency of this approach offers clear advantages as it presents an extremely low barrier to adoption and does not require the user to adopt new workflows, as established by Boettiger (2015).

Similarly to Docker, Singularity defines the entry point to the container as a script that will be executed upon starting (G. M. Kurtzer, Sochat, & Bauer, 2017). This script describes the default behaviour of the container, unlike a traditional VM, which would require booting, logging-in and subsequent manual launching of the desired processes. Typically, a Singularity container follows the layout of a Linux root filesystem and several minimal Linux images are provided, in addition to the ability to import Docker images. The rest of the container definition is left to the user to decide. Whilst the user may include as much or little supporting software as desired, Singularity differs from Docker in that, by default, it recommends that many aspects of the host system will be shared with the container. For example, a container for an MPI application does not need to include the MPI libraries or environment - the host's version of these can be used (*Using Host libraries: GPU drivers and OpenMPI BTLs*, 2017). This is presented as a strength of the Singularity model since the host will typically provide optimised versions of high performance libraries to complement its hardware. However, whilst this solves the software environment flexibility problem within the scope of a single HPC system, the reduced portability of this model is overlooked - what if the system executing the container does not provide a native MPI implementation? A method to express such contextual depen-

dencies within the container is not available. Therefore, container execution will fail on a system that does not meet these requirements, and it limits the scope to which they can be shared and reused.

Despite this, a large amount of software is already packaged for Singularity. Ghosh (2017) present a training environment that must be executed on a cluster providing a resource manager and GPU hardware resources. Elghraoui (2017) present a containerised version of TensorFlow, which depends on filesystems available only on the National Institutes of Health HPC system. On the other hand, Shiratori (2017) present a containerised version of OpenFOAM, which offers no integration with the host system and thus, good potential for reuse. Therefore, it is apparent that, in a relatively short amount of time, a distinct class of containers have emerged that follow the runtime environment virtualisation model and are *exclusive to HPC*, whilst others can be shared, reused and executed on a variety of host systems.

Charliecloud is an alternative container runtime designed for the HPC use case (Priedhorsky & Randles, 2017). Similarly to Singularity, it implements runtime environment virtualisation aimed specifically at the HPC use case. Unlike Singularity, it adopts the Docker workflow for generation and dissemination of container images, providing only the minimum functionality needed in order to run the container in a way that is consistent with regular applications on an HPC resource. This provides an advantage if the user is already familiar with Docker, maintaining a high level of accessibility and low disruption to existing scientific workflows. However, this minimal approach suffers from the same limitations: While it allows end user customisation of the environment to suit an application, it does not ensure that they will be portable between systems. The weak level of abstraction from the underlying software environment limits the potential for re-use of such images in different execution contexts, especially those where the host cluster is not of the same type

required by the application.

2.3.2 Full Stack Environment

Containers do not have to be used exclusively for run time environment virtualisation, and may also be utilised in a way which reflects traditional virtualisation techniques. In this mode, the container is configured to run a full software stack so that the guest environment is more typical of a running OS, rather than a single process. Therefore, a complete cluster nodes can be virtualised within containers, similar to the methods outlined in Section 2.2.

In Yu and Huang (2015), a prototype implementation is presented for an auto-scaling virtual cluster encapsulated within Docker containers. The architecture of this virtual cluster is outlined in Figure 2.6. The deployment model uses separate images for head node and worker node containers. It supports dynamic rendering of a host file, which lists the IP addresses of the worker node containers. This is achieved using an external key-value store service, Consul (*Consul by HashiCorp*, n.d.). An agent runs within the container which registers the current container's IP address and polls the key-value store service for changes, in order to update the host file when the cluster topology changes. Execution of a *hello world* parallel code, using Message Passing Interface (MPI), is demonstrated across 2 containers. From the head node container, the program is executed using the automatically generated host file. When another worker node container is added to the system, the program can be executed again taking advantage of the new worker node - the host file is automatically updated without having to stop and restart the entire virtual cluster. This approach encapsulates all the application layers required for parallel execution within the container itself. This offers better portability when compared to run-time environment virtualisation, as the execution method on which the application

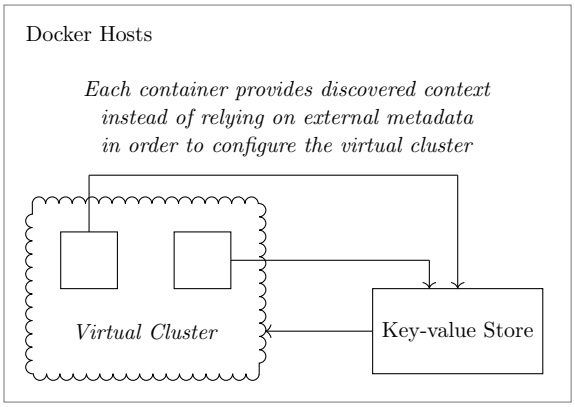


Figure 2.6: Architecture of a prototype Virtual Cluster using Docker

depends is included within the container, rather than being an undocumented dependency that the host system is expected to provide. In addition, configuration of the execution method is documented by the ability to dynamically render the required host file. Therefore, these containers could be deployed on a system that is not natively a cluster and the application would function as expected.

However, the proposed implementation is based on unrealistic assumptions about the underlying environment. Firstly, the containers of the virtual cluster are bridged onto the physical network of the host, and it is assumed that the network will dynamically assign an IP address to them. This may not be the case for all environments and potentially offsets the portability gain. Secondly, whilst the solution appears to implement some of the principles of contextualisation (Keahey & Freeman, 2008), only the model of a head node initiating communication to worker nodes can be represented. There is no provision for the worker nodes to discover the IP address of the head node, for example. Finally, whilst a method is available for configuration files to be updated when worker nodes are joined or removed from the virtual cluster, other configuration actions may need to take place in the context of a full software stack, such as service reloading.

In general, this auto-scaling cluster model is situated halfway between run time environment virtualisation, and virtualisation of an entire cluster software stack. It does not provide adequate capability to configure the non-trivial environment posed by a full cluster middleware, but in the context of a simple MPI application, it is shown to offer improved portability and dynamicity than other solutions.

In contrast, Kniep (2014) propose a containerised virtual cluster that more closely aligns with the microservice model, in order to exploit the deployment and orchestration capabilities of Docker. The initial implementation demonstrates a cluster that mixes several OS environments, where the environment can be chosen at the time of job submission, and is based on the SLURM resource manager (Jette & Auble, 2010). However, it requires statically launching the containers on each node, and configuring the partitions in the resource manager in order to direct jobs to those containers.

An improvement to the model is demonstrated where services within the cluster are represented by separate containers that are launched when required (Kniep, 2016). The resource manager's process launcher resides within a container on each node, and new containers are launched in order to provide the environment for each job. The advantage of this model is that it supports using the wide range of already mature tools for orchestration and management that exist for Docker outside of the HPC use case. This includes network virtualisation, allowing the containers to communicate within a private virtual network across multiple hosts.

However, this model presents a departure from typical process orchestration within HPC systems. Whilst it addresses issues such as environment customisation and performance, it potentially poses new usability issues in terms of system administration by using tooling and techniques that will be unfamiliar to the vast majority of existing systems. In addition, it does not detail the modifications required to

the SLURM resource manager, or the requirements in order to build the container images for job execution.

In de Alfonso, Calatrava, and Moltó (2017), *EC4Docker* is presented as a tool in order to build virtual clusters within a container environment. The authors present a model where each container represents a worker node in the system. However, it does not provide a complete full stack environment: a head node container is deployed manually and managed by a separate process that facilitates dynamic addition or removal of the worker nodes depending on the queue length. This functionality is presented as an innovative approach, despite affording the same capability as Yu and Huang (2015) and applying the same configuration principle used by Dynamic Torque (S. Zhang et al., 2014), where the required contextual information in order to scale the system is obtained by polling an external API. Therefore, it does not resolve the fundamental challenge of portable configuration and adaptation of the environment based on the deployment context, which is evident from previous work. Furthermore, the topology of the virtual cluster model that is implemented is not flexible, and must conform to the traditional batch scheduling cluster paradigm, using NFS to share data between virtual nodes. This fixed architecture is not appropriate as a general purpose virtualisation solution for HPC.

2.4 Synthesis

A taxonomy of virtual clusters within HPC is constructed in Figure 2.7 based on the reviewed work. The implementations that use traditional techniques, i.e. classical or paravirtualisation, are based around the model of a *full stack* cluster software environment running within the VM instance. This represents either a standalone virtual cluster (Keahey et al., 2007; *StarCluster*, n.d.) or virtual worker nodes that are in-

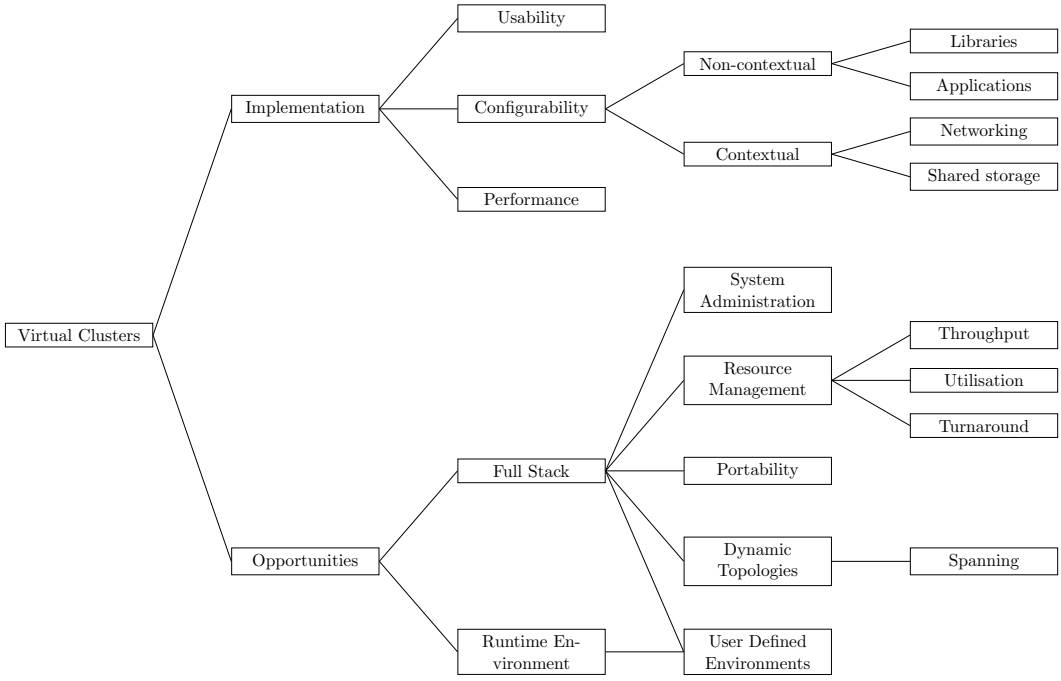


Figure 2.7: Taxonomy of Virtualised Clusters

tegrated into an existing physical cluster (Emeneker & Stanzione, 2007; S. Zhang et al., 2014). By virtue of the fact that a VM must be configured as though it were a real machine, this model can be used to solve the software inflexibility problem in HPC systems by allowing the user to provision a cluster environment tailored to the requirements of a specific application. However, the literature also identifies opportunities to realise other benefits. Virtualisation promotes better portability of software environments between different execution fabrics. This capability can be used to facilitate cluster spanning and job forwarding in order to improve system performance, in terms of utilisation, job throughput and turnaround time (Emeneker & Stanzione, 2007). In addition, it is suggested that VMs can be used to ease the burden of system administration by allowing changes to the system-level libraries and middleware to be tested and deployed in a controlled way, without putting an already established cluster at risk (Youseff et al., 2006).

A common workflow is present throughout the solutions, composed of image procurement, configuration, VM deployment, workload execution and subsequent VM destruction. This workflow, detailed in Figure 2.8a, represents the steps that must be taken in order to invoke a virtual cluster environment on a particular resource, regardless of the underlying implementation or fabric. However, no one solution satisfies the requirements of each step in this workflow for the HPC use case.

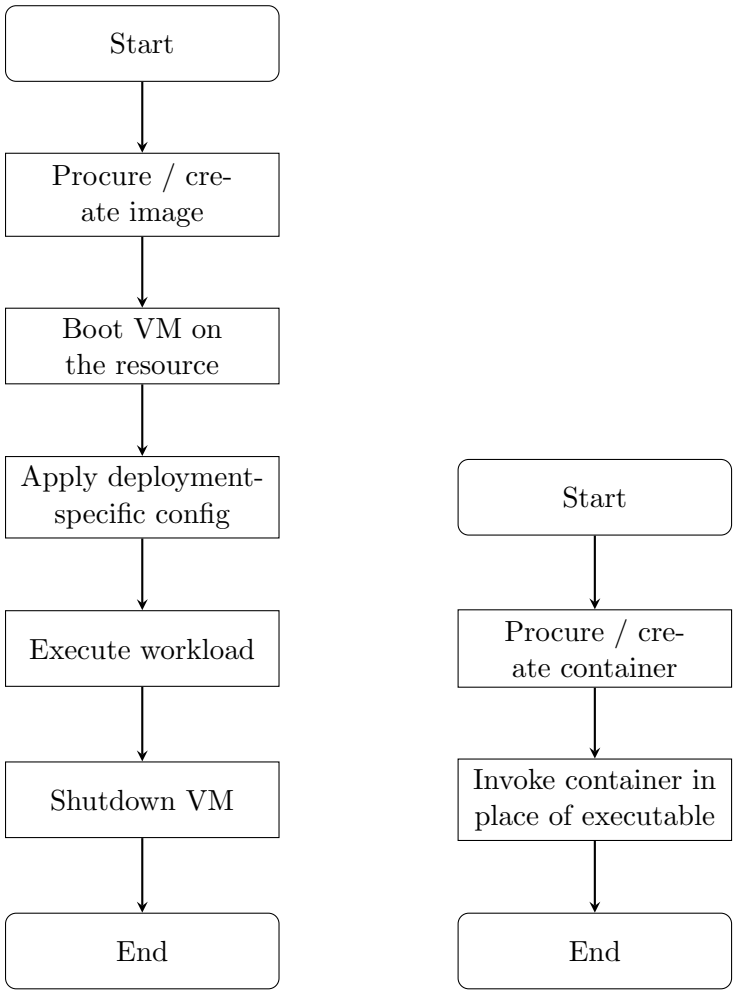
The DVC and Virtual Workspace models require the user to supply a VM image in order to deploy the virtual cluster (Emeneker & Stanzione, 2007; Keahey et al., 2007). This poses a significant usability challenge due to the knowledge required in order to create the VM image, which the average user is unlikely to hold. Typically these images are several Gigabytes in size as they consist of an entire OS and software environment installation. This creates not only a problem for transferring the image to the target for execution, but is an inconvenient file format for procurement and customisation by others.

Furthermore, there is inconsistency in the separation between configuration performed at the time of VM definition and configuration performed at the time of VM deployment. For example, DVC does not specify any separation of configuration responsibility (Emeneker & Stanzione, 2007). Therefore, these virtual clusters have to be manually reconfigured for each deployment scenario. On the other hand, contextualisation defines an approach for configuration of Virtual Machines deployed within a Grid. This allows deployment-specific configuration to be performed automatically that may not be known in advance at the time of definition, such as IP addresses and hostnames (Keahey & Freeman, 2008). However, this relies on metadata obtained from a grid resource manager which will not be available in other fabrics. Cloud based solutions have a relatively rigid configuration mechanism as they are closely mapped to the platform-specific APIs provided by the service

on which they are implemented. For example, Dynamic Torque effectively implements cluster spanning between a local and cloud resource, but instruments this is in a inflexible way, as the virtual cluster nodes must be configured identically as an extension of the physical cluster nodes (S. Zhang et al., 2014).

Containers are defined in a text file which describes the commands the user would issue in order to configure the virtual environment (Boettiger, 2015; G. M. Kurtzer et al., 2017). A container can have a single dependency in that it extends another container, using it as a base image. The Docker and Singularity hubs offer a service for procuring, sharing and asserting provenance of the container images built from such files (Boettiger, 2015; Sochat, 2017). This approach is more accessible to the average user, as it does not require the system administration experience needed to construct an image from scratch, such as installing and configuring an OS. It also optimises the transfer of data by reusing common image layers between containers, reducing both the on-disk storage requirements and the time taken to propagate images to the execution system (Merkel, 2014). Furthermore, the performance overhead of containers is very low (Felter et al., 2014; Le & Paz, 2017), and the workflow in order to deploy them requires less steps than the VM approach, thus smaller adjustments to the normal working patterns of the user, as shown in Figure 2.8b. Therefore, it is clear that the limitations in terms of usability and performance in the traditional full stack approaches can be improved using container virtualisation.

However, the typical usage model of containers is to virtualise only the run time environment of an application - a paradigm that was not possible with traditional virtualisation techniques. Whilst this solves the user-defined environment problem, it does not allow the other opportunities to be realised, as the container does not consider the lower levels of the cluster software stack where they would be imple-



(a) Full stack virtualisation (b) Runtime virtualisation

Figure 2.8: Workflow comparison of full stack cluster node versus runtime environment virtualisation

mented. Yu and Huang (2015) and Kniep (2014) introduce solutions that employ full stack virtualisation within a container environment. However, these solutions are presented by the authors as innovative without fully considering the HPC context in which they are placed. Therefore, they similarly do not appreciate the opportunities identified in the literature that are shown to have a quantifiable improvement in usability or performance when deployed in a cluster environment.

Counterintuitively, both container and traditional virtualisation solutions suffer from low portability. The services that enable good configurability for VMs are not portable, requiring complex grid infrastructure in order to be deployed. This means that whilst the VM is inherently portable due to the high level of abstraction from the underlying system, the orchestration framework that enables it to be used effectively within an HPC system is not. While containers do not necessarily rely on external services in order to be deployed or configured, they similarly inhibit the portability by introducing dependencies on services provided further down the software stack on the host system (Ghosh, 2017; Elghraoui, 2017). For example, a containerised MPI application that can only execute on a system providing the interfaces for MPI execution.

Based on the taxonomy in Figure 2.7, implementation-specific features that are required in order to exploit the recognised benefits of virtualised clusters are used for comparison between the solutions, detailed in Table 2.2. The solution is scored 0 if it does not consider the relevant feature. If the solution considers the feature but has limitations within the HPC context, it is scored 1. If the solution considers the feature without limitations in the HPC context, it is scored 2.

It can be seen that the container implementations offer strength in features that affect usability, but have configurability limitations that prevent them from being used in order to realise benefits other than a user-defined software environment.

	Ideal solution	*DVC (Emeneker & Stanzione, 2007)*	*VSE (Vallée et al., 2008)*	*Virtual Workspace (Keahey et al., 2007)*	*Contextualisation (Keahey & Freeman, 2008)*	*StarCluster (StarCluster, n.d.)*	*Dynamic Torque (S. Zhang et al., 2014)*	*Autoscaling Docker (Yu & Huang, 2015)*	*qmib (Kniep, 2016)*	*EC4Docker (de Alfonso et al., 2017)*	*Singularity (G. M. Kurtzer et al., 2017)*	*Charliecloud (Priedhorsky & Randles, 2017)*
User customisable image	2	1	1	1	1	1	0	2	1	1	2	2
Supports multiple nodes	2	2	0	2	2	2	2	2	2	2	2	2
Standalone deployment	2	0	0	0	0	0	0	2	1	2	2	2
Span multiple fabrics	2	2	0	0	0	0	2	1	1	0	1	1
Separate deploy / run config	2	0	0	0	2	1	0	0	0	0	0	0
Contextual aware config	2	0	0	1	2	2	2	2	0	0	0	0
Dynamic adjustment	2	0	0	0	1	2	2	2	0	1	0	0
Custom topologies	2	0	0	0	2	0	0	0	0	0	0	0
Provides image repository	2	0	0	0	0	1	0	2	2	0	2	2
Improve portability	2	1	0	0	1	0	0	0	0	0	0	0
Improve performance	2	1	1	1	1	0	0	1	1	0	1	1
Improve administration	2	1	0	0	0	0	0	1	1	1	1	1
Total	24	8	2	5	11	9	4	15	9	7	11	11

Table 2.2: Feature comparison of virtualised cluster implementations

Conversely, the traditional approaches overall have better configurability but suffer from poor usability. However, a combination of these approaches has the potential to provide an ideal solution that addresses each aspect of virtualised clusters more comprehensively: specifically, a solution that employs containers as an incremental improvement over the existing full stack cluster virtualisation approach. In addition, solving the limitations in terms of configurability and portability will allow deployment models to be considered which are otherwise not possible with existing container solutions, such as deploying cluster applications on a host system that does not provide the interfaces required for cluster execution, spanning geographically distributed and heterogeneous execution fabrics, and nested virtualisation of cluster environments.

2.5 Summary

In this chapter, a review of related work was conducted. Two types of virtualisation approaches within HPC were identified: full stack virtualisation using traditional VMs and run time environment virtualisation using Linux containers. Both of these approaches offer a solution to the software environment flexibility problem. The limitations within the existing work addressing these two types of virtualisation can be summarised by those that require the user to hold an unreasonable amount of system administration knowledge, and those that rely on bespoke or deployment specific services which cannot be ported to arbitrary deployment fabrics. However, whilst containers offer a more accessible virtualisation technique for the average user, the full stack approach demonstrates a wider range of opportunities to improve HPC systems using virtualisation:

- Performance, in terms of job throughput, turnaround time and resource utili-

sation

- Administration and Deployment
- Portability of compute environments

Whilst there is limited previous work that addresses implementation of the full stack approach using container virtualisation, it does not adequately consider the context of previous efforts in this area, and thus, does not facilitate the potential benefits and opportunities. Furthermore, an examination of both approaches showed that they suffer from poor portability despite the inherent abstraction from the underlying system. The contemporary container implementations in HPC that target run time environment virtualisation do not specifically define the best practice for constructing a container, instead mainly focusing on *how* to technically support containers on an HPC system, rather than *why*.

A taxonomy describing virtual clusters is synthesised based on the work reviewed in this chapter, shown in Figure 2.7. From this, critical implementation details were identified and used to compare the existing solutions in Table 2.2, demonstrating that no single system is suitable for general purpose virtualisation in the HPC context and providing a framework for the design and evaluation of a new solution.

A container-based full stack virtualisation approach is proposed in order to satisfy these implementation details in a unified virtual cluster model. This solution addresses the following gaps in knowledge:

- In-depth analysis of virtual cluster spanning performance in a range of laboratory and real world network conditions, using a well-known HPC benchmarking toolkit
- Best practice for defining and constructing a container to encapsulate a cluster application

- Discovery, configuration and contextualisation services without dependency on external infrastructure
- Novel topologies
  - Virtual clusters on a single workstation
  - Geographically and multi-fabric distributed virtual clusters
  - Nested virtual clusters
- Objective analysis of the usability and portability impact of full stack container virtualisation in HPC

Chapter 3

Virtual Container Cluster Framework

This chapter describes the design and implementation of the novel *Virtual Container Cluster* (VCC). The VCC framework allows the construction of container images that encapsulate one or more parallel applications within their own full stack cluster environments. The main contribution of this design is in the standalone, self-hosted services for configuration and contextualisation. This facilitates increased portability and the ability to automatically adapt the configuration of the virtual cluster to the deployment context, considering a wide range of services such as DNS resolution, Public Key Infrastructure and dynamic reconfiguration after deployment. Together, these services aim to meet the requirements of the ideal virtualisation solution proposed in Table 2.2. The requirements address the portability limitations that affect previous solutions due to dependencies on underlying infrastructure, and usability limitations that require an unreasonable amount of knowledge in order to operate and customise the solution. A key innovation in this model is defining the boundary of a virtual cluster instance by the network connectivity rather than the software

environment. This boundary can be established across networks with variable levels of connectivity by utilising Software Defined Networking (SDN) technologies.

Firstly, a high level overview of the VCC architecture is discussed, followed by a detailed analysis of the design decisions that compose this architecture. Finally, the low level implementation detail of each service is described. The aspects of the design presented in this chapter are documented in the publication *"Autonomous Discovery and Management in Virtual Container Clusters"* by Higgins, Holmes, and Venters (2017a).

3.1 Architecture

An overview of the general architecture is shown in Figure 3.1. Unlike previous container implementations that provide runtime environment virtualisation, the foundation of every VCC container is a service layer which provides discovery, contextual-aware configuration and dynamic scaling capabilities. Above this, the middleware layer contains the full software stack that is required in order to support execution of the runtime environment.

The service layer discovers relevant information about the container, such as net-

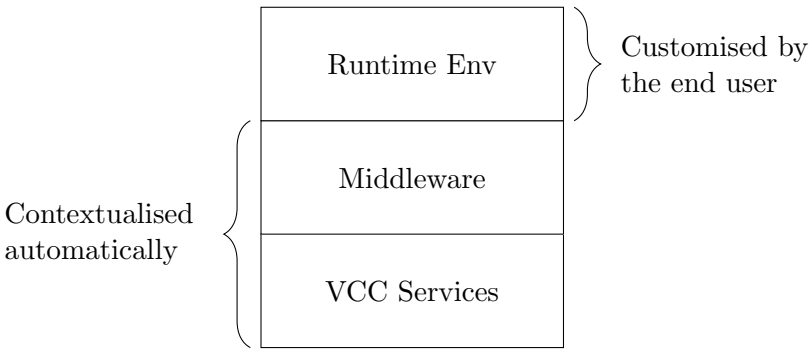


Figure 3.1: Overview of VCC Container Architecture

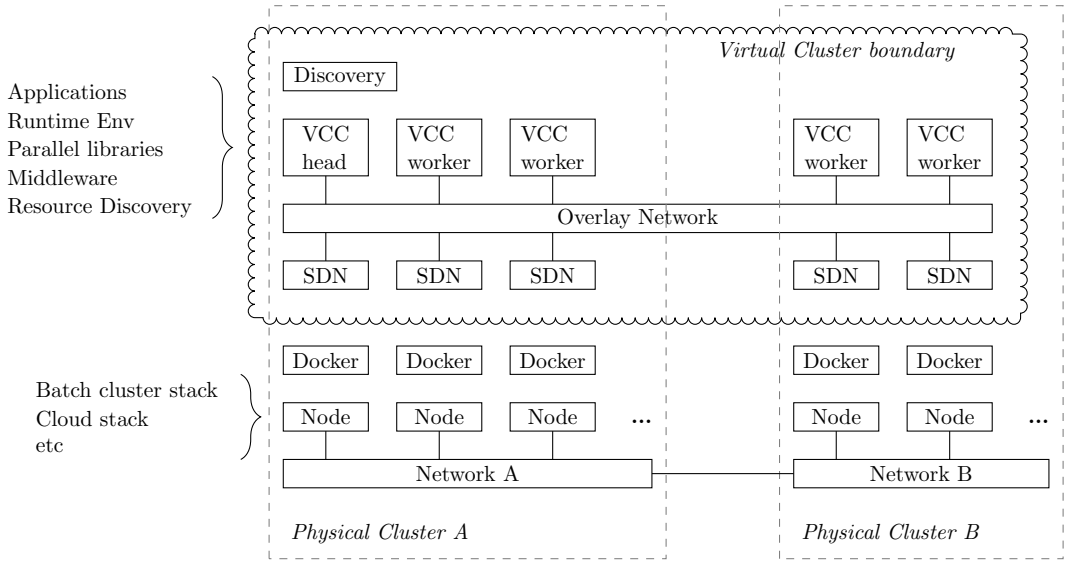


Figure 3.2: VCC Architecture (Higgins et al., 2017a)

work identification, sharing this with other containers through the discovery service. This information is used to dynamically configure the middleware layer, so that the configuration is adapted correctly to the deployment context.

Therefore, the applications within the runtime environment layer can be executed as intended, regardless of the software stack provided by the underlying system. This allows the user to provide customisations without requiring knowledge of how the middleware must be configured in advance.

Figure 3.2[1] outlines a possible deployment for an application which runs in a batch scheduling cluster environment (Higgins et al., 2017a). The key distinguishing features of the VCC, which enable the solution to offer an incremental improvement over existing work, are summarised:

- Encapsulation of a full software stack regardless of the underlying system
- Discovery service providing contextually-relevant metadata without relying on

[1]Permission to reproduce this figure has been granted by the Oxford University Press

external services

- Assigning roles to individual containers in order to define both the unit of scale and how they will be configured
- Facilitating communication through a Software Defined Network, scalable between multiple nodes and network segments
- Retaining the ability for the end user to customise only the topmost (runtime) layer of the container environment

In this example, the containers are orchestrated such that the worker node role is scaled out one per physical host. The container roles, and the relationship between roles, are defined in the configuration of the virtual cluster. Therefore, the VCC can be used to model the configuration of arbitrary application topologies that may not conform to the traditional HPC or client/server paradigms. In practice, this potentially replicates layers of the software stack within the container that are already available on the host system. However, where the host may not provide the required software stack, this is necessary in order to achieve good portability. This is especially applicable to execution within the cloud, where it would be unreasonable to require that the user configures the compute resource as a logical cluster before being able to execute the container.

To facilitate dynamic scaling and spanning features, the responsibility is placed at the services layer, rather than requiring bespoke middleware or patching applications in order to communicate over an inter-cluster interconnect. An SDN solution is used to provide an overlay network, allowing communication between containers distributed among many physical hosts. Where the hosts only have partial connectivity, the overlay network provides a coherent view to the application as though full connectivity is available, routing the communication through a common path

as appropriate. This can be utilised as the virtual inter-node interconnect, in order to offer native spanning capabilities that are transparent to the middleware and runtime environment.

This architecture allows the full potential of containers to be realised within an HPC context - it affords the same level of software environment flexibility as existing container implementations, whilst enabling the dynamic features available in full stack Virtual Machine-based approaches that can improve resource management and performance. Furthermore, each layer of the architecture promotes portability: the networking configuration is decoupled from the physical infrastructure, and the self-contained services for discovery do not require deployment-specific knowledge to be provided manually or by an external, fabric-dependent service. This means that the virtual environment can easily be transported and reused between different execution systems and fabrics.

The virtualisation of networking functions within an HPC system is expected to incur a communication overhead. In Chapter 2, previous work demonstrates that containers do not introduce an overhead in terms of CPU execution performance, however, the latency and bandwidth of the interconnect within an HPC system has a profound effect on the application performance. The proposed solution is expected to increase communication latency between processors in the virtual cluster. In terms of application performance, the effect of this will be slower execution and longer wall time. The size of this effect will depend on the application and type of parallelism that it employs, however, it is anticipated that the dynamic nature of the solution will provide an offset to the potential performance cost.

3.2 Design Decisions

In this section, the design decisions that compose the architecture described in Section 3.1 are justified in detail, in terms of the requirements set out at the end of Chapter 2.

3.2.1 Contextual-aware Configuration

Previous work demonstrates that a separation of configuration responsibility, such as between customisations performed by the end user and the adaptations required at run time, leads to overall improved portability and a lower barrier to entry. The adaptations required at run time typically must be performed by the administrator, requiring specialised knowledge about the underlying infrastructure. Contextualisation defines a methodology where this knowledge can be obtained programatically and used to automate the adaptation of the virtual environment. However, the implementations available in literature depend on metadata services only available within a grid fabric, or provide a simplified model that only allows a client to be configured with the IP address of a server (Keahey & Freeman, 2008; Yu & Huang, 2015).

In the VCC, the separation between user customisation and deployment-specific adaptations is maintained. In order to avoid the requirement for an external metadata service to provide contextually-relevant information about the deployment, each container within a VCC runs a service that discovers information about itself. This information, such as hostnames and IP addresses, is shared with the other containers using a simple key-value store. Each container can monitor the key-value store in order to take action in response to events, such as a virtual node being added or removed.

It is clear that dynamic configuration and dynamic scaling are not interdependent and one is often implemented without the other - for example, Dynamic Torque allows dynamic scaling of a cluster with a rigid configuration, orthogonal to the idea of contextualisation (S. Zhang et al., 2014). However, there is no doubt that a general purpose virtualisation framework requires both of these capabilities. The VCC extends the contextualisation methodology to introduce *roles*, which allow a container to advertise that they offer specific services within the virtual cluster. Therefore, it is possible to distinguish between events that involve a change to the layout of the cluster and events that involve changes to the configuration of the environment, introducing the terminology *scale-based contextualisation* and *role-based contextualisation* respectively. Scripts can be hooked into these events in order to provide the actions that must be taken to configure the environment. This facilitates an innovative mechanism that not only allows dynamic configuration of an environment when it is launched, but subsequent reconfiguration when the environment is scaled up or down, and when the providers of services within the cluster change.

The execution of scripts in order to modify configuration files with the contextually-relevant information limits the ability to create generalisable, immutable base images. The implication is that if the user is required to consider the lower layers of the container environment, rather than just the runtime environment, the barrier of entry is rapidly increased. The design of the VCC overcomes this by providing a dynamic DNS service which allows the contextualisation data to be accessed through DNS queries. For example, an NFS mount could be configured within the container as shown in Figure 3.3. The application is able to resolve a role name to the IP address of the container which is providing that role, rather than substituting the address into the command at run time or hard-coding a real container name. Therefore, this command can now be committed to an immutable image, whilst still being correctly contextualised with the deployment-specific information at run time. Fur-

```
mount -t nfs storageservice:/home /home
```

Figure 3.3: Example NFS mount using role name for dynamic DNS query

thermore, if the container providing the storage service changes, subsequent queries will transparently return the new container's IP address.

3.2.2 Many-Fabric Spanning

The ability for a job to potentially span multiple fabrics, either of the same or different type, is key to realising the enhanced benefits of virtualisation in terms of resource management, performance and collaboration. There are two key challenges that the design must consider in order to facilitate spanning: software environment compatibility and network connectivity.

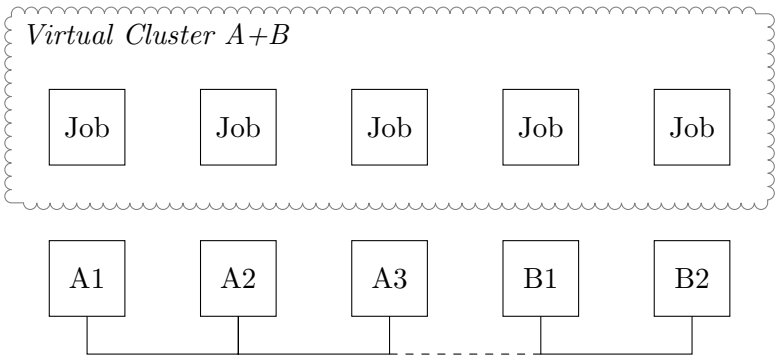
Whilst there are a relatively small number of viable processor architectures for use in contemporary HPC systems, there is a large amount of heterogeneity among software stacks. Even between cluster systems, it is common to find sufficient differences in the software environment to ensure that an application running on one system cannot run unmodified on another of the same type. However, given two mutually incompatible clusters, a third virtual cluster can be created using nodes borrowed from each physical cluster, allowing a coherent parallel distributed execution to take place on the combined resources. Both VM and container virtualisation methods provide adequate abstraction of the software environment in order to achieve this.

Regardless of the virtualisation method used, in order to deploy a spanned cluster, there must be connectivity between the nodes of the underlying clusters. This is not typically the case in HPC systems, and the DVC requires that an inter-

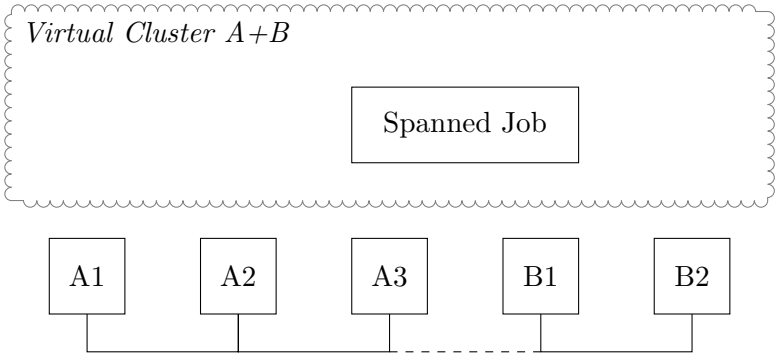
cluster interconnect is added to join the networks of the underlying clusters together (Emeneker & Stanzione, 2007). However, when such connectivity is available, the middleware layer of the software environment will likely need to be modified in order to support the unusual topology.

Therefore, the design of the VCC incorporates an SDN technology in order to provide an overlay network for spanned connectivity. From the container perspective, the overlay network appears to directly connect every container within a VCC on a single subnet, regardless of the underlying physical topology. This shifts the responsibility for implementing spanning from the application layer to the service layer, avoiding modifications to core components of the software stack and providing a virtual cluster interconnect that natively supports spanning. The SDN solution chosen for the VCC implementation is Weave Net, as it can be distributed within a container itself and supports a wide variety of engines, such as Docker (*Weave Net*, n.d.).

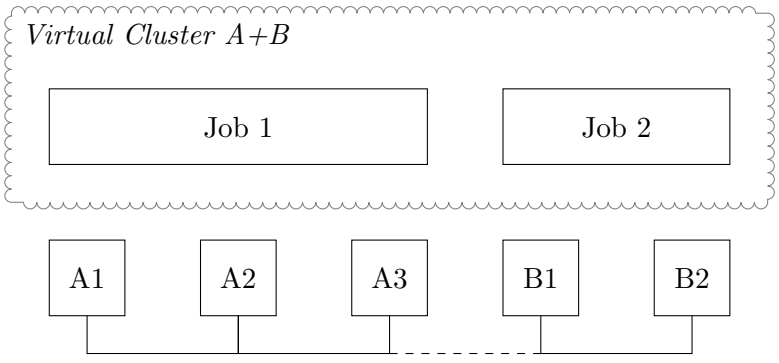
The discovery service in the VCC exposes relevant contextual information from the overlay network, such as IP address assignments and boundaries of the physical network. This allows the overlay network to be utilised in several ways: firstly, it can be used analogous to a management network within a cluster, where only the cluster state and resource management is maintained through the overlay network. This would be appropriate for high throughput, transactional jobs, where there is no communication between nodes, as shown in Figure 3.4a. Secondly, the overlay network can simply be used as the main inter-node interconnect for the virtual cluster, and this is the default use case as shown in Figure 3.4b - where two processes of a parallel application may communicate transparently over the inter-cluster interconnect. Finally, a combination of these modes may be used as shown in Figure 3.4c. In this case, the cluster environment is established using the overlay network, but contextual information from the discovery service is used to configure the re-



(a) No Communication Between Nodes



(b) Spanning Across Inter-Cluster Interconnect



(c) Avoiding Inter-Cluster Interconnect

Figure 3.4: Communication Models within the VCC

source manager such that jobs will not be scheduled to incur communication over the inter-cluster interconnect. Therefore, it allows entire job forwarding, in addition to spanning, to be implemented using the same networking services within the VCC.

3.2.3 Multi-node Parallel Execution

In a distributed parallel application, where processes related to the same program will reside on mulitple nodes, careful orchestration and synchronization of these processes is required in order for a successful execution. In an HPC system, this capability can be provided by a cluster resource manager. Runtime environment containers expect the host cluster to orchestrate the container as though it was the same as a regular executable. For example, a containerised MPI code can be executed through the host's *mpirun* command (*Using Host libraries: GPU drivers and OpenMPI BTLs*, 2017). However, a virtual environment with such a dependency cannot be executed on a host system that does not provide these interfaces. Therefore, in the VCC full stack environment, multi-node process orchestration must be performed as a function of the virtual cluster itself.

The service layer of the container provides the information typically required in order to run distributed parallel applications using a library such as MPI - automatically generating a list of nodes in the virtual cluster in both `/etc/hosts` and machine file compatible formats. In the simplest case, the middleware layer may only contain the appropriate MPI library. For a complex workflow, it is possible to incorporate a middleware layer which provides a resource manager, to allow individual jobs within the VCC to be scheduled across the virtual nodes. In this case, the resource manager will be configured automatically using the contextualisation methods in the service layer, orchestrating parallel execution of jobs as though it was a physical

cluster.

Public Key Infrastructure is a critical component of multi-node execution, as the means by which process launching on remote nodes can be performed securely and safely, with the correct authorisation. Even though the implementations presented by Emeneker et al. (2006), Keahey et al. (2007) and Yu and Huang (2015) demonstrate multi-node execution, they do not consider how to distribute public and private keys within the virtual cluster. A shared filesystem is commonly used for this purpose, but may not be natively available within all execution fabrics, or accessible when spanning multiple systems. Therefore, this is a central design consideration of the VCC; The discovery service is utilised in order to publish public keys to all nodes within a virtual cluster. Where no key exists, or a key was not supplied, one will be generated automatically so that process launching may still take place. This functionality follows the same flexibility as other VCC services, so that when the cluster is dynamically changed, the keys will be maintained accordingly. For example, adding a new public key to the discovery will result in it being automatically pushed to each container, and a new container added to an existing cluster will be automatically populated with the current set of keys.

3.2.4 User Defined Image

One of the motivations for integrating virtualisation into HPC systems is to enable user-defined customisations to the software environment, in order to resolve the often conflicting requirements of different user communities on a shared computational resource. For example, multiple versions of the same software or libraries may be required to coexist within the same environment. Incompatibilities can arise which can be difficult, or in some cases impossible, to resolve, as in the case of libraries providing critical OS functions such as the standard C library. Furthermore,

transporting the software to another environment once built poses an additional challenge. This places a significant burden on both users and administrators of HPC resources.

Virtual Machines inherently support customisation since they must be provided with a filesystem or hard disk image, but this does not immediately offer a satisfactory solution to the software environment flexibility problem. In a full stack approach, the user is additionally required to know how to perform associated system administration tasks, such as configuring the OS and underlying software stack. Container virtualisation addresses this problem by considering only the runtime environment - utilising the host system to orchestrate it as though it were any other process. This significantly reduces the required effort and knowledge such that the difficulty of customising a software environment within a container is comparable to without.

The VCC employs a full stack approach within container virtualisation. In order to retain an accessible method for users to customise the environment, the Docker workflow is used so that the image can be generated as a series of layers which are combined into a single view when the container is executed. This allows the lower layers of the software stack, containing the discovery, middleware and libraries that the host would otherwise provide, to be created seperately from the applications and run time environment layers. This approach means that the user can easily extend a VCC container by adding only the top layer containing their application, in a process almost identical to creating a runtime environment container.

The advantage of using the Docker workflow for generating container images is that it will be familiar to existing users and interoperable with applications already packaged in the format. Furthermore, it is accessible both in terms of the knowledge required to customise an image, and the size of the data which must be transferred:

image layers are defined in a plain text file which describes the commands used to generate each layer. A disadvantage is that the system administrator, or a person with appropriate knowledge, will still need to define the lower layers of the VCC before the user can customise it. However, unlike previous work where administrator intervention was required, as in Vallée et al. (2008), configuration in the VCC is performed automatically based on discovered contextual information, and containers may be created and destroyed by the end user. Therefore, it is practical to create a base VCC image which can be shared and reused without requiring further customisation, except that by the end user. This means that the continued administrative burden of supporting full stack VCC containers is no more than runtime environment containers, unlike VM-based virtual clusters.

3.2.5 Image Repository and Provenance

Traditional VM-based solutions suffer from the lack of easy methods to create and distribute images once defined by the user: a monolithic filesystem image that can exceed several Gigabytes is inconvenient to process and share. This problem is largely solved by containers, by allowing the image to be defined in a text file and built as a series of layers which are combined at run time. This allows common layers between containers to be shared, reducing the filesystem footprint, and allows new layers to be extended on top of an existing image without rebuilding or distributing an entire new image. Online repositories that allow users to publish and share images are available for Docker and Singularity (Boettiger, 2015; Sochat, 2017). Furthermore, Docker allows images to be cryptographically signed to ensure that the integrity and publisher of the data when shared can be verified (*Content trust in Docker*, 2018).

In order to utilize these existing capabilities, the containers built using the VCC

framework follow the Docker workflow. This decision is based on the compatibility offered by other container engines which are suitable for HPC deployment, which support executing container images in the Docker format (Jacobsen & Canon, 2015; G. Kurtzer, 2016; Priedhorsky & Randles, 2017). This provides an accessible method for customising and sharing VCC containers using the online repository, in a format which has support among a variety of container engines. When choosing to download and run an image, or when incorporating layers from other images in your own, the ability to sign Docker images provides a method in order to establish the provenance of each component in the container. This is essential when transferring images across an untrusted medium, such as the internet, to ensure that malicious changes cannot be performed in transit.

3.2.6 Standalone Deployment

A key design goal of the VCC is to create containers with a high degree of portability, that can be orchestrated without depending on a specific execution fabric, such as a cluster, grid or cloud. This promotes reuse of virtual environments, allowing users who otherwise would not have access or hold the knowledge to configure the required environment, to deploy it on their own compute resource. The usefulness of the container in this regard is greatly diminished if the user does not possess a specific type of system that container depends on - the knowledge and expertise required in order to configure the underlying system outweighs the convenience of running the application in a container.

The full stack approach within the VCC does not assume that any functionality is provided by the underlying system, except support for a container execution engine such as Docker. Where local resources are provided, such as optimised parallel libraries, this approach does not prevent them from being utilised. However, where

they are not available, the container includes equivalent functionality. This strategy satisfies both scenarios where it may be beneficial to borrow such resources from the host to improve performance, but detrimental to portability and reusability when it comes to sharing the virtual environment with others.

In a VM-based approach, the full stack must be encapsulated in the virtual environment in order to function, however, existing configuration methods which make it more accessible to the average user introduce dependencies on the underlying system, such as to provide deployment-specific metadata (Keahey & Freeman, 2008). The VCC avoids this problem by using a contextualisation mechanism that relies on discovered data rather than an external source. The other services typically required for execution, such as domain name resolution, resource management and process launching, can also be provided by the VCC following the same principle. This means that a virtual environment designed for a particular execution fabric can be executed on a different fabric, and still work as intended without an additional burden on the user.

3.3 Implementation

This section details the implementation of the VCC services. The services are implemented using the NodeJS runtime, chosen due to the asynchronous nature of the JavaScript language. The code artefact is published in Higgins, Holmes, and Venters (2017b) and archived online in Higgins, Holmes, and Venters (2017c). Where appropriate, references to individual files within the source code are indicated in footnotes.

```
/cluster/
/cluster/test
/cluster/test/hosts
/cluster/test/hosts/host1 => 10.0.0.1
/cluster/test/hosts/host2 => 10.0.0.2
/cluster/test/services/headnode => host1
/cluster/test/services/workernode => host2
```

Figure 3.5: Typical discovery data stored in the key-value store

3.3.1 Discovery

The discovery mechanism is implemented using a key-value store. It allows individual container instances to share state, such as network identification, in order to facilitate the dynamic and autonomous configuration of the virtual cluster. The structure created in the key-value store during a typical run is detailed in Figure 3.5. Each instance of a VCC is given a unique cluster name, used to namespace the discovery data, in order to allow the same key-value store to be used to deploy multiple clusters if required.

Any key set by a container within the key-value store is subject to a Time-To-Live (TTL), and the container is responsible for refreshing the key periodically to ensure that it does not expire. This ensures that, should the container crash, the keys will be removed after a short timeout and the cluster can be reconfigured appropriately.

In order to implement this functionality, the key-value store must provide primitive functions for setting, getting and notifying when keys are changed. Etcd (*Using etcd*, n.d.) has been chosen as a lightweight database that meets these requirements and can easily be distributed as part of the container itself. However, a simple

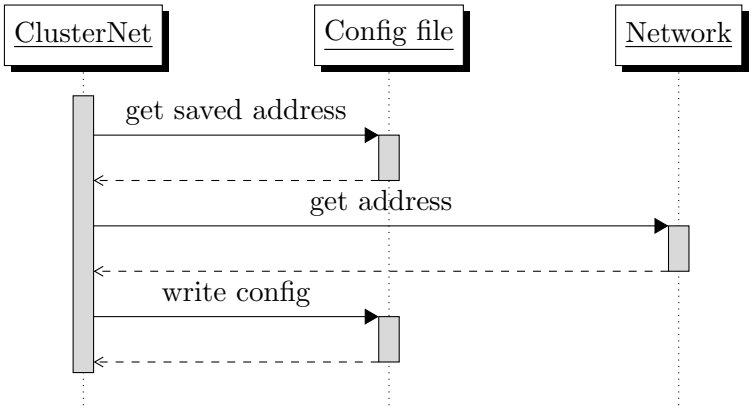


Figure 3.6: Sequence diagram of the ClusterNet service

interface[2] has been created to abstract the underlying key-value store, exposing `get`, `set` and `watch` functions for other VCC services to use. In addition, it provides a convenience function to register a key with a TTL, automatically refreshing the key before the TTL time has passed. Therefore, this allows both the implementation of VCC services to be simplified, and for the discovery mechanism to be easily ported to other storage backends if desired.

3.3.2 Network Identification

The ClusterNet[3] service performs network identification discovery within the VCC. It is responsible for populating the IP address and hostname of the current container instance. This service does not persist within the container as a daemon, rather, it writes the data to a local configuration file and exits. The use of a local configuration file ensures that the discovery key-value store does not need to be queried in order to get information about the local container.

The sequence diagram is shown in Figure 3.6. Initially, a function is called which checks to see if there is a saved IP address. This allows the user to bypass detection

[2] `kvstore.js` (Higgins et al., 2017b)
[3] `clusternet.js` (Higgins et al., 2017b)

and provide the network identification manually, such as if the physical host has a multi-homed network configuration and a specific network interface must be used. Otherwise, a function is called which determines the IP address of the container. Once the IP address is determined, it is written along with the hostname to the configuration file.

In order to determine the IP address of the container, the appropriate system calls are used to enumerate the network devices and their assigned addresses. The devices are checked in a specific order of preference, and the first acceptable device found will be selected:

1. Virtual network interfaces assigned to the container, such as when running in an overlay network
2. Primary network interface as defined by the OS
3. Any available network interface

If no suitable device is found, network identification cannot take place, and thus the VCC is halted.

3.3.3 DNS

The ClusterDNS[4] service provides dynamic name resolution for the VCC. It must allow domain name resolution of all containers within a particular virtual cluster, in addition to exposing names which will be resolved to the contextually-relevant value at run time.

The sequence diagram of the implementation is shown in Figure 3.7. At startup, the service will read the network identification stored in the configuration file from

[4] `clusterdns.js` (Higgins et al., 2017b)

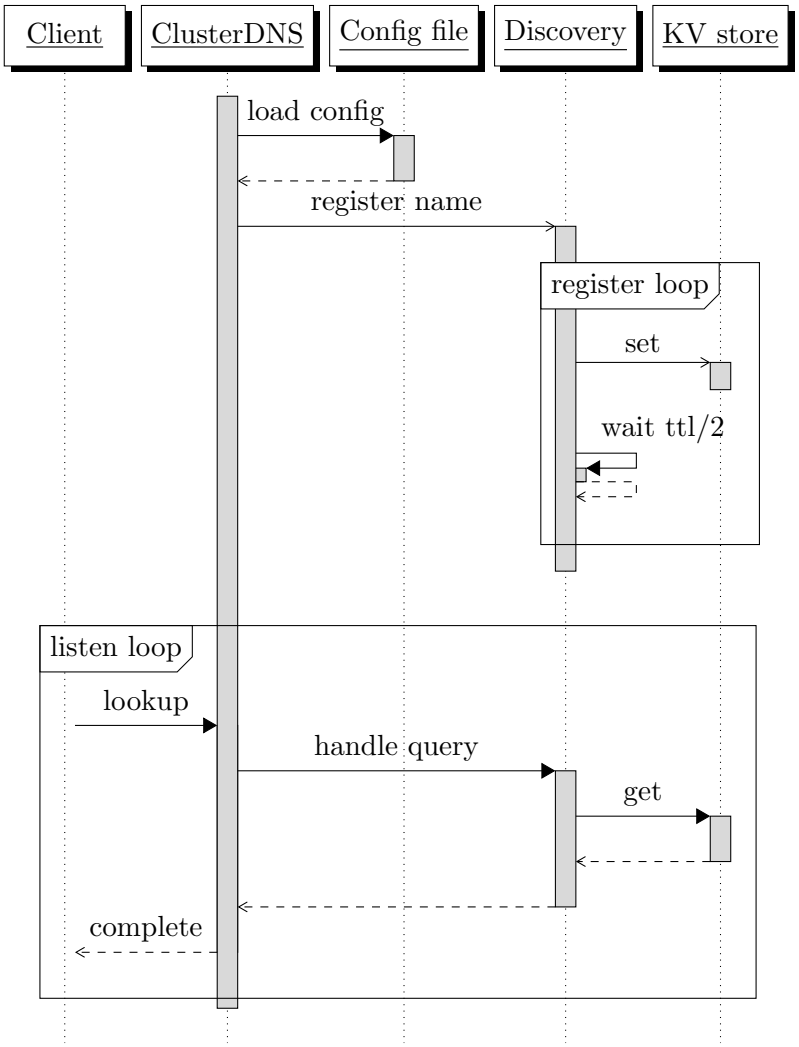


Figure 3.7: Sequence diagram of the ClusterDNS service

the ClusterNet service, and register those in the discovery key-value store. The registration is a loop which sets a key in the format

```
/cluster/[cluster name]/hosts/[host name] => [ip]
```

with a TTL time by default of 60 seconds. After half of the TTL time has passed, the next iteration of the loop begins and the key is set again in order to reset the TTL to 60 seconds. Refreshing each key early is essential in order to ensure that virtual nodes are not unnecessarily removed from the cluster and immediately added again when the TTL reaches zero. Additionally, it allows for a period of grace to deal with transient problems that may delay the refreshing process but not interrupt execution, such as high CPU or network load transients. This functionality is included in the key-value store interface[5] so that its implementation can be shared by each VCC service.

The main loop is entered which listens on UDP port 53 for DNS queries from client applications. Each DNS query contains a question, which includes the name to be resolved. The function invoked in order to handle each lookup query searches the following locations:

- Get the answer for the name from local cache
- Get the key `/cluster/[cluster]/hosts/[name]` from discovery
- Get the key `/cluster/[cluster]/services/[name]` from discovery

These locations are not searched in a particular order, and whichever contains the first valid answer will be returned to the client. By searching the `services` directory, it allows a query to contain the name of a service rather than a specific container, which will be correctly resolved to the container providing that service. Thus, the

[5] `kvstore.js` (Higgins et al., 2017b)

runtime environment can be easily contextualised without substituting values into configuration files, using a scheme as shown in Section 3.2.1.

If the name in question contains the `vnode` prefix, it will be stripped and the query handled as normal. This is required as a hashing algorithm is often used to generate unique references to containers, which can also be used as the hostname. However, some applications do not correctly recognise hostnames that begin with a numeric character, despite it being valid according to the RFC 1123 (Braden, 1989). Therefore, this provides compatibility with such software by allowing a name to be looked up using the alias `vnode_name`.

3.3.4 PKI

One of the design decisions of the VCC is that it should not depend on a shared filesystem to be available between the host systems. Therefore, it cannot be used to share SSH keys between every node in order to establish a trusted mechanism for process launching in the runtime environment layer. To solve this problem, the ClusterKeys[6] service within the VCC allows public keys to be published in the discovery service and automatically distributed among the virtual cluster.

The sequence diagram of the implementation is shown in Figure 3.8. Firstly, the service will check if the invoking user has already generated an SSH key pair, creating one if it does not exist. Secondly, it will register the public key in the discovery service using the TTL/refresh method (abbreviated in Figure 3.8). Finally, the service enters the main loop. The main loop enumerates all public keys within the discovery service and writes an `authorized_keys` file for the invoking user, which contains the public keys from all other containers within the virtual cluster. Iteration of the loop is then paused until a change is detected on the discovery service, such

[6] `clusterkeys.js` (Higgins et al., 2017b)

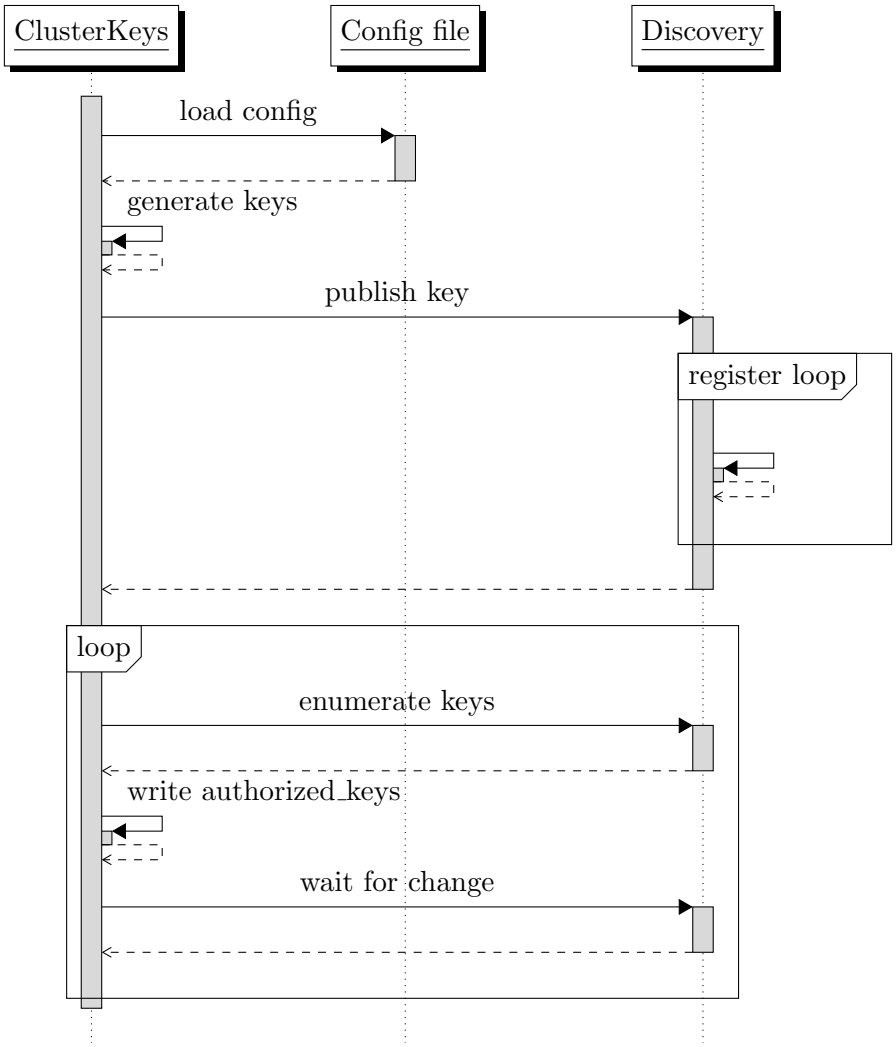


Figure 3.8: Sequence diagram of the ClusterKeys service

as a new public key is added.

Two strategies to detect changes are provided: *watch* or *poll*. The watch strategy relies on the key-value store to provide a function which blocks until a change is detected. The poll strategy polls the key-value store at a specified interval and compares the new result to the previous result to detect changes. If multiple changes are detected over several polling intervals, a change is only considered once the next poll returns to 0, or the settle limit is reached. This improves the efficiency of the configuration process, allowing changes to be batched together and optimising the number of times the process is executed. In large clusters, where there is always a key being changed at any point in time due to the refreshing required in order to avoid TTL expiry, a constant stream of changes can be generated. In this case, the poll strategy must be used and therefore is the default.

3.3.5 Dynamic Configuration

The design of the VCC facilitates dynamic configuration of services within the virtual cluster using a role-based contextualisation method. The implementation is split into two services: Wait4Deps[7] and RegisterService[8]. Each container is designated a role and dependency between roles is defined in a `dependencies.yml` text file. Figure 3.9 shows an example for a cluster that has a head node role and a worker node role. *Providers* define which services must be running within the container for the role to be considered ready. *Depends* specifies which roles must be ready before the current one can be started. Therefore, this describes that the `headnode` role is fulfilled once a container is running a `pbs_server` service, and that the containers which are designated as `workernode` role can only be configured once the head node is ready. In order to perform the appropriate configuration, service hook scripts

[7] `wait4deps.js` (Higgins et al., 2017b)
[8] `registerservice.js` (Higgins et al., 2017b)

```
---
headnode:
    providers:
        - pbs_server
workernode:
    depends:
        - headnode
```

Figure 3.9: An example dependencies.yml file

are executed based on which role has changed state. Following this example, the worker node container will execute the `headnode` service hook in order to configure its connection with the head node container. The interactions between the services which implement this functionality are shown in Figure 3.10.

Firstly, Wait4Deps waits until the services within the remote containers that provide the dependent roles are ready, by polling the following keys in the discovery service:

```
/cluster/[cluster]/services/[role name]
```

It receives a response of either 'False', or the name of the container which is providing the role. Once all role dependencies are ready, the service hooks are executed, where the hook script has the same name as the role name, and the name of the container providing the role is supplied as an argument.

```
/etc/vcc/service-hooks.d/[role name].sh
```

The purpose of the hook script is to perform the required adaptations to the configuration, based on contextual-specific knowledge about which container is providing a specific role. The implementation of the hook script is dependent on the software stack being encapsulated in the container, discussed in Chapter 4.

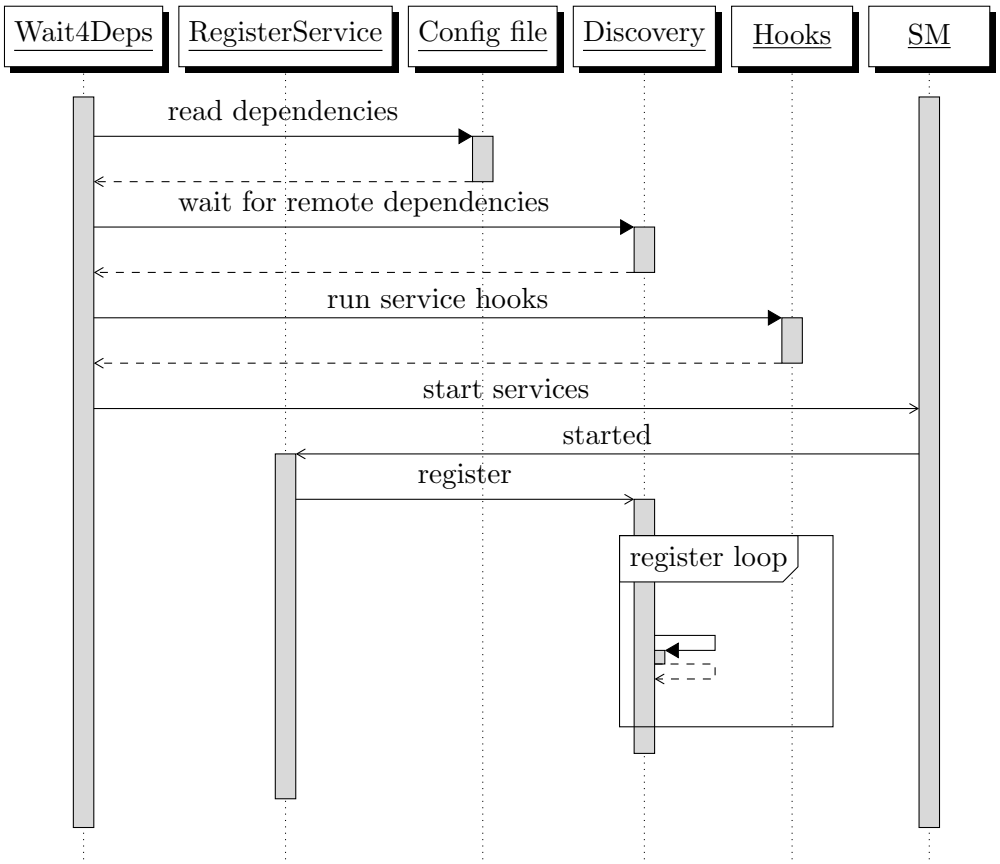


Figure 3.10: Sequence diagram of the dependency related services

Secondly, the service manager can now start the local services which must be running within the container. To complete the example from Figure 3.9, once the worker node container has executed the hook for the `headnode` role, the local `pbs_mom` service can be started using its fully contextualised configuration.

Finally, the container role, as defined in the `dependencies.yml` file, must be registered within the discovery key-value store to advertise that the local services have been started and that the role is now fulfilled. This is implemented by the RegisterService service. The container writes its host name to the key

```
/cluster/[cluster]/services/[role name] => [container]
```

so that other containers can now associate it with the specific role.

3.3.6 Dynamic Scaling

In the previous section, the implementation of role-based contextualisation within the VCC is described. In this section, the implementation of scale-based contextualisation, in order to facilitate dynamically adding or removing nodes from a virtual cluster, is presented. The configuration actions performed in response to these events are usually only desired on designated nodes. For example, when a new worker node is added to the cluster, the list of nodes within the head node is updated, however, no equivalent action needs to be performed on every other node. The ClusterWatcher[9] service is responsible for maintaining this configuration and responding to changes when the cluster is dynamically resized.

Figure 3.11 details the sequence diagram for this service. An instance of the service will be running in every container that is part of a VCC. Firstly, it enumerates all the nodes in the cluster by querying the discovery service. It then writes out a hosts

[9] `clusterwatcher.js` (Higgins et al., 2017b)

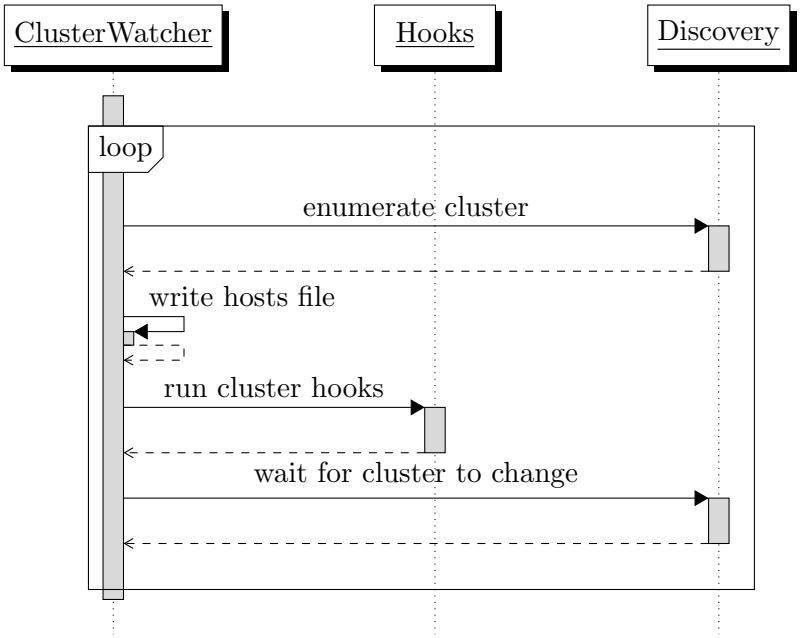


Figure 3.11: Sequence diagram of the ClusterWatcher service

file in a `/etc/hosts` compatible format. Typically, the directory where the hosts file is saved is `/run` in order to not conflict with the system `/etc/hosts`. However, when executing as an unprivileged user, this directory can be customised by setting the environment variable `INIT_RUN_DIR`. This allows the file to be scoped to a particular user, such as `$HOME/.local/run`.

Secondly, the cluster hook scripts are executed. The hook scripts are stored in the directory `/etc/vcc/cluster-hooks.d`. These scripts perform the adaptations to the configuration which are required when the boundary of the cluster has changed. The host file can be referred to in the hook scripts in order to access data about the entire virtual cluster without additional requests to the discovery service. The execution of the hook scripts occurs in parallel, in order to speed up the configuration where many scripts are present.

Finally, once the execution of all cluster hook scripts is complete, the loop is paused until a change is detected on the discovery service. The same detection strategies

as implemented in the ClusterKeys services are used: *watch* or *poll*. The default strategy is polling, where the nodes in the cluster are enumerated at a set interval and compared to the previous list. When using this method of change detection, the time to react to changes within the cluster, or the *configuration latency*, is increased up to the polling interval in the worst case, depending on when the change occurred during the polling cycle. However, it allows more efficient processing in one iteration of all changes that occur during the interval, increasing performance of the service for large cluster sizes.

3.3.7 Service Management

The VCC services are implemented as separate daemons and must be started within the container. Some services are dependent on other services, so the startup order is critical for successful initialisation of the cluster. This differs from other container solutions which are typically design to run the application as the only process within the container. Inside a VCC container, a service manager is required in order to supervise the execution of the VCC services in order to realise the full stack virtualisation approach.

The startup of the services is sequential and therefore straightforward to implement. The correct dependency order is detailed in Figure 3.12. After the Wait4Deps service, but before the RegisterService, the non-VCC related services within the container must be started. For example, if the container will provide a PBS or NFS server, those services should be started at this point. The last service to be executed is the ClusterWatcher, which will maintain the cluster state throughout its lifetime. The ClusterDNS, RegisterService and ClusterWatcher services are daemons which will persist for the duration of the container's execution.

A simple NodeJS-based service manager is provided in the VCC framework, however,

it is usually more convenient to use the systemd service manager within the container as the service files for non-VCC services will be available in this format. Systemd performs parallelisation of service startup, so the VCC framework provides a set of unit files in order to control the order as required.

An example unit file for the ClusterDNS service is shown in Figure 3.13. Each unit defines the `After=` argument in order to ensure it is only executed once the dependencies are satisfied. The VCC framework also installs a VCC services target, as shown in Figure 3.14. This defines the point during service startup where non-VCC services should be started by systemd.

Furthermore, when using systemd as the service manager, each service can send a

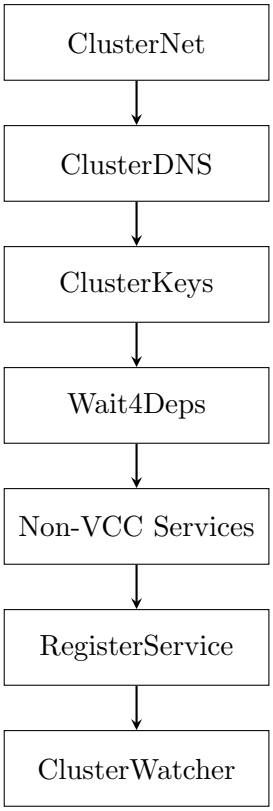


Figure 3.12: Service Dependency Ordering

```
[Unit]
Description=VCC DNS resolver
PartOf=vcc.service
After=vcc-net.service

[Service]
Type=notify
NotifyAccess=all
ExecStart=/usr/local/bin/node /vcc/clusterdns.js
Restart=on-failure

[Install]
RequiredBy=vcc.service
```

Figure 3.13: Unit File for ClusterDNS Service under systemd

```
[Unit]
Description=VCC launch service units
Requires=vcc-wait4deps.service
After=vcc-wait4deps.service
Before=vcc-register.service
```

Figure 3.14: VCC Service Target under systemd

notification when it is ready. This allows the service to perform a certain action, such as writing a configuration file or running an iteration of the main loop, before it is considered ready and the system starts the next service. The capability is implemented in the VCC utility library [10], which is included by each service. It writes the string `READY=1` to the socket specified in the environment variable `NOTIFY_SOCKET`. This environment variable is populated automatically by systemd.

3.4 Summary

In this chapter, the design and implementation of the VCC framework was presented, outlining the principles on which the architecture is based, and describing the low level implementation details of each service within the framework.

The design introduces key innovations in order to resolve limitations present in existing work: firstly, a novel contextualisation methodology allows the container middleware layer to be automatically configured at run time, using deployment-specific knoweldge obtained through a discovery process. This knowledge is shared with other containers within a VCC, to facilitate both dynamic scaling of the virtual cluster and role-base dynamic reconfiguration throughout the lifetime of the deployment. Contextually-relevant information is exposed via DNS, enabling an immutable image to be defined whilst still allowing the required adaptations to the configuration to be performed. Therefore, the VCC facilitates encapsulation of a full stack software environment within containers, whilst retaining the accessibility, ease of use and customisation workflow of runtime environment containers.

Secondly, a Software Defined Network technology is utilised in order to provide the inter-node interconnect when a VCC is deployed across multiple nodes. This

[10] `vccutil.js` (Higgins et al., 2017b)

removes the requirement for connections to be made between private networks when spanning a VCC across several physical clusters - as long as there is a common route between the clusters, an overlay network can be created to facilitate direct communication between the containers. In contrast to previous work, this moves the responsibility for configuring inter-cluster connectivity from the application layer to the service layer, therefore, no modifications to existing software is necessary in order to take advantage of spanning. This allows the full potential of containers to be exploited in the context of an HPC resource, bringing the same dynamicity and flexibility as VM-based approaches, without the associated performance and management overheads.

Finally, the implementation of the VCC services is presented. The VCC services provide the foundation layer of each container, exposing the capabilities for discovery and contextualisation. The services follow a general pattern where contextually-relevant information is discovered about the container, published to the key-value store, and then the main loop is entered in order to monitor the key-value store and react accordingly to changes within the cluster. A key design aspect of each service is portability - the VCC does not rely on external services to provide any knowledge required for deployment and dynamic configuration. This allows a standalone VCC to be deployed within a variety of fabrics and execute as intended, without a significant burden in order to configure the underlying system before being able to execute the container.

Chapter 4

Building a Container Cluster

In Chapter 3, the design and implementation of the VCC was presented. The innovations in this design allow a wide range of flexibility in terms of deployment contexts, configurability and level of skill, using a unified cluster virtualisation model. In order to achieve this, the VCC has been designed as a framework providing services and functionality that can be used to build containers for a specific use case or application. This chapter details the process of building a container that encapsulates a parallel application executing on a batch scheduling cluster - a typical execution paradigm in the HPC context. The workflow is detailed from definition of the base image to subsequent extension including the runtime application layer and experimental data.

The processes described in this chapter are based on the technical documentation of the VCC framework, which is partially published along with the code in Higgins et al. (2017b). This reference implementation of a VCC container is used in Chapters 5, 6 and 7 in order to evaluate the system.

4.1 Base Image

This section describes building a base container image that includes the VCC service layer and the middleware layer. You do not have to separate the build process into two parts, but it is convenient to do so because they require distinct expertise - the base image requires system administration knowledge in order to construct, whilst the runtime layer can easily be edited by the average user. Firstly, the VCC service layer will be installed into the container image. Secondly, the middleware layer can be installed. The installation of the middleware follows the same patterns as it would on a native system, i.e. outside of a container. Finally, the VCC configuration is created, describing the roles provided by the middleware, dependencies and hook scripts in order to facilitate dynamic configuration. This allows the middleware layer to be contextualised automatically by the framework at runtime.

4.1.1 VCC Installation

The installation of the VCC service layer within the container is detailed in Figure 4.1. It is necessary to decide which Operating System will be the foundation of the container image, specified on the first line using the `FROM` directive. The commands that need to be executed in order to build the image are specified using the `RUN` directive. In this case, the VCC framework is cloned from the Git repository, however, it is also possible to download and install from a tarball. The advantage of using the repository is that a specific point in time can be checked out, without storing the entire source tree for that version in the container source. This ensures that on subsequent builds of the container, the versions of the software are guaranteed to be the same - an important consideration if the container will be used to reproduce an environment in the future.

```
FROM centos:7

# install vccjs
WORKDIR /
RUN git clone https://github.com/hpchud/vccjs.git \
        && cd vccjs \
        && git checkout -q eb26268
WORKDIR /vccjs
RUN npm install

# install systemd services
RUN cp -r /vccjs/systemd/*.service /etc/systemd/system/
RUN cp -r /vccjs/systemd/*.target /etc/systemd/system/
RUN cd /etc/systemd/system && systemctl enable vcc*

# launch script
WORKDIR /
ENTRYPOINT ["/vccjs/launch.sh"]
```

Figure 4.1: Dockerfile excerpt to install VCC service layer

As detailed in Chapter 3, the VCC services must be launched in the correct order when the container is started. For this purpose, the systemd service manager will be used as it is already provided by default in CentOS 7 (the base OS). The units provided by the framework are copied into the correct location within the container filesystem.

The service layer must also provide an entrypoint which dictates the startup behaviour of the container. In this case, the launch script[1] provided by the VCC framework is used. The purpose of the script is to read environment variables provided by the container runtime, such as the address of the discovery service, and apply them before the service manager is started. The entrypoint script also checks the configuration variables to ensure that launching the container will be successful. For advanced use cases an alternative entrypoint script can be defined in order to override the startup behaviour, however, for the majority of use cases this will not

[1] `launch.js` (Higgins et al., 2017b)

be necessary.

Once the Dockerfile is defined, the image is built using a command such as

```
docker build -t hpchud/vcc-base-centos:7 .
```

in order to execute the directives in the Dockerfile and generate the image layers. The image can optionally be pushed to an online repository, and the source code can be stored in version control. Subsequent images can be created to extend this base image with additional layers by referring to the tag `hpchud/vcc-base-centos`.

4.1.2 Middleware Installation

The installation of the middleware layer is divided into two parts: firstly, the required packages and libraries are installed using the same process as would be followed on a real system - typically this involves procuring the source code either from a repository or tarball, and executing `make && make install` commands. Secondly, hook files that describe how the middleware components will be configured with contextually-relevant information at runtime need to be created.

These two processes are detailed in Figure 4.2 for installing the Torque/PBS resource manager, MPICH and associated VCC hooks. This middleware layer represents a traditional HPC execution paradigm where the resource manager is responsible for job queuing and launching of distributed parallel code using the MPI library for communication.

4.1.2.1 Roles and Dependencies

The `dependencies.yml` file defines the roles which are provided by the container. A single container may provide several roles, with each role resulting in a different

```
FROM hpchud/vcc-base-centos:7

RUN cd /tmp \
    && git clone https://github.com/adaptivecomputing/torque.git \
        -b 5.1.1.2 torque-src \
        && cd torque-src \
        && ./autogen.sh \
        && ./configure --prefix=/usr --disable-posixmemlock \
            --disable-cpuset \
        && make \
        && make install \
        && ldconfig \
        && cd .. \
        && cp torque-src/torque.setup . \
    && rm -r torque-src

RUN cd /tmp \
        && curl -O https://www.mirrorservice.org/sites/
distfiles.macports.org/mpich/mpich-3.2.tar.gz \
        && tar xf mpich-*.tar.gz \
        && cd mpich-* \
        && ./configure \
        && make \
        && make install \
        && cd / \
    && rm -rf /tmp/mpich-*

COPY dependencies.yml /etc/vcc/dependencies.yml

ADD hooks/pbsnodes.sh /etc/vcc/cluster-hooks.d/pbsnodes.sh
ADD hooks/pdsh.sh /etc/vcc/cluster-hooks.d/pdsh.sh
RUN chmod +x /etc/vcc/cluster-hooks.d/*

ADD hooks/headnode.sh /etc/vcc/service-hooks.d/headnode.sh
RUN chmod +x /etc/vcc/service-hooks.d/*
```

Figure 4.2: Dockerfile excerpt to install Middleware layer

set of services started within the container in order to fulfil that role. It also defines dependent roles which must be fulfilled by other containers before the current one may start, as described in Section 3.3.5.

For the implementation of the virtual cluster, `headnode` and `workernode` roles are defined in the image, using the example dependency file presented previously in Figure 3.9. Compared to requiring the user to define two distinct images for the cluster, the multi-role capability of a VCC container image reduces the amount of data to be transferred when sharing and deploying the images, in addition to ensuring that updates to the environment are consistent and do not have to be repeated unnecessarily. The head node role requires the `pbs_server` service to be started in the container, after which it will be advertised to other containers as being ready, through the discovery service. The worker node role defines that the `pbs_mom` service should be started within the container, but that it depends on the head node role. Thus, the services for the worker node role will not be started until a container within the VCC has published to the discovery service that it is providing the head node role.

This flexibility of this scheme allows it to be extended to configure different roles of nodes within the cluster, such as login nodes and post-processing nodes, or different types of clusters entirely.

4.1.2.2 Cluster Hooks

Cluster hooks are executed when containers are added or removed from a running instance of a virtual cluster. A cluster hook can be used to configure a service provider using contextual information about the entire cluster. In this cluster being built throughout this chapter, Torque/PBS is used as the resource management middleware. It requires a file to be created which describes the list of nodes within the

```
#!/bin/bash

echo -n > /var/spool/torque/server_priv/nodes

INIT_RUN_DIR="${INIT_RUN_DIR:-/run}"

cat $INIT_RUN_DIR/hosts.vcc | while read line; do
        hn="`echo $line | awk '{print $2}'`"
        if [ "$hn" != "`hostname`" ]; then
                echo "vnode_$hn" >> /var/spool/torque/server_priv/nodes
        fi
done

qterm -t quick
```

Figure 4.3: Cluster hook script to generate PBS node file

cluster onto which batch jobs can be scheduled. This file must be created on the head node, and updated each time the layout of the cluster changes. Therefore, a cluster hook is used to generate the list of nodes on the head node, shown in Figure 4.3. The hook script is placed in the configuration directory `/etc/vcc/cluster-hooks.d` within the container. It will be executed by the VCC service layer whenever the layout of the virtual cluster changes.

The ClusterWatcher service, detailed in section 3.3.6, generates an `/etc/hosts`-compatible file describing the cluster layout, which is available for reference in order to avoid multiple calls to the discovery service directly from the script. This file is parsed by the cluster hook script in order to transform each line into the correct format for the Torque/PBS node list. Each host is prefixed by the `vnode_` prefix in order to ensure the hostname begins with a non-numeric character. After the script has been executed, the Torque/PBS service is restarted in order to load the new node list.

4.1.2.3 Service Hooks

Service hooks are executed when the provider of a service is changed. In order to integrate Torque with the VCC service layer, a hook script is required for the `headnode` role. The hook must perform the tasks detailed in the Torque Administrator Guide (*Configuring TORQUE on compute nodes*, 2012):

1. Configure the server name in `/var/spool/torque/server_name`
2. Configure the server name of the MOM by setting the `$pbsserver` variable in `/var/spool/torque/mom_priv/config`
3. Configure the name of the MOM with the `vnode_` prefix by setting the `$mom_host` variable in `/var/spool/torque/mom_priv/config`

This hook script will be executed at first launch by every container within a VCC which depends on the `headnode` role. If the container providing the `headnode` role changes throughout the lifetime of a VCC deployment, the hook will be executed again to reconfigure the container with the new context.

The hook script is shown in Figure 4.4. The DNS name of the container providing the `headnode` role, resolvable through the ClusterDNS service, is provided to the hook script as the first argument. The script is added to the container in the VCC configuration directory under `/etc/vcc/service-hooks.d/headnode.sh`.

4.2 Runtime Environment

The container built in this chapter will be used to conduct the benchmark and case study activities, therefore, it must provide the tools outlined in the methodology as part of the run time environment. As the run time environment is extended from the VCC service and middleware layers, which is in turn extended from the CentOS

```
#!/bin/bash

# service hook for torque server

echo $1 > /var/spool/torque/server_name

# configure the mom if we have it

echo "\$pbsserver $1" > /var/spool/torque/mom_priv/config
echo "\$logevent 255" >> /var/spool/torque/mom_priv/config
echo "\$mom_host vnode_`hostname`" >> /var/spool/torque/mom_priv/config
```

Figure 4.4: Service hook script for the headnode role

base image, the installation process is in no way distinguished from that of a real CentOS system, except that the commands are placed in the Dockerfile and executed at container build time.

OpenFOAM is installed using the recommended approach, as detailed in Figure 4.5 (*OpenFOAM Build Guide*, n.d.). Version v1612+ of the source tarball is downloaded from the upstream repository as part of the Dockerfile. This ensures good reproducibility if the container is rebuilt in the future to ensure that versions of the software stack remain consistent.

HPC Challenge version 1.5.0b is built against the Intel Math Kernel Library (MKL) version 2017.1.013. The MKL is used to provide optimised libraries for Basic Linear Alegebra Subroutines (BLAS) and Fast Fourier Transform (FFT) functions. As this toolchain has limited distribution rights, it is not permissable to include the full development environment and compile the executable as part of the container build process. However, the Makefile - detailed in Appendix - is configured to link against the MPICH library that is installed as part of the middleware image layers detailed in Section 4.1.2. Therefore, the binary can be produced separately and copied into the container image.

```
FROM vcc-torque-mpi:latest

RUN mkdir /opt/OpenFOAM
RUN cd /tmp/ \
        && curl -O -L https://sourceforge.net/projects/openfoamplus/files
/v1612+/OpenFOAM-v1612+.tgz \
        && curl -O -L https://sourceforge.net/projects/openfoamplus/files
/v1612+/ThirdParty-v1612+.tgz \
        && tar -xf /tmp/OpenFOAM-*.tgz -C /opt/OpenFOAM \
        && tar -xf /tmp/ThirdParty-*.tgz -C /opt/OpenFOAM \
        && rm -rf /tmp/*.tgz

RUN source /opt/OpenFOAM/OpenFOAM-*/etc/bashrc \
        && export USER=root \
        && foamSystemCheck \
        && cd /opt/OpenFOAM/OpenFOAM-*/ \
        && ./Allwmake
```

Figure 4.5: Dockerfile excerpt for installing OpenFOAM run time environment

Both OpenFOAM and HPC Challenge utilize the MPI library to facilitate inter-node communication. At the top layer of the VCC, it is only necessary to install the software and any experimental data within the image; the middleware layer will perform the required configuration of the environment and ensure that contextual information, such as a host file for execution, is available to the MPI applications at run time.

Alternatively, the Dockerfile which is used to build the run time environment layers can invoke a package manager such as EasyBuild (Geimer, Hoste, & McLay, 2014) in order to manage the installation of the required tools and packages within the virtual cluster. By allowing modular installation of packages in standard filesystem directories, this approach is more suitable for creating a virtual cluster image that is intended to be deployed as a multi-user system, rather than a transient cluster existing only for the duration of a single job.

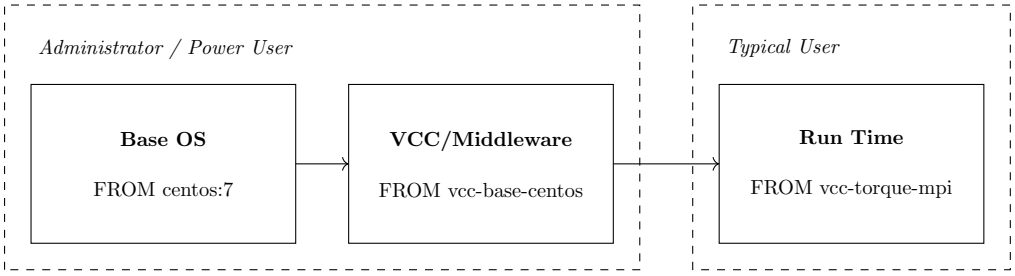


Figure 4.6: Summary of VCC Image Build Process

4.3 Summary

The process of building a containerized virtual cluster using the VCC framework was outlined in this chapter. Firstly, the base image is defined using the OS which will be the foundation for the container, such as CentOS. This is extended to generate another image which adds the middleware and configuration for a virtual Torque/PBS cluster to this foundation. The configuration defines how the environment will be contextualised at run time using the hooks provided by the VCC, such as adding worker nodes to the node list on the PBS server. The run time environment is the final layer of the VCC container. It includes the applications, libraries and supporting software that will be executed within the virtual cluster.

In the design of the solution detailed in Chapter 3, it is anticipated that the use of a Software Defined Network as the interconnect for the virtual cluster will have an impact on application performance, in terms of increased communication latency and increased job execution time. The container built in this chapter details the installation of OpenFOAM and HPC Challenge into the run time environment, which will be used to conduct a performance analysis of the VCC cluster through the benchmarking and case study activities.

An outline of the chain of images that are created and extended in order to produce

the full stack VCC image is presented in Figure 4.6. This methodology provides clean separation between the base OS, VCC-specific service and middleware configuration, and the run time environment. This promotes a high level of reusability, for example, by allowing many images providing different run time environments to be extended from the same `vcc-torque-mpi` image. Furthermore, the responsibility for creating each layer can be divided between administrators or power users, and the typical end user of a HPC system. Where a user is already provided with the base VCC images, they only need to add the run time environment layers on top. Therefore, it does not significantly increase the barrier of entry to container virtualisation.

Chapter 5

Performance Benchmarking

This chapter details the performance evaluation of the virtual cluster model proposed and implemented in Chapters 3 and 4. Based on previous work that considers the performance of containers within HPC, it is expected that the virtualised execution performance will be equivalent to the non-virtualised performance in the context of a single cluster fabric. Benchmarking this scenario is not likely to contribute any insight that is not already apparent through existing run time environment virtualisation techniques.

However, the VCC full stack approach introduces the concept that the boundary of the virtual cluster is defined by the network connectivity, rather than the software environment - mirroring one of the main characteristics of an HPC system. The novel application of a Software Defined Network (SDN) allows this connectivity, and thus a virtual cluster, to be established across heterogeneous and disparate fabrics in order to facilitate transparent job spanning and forwarding capabilities. Nonetheless, SDN technologies are known to introduce a performance penalty of their own; where inter-cluster communication spanning multiple fabrics is required, the performance cost to an individual job must not be so great that it outweighs the

opportunities to improve global resource performance. Therefore, it is necessary to profile the performance of the VCC under these scenarios, in order to understand the feasibility and potential to realise these opportunities.

5.1 Benchmarking Tools

The benchmarking instruments have been chosen in order to exercise the CPU and network subsystems, both independently and together.

Linpack

High Performance Linpack (HPL) is a benchmark based on matrix multiplication and solving a system of linear simultaneous equations (Dongarra et al., 2003). The result produces a single figure, the floating point rate of execution, and is therefore easily and widely used to compare HPC cluster and supercomputing systems (*Top500 FAQ*, n.d.). However, this benchmark is CPU bound and does not exercise the cluster interconnect.

Effective bandwidth

The effective bandwidth (`b_eff`) benchmark measures the latency and bandwidth of a parallel communication network using basic MPI functions (Rabenseifner, Koniges, & Livermore, 2001). In order to perform the latency timing and bandwidth measurements, the algorithm sends both short and long messages in either simultaneous or non-simultaneous patterns between pairs of processors. The randomly ordered ring pattern is used for the benchmarking in this chapter, as it represents the maximum level of contention, and thus the worst case scenario - where every processor

is communicating with a randomly chosen processor simultaneously.

HPC Challenge

The HPC Challenge (HPCC) is a distribution of benchmarks designed to assess the performance of different subsystems in an HPC cluster, including Linpack for CPU execution performance and the effective bandwidth for network performance (Luszczek et al., 2006). In addition to executing the set of benchmarks, HPCC produces a report containing the results of each benchmark in a standard format. Therefore, it has been chosen as a consistent and reliable method of executing the set of benchmarks across the range configurations and scenarios.

OpenFOAM

OpenFOAM is a set of open source numerical solvers for Computational Fluid Dynamics (Jasak, 2009). It has been chosen as a benchmarking tool in order to reflect real world application usage that places demand on both CPU and networking subsystems simultaneously. The OpenFOAM distribution contains example simulations of a propeller and a motorbike. These two scenarios are used as a benchmarking instrument by refining the meshes to between 1.5-2 million cells (a common problem size observed on the campus grid at the University of Huddersfield) and measuring the execution wall time running under different VCC configurations.

5.2 Native vs SDN Interconnect

The service and discovery layer within a VCC container allows the same virtual cluster environment to be deployed regardless if each node resides within the same

Name	Nodes	Cores per node	Processor	Interconnect
Iceotope	4	8	Xeon E5-2670	Gigabit Ethernet
Ascella	4	16	Xeon E7320	IB DDR
Eridani	32	4	Core2 Q8300	Gigabit Ethernet
UCBC	16	4	Opteron 280	IB SDR

Table 5.1: Systems used for performance benchmarking (Higgins et al. (2017a))

fabric or is spanned across many, performing the required contextualisation - such as IP address configuration - automatically. This allows the container to take advantage of a native high performance interconnect if deployed within the boundary of a single fabric, where it is expected that the performance will be comparable with or without virtualisation (Felter et al., 2014). However, once an environment is spanned between two systems, the performance of the SDN compared to the native interconnect becomes a critical consideration.

An evaluation of the VCC performance on Ethernet and InfiniBand interconnects has been conducted and is published in Higgins et al. (2017a). The benchmarking suite set out in the methodology is used to measure the performance of a VCC deployment on 4 different systems, using the HPC Challenge benchmark, as shown in Table 5.1. Each physical node holds a single virtual node, and the benchmark is executed increasing the number of nodes each time, up to the maximum nodes in the respective cluster. The Linpack, random ring bandwidth, and random ring latency results are reported. This process is repeated using both the Weave and native interconnects. In order to draw a fair comparison, IP-over-InfiniBand (IPoIB) is used as the SDN operates on the IP layer, and does not support any RDMA protocols (*Weave Net*, n.d.).

Figures[1] 5.1 and 5.2 outline the random ring bandwidth and latency results (Higgins et al., 2017a). On the Eridani and Iceotope systems, which utilise a Gigabit Ethernet

[1]Permission to reproduce these figures has been granted by the Oxford University Press

interconnect, it can be seen that the SDN overhead in terms of bandwidth is between 5% and 7%, whilst the overhead in terms of latency is 20-30%, 18μs slower compared to the average latency for the native execution. This suggests that while the SDN introduces additional time in order to transmit the data, similar bandwidth to the native interconnect can be achieved.

On the Ascella and UCBC systems, which utilise an InfiniBand interconnect, the bandwidth overhead is significantly increased to 42-45%. Furthermore, the native performance on the Ascella system is less than half that of the UCBC system, despite the interconnect having twice the data rate. This highlights the significance of offloading capabilities when utilising SDN on high bandwidth interconnects - the UCBC system supports IPoIB offloading, but not SDN offloading, whereas the Ascella system does not support IPoIB or SDN offloading. Therefore, the CPU must perform the required IPoIB or SDN functions for each network packet, which incurs a high performance cost. In Figure 5.1, the difference between the native bandwidth of the two systems demonstrates the cost of IPoIB offloading, whilst the difference between SDN and native within each system demonstrates the cost of the SDN encapsulation.

In Figures[2] 5.3 and 5.4, the bandwidth and latency results across VCC sizes are shown (Higgins et al., 2017a). They demonstrate that the SDN performance generally offsets the native by a fixed amount, regardless of the cluster size and underlying interconnect. This suggests that the SDN solution has good scalability, compared to other methods where the cost rapidly increases as the number of nodes increases (Higgins et al., 2017a). Furthermore, it suggests that available strategies in order to reduce the cost, such as hardware offloading capabilities in the network card, will offer a steady performance gain across a range of deployment scales.

[2]Permission to reproduce these figures has been granted by the Oxford University Press

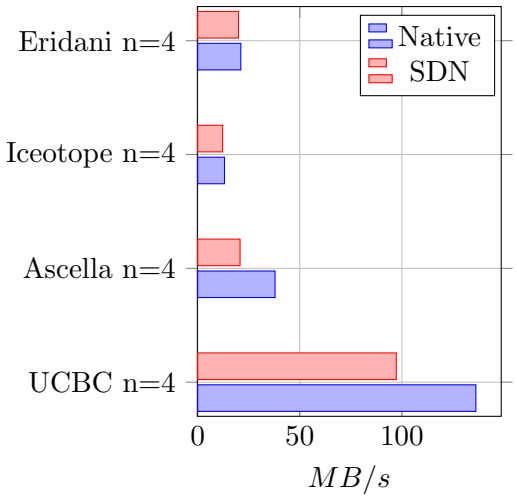


Figure 5.1: Random ring bandwidth benchmark (Higgins et al. (2017a))

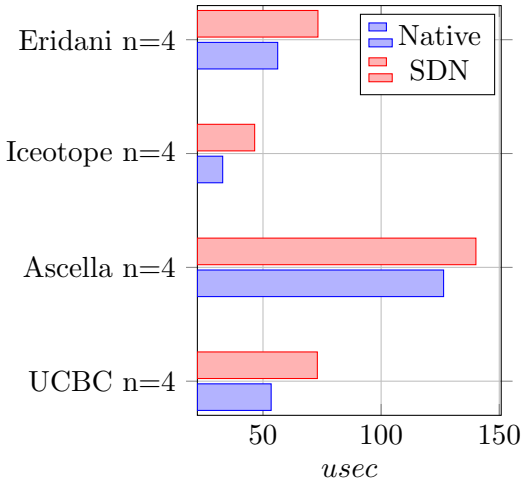


Figure 5.2: Random ring latency benchmark (Higgins et al. (2017a))

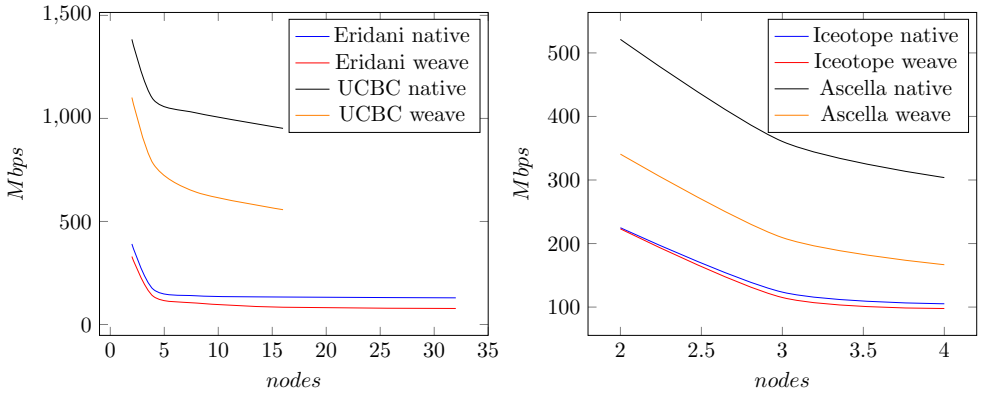


Figure 5.3: Random Ring Bandwith over VCC sizes (Higgins et al. (2017a))

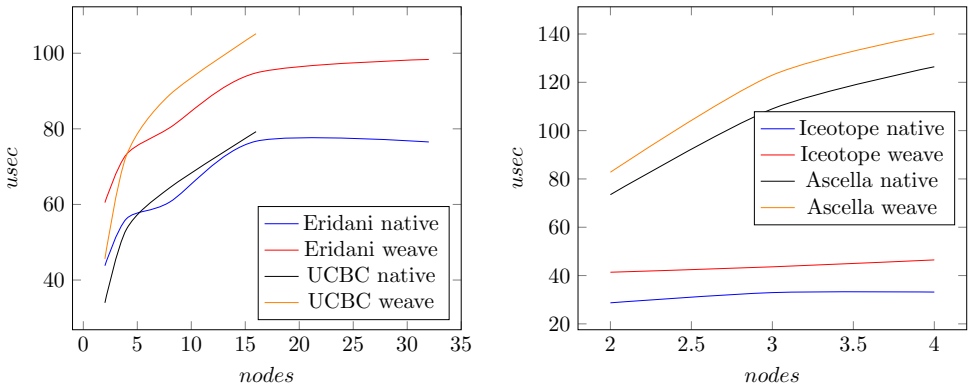


Figure 5.4: Random Ring Latency over VCC sizes (Higgins et al. (2017a))

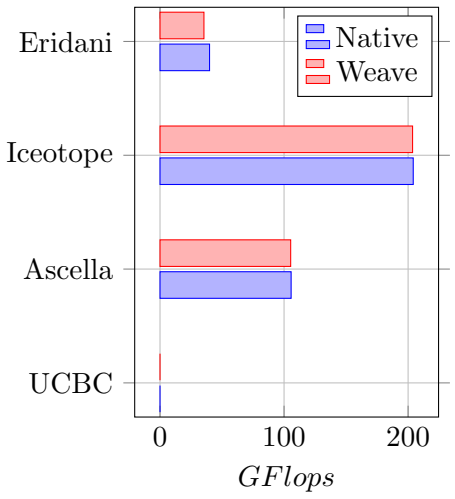


Figure 5.5: Linpack benchmarking results (Higgins et al. (2017a))

The Linpack results are shown in Figure[3] 5.5. Despite the networking overheads introduced by the SDN networking model, the execution performance is not compromised. The Linpack benchmark is typically compute bound but still requires inter-node communication during execution (Dongarra et al., 2003). For applications with a similar profile, it is expected that they will perform without overhead. In the case of an application which is heavily dependent on network communication or sensitive to latency, it is certain to suffer from lower performance when executed within the SDN. However, this effect must only be appreciated when execution occurs spanning two fabrics, as the SDN is not required when execution occurs within the bounds of a single fabric - where the native interconnect can be utilised directly.

5.3 Inter-cluster Interconnect

The benchmark in Section 5.2 demonstrates that, with effectively no physical distance between processes communicating over the SDN interconnect, the execution performance within the VCC in terms of a typical HPC benchmark is acceptable, whilst incurring a small increase in latency.

An inter-cluster interconnect adds additional uncertainty into this model: It is more than likely that a connection can be provisioned in order to meet the bandwidth requirements for communication between two clusters - for example, Janet, the National Research and Education Network in the UK, delivers multiple 400Gbps backbone circuits (*Janet Network*, n.d.). However, this connection may be subject to latency limitations, either due to artificial or physical phenomena, such as network routing decisions or transmission speed in the medium. Previously, this has been considered the primary factor that prevents spanned job execution between two such

[3]Permission to reproduce this figure has been granted by the Oxford University Press

Scenario	Distance	RTT
Cluster	<10 meters	0.17 ms
Campus Grid	500 meters	0.24 ms
Inter-institution	30 miles	14.8 ms
West US to Central US	1000 miles	44 ms
East US to West US	2000 miles	76 ms
Europe to East US	4000 miles	135 ms
Europe to Central US	5000 miles	142 ms

Table 5.2: Survey of typical Round Trip Time (RTT) per scenario

interconnected clusters (Richling, Hau, Kredel, & Kruse, 2011).

5.3.1 Latency Scalability

In order to understand the range of latencies exhibited by the deployment scenarios proposed in this research, a survey was carried out measuring round trip time between various systems that represent different geographical distances. The results are shown in Table 5.2. All measurements were taken using the `ping` utility and averaged over 10 packets. For the long distance scenarios, VMs were provisioned in the Microsoft Azure cloud in the respective regions in order to conduct the measurements.

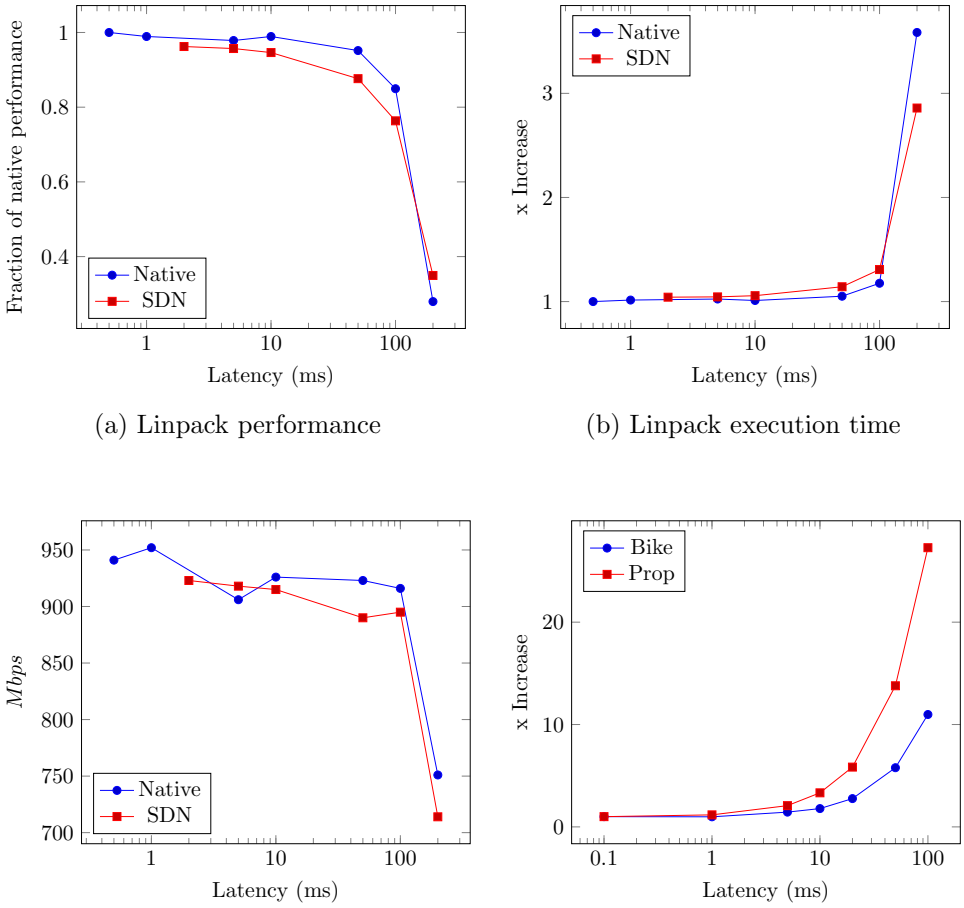
Based on these measurements, a possible range from 0.1-150ms has been used when assessing the performance and scalability of the VCC in terms of latency rather than cluster size. In order to execute the benchmark suite with increasing latency between processes, a fixed size VCC environment has been deployed on a pair of VMs within the Microsoft Azure cloud. The `D8S_V3` machine type was chosen for the compute workloads, consisting of 8x Xeon E5-2673v4 cores, 32GB RAM and 64GB SSD based storage. For the long distance scenarios, the VMs were provisioned in the respective regions. However, in order to conduct testing for the short distance scenarios, the

VMs were deployed in the same region and the network emulation capabilities of the Linux kernel were utilised in order to add the required latency (Ludovici & Pfeifer, 2011).

The Linpack benchmark results are detailed in Figures 5.6a and 5.6b. As previously observed, the SDN approach introduces an overhead mainly due to the additional latency of encapsulating the virtual cluster traffic before it is transmitted over the network. Nonetheless, the results suggest that up to 10ms, the spanned execution performance of the Linpack benchmark is within 5% of the native performance. In addition, the execution time - a critical factor for exploiting job spanning - is not significantly extended up until this point. However, above 100ms of latency, it was not possible to sustain a comparable end-to-end bandwidth, as shown in Figure 5.6c. This is reflected in a performance tail off and respective increase in execution time.

The OpenFOAM benchmark results are detailed in Figure 5.6d. Compared to the Linpack performance, it can be seen that the execution time is greater affected by the network latency: 5ms is approximately 2x slower, and 10ms is approximately 3.5x slower. This is due to the fact that this computation is more dependent on the network interconnect, where the magnitude of the data that must be exchanged between processors is higher and more frequent than the Linpack benchmark. It is also apparent that some cases scale better with higher latency - the Bike benchmark uses the simple OpenFOAM solver and only suffered approximately half the performance degradation as the Prop benchmark, however, this mesh also contained 25% less cells (1.5 million compared to 2 million).

Overall, these results suggest that latency, as a metric describing geographical distance, provides a simple predictor of spanning performance and associated cost, given a constant end-to-end bandwidth. HPC clusters are typically characterised by



(a) Linpack performance

(b) Linpack execution time

(c) Bandwidth

(d) OpenFOAM execution time

Figure 5.6: Latency scalability benchmarking

exceptionally low-latency interconnects, however, the results indicate that the range of acceptable latency for spanned job execution is modest. Depending on the communication patterns of the application, a latency overhead of up to 10ms may still provide an acceptable execution time for the job. Beyond this, the execution time appears to increase rapidly, and the usefulness depends on whether the longer wall time remains less than the time the job would otherwise have been queued.

5.3.2 Spanning Simulator

In order to discover the feasibility of an improvement in resource management given a particular spanning cost, a simple simulation model was devised in order to analyse the historical workload from two clusters in the campus grid at the University of Huddersfield. The design of the simulator is not to arbitrarily reschedule the workload in order to achieve a desired job throughput or efficiency. In this case, one can simply defragment the existing jobs into contiguous periods. However, this does not reflect real world usage patterns where jobs are submitted by users when they wish. Therefore, the purpose of the simulator is to identify points in the actual scheduling timeline where it would have been possible to exploit forwarding or spanning.

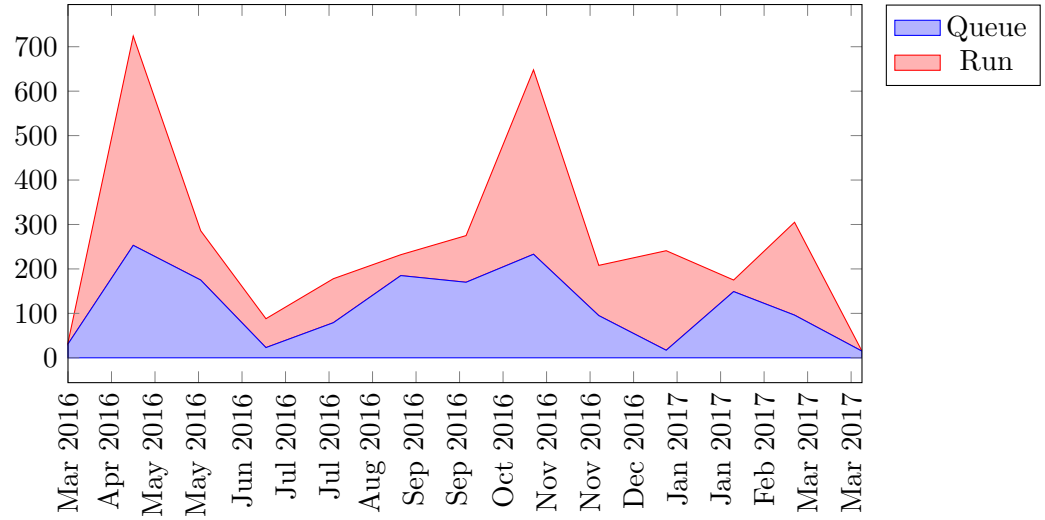
The simulator will recognise where a job was originally queued on the cluster it was submitted from, but where at the same time, enough nodes were idle for an adequate period in order to satisfy its execution on another cluster. In this case, it will report that the job was eligible to be forwarded. Similarly, the simulator will recognise periods where a job was queued upon submission, but where at the same time, the summation of idle nodes between the two clusters is adequate in order to satisfy its execution. In this case, it will report that the job can be spanned. However, in both cases it does not reschedule all subsequent jobs or introduce new jobs into the system.

The simulator has been run using the scheduling logs from the campus grid at the University of Huddersfield, covering the 12 month period from March 2016 to March 2017. It represents usage from a variety of domains, such as Bioinformatics, Computational Fluid Dynamics and Particle Physics. The scheduling logs include a list of jobs executed by each system, including the following metrics:

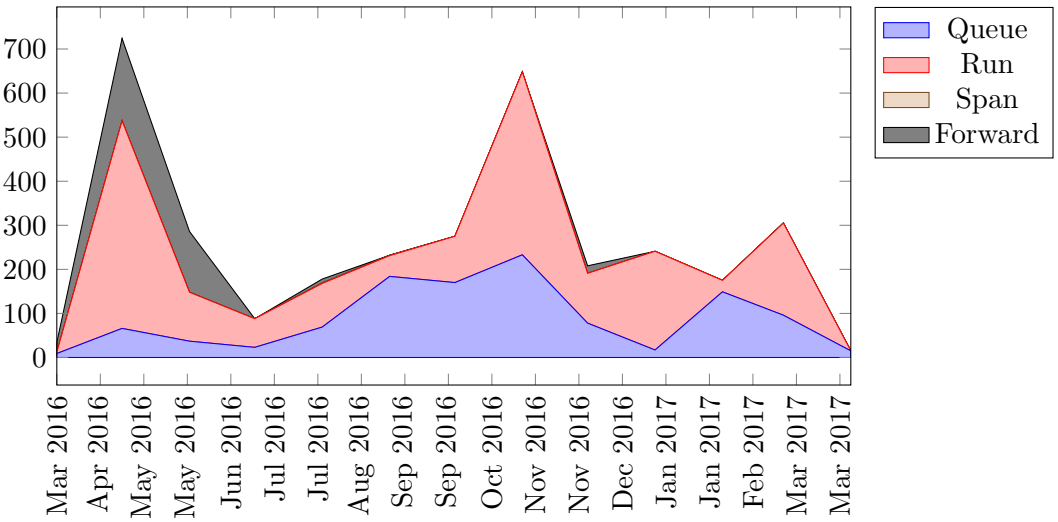
1. Submission time
2. Start time
3. End time
4. Number of nodes required

The results are shown in Figures 5.7 and 5.8. It can be seen that there were opportunities to exploit both spanning and forwarding techniques within this workload profile. By re-balancing jobs between the two clusters, the queue wait times were almost eliminated in the first 4 months of the workload profile (the busiest period of the resource in terms of number of job submissions). The overall number of jobs executed remains the same as the simulator does not introduce new jobs into the system - thus the utilisation remains constant. However, this creates a vacuum of jobs towards the end of each scheduling period, as the originally submitted jobs are completed earlier in time. This demonstrates a decrease in job turnaround time, and the potential for backfilling new jobs within the time gained in order to increase the utilisation of the system in a real world context.

The second aspect evaluated by the simulator is, where a job has been identified as a candidate for spanning, would it have been able to complete execution within the available idle time given a spanned performance penalty. It was seen that, even with a 3x performance penalty, 42% of jobs that were eligible for spanning would still have completed execution within a period of available idle time. Based on an

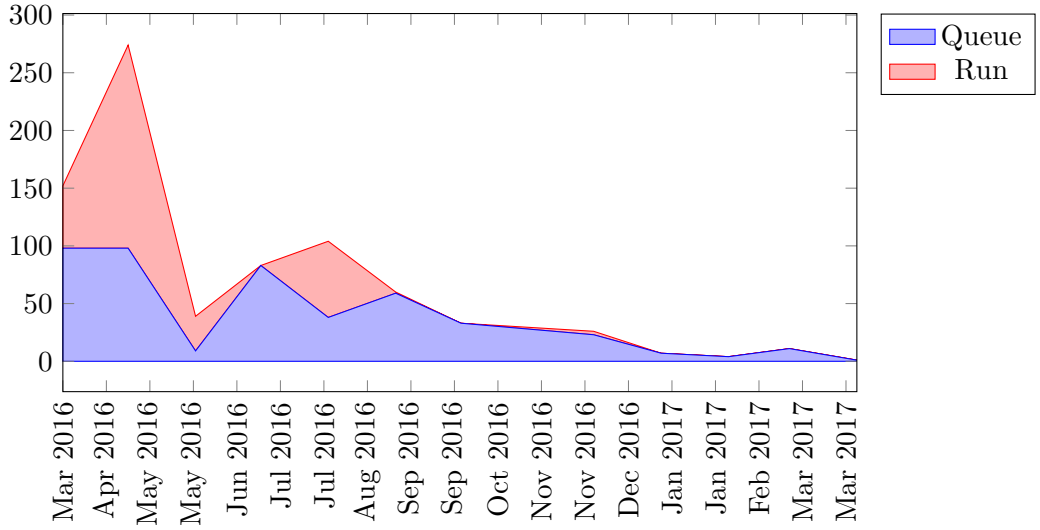


(a) Original submitting job states

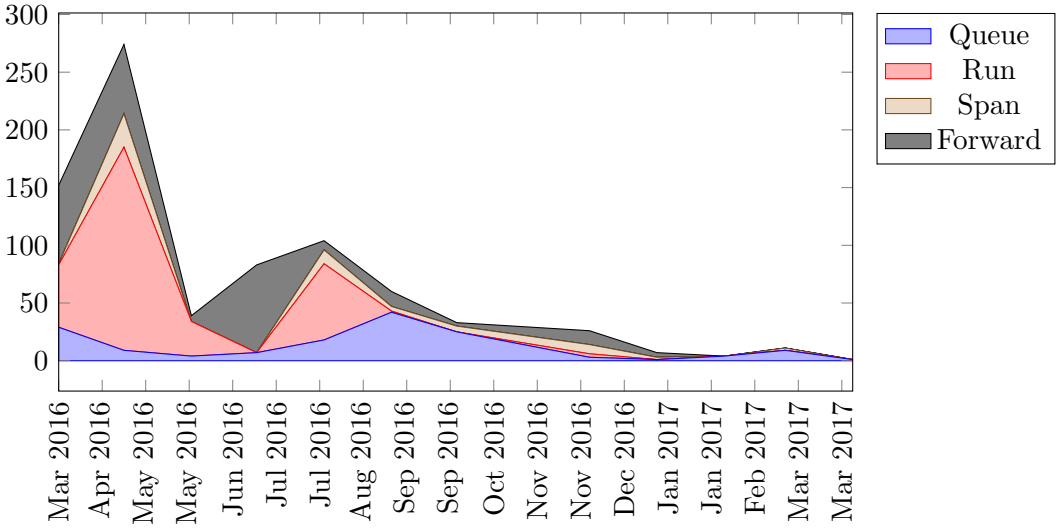


(b) Spanned submitting job states

Figure 5.7: Cluster 1 (Eridani) simulator results



(a) Original submitting job states



(b) Spanned submitting job states

Figure 5.8: Cluster 2 (Ascella) simulator results

application with a similar profile to Linpack, this would equate to approximately 100ms of latency as being acceptable, or 10ms based on the profile of the OpenFOAM benchmark.

5.4 Summary

In this chapter, a robust performance evaluation of the VCC was conducted. As the performance characteristics of containers within the run time environment are already well understood in literature, the benchmarking focus is on the networking model and subsequent effect on HPC application performance; the innovative full stack approach of the VCC introduces a virtualised SDN interconnect in order to enable transparent job spanning and forwarding within a containerised cluster environment.

Firstly, the CPU execution performance within the VCC is evaluated. A comparison between native, run time environment and VM-based approaches suggests that the full stack container design, and associated services for discovery, configuration and management, does not have a negative impact on the application performance.

Secondly, the communication performance of the VCC networking model is evaluated within the boundary of a single cluster fabric. It was observed that the overhead of the SDN typically manifests as a fixed offset from the native performance. On commodity Ethernet networks, it is possible to achieve equivalent communication bandwidth. On a high bandwidth InfiniBand network, the results highlight that support for offloading the processing of the IP and SDN protocols onto the networking hardware is essential in order to achieve good performance. For both interconnection technologies, the latency overhead is similar and shown to be offset between 20-30% from the native performance.

Finally, the additional latency uncertainty introduced by the inter-cluster interconnect is considered. By measuring the latency between systems which are representative of the cluster, campus grid and inter-institution scenarios, the additional time incurred was found to fall on a range between 0.1-150ms depending on geographical distance. The benchmarking suite was repeated with increasing latency added between the communicating nodes in order to understand the effect on execution performance.

Previous work rejects such network environments as inappropriate for HPC execution due to this latency (Richling et al., 2011). Rather than considering performance on an individual job level, it is clear that there are opportunities for slower execution of spanned jobs to improve global resource performance, in terms of job throughput and utilisation. A simulator was presented based on a historical workload profile, which identifies common periods of idle node time between several HPC clusters and applies spanning or forwarding to jobs that would otherwise have been immediately queued upon submission. Aggregating the idle capacity and using it for job execution provides the ability to start the job earlier in time, potentially offsetting the performance cost and improving utilisation of the individual clusters. The simulation results demonstrate that even with up to 3x slower execution, over 40% of spanned jobs would still have been able to complete within a period of idle time. However, this effect is dependent on the application and its communication patterns - for example, at 100ms of latency, the Linpack run time was 1.2x slower whilst maintaining 80% of the native performance, whereas the OpenFOAM benchmark was approximately an order of magnitude slower.

Chapter 6

Geographically Distributed Spanning

In this chapter, a case study composed of a VCC deployment within two real world scenarios - campus grid and inter-institution grid - is presented. Previous work has demonstrated the potential of job forwarding and spanning in order to improve global resource performance, in the context of a controlled experimental environment. Despite this potential, it does not adequately consider the feasibility of achieving such performance gains, given realistic workload patterns and communication constraints. In Chapter 5, it was demonstrated that the VCC can be used to facilitate the same meta-scheduling-like capabilities. Network-orientated benchmarking and workload simulation based on historical job submissions suggested that, not only are the inter-cluster interconnect performance demands modest, but that even with a spanning cost of up to 300%, there were still many scheduling periods where a performance improvement was attainable. This performance improvement is characterised by the ability to start job execution earlier on spanned or forwarded assets, thus, removing the queue wait time, decreasing turnaround time and increasing job throughput.

Therefore, it is necessary to evaluate whether this result is reproducible in practice.

Furthermore, in the synthesis it is identified that a virtual cluster on its own, whilst portable, does not immediately resolve the software environment flexibility problem; there still exists the requirement for a single system image within the virtual cluster environment. Thus, the case study also demonstrates how a novel nested cluster deployment topology is able to retain both portability and software flexibility aspects afforded by virtualisation within the HPC context.

Firstly, the meta-cluster deployment topology is described. Secondly, the design of the experiment, including the connectivity requirements of both scenarios and the job queue used for benchmarking, is defined in Section 6.2. Finally, an evaluation of the results and discussion of case study outcomes is presented for each scenario in Section 6.3.

6.1 Nested Meta-Cluster Topology

One of the main problems with traditional HPC software stacks is the lack of flexibility in customising the software environment. The primary motivator for integrating virtualisation into HPC is to address this lack of flexibility. It is clear from the literature review that the flexibility of existing solutions is defined by the persistence of the virtual environment.

With run time environment containers, the virtualised environment lasts only for the duration of a single process execution. This allows the environment to be arbitrarily customised on a per-process level, but introduces portability limitations by depending on the underlying system to provide the necessary interfaces for execution.

In a full stack approach, a virtual cluster can be deployed for a single user, or for the duration of an application execution, as with run time environment containers. This allows the same flexibility whilst additionally increasing the portability of the environment as the full stack is virtualised rather than just the top-most layer. However, if a full stack virtual cluster is deployed in a multi-user or persistent scenario, then whilst it is still inherently portable and self-contained, it does not resolve the software flexibility limitations as a coherent, single system image is still required as would be on a non-virtual cluster.

The design of the VCC enables a unique characteristic where it can be deployed at either level of persistence: as a transient, temporary environment for the duration of execution, or as a permanent, multi-user virtual cluster. In order to introduce software environment flexibility into the persistent full stack approach, the VCC presents a nested deployment topology as shown in Figure 6.1. The outer cluster is the persistent virtual cluster, deployed on top of the physical resource. The software environment of the outer cluster maintains the same limitations as a real cluster, in that each virtual node must conform to the same software environment. However, in order to overcome this limitation, a temporary inner virtual cluster is deployed with a distinct software environment in order to execute the workload.

The ability to nest virtual clusters has a number of advantages; firstly, the software environment can be gradually refined to any extent by simply nesting another cluster. As container virtualisation does not require the layers of emulation or translation required by traditional VMs, there is no additional virtualisation overhead introduced on the nested containers (Morgua, 2015).

Secondly, for a VCC deployed spanning a multiple of underlying physical resources, only the outer cluster is required to establish the SDN interconnect. Therefore, the inner cluster does not have to consider many aspects of the deployment, such as

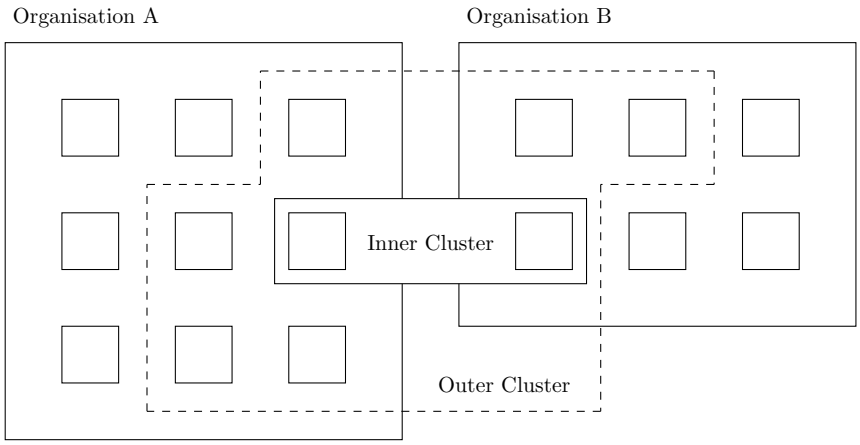


Figure 6.1: VCC Meta-Cluster Deployment

interconnect, DNS name resolution, and other contextual-dependent configuration. This means that the nested clusters do not need to duplicate these services in order to take advantage of spanning and forwarding features, as it will already be guaranteed that the outer cluster will provide them. Furthermore, it simplifies the definition and deployment workflow of the inner cluster in order to maintain accessibility and ease of use for the average user, equivalent to the demands of using run time environment containers. In practice, the administrator could deploy the outer virtual cluster, across as many physical resources as required, and the end user can deploy as many transient, nested virtual clusters as required in order to execute the workload of jobs. In order to operate the inner cluster, the user will not require knowledge of, or direct access to, the underlying physical cluster.

Finally, establishing an outer cluster provides a straightforward method in order to divide the resources of an underlying system and limit the scale at which virtual clusters can be deployed, without requiring dedicated resource management. As shown in Figure 6.1, the persistent outer cluster is simply deployed across as many physical nodes as desired within each organisation's cluster, and this provides a natural boundary on the scale of the nested inner clusters. In a virtual cluster that

spans many underlying physical systems such as this, where each system is potentially hosted by different institutions or organisations, this capability is essential in order to maintain fair use and dedicate only a subset of the physical resource to be made available for virtual cluster execution.

6.2 Methodology

The aim of this experiment is to realise a decrease in turnaround time when executing a queue of jobs on a VCC deployed across many physical resources, by utilising spanning and forwarding techniques. It will be conducted based on two scenarios, campus grid and inter-institution. These two scenarios have been selected based on the experience of past collaborations at the University of Huddersfield, which have demanded connectivity between both computing facilities located around the campus and computing facilities provided by nearby institutions. The process of deploying the VCC in these two scenarios will also provide a useful assessment of the SDN and workflow challenges in practice, such as negotiating firewall and port forwarding requirements with the local IT departments.

Two clusters have been constructed in order to conduct the case study, detailed in Table 6.1. A dedicated head node is located alongside each cluster to provide management functions, which will not be included in computation. Cluster A is installed in the main university data center, whilst cluster B will be relocated across campus and subsequently to another institution in order to test the two scenarios in this case study. Therefore, the hardware and software configuration remains constant in order to observe the effect of spanning and forwarding on real world application and resource performance. The benchmarking suite outlined in the methodology in Section 1.2 will be used in order to measure the performance of the spanned meta-

cluster deployment; the OpenFOAM benchmark provides application performance for a CFD simulation workload.

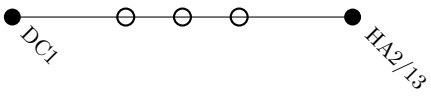
6.2.1 Cluster Connectivity

For the campus grid scenario, the two clusters will be interconnected across the general campus LAN. This provides 1 Gbps end-to-end bandwidth and typical 0.17ms latency. Cluster A is installed in the university data center, DC1, and Cluster B is installed in the HPC Laboratory, HA2/13. As shown in Figure 6.2a, 4 hops are introduced as the traffic is passed by switches within the network infrastructure in order to reach the destination. The physical distance of this link is approximately 300m. However, the nodes within each cluster reside on a private network, with the headnode acting as a NAT router. Direct routing between the private networks of each cluster would not be possible. Direct physical interconnection, as required by other solutions such as Emeneker et al. (2006), is impractical. Therefore, the SDN interconnect will provide a virtual subnet that traverses each network, without exposing access to the private networks from other locations on the LAN.

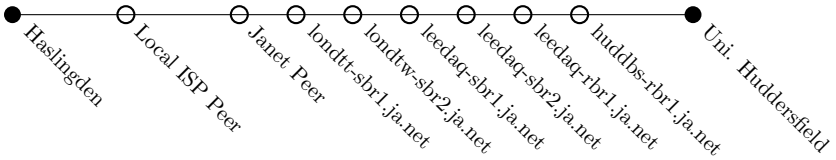
For the inter-institution scenario, Cluster B will be relocated to another institution and the two clusters will be interconnected across the internet. Cluster B is selected to be relocated as it has the fewer nodes and smaller footprint in terms of logistics.

Cluster	Nodes	Type	CPU	Memory
A	5	HP DL380	2x 6C/12T Xeon X5690	128GB
B	3	Dell R710	2x 6C/12T Xeon X5650	192GB

Table 6.1: Cluster Specifications for Spanning Case Study



(a) Campus Grid



(b) Inter-institution

Figure 6.2: Traceroute of Spanning Case Study Connectivity

As shown in Figure 6.2b, a more complex route is taken in order to pass packets between the respective network infrastructures. This connectivity also provides up to 1 Gbps end-to-end bandwidth with a typical latency of 18ms. The physical distance of this link is approximately 30 miles (48 km). Similarly to the campus grid scenario, both institutions utilise a firewall which performs NAT, therefore, direct routing between the private networks of each cluster is not possible. The SDN interconnect established by the VCC will facilitate this communication across the networks in the same manner. However, there is uncertainty introduced into this scenario, as the traffic will be subject to Quality of Service, routing and traffic shaping policies by the respective Internet Service Providers. In the case where these measures are apparent, and spanned execution is likely to be severely affected, the VCC can utilise only job forwarding capabilities.

6.2.2 Job Queue

In order to examine both the global resource performance, and the individual job performance, several OpenFOAM benchmark jobs will be queued on the spanned VCC meta-cluster. The turnaround time of the queue will be measured in order

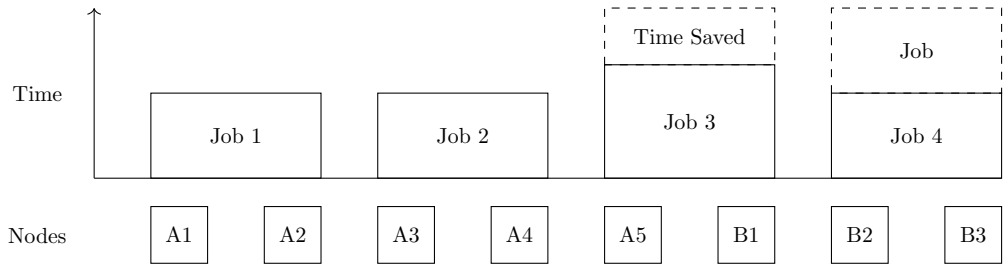


Figure 6.3: Job Queue Pattern for Spanning Case Study

to demonstrate the time saved and the potential increase in job throughput, compared to running the job queue only on cluster A without spanning to cluster B. In addition, the performance of the spanned job can be compared to the non-spanned jobs within the same execution conditions, without having to carry out a second experiment.

Each job is based on the same OpenFOAM Bike benchmark using different starting conditions. This is a common scheme observed on the campus grid at the University of Huddersfield, where a researcher wishes to test a parameter space in order to narrow down the conditions for subsequent investigation. Each job is configured with the same 2 node, 24 cores per node requirement. Therefore, the queue is devised in such a way that when submitted to the cluster, at any time one job will effectively be forwarded and one job will be spanned, as shown in Figure 6.3. In this case, Job 4 is forwarded as it is executed entirely within the bounds of Cluster B, and Job 3 is spanned as it requires nodes from both Cluster A and B.

The benchmarking in Chapter 5 indicates that spanned execution will incur a performance penalty; as long as the runtime of the spanned job does not exceed the queue wait time plus the original execution time, there is a potential to complete the job earlier in time. Thus, the time saved can enable an increase in job throughput as the next job in the queue can also be started earlier. Without the spanning

Cluster	Location	IP	Ports	Type	Direction
A	Huddersfield	161.112.232.42	36783, 36784	TCP + UDP	Both
B	Haslingden	84.21.152.17	36783, 36784	TCP + UDP	Both

Table 6.2: Firewall requirements for Spanning Case Study

capability, a single node from each cluster would be continuously idle as it cannot satisfy the requirements of any jobs in the queue.

6.2.3 Procedure

In general, the outline of the procedure is to establish the SDN network, launch the VCC containers on each node and execute the benchmark suite as detailed in Appendix. Firstly, connectivity between the clusters must be established. For the campus span scenario, the head nodes of clusters A and B can communicate across the campus LAN without any special network configuration. However, for the inter-institution scenario, each head node will require ports to be forwarded from a public IP address in order for the SDN to be established. This configuration is detailed in Table 6.2.

Secondly, the Weave router - which provides the SDN functionality - needs to be launched on every node in order to establish the network. As shown in Figure 6.4, each node must be bootstrapped by providing the IP of another node within the cluster. Once connected, the IPs of the other hosts will be discovered. A new network interface is created on the system which exposes an IP address on the virtual subnet provided by the SDN.

Finally, the VCC container must be launched on every node. It will automatically detect the presence of the SDN network interface and perform the required environment contextualisation so that each container will use the SDN for communication.

```
$ weave launch --port=36783 --ipalloc-range 172.31.0.0/20
$ weave connect 161.112.232.42
$ weave expose 172.31.1.1/20
$ docker run -d --net=host --privileged \
  joshiggins/vcc-exp \
  --service=workernode \
  --storage-host=172.31.1.1 \
  --storage-port=4001 \
  --cluster=test
```

Figure 6.4: Bootstrapping the Weave SDN network and spanned VCC

Even though the underlying physical clusters both have their own head nodes, within the VCC the head node container can be deployed on any node, and it is not necessary to deploy a VCC head node for each underlying head node. For the purpose of consistency, the VCC head node is deployed on the head node for cluster A in both scenarios.

6.3 Evaluation

In both scenarios, the deployment of the VCC was successful and created a coherent cluster environment across the respective geographical distances. The SDN provided a virtual subnet, assigning IP addresses for each VCC container within a private address range which were routable between both clusters, despite the partial connectivity introduced by the firewalls and NAT.

From the application perspective, no changes were required to the communication routines or resource management in order to take advantage of spanning and forwarding. The SDN appears as a global inter-node interconnect, transparently routing and forwarding traffic to the appropriate destination, allowing the *outer cluster* detailed in Section 6.1 to be established for both scenarios. Figure 6.5 shows the

```
[root@headnode /]# ping clusterB3
PING clusterB3 (172.31.3.10) 56(84) bytes of data.
64 bytes from 172.31.3.10 (172.31.3.10): icmp_seq=1 ttl=64 time=10.5 ms

[root@headnode /]# ping clusterA1
PING clusterA1 (172.31.14.0) 56(84) bytes of data.
64 bytes from 172.31.14.0 (172.31.14.0): icmp_seq=1 ttl=64 time=0.700 ms

[u1056048@headnode]$ for i in {1..10}; do qsub submit.job; done

[u1056048@headnode]$ qstat -n
headnode:

11.headnode                u1056048    paraul   prop_original_hi
0      2      48     --   336:00:00 R  00:00:36
   clusterB1/0-23+clusterB2/0-23
12.headnode                u1056048    paraul   prop_original_hi
0      2      48     --   336:00:00 R  00:00:36
   clusterB3/0-23+clusterA5/0-23
13.headnode                u1056048    paraul   prop_original_hi
0      2      48     --   336:00:00 R  00:00:36
   clusterA1/0-23+clusterA2/0-23
14.headnode                u1056048    paraul   prop_original_hi
0      2      48     --   336:00:00 R  00:00:36
   clusterA3/0-23+clusterA4/0-23
```

Figure 6.5: Outer Cluster Job Submission Workflow

workflow of the outer VCC cluster. It demonstrates that the nodes are combined into a single virtual cluster, allowing jobs or nested clusters to be scheduled within. Communication between nodes, regardless of the location, appears as though they are directly connected on the same network. However, the overhead introduced by the underlying networks can be observed by pinging between nodes within the virtual cluster.

From an administration perspective, there was minimal concern from the respective IT departments in order to open the two ports required for the SDN. The virtual subnet traffic is encapsulated using the VXLAN protocol and transmitted using UDP on the data plane port (*Fast Datapath and Weave Net*, n.d.). This methodology proved to easily traverse firewalls and avoid double NAT when facilitating routing

between geographically distributed sites, whilst maintaining the security of each private network. However, it is anticipated that if the system were to be used in production, the wider implications of the bandwidth and traffic usage patterns, and the potential impact on other services, would become a concern.

Table 6.3 details the queue turnaround time for both scenarios.

Scenario	Queue turnaround (mins)
Cluster A only	226
Campus Grid	134 (41% faster)
Inter-institution	145 (36% faster)

Table 6.3: Campus spanned job queue turnaround time

Compared to execution only on cluster A, the queue was executed faster by utilising the spanning and scaling techniques in both cases. This is because the virtual cluster is effectively a summation of both cluster A and B with a single queue, thus, making more nodes available for processing the queue of jobs. The result demonstrates that the penalty of combining the two clusters using this approach does not outweigh the benefit.

However, only the forwarding capability was utilised in the inter-institution scenario, as the actual throughput of the connectivity was lower than expected, measured at 200 Mbps. It was not deemed suitable for spanning execution: the performance benchmarking and feasibility study is calibrated based on a constant end-to-end bandwidth, therefore, it would not be a fair comparison. In addition, the reduced bandwidth is likely to significantly extend the execution time of the OpenFOAM benchmark. Regardless, attempting to execute the spanned jobs using this connection highlighted that the Message Passing Interface (MPI) library used for parallelisation of the OpenFOAM code is not tolerant of sustained adverse network conditions.

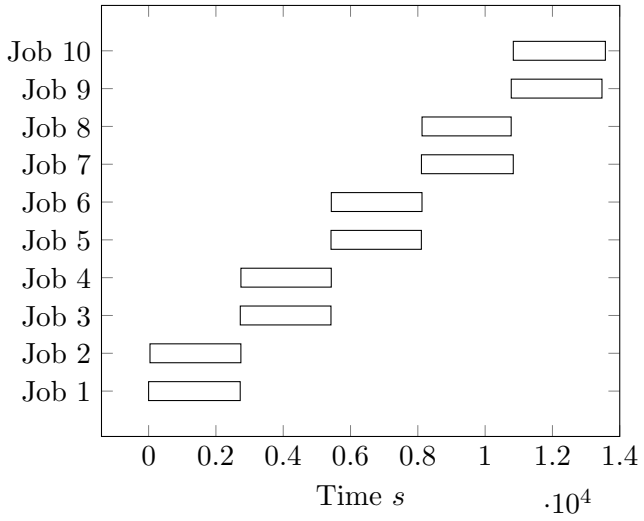
6.3.1 Campus Grid

Figure 6.6 plots the detailed execution times of each job within the queue for the campus grid scenario. As expected, the performance overhead of spanning is minimal: the inter-cluster interconnect matches the bandwidth specification of the local cluster interconnect (1 Gbps), and the latency is small due to the relatively close proximity of the two systems.

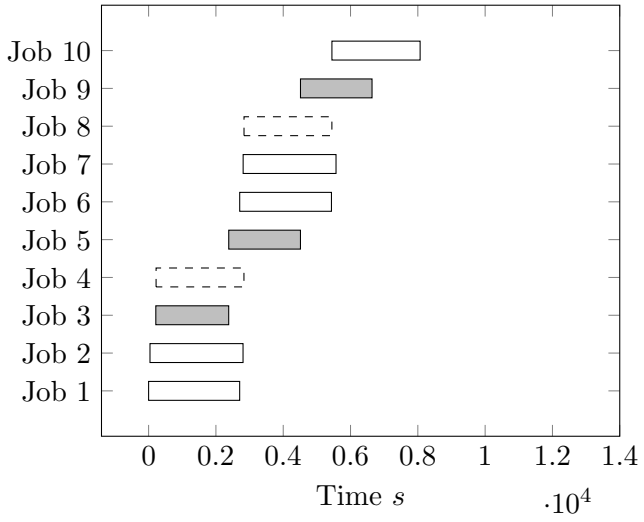
Without the VCC, the concurrency for job execution scheduled in the queue on cluster A is limited to two, whilst one node remains idle. Execution on cluster B was slightly faster than cluster A due to differences in the hardware, however, the concurrency is limited to one job at a time. The spanned job, highlighted by dashes, was approximately 20% slower in comparison to the fastest execution. The forwarded job is highlighted by a filled box. With the spanned VCC providing both of these capabilities, four jobs may run concurrently, and hence, a subsequent reduction in the queue turnaround time is observed.

On an individual job level, the overhead of the spanned execution is not significant enough to extend the runtime such that it negates the advantage of starting earlier. However, this result may only be generalised to high performance jobs; where there is a high throughput of extremely short lived jobs, this overhead may delay the scheduling of each job and thus, have the opposite effect of slowing down the job queue overall. This has been observed by Emeneker and Stanzione (2007), though due to the overheads of traditional virtualisation techniques rather than spanned network performance.

Furthermore, the CFD simulation performed by the OpenFOAM benchmark is composed of a mix of CPU bound processing and storage I/O. At the start and end of the simulation, parallel storage I/O is performed in order to transfer the partitioned



(a) Cluster A only



(b) Spanned Cluster A+B

Figure 6.6: Campus Grid Queue Execution Time Plots

data to and from each node's local scratch space. During the execution, the data is operated in local scratch space only. Where the storage I/O is being performed over the network, the spanning penalty also applies to this traffic in the same way that it applies to data being passed between communicating processes.

In this case, whilst the benchmark operates on several gigabytes of scratch files, the actual data transferred is on the order of 100's of megabytes. Even on the 200 Mbps network, the transfer time is a matter of seconds. When a higher demand is placed by the application on parallelisation of storage I/O, the effect of the SDN overhead can be mitigated when the underlying system provides a native storage service. The same pass-through methodology can be used as when a high performance interconnect, such as InfiniBand, is used within the VCC. Therefore, the container can access the native storage system without incurring the overhead of communicating across the SDN, but the individual job execution cannot span beyond the boundary of the underlying system.

6.3.2 Inter-institution Grid

Figure 6.8 plots the detailed execution times of each job within the queue for the inter-institution scenario. Compared to execution only on cluster A, the time to process the queue when utilising job forwarding is reduced. This is what would be expected if one simply added two nodes to the cluster, which is effectively what the VCC has facilitated, despite the geographical separation between the additional nodes and the rest of the cluster. Due to bandwidth constraints of the inter-cluster interconnect, job spanning capability was not enabled in this scenario. Whilst this results in a node from each underlying cluster remaining idle, the difference in queue turnaround time of just forwarding compared to utilising both spanning and forwarding is only 5%, over this queue of jobs.

```
smoothSolver:  Solving for Ux, Initial residual = 1.58663e-07, ...
smoothSolver:  Solving for Uy, Initial residual = 7.49369e-08, ...
smoothSolver:  Solving for Uz, Initial residual = 1.56562e-07, ...
--------------------------------------------------------------------------
ORTE has lost communication with its daemon located on node:

  hostname:  clusterB1

This is usually due to either a failure of the TCP network
connection to the node, or possibly an internal failure of
the daemon itself. We cannot recover from this failure, and
therefore will terminate the job.

--------------------------------------------------------------------------
write ip4 194.82.37.91->161.112.232.42: write: no buffer space available
```

Figure 6.7: Spanning Errors in Extreme Network Environments

In this case, taking into account external factors such as the cost of network transit and the relatively small gain, it might not be worth utilising spanning even if there is a capability to do so. However, considering just the first four jobs in each scenario, the difference in performance gain is much more significant - 48% faster with forwarding and spanning compared to 19% faster with just forwarding. Therefore, it is clear that whilst the net gain is likely to be positive due to the fact of simply having more nodes available for execution, the extent to which the global resource performance is improved depends on both the spanning penalty introduced by the VCC, and the queue length.

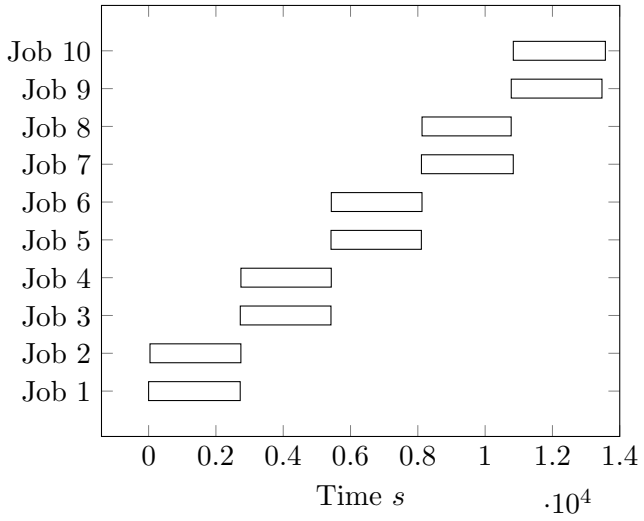
Attempting to execute the spanned workload despite the bandwidth limitation demonstrates that the MPI library used for parallelisation of the code is not robust to poor network connectivity. Therefore, even if the spanning cost is acceptable, in extreme conditions it may not work at all. As the application itself is not natively bandwidth or latency aware, this resulted in excessive packet loss due to the exhaustion of the OS network buffers, as data was sent faster than it is possible to be transmitted. The Weave SDN client recovers from this error, but the subsequent

connection reset causes problems for the MPI application. In the course of carrying out this experiment, the same benchmark program exhibited three different runtime errors caused by the same problem: sometimes, the MPI library reports that a timeout has occurred in a communication routine, which immediately stops the simulation in a manner which is not restartable, as shown in Figure 6.7. Other times, the program appeared to hang and make no further progress, or exited with a segmentation fault. Even though there are algorithms available for collective communications which are bandwidth and latency aware, such as Gong, He, and Zhong (2015), it does not consider this scale of sub-gigabit speeds.

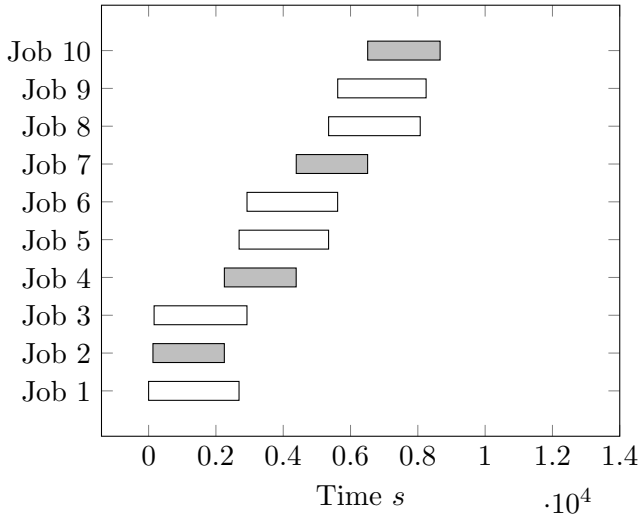
6.4 Summary

In this chapter, a case study is presented in order to evaluate the improvement to global resource performance through deployment of the VCC in two real world environments. The meta-cluster deployment topology is outlined, in which an outer virtual cluster spanning many underlying resources is established. Within the outer cluster, nested, inner virtual clusters can be scheduled in order to execute the job workloads with an abstracted, coherent environment. The inner clusters can be nested in order to gradually refine the software environment, and do not require knowledge of the underlying infrastructure or consideration of aspects already established by the outer cluster, such network connectivity and shared file storage. This offers a novel approach to introducing software environment flexibility within a full stack virtualisation solution, whilst inheriting the good accessibility and portability afforded by run time environment virtualisation approaches.

The performance benchmarking within a controlled environment, conducted in Chapter 5, suggests that it is feasible to realise a performance improvement in terms of the



(a) Cluster A only



(b) Forwarded Cluster A+B

Figure 6.8: Inter-institution Queue Execution Time Plots

overall turnaround time and job throughput - as jobs can be started earlier in time - by utilising the job spanning and forwarding capability provided by the VCC. The purpose of the case study is to demonstrate that it is reproducible in a real world environment. Two scenarios are used in order to assess the resource performance: campus grid and inter-institution spanning. A queue of jobs is constructed based on the OpenFOAM Bike benchmark in order to submit to each spanned VCC for execution, and the turnaround time of the individual jobs and the overall queue is recorded.

The challenges posted by the inter-cluster communication were easy to accommodate using the SDN, requiring only two firewall port forwarding rules to be in place within each institution. In the campus grid scenario, utilising spanning and forwarding capabilities, the queue turnaround time was 41-48% faster than execution on a single cluster only. In the inter-institution scenario, utilising only forwarding capabilities due to the bandwidth constraints, 18-36% faster turnaround time of the job queue was observed. Both spanning and forwarding techniques are facilitated by the fact that the VCC creates a single, coherent virtual cluster across the two physical clusters, using the SDN in order to create a communication channel across partially connected networks - where previous solutions have required dedicated links or proprietary appliances (Emeneker & Stanzione, 2007; Richling et al., 2011).

The results demonstrate that the performance gains suggested by the benchmarking are possible to achieve in practice. In this case study, a CFD code was used and thus, the result can be generalisable to other codes with a similar profile in terms of CPU and network performance requirements. However, it is highlighted that in extreme network conditions, spanning may not work due to the parallelisation libraries used by the code, or for high throughput applications, where the latency introduced may

reduce performance by preventing the rapid scheduling of jobs.

Chapter 7

Virtual Clusters in the Classroom

The vast amount of computational power offered by the current petascale, and future exascale, computing resources pose significant problems from both programming and system administration perspectives and is an active area of research. However, accessibility to courses that enable students to study High Performance Computing from an architecture, administration and technology platform perspective does not reflect this level of activity. Based on the experience at the University of Huddersfield in delivering the Parallel Computer Architectures module, one of the main challenges is in developing an approach that provides a suitable environment in order to achieve the learning outcomes without putting a production resource at risk. It is often necessary to implement specialised laboratories in order to support the computational hardware and software requirements of university courses that offer hand-on experience with building, maintaining and programming HPC systems, which poses administrative and sustainability limitations.

It is suggested that the VCC can be used to address these concerns. In this chapter, a case study is presented which implements the VCC in the classroom environment. It is designed to evaluate both the deployment of the full stack approach on one or more standalone machines, and the effectiveness of the virtualised clusters for the purpose of teaching and training. Sustainability, workflow, usability and attainment parameters form the basis of this evaluation and are compared with the existing OSCAR-based method used to deliver the module.

Firstly, the background of the module and implementation of the VCC into the teaching method is introduced. Secondly, the design of the data collection and processing activities are detailed in Section 7.3, followed by an evaluation of the results in Section 7.4. Finally, a discussion of the case study outcomes in relation to the research objectives, and the extent to which these outcomes are generalisable, are summarised.

7.1 Parallel Computer Architectures Module

There is a notable lack of undergraduate and postgraduate courses that address both usage and administration of HPC systems, given the proliferation of fast computing in science, as discussed in Chapter 1. One of the reasons for this, based on the experience at the University of Huddersfield, is the difficulty in providing appropriate computational resources; an environment is required that facilitates experiential learning, where the student is free to explore and interact with the system without putting a production cluster at risk.

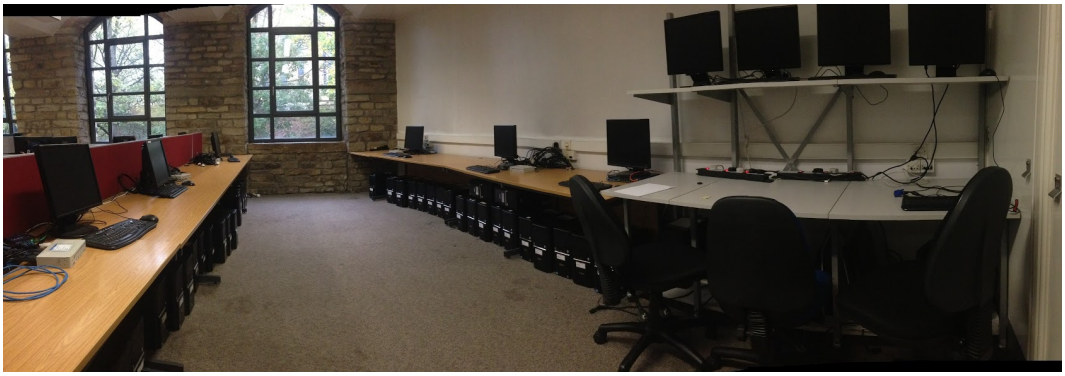
The PCA module at the University of Huddersfield aims to equip students with a comprehensive understanding of the practical aspects of cluster administration, program parallelisation and job management. It is offered to postgraduate and un-

dergradute students, in two separate classes that run for 12 or 22 weeks respectively (Higgins & Holmes, 2017). The learning outcomes are designed to fulfil this aim within the context of modern HPC environments and their application in scientific computing:

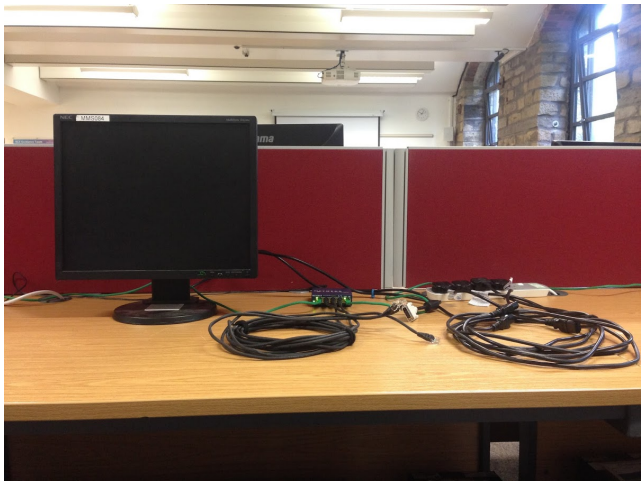
- Understand the demand for HPC systems and evolution of their development, in the context of speeding up applications.
- Gain knowledge about the fundamental components of cluster software environments, networking, middleware, resource management and parallel programming.
- Identify scalability issues in an HPC system by profiling the performance of parallel processing applications.

The underpinning administration concepts and programming models are presented through lectures, followed by practical work during the laboratory based sessions. The principle strategy for assessment is through project based learning. It is usually clear to students where the inefficiencies in their system lies, through benchmarking the constructed cluster. They must use this information to develop a critical evaluation of the performance of the designed system. In order to compare and contrast the laboratory cluster performance, the students are given access to the university research clusters. They are expected to write shell scripts, submit jobs for execution and analyse the outputs. In addition to project work, the final element of assessment is examination.

The course is designed around the Open Source Cluster Application Resources (OSCAR) toolkit, which is utilised by the students in order to build and test a cluster from scratch (Des Ligneris et al., 2003). The HPC laboratory is based on decommissioned workstations in order to provide the students with the hardware needed



(a) Lab



(b) Work area

Figure 7.1: Layout of the cluster laboratory

to build their own computer clusters, typically anywhere between 30 and 50 workstations in each cohort. In addition, each cluster requires supporting infrastructure such as network connectivity, an Ethernet switch for the cluster interconnect, a keyboard, video and mouse (KVM) console and the required cabling. The typical layout of the laboratory is shown in Figure 7.1.

However, this methodology suffers from hardware and software limitations: Firstly, the OSCAR software is no longer actively maintained (*OSCAR Homepage*, 2005).

This introduces compatibility problems with modern PC hardware and software, as an unmaintained Linux distribution is required in order to run it.

Secondly, the nature of the module activities dictate a high resource requirement per-student in order to construct several individual clusters. Currently this demand can be met due to the turnover of decommissioned systems, however, this strategy is unlikely to be sustainable in the long term or easily reproducible by other institutions, and offers limited scope for building clusters with a large number of nodes each.

Furthermore, while this method affords a high degree of hands-on experience, when problems are encountered, in some cases persisting over several weeks, the exploration and application of the theoretical aspects of the course is compromised. As the process of building a cluster is non-trivial, a large number of remediation steps are typically required, often reinstalling entire components of the cluster - this activity consumes a significant amount of time and does not contribute a representative amount of value to the learning experience.

Finally, a critical path exists in the cluster deployment workflow which means that a student must complete all steps of the cluster building phase successfully, otherwise they cannot complete the remaining programming activities even if they are capable. Instances of the critical path problem have been observed based on failures in both hardware and software environments (Higgins & Holmes, 2017). Therefore, a balance must be found between the practical and theoretical outcomes of the module.

It is suggested that the VCC can be used to resolve these limitations - providing an accessible virtual cluster environment that can be quickly created and destroyed on one or more standard laboratory PCs, regardless of the underlying machine and OS combination. Containers allow a higher density of virtual instances to be deployed per machine compared to VMs due to the lower execution overhead, thus, a large

number of virtualised cluster nodes can be created using a relatively small number of machines. This environment gives the user the same interactions as a physical cluster, and can be constructed to exactly replicate the environment of a production cluster provided by the institution, without requiring a dedicated laboratory or hardware resources. In addition, the use of container virtualisation and Dockerfile format allows a known good configuration to be packaged and distributed to students without obfuscating the steps necessary in order to create it. This offers a potential solution to the critical path problem.

In order to implement the VCC as a replacement for the OSCAR toolkit, a virtual cluster was built using the same middleware components and Torque/PBS resource manager as the university research cluster, using the process described in Chapter 4. Apart from the documentation and learning resources for cluster installation, no other supporting infrastructure changes are required by the implementation. The delivery and assessment aspects of the module - lectures, laboratory sessions and examination schedule - remain unchanged.

7.2 VCC on a Single Machine

The students have an option to deploy many instances of a worker node on the same physical machine, in order to increase the size of the virtual cluster beyond the hardware that is available in the lab. This topology is enabled by the container virtualisation technique, as the overhead is lower than traditional virtualisation techniques, thus, the density of instances can be higher. The typical RAM consumption of the container is 180 MB per instance, compared to the 2GB minimum system requirements of the CentOS distribution on which it is based. Therefore, even though this topology is not suitable for high performance execution, the student can expe-

Parameter	Activity	Data
Sustainability	Metrics	Hardware and software requirements
Workflow	Metrics	Number of steps to build cluster Time to first cluster
Usability	Survey	Perceived efficiency and effectiveness
Attainment	Survey	Skills Audit

Table 7.1: Classroom Case Study Measurable Objects and Activities

rience a wider variety of scheduling strategies, resource management scenarios and other interactions such as process launching, at a scale which would otherwise not be practical in the shared laboratory setting.

7.3 Methodology

An overview of the measurable objects and related activities in order to carry out this case study are detailed in Table 7.1. The same activities will be carried out against the VCC and OSCAR toolkits in order to draw a comparison. The data required for the sustainability and workflow parameters consists of metrics such as system requirements, number of steps to build the cluster and time to first cluster. No special methodology is required in order to collect this data. However, survey methods will be used in order to quantify the usability and attainment. An accepted technique needs to be used to conduct the survey in order to measure the usability in a meaningful way. A skills audit has also been designed in order to provide insight on attainment based on the students' own self-assessment.

7.3.1 Usability Survey

A common definition of usability is how "easy to use" a system is. However, this definition is not precise enough to gain an understanding of the user requirements and formulate a technique for its evaluation (Quesenbery, 2001). Usability is defined as "The extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use" by the International Organization for Standardization (ISO 9241-11, 1998). Thus, the usability does not consider the functionality or technical capabilities of a product, but the users' perception when performing the desired goals. An evaluation technique is required in order to gauge how easy to learn the system is, satisfaction and user attitudes towards the system (Quesenbery, 2001). In order to quantify these aspects, two techniques have been considered: System Usability Scale (SUS) (Brooke, 1996) and Computer System Usability Questionnaire (CSUQ) (Lewis, 1995).

The CSUQ is designed to assess the overall satisfaction of a user when interacting with computer systems (Lewis, 1995). It is a variant of the Post-Study System Usability Questionnaire (PSSUQ) which changes the wording of the questions to remove lab research specific terminology. The survey is composed of 16 statements for which the user must indicate their agreement based on a Likert-type scale of 1 to 7, representing strong agreement or disagreement respectively. It includes statements such as *"Whenever I make a mistake using the system, I recover easily and quickly"*. The CSUQ is able to diagnose some classes of usability problems, such as inadequate explanation of on-screen items or error messages (Sauro & Lewis, 2016). The results are characterised by a high level of scale reliability, with corresponding Cronbach's alpha typically exceeding 0.9 (Lewis, 1995).

The SUS is a subjective measure of the usability (Brooke, 1996). The survey is composed of 10 questions with responses graded on a scale of 1 to 5, representing

strong disagreement and agreement respectively. It is similarly characterised by high internal reliability without requiring a large number of participants, but can only recognise that a problem exists within the system, rather than diagnosing a specific area of usability that is problematic. However, the SUS has been shown to provide insight on two factors - usability and learnability - through statements such as *"I think that I would need the support of a technical person to be able to use this system"* (Lewis & Sauro, 2009).

While both the CSUQ and SUS surveys are regarded as appropriate techniques for the purpose of evaluating the usability of the VCC in a classroom environment, the SUS has been chosen as it is quick to administer, so the risk of response fatigue due to filling out multiple surveys is mitigated. In addition, it is demonstrated that despite rapid changes in technology since its inception, the SUS remains a technology agnostic and effective method in order to distinguish between usable and unusable systems (Bangor, Kortum, & Miller, 2008; Orfanou, Tselios, & Katsanos, 2015). Finally, there is a large body of published and unpublished test results from a variety of systems that allow a score to be converted into a percentile rank, and benchmarked against the usability of other systems evaluated with the SUS (Sauro & Lewis, 2016).

7.3.2 Skills Audit

A skills audit has been devised in order to capture the students' own self-assessment of proficiency, shown in Appendix. It assesses 12 key skills ranging from basic computing knowledge to advanced parallel computing topics:

1. Troubleshooting PC hardware
2. Measuring performance of a PC

3. Installing software from the internet
4. Compiling and installing software from source code
5. Basic networking (IP addressing, host names, ping)
6. Linux or Unix-like Operating Systems
7. Programming in C
8. Programming in any other language
9. Parallel programming
10. Code execution models (serial / parallel)
11. Parallelisation strategies
12. CPU level parallelism (Flynn's taxonomy)

The student must rate each skill based on a scale of 0 to 5, where a score of 0 represents no knowledge of the topic, and 5 represents an expert in that skill. The key skills are defined based on the learning outcomes of the module and the prerequisite knowledge required. It is expected that most students will be familiar with basic computing skills, and that the perceived knowledge regarding advanced topics will improve over the course of the module.

7.3.3 Procedure

The schedule for conducting each activity is outlined in Table 7.2. In the first week of the module, the skills audit was administered within both classes in order to establish a baseline for comparison with the second skills audit. It provides definitions of each point on the scale and instructs the student to read and understand them before completing the skills audit. In the 11th week (at the end of the first term), the second

Week	UG Class	PG Class
1	1st skills audit	1st skills audit
11	2nd skills audit SUS survey	2nd skills audit SUS survey
12		Examination
22	Examination	

Table 7.2: Timeline of Classroom Case Study Activities

skills audit was issued to both classes of students. At this point, both classes have gained similar experience with using the respective toolkits in order to build and test a cluster. Therefore, the SUS survey is also administered at this time before the students have reflected on their experience with the system. The survey is carried out in accordance with the method detailed by Sauro and Lewis (2016), where the instructions state that the student must not spend a long time considering each statement beyond an initial impression, and that the response should be marked in the middle of the scale if the student is unsure of their agreement or disagreement with a statement.

In the 16/17 cohort, the VCC toolkit was implemented in the undergraduate PCA class whilst the postgraduate class continued with the OSCAR method. In the 17/18 cohort, both undergraduate and postgraduate PCA classes migrated to the VCC framework as the cluster building method. Therefore, whilst the same procedure is followed for both cohorts, in the second cohort both SUS surveys report results only for the VCC.

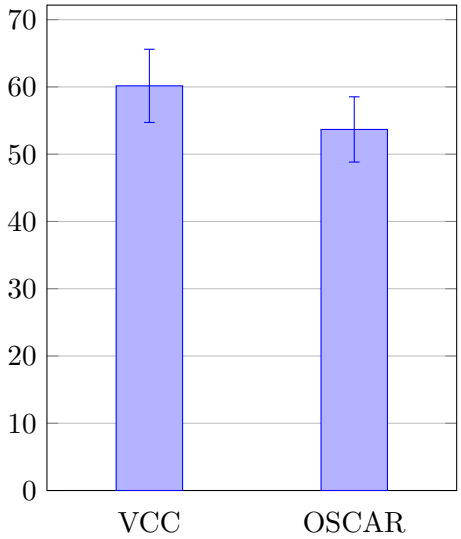


Figure 7.2: System Usability Scale Scores for VCC and OSCAR

7.4 Evaluation

In this section, the results and evaluation from each activity conducted as part of the case study are presented.

7.4.1 System Usability Scale

The SUS score results from both years of the study are summarised in Figure 7.2, including those from the 16/17 cohort published in Higgins and Holmes (2017). In total, there were 47 respondents to the VCC usability survey and 17 respondents to the OSCAR survey. The larger population for the VCC survey is due to student enrolment numbers in each class. As expected, the responses from the SUS surveys have good internal reliability: 0.78 to 0.88. However, one result was discarded due to having inconsistent responses, where the student agreed to both extremes of poor and excellent usability within the same survey. The unprocessed responses are detailed in Appendix.

		VCC	**OSCAR**
N		47	17
α		0.88	0.78
Mean SUS score		60.16	53.68
Std Dev		18.56	9.45
Margin of error (95% CL)		5.4	4.9
VCC > OSCAR p-value	0.036		
Effect Size d	0.44		

Table 7.3: Descriptive Statistics for SUS 2 Sample T-Test

At a glance, it can be seen that the usability score for the VCC is greater than OSCAR. A 2 Sample T-Test was conducted between the VCC and OSCAR results using the SUS Calculator Package version 1.42 (Sauro, 2011). A summary of the descriptive statistics is detailed in Table 7.3. The mean SUS score for VCC exceeds OSCAR by 7 points and the results demonstrate that this difference is statistically significant, rather than one which may occur by chance. The effect size measure of 0.44 gives an indication of the magnitude of this difference (Cohen, 1988). The effect size is categorised as "small", where 67% of the SUS scores for the VCC are above the mean score for OSCAR.

However, in order to determine the practical significance, the scores can be translated into a percentile rank in order to compare how usable the software is relative to other systems. A SUS benchmark has been used, based on a large database of over 5000 SUS results collected from various types of systems such as websites, business and consumer facing software, mobile applications and interactive voice applications (Sauro, 2011). Based on this benchmark, the average overall SUS score is 68. The VCC and OSCAR scores fall within the 28th and 16th percentile ranks respectively. The scores projected onto several grading scales are shown in Table 7.4.

It is clear that, while the VCC has greater usability than OSCAR, this increase is only marginally significant in a practical sense; both toolkits exhibit below average

	VCC	OSCAR
Percentile Rank (Sauro, 2011)	29th	17th
Adjective (Bangor, Kortum, & Miller, 2009)	OK	OK
Grade (Bangor et al., 2009)	D	F
Grade (Sauro & Lewis, 2016)	D	D

Table 7.4: Percentile Rank of SUS scores

		S_1	S_2	$S_2 - S_1$	S_2/S_1
Average overall skill	VCC	1.57	2.86	1.29	1.82
	OSCAR	2.30	3.26	0.96	1.42
Questions 9-12	VCC	0.44	2.16	1.72	4.89
	OSCAR	0.82	2.66	1.84	3.25

Table 7.5: Summary of skills audit results

usability, and in the context of a large number of products evaluated using the SUS, are graded within the same or adjacent level of usability.

7.4.2 Skills Audit

A summary of the skills audit results is presented in Table 7.5. The unprocessed data for the skills audit is detailed in Appendix. There is a positive change in the average across all skills between the first skills audit, S_1, and the second skills audit, S_2. This is also true when considering only the questions that relate specifically to the learning outcomes of the module (questions 9-12). Figures 7.3 and 7.4 display the results per question for both the VCC and OSCAR classes respectively.

As the postgraduate class used the OSCAR toolkit, it is expected that these students have experience greater than those in the undergraduate class, and this is reflected in a higher degree of proficiency for each skill during the first skills audit. However, whilst this means that the relative change (S_2/S_1) for OSCAR is lower, the actual

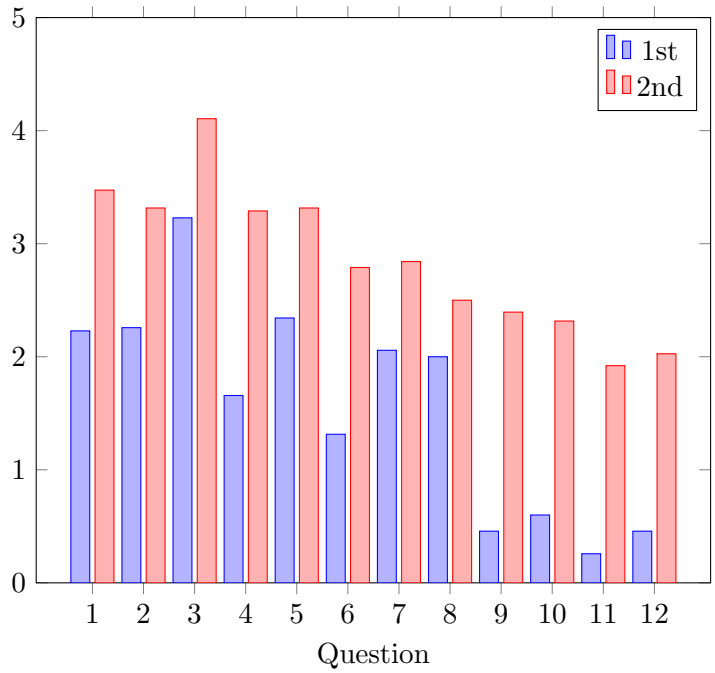


Figure 7.3: Combined skills audit results for VCC

difference in attainment ($S_2 - S_1$) when considering questions 9-12 is comparable. Therefore, from the perspective of the students' own self-assessment, it suggests that a similar level of knowledge was attained when using the VCC toolkit.

7.4.3 Workflow

A comparison between the workflows required to create a working cluster using each respective middleware is shown in Figure 7.5. Based on the VCC design, there are fewer steps in order to deploy the cluster, and this is reflected in practice. In the context of the skills audit and survey results, it is clear that the reduced number of steps contributes to improving the usability and reducing the number of problems encountered during the cluster building phase of the course, without compromising the learning outcomes.

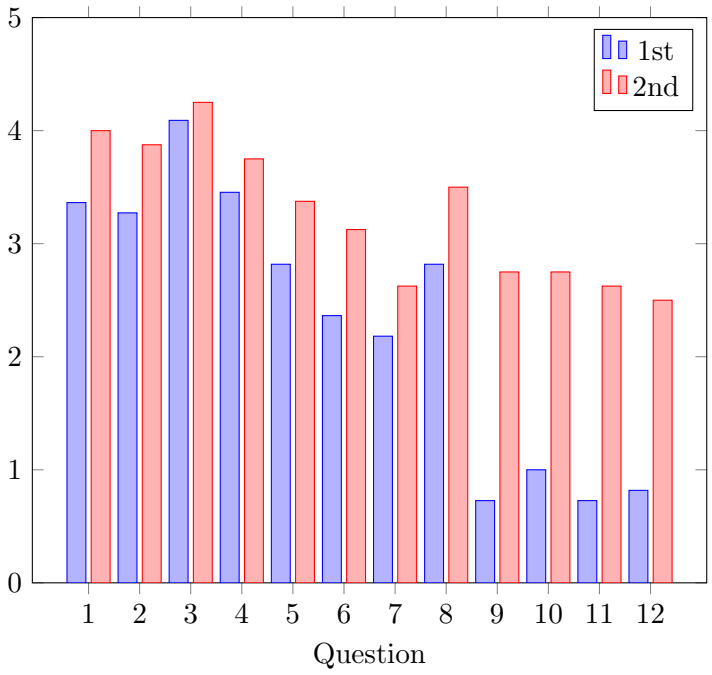
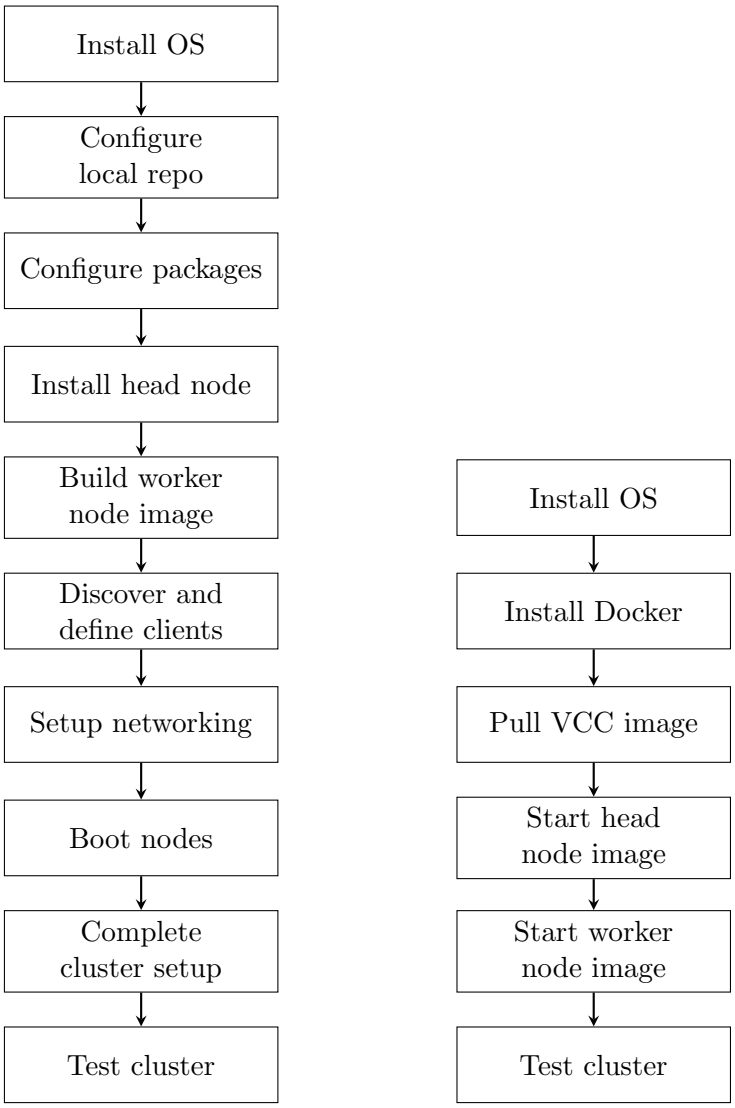


Figure 7.4: Combined skills audit results for OSCAR

In addition to reducing the number of steps to build the cluster, the streamlined workflow of the VCC enabled students to create a working environment earlier - week 2 of the undergraduate class compared to week 4 in the postgraduate class. This provides a more balanced workload for the student in terms of cluster building, parallel programming and evaluation, and offers a solution to the critical path problem identified in the previous teaching method.

7.4.4 System Requirements

The system requirements of the VCC framework compared to OSCAR provide improved sustainability and better compatibility with modern Operating System versions. For example, whilst OSCAR requires an old Red Hat derived Linux distribution, the VCC supports any OS which can run the Docker runtime, including Microsoft Windows. This means that standard PC laboratory equipment can be



(a) OSCAR (b) VCC

Figure 7.5: Workflow comparison of OSCAR and VCC

used, and as this equipment is upgraded on the regular cycle, the VCC will continue to support it.

Furthermore, the original teaching method required the student to build a cluster composed of 1 head node and 2 worker nodes. Using the VCC, this system can be virtualised entirely on a single node, requiring approximately 180 MB of RAM per instance. This has the potential to support large environments on a single machine, comfortably fitting 64 nodes on a typical machine with 16 GB of RAM. Whilst not all students utilised this capability during the course of this case study, it provides an option for future development of the course in order to introduce more complex scheduling problems and system administration tasks at scale.

7.5 Summary

In this case study, the VCC was applied in a classroom setting in order to address deployment on a new topology which represents a single or small number of machines. It was used for the 15/16 and 16/17 academic years in order to deliver the Parallel Computer Architectures course at the University of Huddersfield. The usability was assessed using the System Usability Scale (SUS) and compared to the previous teaching method using the OSCAR middleware. The usability of the VCC falls within the 29th percentile, compared to 17th for OSCAR, benchmarked against a large body of SUS results from a variety of information technology systems. Whilst the usability is demonstrably higher, the practical significance of the increased usability has minimal impact: both solutions exhibit lower than average usability. The skills audit suggests that there is an inherent difficulty in the subject, where both undergraduate and postgraduate students attained a similar level of understanding in the same HPC-specific topics. In addition, the workflow demonstrates that, whilst

there are less steps to deploy and configure a working cluster using the VCC, the high level processes are similar. These two factors contribute to both the usability being relatively low, and the two systems being similar in a practical sense.

Despite this, it is clear that the VCC framework is a sustainable solution for delivering the course in the future: not only was it sufficient to meet the learning outcomes whilst being perceived to be slightly easier to use, the single machine topology and system requirements enable it to be used on a variety of systems and platforms that are already existing in typical IT laboratories in the university setting.

Chapter 8

Conclusion

This research addresses the implementation of full stack cluster environments within containers. Traditionally, Virtual Machines (VMs) have required a full software environment by design, including an OS, middleware and management software, in addition to the application which the user wishes to run. The VM poses several challenges in the HPC context: they are hard to use, large filesystem images require a lot of disk space at rest and high bandwidth while being transferred, and there is a potential performance overhead depending on the application. Containers provide a virtualisation technique that is more accessible to the average user, as they only require the definition of the top-most layer of the software environment - the run time layer. The container can be orchestrated within the system similar to non-virtualised applications, reducing the disruption to the normal workflow and patterns, and is defined as a composition of text files describing each layer of the software environment which can be easily version controlled and shared. As no machine emulation takes place during the execution of a container, the execution performance of the virtualised processes is equivalent to the non-virtualised performance. The primary motivation for implementing run time environment virtualisation is to improve the

portability of a particular application.

However, the full stack VM approach offers more benefits than just portability; as it represents a full cluster environment that is independent from the host system, it offers the ability to instantiate that environment regardless of the underlying system and it's capabilities. Therefore, in the context of a group of integrated resources, such as in a grid, it enables the implementation of meta-scheduling features in order to transfer the entire burden of a job's execution to another system - without any uncertainty in terms of compatibility - and to combine idle capacity among many resources to execute the job in a spanning mode. Previous work has demonstrated that these two capabilities offer an improvement in global resource performance, by reducing the job turnaround time and increasing cluster utilization. Despite this, the full stack solutions are not without limitation: the literature does not adequately address the performance implications of introducing an inter-cluster interconnect, which is required to facilitate job forwarding and spanning. Nonetheless, these capabilities cannot be implemented using the run time environment container techniques that have gained popularity within HPC communities.

Furthermore, a comprehensive review of existing virtualisation solutions reveals that even though virtualised environments are inherently portable due to the abstraction provided by the VM or container technique, in the HPC context they rely on complex orchestration strategies which are not. The net result is that, while virtual environments offer flexibility within the boundary of a small collection of systems, they are not fit for general purpose, reusable, dynamic HPC deployment.

Accordingly, the aim of this research was to develop a novel, container-based full stack virtualisation solution, which will enable the capabilities to improve global resource management and software environment flexibility, whilst inheriting the improved accessibility and performance of the modern container virtualisation tech-

nique. This thesis contributes a new, innovative design for a full stack virtualised cluster environment, the *Virtual Container Cluster*, which satisfies the requirements of a general purpose solution synthesized from the analysis of existing work.

This approach is addressing the shortcomings of the existing virtualisation systems within HPC. The following conclusions are drawn based on the objectives and insight gained from this research:

Improved Portability of Virtual Environments

It is identified that the full stack virtualisation approach is essential in order to avoid compromising the portability, where the application must be shipped along with the middleware and management stack required to execute it. However, the level of system administration knowledge required to operate the VM, and the reliance on external services to perform the configuration, are found to pose the most significant barriers to entry.

In order to resolve this limitation, the VCC design provides a unique, self-contained configuration mechanism which removes the requirement for the user to perform contextual specific adaptations to the rest of the software stack when running a container. The gap that this fills is two-fold; firstly, it improves the portability by allowing the full stack container to be standalone from the underlying system and any external services, and secondly, it lowers the barrier to entry by maintaining the accessibility of the container virtualisation technique so that it can be operated by the average user.

The portability of the VCC is evidenced in the performance benchmarking and case studies, where the same image is deployed across a variety of cluster, grid and cloud systems.

Containers can be used to realise benefits of meta-scheduling

Using an established experimental methodology, this research demonstrates that containers can be used to realise the benefits of meta-scheduling in the context of a campus grid. By utilising idle capacity among a group of clusters in order to run an additional job, the utilization of each individual cluster is increased, and the queue turnaround time can be decreased despite the performance cost of spanned execution. Unlike VMs, where the performance cost results from the virtualisation technique, the inter-cluster interconnect is demonstrated to be the limiting factor.

The Software Defined Network approach does not require a large number of participants, and thus a diverse number of connections, in order to achieve acceptable performance which was demonstrated to scale from 300 meters on the campus grid, to 48 km (30 mi) between two institutions. In the campus grid, it was identified based on 12 months of historical data that there were many opportunities to exploit job forwarding and spanning. In practice, this resulted in a 41% faster queue turnaround time - effectively combining the idle resource in both clusters without any changes to the underlying connectivity or increased administrative cost. Between institutions, utilizing just job forwarding resulted in a 36% faster queue turnaround time - without implementing any specialised network infrastructure or changes to the host software environment.

Latency is a key factor in spanning performance

Previous work considers the bandwidth requirements of the inter-cluster interconnect, but often neglects the latency characteristics of network connections at this scale. This thesis establishes the acceptable range of latency for spanning an appli-

cation execution across an inter-cluster interconnect when loaded with typical HPC workloads: a compute bound benchmark (Linpack) and a network-dependent CFD code (OpenFOAM).

At latencies up to 10ms, which would not typically be considered appropriate for HPC execution, the execution time of the Linpack benchmark is not significantly extended. At 5ms, the wall time of the OpenFOAM benchmark was approximately 2x slower, increasing to 3.5x slower at 10ms. To achieve a similar slow down on the Linpack benchmark, the latency must be increased to well beyond 100ms. Based on the historical workload data, almost half of the opportunities to exploit spanning would still have resulted in a turnaround time equal to or less than the original queue wait time plus execution time, even with a 3x performance cost.

Virtual cluster is an effective environment for teaching

The VCC was used to deliver the Parallel Computer Architectures course for the 15/16 and 16/17 academic years. A usability survey was conducted and compared to the original teaching method using the OSCAR middleware. It showed that the usability of the VCC is comparable and in some respects better than the OSCAR tool, however, the inherent difficulty of the subject matter means that both systems exhibit below average usability. Furthermore, a skills audit demonstrated that the student attainment using the VCC was sufficient to meet the learning outcomes of the course.

Full stack containers can implement novel topologies

Two novel topologies are successfully implemented using the full stack container which are not feasible with traditional virtualisation techniques: nested clusters

and single machine clusters. The spanning case study demonstrated that containers can be nested in order to gradually refine the software environment. This provides portability and flexibility, in a unified solution that retains the accessible nature of container virtualisation.

The teaching case study demonstrated that a large cluster can be virtualised on a single machine, and whilst clearly this is not suitable for high performance execution, it is a useful traning environment. Therefore, the user can gain experience with resource management, job submission and the other activities that characterise an HPC system, in a realistic environment that does not put the production system at risk.

The novelty of the work presented in this thesis is demonstrated in the design of the framework, and the ability to deploy it in environments not previously possible with traditional virtualisation approaches in HPC. The usefullness of the solution is validated through case studies in two applications - grid and classroom - and the resulting impact of this research in these applications is clear.

Firstly, the VCC framework is now being used at the University of Huddersfield in order to deliver the Parallel Computer Architectures module. It provides a future proof platform for teaching HPC and cluster computing concepts, in a realistic environment that can be constructed by the student with minimal prior experience, that can be kept up to date and in line with the current trends in HPC software environments. The hands-on approach provides a seamless transition from the training environment to the production resource. This will ensure the sustainability of delivering these courses, and contribute to maintaining the increase in student recruitment and generation of research projects that stem from students studying the course.

Secondly, it facilitates accessible integration and sharing of computational resources without a high administrative or knowledge burden. This supports collaborations between institutions or departments without the complexity of establishing grid infrastructure, and the associated responsibility to maintain Public Key Infrastructure, Single System Image software stack and public facing network endpoints. On a functional level, grid-like infrastructures allow sharing of computational resources, but they also empower cross-domain transfer of knowledge, skills and techniques. This research demonstrates that the VCC can facilitate the same kind of collaboration whilst keeping the technical overhead low. In the same way that the EPSRC investment aims to encourage horizontal integration between Tier-2 assets, a novel framework is provided to achieve this in the long tail of science, at Tier-3 and below. There is a potential to provide significant impact by enabling horizontal intergration between these resources which are otherwise disparate, offering an alternative to scaling vertically to regional and national HPC systems. For example, on research projects where budget or time constraints make the integration with traditional grid technologies prohibitive to implement, the VCC can be used to faciliate resource sharing. The case study carried out between HE and FE institutions demonstrates that even with poor inter-cluster connectivity, a valuable contribution to a shared resource can still be made by offering job forwarding capabilities. This was demonstrated when attempting to run the VCC on Tier 1 and 2 resources, which did not support the containerization technology required to conduct the study at scale. In this case, the only option was to scale horizontally between willing Tier 3 facilities, and indeed, institutions that are not part of the HPC landscape at all.

Finally, the community developing around the software demonstrates that is it fulfilling the need for a flexible system framework that is able to accomodate a wide variety of applications and requirements. In addition to the refereed conference and journal publications detailed in Appendix, the project has had over 600 downloads

on Docker Hub and GitHub, has been forked 3 times with 2 unique contributions in order to improve the code, and presented at various workshops throughout the UK. Therefore, practitioners in the field have been engaged at every stage of this research in order to inform, continuously validate and maintain the currency of this work, establishing its novelty and usefulness within the HPC community.

Chapter 9

Future Work

The work in this thesis has been published as an open source project, freely available to the community, and will be maintained through this collaborative framework. A richer library of base images will be created in order to reflect the more diverse range of systems that modern HPC encompasses, in addition to the traditional PBS batch scheduling system demonstrated in this work.

The current petascale class systems are composed from large, expensive, nationally supported supercomputer implementations, which may be out of reach of the average researcher or institution. However, the summation of many smaller, distributed systems has the potential to reach the same level of performance and utility. The VCC is not a disruptive technology from the implementation perspective, promising to unify Tier 3 and below resources without intrusive actions on existing production systems. Therefore, the first barrier in order to realise this alternative view of the HPC estate is being overcome. A future direction for this research is to investigate the remaining barriers to enable a distributed VCC meta-cluster with performance akin to contemporary petascale systems. In the same way that virtualisation enables capabilities that can offset the performance cost, the geographical distribution of

processing, power, cooling and network transit enabled by the VCC will need to be evaluated.

Finally, it is clear that even when it is desirable to execute a job over a slower, spanned inter-cluster interconnect - in order to improve the global resource performance among a group of clusters - it may not be possible if the code is not robust to adverse network conditions. Mechanisms in order to improve the fault tolerance of parallel process launching and communication when operating in high latency network environments will be investigated, to enable a wider range of network-bound applications to take advantage of the transparent VCC job spanning capabilities, in addition to the application and synthetic benchmarks demonstrated in this work.

References

Archer. (n.d.). EPCC. Retrieved from `http://www.archer.ac.uk/` (Accessed: 2018-09-01)

Bailey, D. H., Barszcz, E., Barton, J. T., Browning, D. S., Carter, R. L., Dagum, L., ... Schreiber, R. S. (1991). The NAS parallel benchmarks. *The International Journal of Supercomputing Applications*, *5*(3), 63–73.

Bangor, A., Kortum, P., & Miller, J. (2009). Determining what individual SUS scores mean: Adding an adjective rating scale. *Journal of Usability Studies*, *4*(3), 114–123.

Bangor, A., Kortum, P. T., & Miller, J. T. (2008). An Empirical Evaluation of the System Usability Scale. *Intl. Journal of Human–Computer Interaction*, *24*(6), 574–594.

Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., ... Warfield, A. (2003). Xen and the Art of Virtualization. *ACM SIGOPS Operating Systems Review*, *37*(5), 164–177.

Birkenheuer, G., Brinkmann, A., Kaiser, J., Keller, A., Keller, M., Kleineweber, C., ... Wilhelm, M. (2012). Virtualized HPC: a contradiction in terms? *Software: Practice and Experience*, *42*(4), 485–500.

Boettiger, C. (2015). An introduction to docker for reproducible research. *ACM SIGOPS Operating Systems Review*, *49*(1), 71–79.

Braden, R. (1989, October). *Requirements for Internet hosts-application and support* (RFC No. 1123). Internet Requests for Comments. Retrieved from https://www.rfc-editor.org/rfc/rfc1123.txt (Accessed: 2018-09-02)

Brooke, J. (1996). SUS: A 'quick and dirty' usability scale. In P. W. Jordan, B. Thomas, I. L. McClelland, & B. Weerdmeester (Eds.), *Usability Evaluation in Industry* (pp. 189–194). CRC Press, LLC.

Buyya, R., Cortes, T., & Jin, H. (2001). Single system image. *The International Journal of High Performance Computing Applications*, *15*(2), 124–135.

Carnes, B. (2002, Apr). *The ASCI Presta Stress Benchmark code.* LLNL. Retrieved from https://asc.llnl.gov/computing_resources/purple/archive/benchmarks/presta/

CentOS Linux 5 EOL. (2017, Apr). CentOS-announce Mailing List. Retrieved from https://lists.centos.org/pipermail/centos-announce/2017-April/022350.html

Chamberlain, R., & Schommer, J. (2014, Jul). *Using Docker to Support Reproducible Research.* Retrieved from https://figshare.com/articles/Using_Docker_to_Support_Reproducible_Research/1101910/1 doi: 10.6084/m9.figshare.1101910.v1

Cohen, J. (1988). Statistical power analysis for the behavioural sciences.

Configuring TORQUE on compute nodes. (2012). Adaptive Computing. Retrieved from http://docs.adaptivecomputing.com/torque/4-1-3/Content/topics/1-installConfig/configOnComputeNodes.htm (Accessed: 2018-08-28)

Consul by HashiCorp. (n.d.). Retrieved from https://www.consul.io/ (Accessed: 2018-09-01)

Content trust in Docker. (2018, Jun). Retrieved from https://docs.docker.com/engine/security/trust/content_trust/ (Accessed: 2018-09-01)

Creasy, R. J. (1981). The origin of the VM/370 time-sharing system. *IBM Journal of Research and Development*, *25*(5), 483–490.

de Alfonso, C., Calatrava, A., & Moltó, G. (2017). Container-based virtual elastic clusters. *Journal of Systems and Software*, *127*, 1–11.

Des Ligneris, B., Scott, S. L., Naughton, T., & Gorsuch, N. (2003). Open Source Cluster Application Resources (OSCAR): design, implementation and interest for the scientific community. In *Proceeding of 17th Annual International Symposium on High Performance Computing Systems and Applications (HPCS 2003)* (pp. 241–246).

DiRAC. (n.d.). Science and Technology Facilities Council. Retrieved from `https://dirac.ac.uk/` (Accessed: 2018-08-23)

Dixit, K. M. (1991). The SPEC benchmarks. *Parallel computing*, *17*(10-11), 1195–1209.

Dongarra, J. J., Luszczek, P., & Petitet, A. (2003). The LINPACK benchmark: past, present and future. *Concurrency and Computation: practice and experience*, *15*(9), 803–820.

Elghraoui, A. (2017, Nov). *NIH-HPC/singularity-examples.* National Institutes of Health. Retrieved from `https://singularity-hub.org/collections/267` (Accessed: 2018-08-23)

Emeneker, W., Jackson, D., Butikofer, J., & Stanzione, D. (2006). Dynamic Virtual Clustering with Xen and Moab. In G. Min, B. Di Martino, L. T. Yang, M. Guo, & G. Rünger (Eds.), *Frontiers of High Performance Computing and Networking – ISPA 2006 Workshops* (pp. 440–451). Springer Berlin Heidelberg.

Emeneker, W., & Stanzione, D. (2006, Sept). HPC cluster readiness of Xen and User Mode Linux. In *2006 IEEE International Conference on Cluster Computing* (pp. 1–8). doi: 10.1109/CLUSTR.2006.311870

Emeneker, W., & Stanzione, D. (2007, Sept). Dynamic Virtual Clustering. In *2007 IEEE International Conference on Cluster Computing* (p. 84-90). doi: 10.1109/CLUSTR.2007.4629220

EPSRC strategy for the developing landscape of Tier-2 HPC in the UK. (n.d.). Engineering and Physical Sciences Research Council. Retrieved from `https://epsrc.ukri.org/files/research/tier2hpcstrategy/` (Accessed: 2018-09-01)

Fast Datapath and Weave Net. (n.d.). Weaveworks. Retrieved from `https://www.weave.works/docs/net/latest/concepts/fastdp-how-it-works/` (Accessed: 2018-09-02)

Felter, W., Ferreira, A., Rajamony, R., & Rubio, J. (2014). IBM Research Report: An Updated Performance Comparison of Virtual Machines and Linux Containers. *IBM Research Division, Austin Research Laboratory*.

Figueiredo, R. J., Dinda, P. A., & Fortes, J. A. (2003). A case for grid computing on virtual machines. In *23rd International Conference on Distributed Computing Systems, 2003. Proceedings.* (pp. 550–559).

Fink, J. (2014). Docker: a software as a service, operating system-level virtualization framework. *Code4Lib Journal*, *25*.

Foster, I., Freeman, T., Keahy, K., Scheftner, D., Sotomayer, B., & Zhang, X. (2006). Virtual Clusters for Grid Communities. In *Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06)* (Vol. 1, pp. 513–520).

Gable, G. (1994, Jan). Integrating case study and survey research methods: an example in information systems. *European Journal of Information Systems*, *3*(2), 112–126. doi: 10.1057/ejis.1994.12

Geimer, M., Hoste, K., & McLay, R. (2014). Modern scientific software management using EasyBuild and Lmod. In *Proceedings of the First International Workshop*

on HPC User Support Tools (pp. 41–51). IEEE Press. doi: 10.1109/HUST .2014.8

Ghosh, S. (2017, Feb). *satra/om-images.* Retrieved from `https://singularity-hub.org/collections/87` (Accessed: 2018-08-23)

Gong, Y., He, B., & Zhong, J. (2015). Network performance aware MPI collective communication operations in the cloud. *IEEE Transactions on Parallel and Distributed Systems*, *26*(11), 3079–3089.

Hanenberg, S. (2010, October). Faith, Hope, and Love: An Essay on Software Science's Neglect of Human Factors. *SIGPLAN Not.*, *45*(10), 933–946. doi: 10.1145/1932682.1869536

Hess, B., Kutzner, C., Van Der Spoel, D., & Lindahl, E. (2008). GROMACS 4: algorithms for highly efficient, load-balanced, and scalable molecular simulation. *Journal of Chemical Theory and Computation*, *4*(3), 435–447.

Higgins, J., & Holmes, V. (2017). Teaching parallel and distributed computing with container virtualization. In *Proceedings of EduHPC-17: Workshop on Education for High-Performance Computing at SC17.*

Higgins, J., Holmes, V., & Venters, C. (2015). Orchestrating Docker Containers in the HPC Environment. In J. M. Kunkel & T. Ludwig (Eds.), *ISC High Performance 2015: High Performance Computing* (pp. 506–513). Springer International Publishing.

Higgins, J., Holmes, V., & Venters, C. (2016, July). Securing user defined containers for scientific computing. In *2016 International Conference on High Performance Computing Simulation (HPCS)* (pp. 449–453). doi: 10.1109/HPCSim.2016.7568369

Higgins, J., Holmes, V., & Venters, C. (2017a). Autonomous Discovery and Management in Virtual Container Clusters. *The Computer Journal*, *60*(2), 240-252. doi: 10.1093/comjnl/bxw102

Higgins, J., Holmes, V., & Venters, C. (2017b, March). VCC: A framework for building containerized reproducible cluster software environments. *The Journal of Open Source Software*, *2*(11). doi: 10.21105/joss.00208

Higgins, J., Holmes, V., & Venters, C. (2017c, March). *VCC: A framework for building containerized reproducible cluster software environments (code repository).* doi: 10.6084/m9.figshare.4763857.v1

Holmes, V., & Kureshi, I. (2015). Developing High Performance Computing Resources for Teaching Cluster and Grid Computing Courses. *Procedia Computer Science*, *51*, 1714–1723.

Huang, W., Liu, J., Abali, B., & Panda, D. K. (2006). A Case for High Performance Computing with Virtual Machines. In *Proceedings of the 20th Annual International Conference on Supercomputing* (pp. 125–134). ACM. doi: 10.1145/1183401.1183421

ISO 9241-11 Ergonomic Requirements for Office Work with Visual Display Terminals (VDTs): Part 11: Guidance on Usability (Vol. 1998; Standard). (1998). International Organization for Standardization.

Jacobsen, D. M., & Canon, R. S. (2015). Contain this, unleashing Docker for HPC. *Proceedings of the Cray User Group*.

Janet network. (n.d.). Retrieved from `https://www.jisc.ac.uk/janet` (Accessed: 2018-09-02)

Jasak, H. (2009). OpenFOAM: open source CFD in research and industry. *International Journal of Naval Architecture and Ocean Engineering*, *1*(2), 89–94.

Jette, M., & Auble, D. (2010). SLURM: Resource Management from the Simple to the Sophisticated. In *Lawrence Livermore National Laboratory, SLURM User Group Meeting.*

Keahey, K., Doering, K., & Foster, I. (2004). From Sandbox to Playground: Dynamic Virtual Environments in the Grid. In *Proceedings of the 5th*

IEEE/ACM International Workshop on Grid Computing (pp. 34–42). doi: 10.1109/GRID.2004.32

Keahey, K., Foster, I., Freeman, T., Zhang, X., & Galron, D. (2005). Virtual Workspaces in the Grid. In J. C. Cunha & P. D. Medeiros (Eds.), *Euro-Par 2005 Parallel Processing* (pp. 421–431). Springer Berlin Heidelberg.

Keahey, K., & Freeman, T. (2008). Contextualization: Providing one-click virtual clusters. In *2008 IEEE Fourth International Conference on eScience* (pp. 301–308).

Keahey, K., Freeman, T., Lauret, J., & Olson, D. (2007). Virtual workspaces for scientific applications. In *Journal of Physics: Conference Series* (Vol. 78, p. 012038).

Kivity, A., Kamay, Y., Laor, D., Lublin, U., & Liguori, A. (2007). KVM: the Linux virtual machine monitor. In *Proceedings of the Linux Symposium* (Vol. 1, pp. 225–230).

Kniep, C. (2014). Containerization of High Performance Compute Workloads using Docker. Retrieved from `http://doc.qnib.org/2014-11-05_Whitepaper_Docker-MPI-workload.pdf`

Kniep, C. (2016). *Multi-host containerised hpc cluster.* Retrieved from `https://archive.fosdem.org/2016/schedule/event/hpc_bigdata_hpc_cluster/` (Presented at FOSDEM 2016)

Kolyshkin, K. (2006). Virtualization in Linux. *OpenVZ Whitepaper*. Retrieved from `https://download.openvz.org/doc/openvz-intro.pdf`

Kurtzer, G. (2016). *Singularity 2.1.2 - Linux application and environment containers for science.*

Kurtzer, G. M., Sochat, V., & Bauer, M. W. (2017). Singularity: Scientific containers for mobility of compute. *PloS one*, *12*(5), e0177459.

Kutch, P. (2011). PCI-SIG SR-IOV primer: An introduction to SR-

IOV technology. *Intel application note*, 321211–002. Retrieved from https://www.intel.sg/content/dam/doc/application-note/pci-sig-sr-iov-primer-sr-iov-technology-paper.pdf

Le, E., & Paz, D. (2017). Performance Analysis of Applications using Singularity Container on SDSC Comet. In *Proceedings of the Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact* (p. 66).

Lewis, J. R. (1995). IBM computer usability satisfaction questionnaires: Psychometric evaluation and instructions for use. *International Journal of Human–Computer Interaction*, 7(1), 57-78. doi: 10.1080/10447319509526110

Lewis, J. R., & Sauro, J. (2009). The Factor Structure of the System Usability Scale. In M. Kurosu (Ed.), *Human Centered Design* (pp. 94–103). Springer Berlin Heidelberg.

Li, P. (2010). Selecting and using virtualization solutions: our experiences with VMware and VirtualBox. *Journal of Computing Sciences in Colleges*, *25*(3), 11–17.

Liu, J., Huang, W., Abali, B., & Panda, D. K. (2006). High Performance VMM-bypass I/O in Virtual Machines. In *Proceedings of the Annual Conference on USENIX '06 Annual Technical Conference* (pp. 3–3). USENIX Association.

Ludovici, F., & Pfeifer, H. P. (2011, Nov). tc-netem(8) - Linux manual page [Computer software manual].

Luszczek, P., Bailey, D. H., Dongarra, J. J., Kepner, J., Lucas, R. F., Rabenseifner, R., & Takahashi, D. (2006). The HPC Challenge (HPCC) benchmark suite. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing* (p. 213).

Luszczek, P., Meek, E., Moore, S., Terpstra, D., Weaver, V. M., & Dongarra, J. (2012). Evaluation of the HPC Challenge Benchmarks in Virtualized Environments. In M. Alexander et al. (Eds.), *Euro-Par 2011: Parallel Processing*

Workshops (pp. 436–445). Springer Berlin Heidelberg.

Merkel, D. (2014). Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, *2014*(239), 2.

The MIT license. (2006). Retrieved 2018-02-18, from `https://opensource.org/licenses/MIT`

Morgua, A. (2015). *Nesting Containers: Real Life Observations.* (Presented at DockerCon EU 2015)

Musleh, M., Pai, V., Walters, J. P., Younge, A., & Crago, S. (2014). Bridging the virtualization performance gap for HPC using SR-IOV for InfiniBand. In *2014 IEEE 7th International Conference on Cloud Computing* (pp. 627–635).

Nussbaum, L., Anhalt, F., Mornard, O., & Gelas, J.-P. (2009). Linux-based virtualization for HPC clusters. In *Montreal Linux Symposium.*

OpenFOAM Build Guide. (n.d.). OpenCFD Ltd. Retrieved from `https://www.openfoam.com/code/build-guide.php` (Accessed: 2018-09-02)

Open Grid Scheduler/Grid Engine. (n.d.). `http://gridscheduler.sourceforge.net/`. (Accessed: 2018-09-02)

Orfanou, K., Tselios, N., & Katsanos, C. (2015). Perceived usability evaluation of learning management systems: Empirical evaluation of the system usability scale. *The International Review of Research in Open and Distributed Learning*, *16*(2).

OSCAR homepage. (2005, Apr). Retrieved from `http://www.csm.ornl.gov/oscar/` (Accessed: 2018-09-02)

Overview of Docker Hub. (n.d.). Docker Documentation. Retrieved from `https://docs.docker.com/docker-hub/` (Accessed: 2018-08-23)

Priedhorsky, R., & Randles, T. (2017). Charliecloud: Unprivileged containers for user-defined software stacks in HPC. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*

(pp. 36:1–36:10). doi: 10.1145/3126908.3126925

Quesenbery, W. (2001). What Does Usability Mean: Looking Beyond 'Ease of Use'. In *Proceedings of the 48th Annual Conference, Society for Technical Communication* (Vol. 48, pp. 432–436).

Rabenseifner, R., Koniges, A. E., & Livermore, L. (2001). The Parallel Communication and I/O Bandwidth Benchmarks: b_eff and b_eff_io. In *Proc. of 43rd cray user group conference, indian wells, california, usa.*

Richling, S., Hau, S., Kredel, H., & Kruse, H.-G. (2011). Operating two InfiniBand grid clusters over 28 km distance. *International Journal of Grid and Utility Computing*, *2*(4), 303–312.

Rodríguez-Haro, F., Freitag, F., Navarro, L., Hernánchez-sánchez, E., Farías-Mendoza, N., Guerrero-Ibáñez, J. A., & González-Potes, A. (2012). A summary of virtualization techniques. *Procedia Technology*, *3*, 267–272.

The R Project for Statistical Computing. (n.d.). The R Foundation. Retrieved from `https://www.r-project.org/` (Accessed: 2018-08-23)

Runeson, P., & Höst, M. (2009). Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, *14*(2), 131.

Sauro, J. (2011). *A Practical Guide to the System Usability Scale: Background, Benchmarks & Best practices.* Measuring Usability LLC.

Sauro, J., & Lewis, J. R. (2016). *Quantifying the user experience: Practical statistics for user research.* Morgan Kaufmann.

Shiratori, T. (2017, Dec). *TakahisaShiratori/openfoam.* Retrieved from `https://singularity-hub.org/collections/386` (Accessed: 2018-08-23)

Sochat, V. (2017, Oct). Singularity Registry: Open Source Registry for Singularity Images. *The Journal of Open Source Software*, *2*(18), 426. doi: 10.21105/joss.00426

Soltesz, S., Pötzl, H., Fiuczynski, M. E., Bavier, A., & Peterson, L. (2007).

Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors. In *ACM SIGOPS Operating Systems Review* (Vol. 41, pp. 275–287).

Sotiriadis, S., Bessis, N., Xhafa, F., & Antonopoulos, N. (2012). From meta-computing to interoperable infrastructures: A review of meta-schedulers for HPC, grid and cloud. In *2012 IEEE 26th International Conference on Advanced Information Networking and Applications(AINA)* (pp. 874–883).

StarCluster. (n.d.). `http://star.mit.edu/cluster/`.

StarCluster Plugin Documentation. (2011). Retrieved from `http://star.mit.edu/cluster/docs/latest/plugins/index.html` (Accessed: 2015-04-14)

Sterling, T., Anderson, M., & Brodowicz, M. (2017). *High Performance Computing: Modern Systems and Practices.* Morgan Kaufmann.

Tikotekar, A., Vallée, G., Naughton, T., Ong, H., Engelmann, C., & Scott, S. L. (2009). An Analysis of HPC Benchmarks in Virtual Machine Environments. In E. César et al. (Eds.), *Euro-Par 2008 Workshops - Parallel Processing* (pp. 63–71). Springer Berlin Heidelberg.

Top500 FAQ. (n.d.). Top500.org. Retrieved from `https://www.top500.org/resources/frequently-asked-questions/` (Accessed: 2019-01-10)

Turilli, M., Santcroos, M., & Jha, S. (2018). A comprehensive perspective on pilot-job systems. *ACM Computing Surveys (CSUR)*, *51*(2), 43.

Using etcd. (n.d.). Red Hat, Inc. Retrieved from `https://coreos.com/etcd/` (Accessed: 2018-09-02)

Using Host libraries: GPU drivers and OpenMPI BTLs. (2017, May). Retrieved from `http://singularity.lbl.gov/tutorial-gpu-drivers-open-mpi-mtls` (Accessed: 2018-08-23)

Vallée, G., Naughton, T., Ong, H., Tikotekar, A., Engelmann, C., Bland, W., ... Scott, S. L. (2008). Virtual System Environments. *Systems and Virtualization*

Management. Standards and New Technologies, *18*, 72–83.

Vallée, G., Naughton, T., & Scott, S. L. (2007). System Management Software for Virtual Environments. In *Proceedings of the 4th International Conference on Computing Frontiers* (pp. 153–160). ACM. doi: 10.1145/1242531.1242555

Walters, J. P., Chaudhary, V., Cha, M., Guercio Jr, S., & Gallo, S. (2008). A Comparison of Virtualization Technologies for HPC. In *22nd International Conference on Advanced Information Networking and Applications (AINA 2008)* (pp. 861–868).

Weave Net. (n.d.). Weaveworks. Retrieved from `https://www.weave.works/oss/net/` (Accessed: 2018-08-23)

Younge, A. J., Henschel, R., Brown, J. T., Von Laszewski, G., Qiu, J., & Fox, G. C. (2011). Analysis of Virtualization Technologies for High Performance Computing Environments. In *2011 IEEE 4th International Conference on Cloud Computing* (pp. 9–16).

Youseff, L., Wolski, R., Gorda, B., & Krintz, C. (2006). Evaluating the Performance Impact of Xen on MPI and Process Execution For HPC Systems. In *First International Workshop on Virtualization Technology in Distributed Computing (VTDC 2006).* doi: 10.1109/VTDC.2006.4

Yu, H., & Huang, W. (2015). Building a Virtual HPC Cluster with Auto Scaling by the Docker. *CoRR*, *abs/1509.08231*. Retrieved from `http://arxiv.org/abs/1509.08231`

Zhang, S., Boland, L., Coddington, P., & Sevior, M. (2014). Dynamic VM Provisioning for TORQUE in a Cloud Environment. In *Journal of Physics: Conference Series* (Vol. 513, p. 032107).

Zhang, X., Keahey, K., Foster, I., & Freeman, T. (2005). Virtual Cluster Workspaces for Grid Applications. *Argonne National Laboratory, Tech. Rep. ANL/MCS-P1246-0405*.

Appendices

Appendix A

Engagement with Research Community

Refereed Publications

Higgins, J., Holmes, V., & Venters, C. (2015, July). Orchestrating docker containers in the HPC environment. In International Conference on High Performance Computing (pp. 506-513). Springer.

Abstract: Linux container technology has more than proved itself useful in cloud computing as a lightweight alternative to virtualisation, whilst still offering good enough resource isolation. Docker is emerging as a popular runtime for managing Linux containers, providing both management tools and a simple file format. Research into the performance of containers compared to traditional Virtual Machines and bare metal shows that containers can achieve near native speeds in processing, memory and network throughput. A technology born in the cloud, it is making inroads into scientific computing both as a format for sharing experimental applica-

tions and as a paradigm for cloud based execution. However, it has unexplored uses in traditional cluster and grid computing. It provides a run time environment in which there is an opportunity for typical cluster and parallel applications to execute at native speeds, whilst being bundled with their own specific (or legacy) library versions and support software. This offers a solution to the Achilles heel of cluster and grid computing that requires the user to hold intimate knowledge of the local software infrastructure. Using Docker brings us a step closer to more effective job and resource management within the cluster by providing both a common definition format and a repeatable execution environment. In this paper we present the results of our work in deploying Docker containers in the cluster environment and an evaluation of its suitability as a runtime for high performance parallel execution. Our findings suggest that containers can be used to tailor the run time environment for an MPI application without compromising performance, and would provide better Quality of Service for users of scientific computing.

Higgins, J., Holmes, V., & Venters, C. (2016, July). Securing user defined containers for scientific computing. In High Performance Computing & Simulation (HPCS), 2016 International Conference on (pp. 449-453). IEEE.

Abstract: Linux containers and Docker have gained immense popularity as a lightweight alternative to hypervisor based Virtual Machines (VMs). In the context of High Performance Computing and the scientific community, it is clear that containers can serve many useful purposes from system administration, to improved cluster resource management and as a format for sharing reproducible research. However, when compared to VMs, containers seem to trade isolation for performance and ease of use, which poses unique security challenges. In this paper we review how Docker is being used in science, highlight easy to perform exploits, and

evaluate the impact of these on HPC deployments. We also summarise a number of strategies for hardening such a system to reduce the vulnerability of hosting User Defined Containers. Based on these, an original solution to enforce default options and container ownership for nonadministrative users in the HPC use case is presented, in addition to the experience of implementing such a system on a cluster at the University of Huddersfield.

Higgins, J., Holmes, V., & Venters, C. (2017). Autonomous Discovery and Management in Virtual Container Clusters. The Computer Journal, 60(2), 240-252.

Abstract: Global software stacks on scientific cluster computing resources are required to provide a homogeneous software environment which is typically inflexible. Efforts to integrate Virtual Machines (VMs), in order to abstract the software environment of various scientific applications, suffer from performance limitations and require systems administration expertise to maintain. However, the motivation is clear; in addition to increasing resource utilization, the burden of supporting new software installations on existing systems can be reduced. In this paper, we introduce the Virtual Container Cluster (VCC) that encapsulates a typical HPC software environment within Docker containers. The novel component cluster–watcher enables context aware discovery and configuration of the virtual cluster. Containers offer a lightweight alternative to VMs that more closely match the native performance, and presents a solution that is more accessible to customization by the average user. Combined with a Software Defined Networking (SDN) technology, the VCC enables dynamic features such as transparent scaling and spanning across multiple physical resources. Although SDN introduces an additional performance limitation, within the context of a parallel communication network, the benchmarking demonstrates that this cost is application dependent. The Linpack benchmarking shows that

the overhead of container virtualization and SDN interconnect is comparable to the native performance.

Higgins, J., Holmes, V., & Venters, C. (2017). VCC: A framework for building containerized reproducible cluster software environments. The Journal of Open Source Software, 2(11).

Abstract: The problem of portability and reproducibility of the software used to conduct computational experiments has recently come to the fore. Container virtualisation has proved to be a powerful tool to achieve portability of a code and it's execution environment, through runtimes such as Docker, LXC, Singularity and others - without the performance cost of traditional Virtual Machines. However, scientific software often depends on a system foundation that provides middleware, libraries, and other supporting software in order for the code to execute as intended. Typically, container virtualisation addresses only the portability of the code itself, which does not make it inherently reproducible. For example, a containerized MPI application may offer binary compatibility between different systems, but for execution as intended, it must be run on an existing cluster that provides the correct interfaces for parallel MPI execution. As a greater demand to accomodate a diverse range of disciplines is placed on high performance and cluster resources, the ability to quickly create and teardown reproducible, transitory virtual environments that are tailored for an individual task or experiment will be essential. The Virtual Container Cluster (VCC) is a framework for building containers that achieve this goal, by encapsulating a parallel application along with an execution model, through a set of dependency linked services and built-in process orchestration. This promotes a high degree of portability, and offers easier reproducibility by shipping the application along with the foundation required to execute it - whether that be

an MPI cluster, big data processing framework, bioinformatics pipeline, or any other execution model.

Higgins, J., & Holmes, V. (2017) Teaching parallel computing with container virtualization. Accepted In Proceedings of the Workshop on Education for High Performance Computing (EduHPC) at SC17. Denver, CO, USA. Pending Publication.

Abstract: Incorporating modules that equip students with parallel programming and high performance computing skills into core computing and engineering courses poses unique technical challenges. A typical PC laboratory environment may not be suitable, and allowing learners access to a production resource may introduce an unacceptable risk, if it is even possible. For decades, Beowulf clusters have offered an attractive solution where a system can be built using commodity components, often by the students themselves. This can instil in the learner both an understanding of parallel code and how it is orchestrated on the hardware - a strategy used successfully in delivering courses at the University of Huddersfield. However, there are two problems; Firstly, the depth of knowledge required to be learnt in order to successfully build, program and profile a cluster. Secondly, the use of seemingly antiquated tools in order to achieve this. In this paper, we outline some of the experience of integrating parallel and distributed computing into our courses. In addition, we present a novel, cross-platform virtual cluster toolkit developed to address the technical requirements of delivering such courses.

Higgins, J., Aagela, A., Al-Jody, T. & Holmes, V. (2018) Inspiring the Next Generation of HPC Engineers with Reconfigurable, Multi-Tenant Resources for Teaching

and Research. Publication Pending. Submitted to Computer Science Education, Taylor & Francis.

Abstract: There is a tradition at our university for teaching and research in High Performance Computing (HPC) systems engineering. With exascale computing on the horizon, and a shortage of HPC talent, there is a need for new research computing specialists to secure the future of research computing. Whilst many institutions provide research computing training for users within their particular domain, few offer HPC engineering and infrastructure related courses, making it difficult for students to acquire these skills. This paper outlines how and why we are training students in HPC systems engineering skills, including technologies used in delivering this goal. We demonstrate a potential for a multi-tenant system for education and research which can be supported by other institutions, using novel container and cloud based architecture. An evaluation of our activities over the last 2 years is given in terms of recruitment metrics, skills audit feedback from students, and research outputs enabled by the multi-tenant usage of the resource.

Higgins, J., Al-Jody, T. & Holmes, V. (2018) Rapid Deployment of Bare-Metal and In-Container HPC clusters using OpenHPC playbooks. Publication Pending. Submitted to the HPC Systems Professionals Workshop, held in conjunction with SC18.

Abstract: In this paper, we present a toolbox of reusable Ansible roles and playbooks in order to configure a cluster software environment described by the freely available OpenHPC recipes. They can be composed in order to deploy a robust and reliable cluster environment, instilled with the best practise offered by the OpenHPC packaging, and the repeatability and integrity guarantees of the con-

figuration managed approach. With container virtualization setting a new trend in scientific software management, we focus this effort on supporting the deployment of such environments on both bare-metal and container-based targets using the Virtual Container Cluster framework.

Talks

Higgins, J (2017, June). VCC: building reproducible cluster software environments. Talk presented at the Docker Containers for Reproducible Research Workshop (C4RR), Cambridge, UK.

Abstract: Does putting code inside a container really make your experiment reproducible? Scientific software often depends on a system foundation that provides libraries, middleware and other supporting software in order for the code to be executed as intended. With container virtualization, some of these things can be shipped along with the application - clearly reducing the barrier to reproducibility. For example, MPI applications can be containerized and the container itself executed in place of the original program. However, whilst this containerized code may offer compatibility between different system environments, for execution as intended, it must still be run on a cluster that provides the correct interface for parallel MPI execution. The VCC is a framework for building containers that ship the application along with the foundation required to execute it - whether that is an MPI cluster, big data processing framework, bioinformatics pipeline or other execution models. This gives us the ability to quickly create and teardown complex virtual environments tailored for a task or experiment. These can be used as both the primary execution environment and as the method for reproducibility, without requiring the underlying system to provide anything but the container runtime. In this talk we

will introduce the tool and the principles of operation, using example applications to demonstrate it from the point of view of the experiment creator and a person who wants to reproduce an experiment. The tool has been utilised successfully within the University of Huddersfield, and this experience will also be presented – where do we draw the line at how much of the system should be encapsulated in a container, how we adapted our computing resources to support the VCC, and how we convinced users to use it.

Higgins, J (2017, September). Virtual container communities. Talk presented at the Second Conference of Research Software Engineers, Manchester, UK.

Abstract: There is no doubt that container virtualisation is a useful tool for reproducible research. An important result of its adoption is that complex, well documented environments will be accessible for others to reuse. The demand for these portable environments will grow, especially in the long tail of science, to ease the burden of translating experiment to execution and publication. However, the execution system itself is often overlooked when defining them. Where do we draw the line that separates system from experiment? For example, if an experiment requires Hadoop, do we need to distribute Hadoop with the experiment? By encapsulating an execution model along with the code, the utility of containers can be extended beyond reproducibility. In this talk we present a framework that can be used to deploy temporary and permanent cluster software environments within containers. This approach improves the portability and enables dynamic features such as scaling and spanning, transparently of the application. Through nested virtualisation of these environments, we can also move a step closer toward overcoming the technical constraints of facilitating resource sharing at scale – whilst satisfying the needs of every user community.

Appendix B

Surveys and Printed Materials

The System Usability Scale (SUS) and Skills Audit surveys, as issued to students during the Teaching Case Study outlined in Chapter 7, are reproduced in Figures B.1 and B.2 respectively.

Virtual Container Cluster

Student number: _____

For the next set of questions, please think about the usability of the VCC in terms of the grid work we did this term. Don't think too hard, just record your immediate response to the question. **If you are not sure of your answer, or you feel that you cannot answer it, just mark down the middle and pick 3.** This is really important..... don't just pick number 1 or 5 if you don't know!

		1	2	3	4	5	
I think that I would like to use this system frequently	strongly disagree						strongly agree
I found the system unnecessarily complex	strongly disagree						strongly agree
I thought the system was easy to use	strongly disagree						strongly agree
I think that I would need the support of a technical person to be able to use this system	strongly disagree						strongly agree
I found the various functions in this system were well integrated	strongly disagree						strongly agree
I thought there was too much inconsistency in this system	strongly disagree						strongly agree
I would imagine that most people would learn to use this system very quickly	strongly disagree						strongly agree
I found the system very cumbersome to use	strongly disagree						strongly agree
I felt very confident using the system	strongly disagree						strongly agree
I needed to learn a lot of things before I could get going with this system	strongly disagree						strongly agree

These last questions are yes and no answers. If you want to elaborate on your choices, there will be a free text box at the end. Clearly mark you choice by circling it or crossing the other options out.

Did you find it easier to work with the system on just the head node? Yes / No

Do you think it would have been better to use all 3 nodes in each group to set up the grid?
Yes / No

Do you feel like you missed out on anything by not setting up the other nodes when building the grid?
Yes / No

Do you feel like using the VCC has prepared you to use a real cluster or grid? Yes / No

Has your level of proficiency in Linux improved since you started the VCC?
Yes / No / I never used Linux before

Do you think you would know how to use a similar Linux system in future after learning using the VCC?
Yes / No

Finally, please add any other comments you would like me to consider (related to the VCC software itself):

Figure B.1: SUS Survey for Teaching Case Study

Student proficiency survey for NHE2530 and NME3502

Name / student number:

Course / pathway:

Group name:

Please read the skills listed in the table and rate your level of proficiency by ticking the box that is applicable to you. Please read the definitions on the reverse to clarify what each level means.A

	No knowledge	**Basic knowledge**	**Novice**	**Intermediate**	**Advanced**	**Expert**
Troubleshooting PC hardware (BIOS, memory errors, replacing parts)						
Measuring performance of a PC						
Installing software from the internet						
Compiling and installing software from source code						
Basic networking (IP addressing, host names, ping)						
Competency with Linux or other Unix-like Operating System						
Programming in C						
Programming in any other language						
Parallel programming using a library such as MPI						
Code execution models (serial / parallel)						
Paralleisation strategies (embarassing / dependent / inherently serial)						
CPU (instruction) level parallelism (flynn's taxonomy)						

Figure B.2: Skills Audit for Teaching Case Study

Appendix C

Data

Benchmarking Results

The Linpack and OpenFOAM benchmarking results are shown in Tables C.1 and C.2 respectively. The spanning queue simulator results are shown in Tables C.3 and C.4. The case study results from Chapter 6 are shown in Tables C.5 and C.6 for Campus Grid and Inter-institution spanning respectively.

Usability Survey Results

The System Usability Scale (SUS) results are shown in Tables C.7 and C.8 for the VCC and OSCAR solutions respectively.

Skills Audit Results

The Skills Audit questionnaire results are shown in Tables C.9 and C.10 for the VCC, and C.11 and C.12 for OSCAR.

Latency	Native Mbps	Native Gflops	SDN Mbps	SDN Gflops
0.5	941	186		
1	952	184		
2			923	173
5	906	182	918	178
10	926	184	915	176
50	923	177	890	163
100	916	158	895	142
200	751	52	714	65

Table C.1: Linpack Latency Scalability Results

Latency	Bike (hours)	Prop (hours)
0.1	6.20	5.09
1	6.11	5.98
5	8.89	10.58
10	11.09	16.93
20	17.14	29.72
50	17.92	35.07
100	68.07	139.38

Table C.2: OpenFOAM Latency Scalability Results

Original

Date	Queued	Run
2016-03-31	31	3
2016-04-30	253	471
2016-05-31	175	111
2016-06-30	23	65
2016-07-31	79	99
2016-08-31	185	47
2016-09-30	170	105
2016-10-31	233	415
2016-11-30	95	113
2016-12-31	17	224
2017-01-31	149	26
2017-02-28	96	209
2017-03-31	15	0

Spanned

Date	Queued	Run	Spanned	Forwarded
2016-03-31	9	3	0	22
2016-04-30	66	471	0	187
2016-05-31	37	111	0	138
2016-06-30	23	65	0	0
2016-07-31	69	99	1	9
2016-08-31	184	47	1	0
2016-09-30	170	105	0	0
2016-10-31	233	415	0	0
2016-11-30	78	113	0	17
2016-12-31	17	224	0	0
2017-01-31	149	26	0	0
2017-02-28	96	209	0	0
2017-03-31	15	0	0	0

Table C.3: Simulator Results - Eridani

	Date	Queued	Run
	2016-03-31	98	54
	2016-04-30	98	176
	2016-05-31	9	30
	2016-06-30	83	0
	2016-07-31	38	66
Original	2016-08-31	59	1
	2016-09-30	33	0
	2016-11-30	23	3
	2016-12-31	7	0
	2017-01-31	4	0
	2017-02-28	11	0
	2017-03-31	1	0

	Date	Queued	Run	Spanned	Forwarded
	2016-03-31	29	54	1	68
	2016-04-30	9	176	29	60
	2016-05-31	4	30	0	5
	2016-06-30	7	0	0	76
	2016-07-31	18	66	12	8
Spanned	2016-08-31	42	1	4	13
	2016-09-30	25	0	5	3
	2016-11-30	3	3	8	12
	2016-12-31	1	0	2	4
	2017-01-31	4	0	0	0
	2017-02-28	9	0	2	0
	2017-03-31	1	0	0	0

Table C.4: Simulator Results - Ascella

	Job ID	Hosts	Start time	End time	Wall (s)
	o19	clusterA4+clusterA5	1519859367	1519862132	2765
	o20	clusterA2+clusterA3	1519859326	1519862032	2706
	o21	clusterA1+clusterB3	1519859548	1519862159	2611
	o22	clusterB1+clusterB2	1519859540	1519861706	2166
Run 1	o23	clusterB1+clusterB2	1519861707	1519863838	2131
	o24	clusterA1+clusterB3	1519862161	1519864772	2611
	o25	clusterA2+clusterA3	1519862033	1519864759	2726
	o26	clusterA4+clusterA5	1519862135	1519864895	2760
	o27	clusterB1+clusterB2	1519863839	1519865965	2126
	o28	clusterA1+clusterB3	1519864773	1519867394	2621

	Job ID	Execution Host	Start time	End time	Wall (s)
	o53	clusterA4+clusterA5	1519896360	1519899024	2664
	o54	clusterA2+clusterA3	1519896319	1519899021	2702
	o55	clusterA1+clusterB3	1519896541	1519899175	2634
	o56	clusterB1+clusterB2	1519896532	1519898665	2133
Run 2	o57	clusterB1+clusterB2	1519898667	1519900820	2153
	o58	clusterA1+clusterB3	1519899176	1519901814	2638
	o59	clusterA4+clusterA5	1519899025	1519901674	2649
	o60	clusterA2+clusterA3	1519899023	1519901745	2722
	o61	clusterB1+clusterB2	1519900822	1519902957	2135
	o62	clusterA1+clusterB3	1519901816	1519904430	2614

Table C.5: Campus Grid Spanning Case Study Results

	Job ID	Execution Host	Start time	End time	Wall (s)
	o100	clusterB1+clusterB2	1521914112	1521916246	2134
	o101	clusterA2+clusterA3	1521914785	1521917478	2693
	o102	clusterA4+clusterA5	1521914550	1521917209	2659
	o103	clusterB1+clusterB2	1521916248	1521918369	2121
Run 1	o104	clusterA2+clusterA3	1521917480	1521920107	2627
	o105	clusterA4+clusterA5	1521917210	1521919932	2722
	o106	clusterB1+clusterB2	1521918371	1521920523	2152
	o97	clusterB1+clusterB2	1521911990	1521914111	2121
	o98	clusterA2+clusterA3	1521912024	1521914784	2760
	o99	clusterA4+clusterA5	1521911861	1521914548	2687

	Job ID	Execution Host	Start time	End time	Wall (s)
	o107	clusterB1+clusterB2	1521922037	1521924177	2140
	o108	clusterA2+clusterA3	1521922072	1521924646	2574
	o109	clusterA4+clusterA5	1521921909	1521924563	2654
	o110	clusterB1+clusterB2	1521924178	1521926309	2131
Run 2	o111	clusterA2+clusterA3	1521924648	1521927207	2559
	o112	clusterA4+clusterA5	1521924565	1521927155	2590
	o113	clusterB1+clusterB2	1521926311	1521928450	2139
	o114	clusterA2+clusterA3	1521927209	1521929768	2559
	o115	clusterA4+clusterA5	1521927157	1521929780	2623
	o116	clusterB1+clusterB2	1521928451	1521930604	2153

Table C.6: Inter-Institution Spanning Case Study Results

Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	SUS
3	2	4	3	4	5	4	3	4	5	52.5
4	1	4	4	4	3	3	3	3	2	62.5
3	4	3	3	4	3	2	3	4	3	50
3	1	4	2	4	3	3	3	4	4	62.5
4	2	1	3	1	5	3	5	3	3	35
3	3	3	3	4	3	4	2	3	3	57.5
5	1	5	1	5	1	5	1	5	1	100
4	2	3	2	3	3	4	3	5	4	62.5
5	2	5	1	5	2	4	1	5	1	92.5
5	5	5	3	5	1	5	1	3	3	75
4	3	3	4	3	3	3	4	3	5	42.5
4	4	4	3	4	3	4	4	5	3	60
4	3	5	4	4	3	3	3	4	4	57.5
3	4	4	4	4	3	3	3	4	4	50
3	3	4	3	4	3	3	3	4	5	52.5
5	1	5	4	5	1	5	1	5	2	90
4	1	4	1	3	4	4	1	4	2	75
4	3	4	2	4	2	3	3	3	2	65
4	3	4	4	3	3	4	4	4	3	55
4	3	3	5	4	3	3	3	2	4	45
3	4	4	5	3	4	3	3	3	5	37.5
4	3	4	3	3	3	5	3	3	3	60
3	1	4	1	4	2	3	2	4	3	72.5
3	1	4	2	4	2	4	2	4	1	77.5
4	2	4	2	4	2	3	2	4	3	70
5	2	4	2	4	2	4	1	4	5	72.5
3	2	4	2	4	2	3	4	4	2	65
3	2	2	2	3	4	2	3	2	4	42.5
3	4	3	3	3	3	4	3	4	4	50
4	2	4	2	4	2	4	2	4	2	75
3	3	4	4	5	3	4	2	4	5	57.5
3	2	4	2	4	2	4	1	4	3	72.5
3	4	3	3	3	1	3	3	2	5	45
3	4	4	3	3	3	4	3	3	4	50
5	1	5	1	5	1	5	1	5	2	97.5
2	3	2	2	2	2	1	2	2	5	37.5
2	3	2	2	2	2	3	2	2	5	42.5
3	3	4	3	4	3	4	2	4	2	65
2	4	3	4	2	3	2	3	2	5	30
5	1	5	1	5	1	5	1	5	1	100
3	2	2	1	3	2	4	3	2	4	55
4	1	5	1	5	1	3	2	4	2	85
3	5	1	3	1	5	2	5	1	4	15
3	4	3	5	4	3	4	3	2	4	42.5
2	3	4	3	4	3	3	3	2	3	50
4	3	4	3	3	4	4	3	3	4	52.5
5	2	4	2	3	2	4	2	3	5	65

Table C.7: System Usability Scale - VCC Results

Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	SUS
2	2	3	3	3	2	3	2	3	2	57.5
5	3	5	3	4	2	4	3	5	3	72.5
3	3	3	4	3	3	3	3	3	4	45
3	3	3	4	3	3	3	2	4	4	50
4	2	3	2	3	2	3	2	2	2	62.5
3	3	3	5	3	3	4	3	3	4	45
3	3	4	4	3	3	4	3	4	5	50
3	3	3	3	4	2	3	2	3	2	60
3	2	4	2	4	1	4	2	4	2	75
2	3	4	4	3	3	3	3	3	4	45
4	4	3	4	3	3	2	1	4	4	50
3	3	4	4	2	3	4	3	3	4	47.5
2	4	4	3	3	3	2	2	5	4	50
3	3	2	5	3	4	4	2	4	4	45
3	2	3	2	2	3	2	2	1	2	50
2	2	3	3	3	1	3	2	3	2	60
2	3	4	4	3	2	3	3	3	4	47.5

Table C.8: System Usability Scale - OSCAR Results

Cohort	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12
16/17	1	1	1	1	1	0	1	1	1	0	0	0
	0	3	5	1	5	0	4	0	0	0	0	2
	0	3	5	1	5	0	4	4	0	0	0	2
	0	2	0	5	5	3	2	1	0	1	0	0
	2	2	3	1	3	2	1	3	2	2	1	1
	2	3	3	2	2	1	2	3	1	1	1	1
	3	3	5	3	3	3	0	3	2	2	0	0
	3	3	5	3	2	3	0	3	0	1	0	0
	2	2	3	1	0	0	2	0	0	0	0	0
	3	2	2	0	1	0	2	0	0	0	0	0
	3	2	2	0	1	0	1	0	0	0	0	0
	3	2	2	0	1	0	3	0	0	0	0	0
	1	3	5	2	1	0	3	3	0	1	0	1
	4	4	4	3	3	3	4	4	2	1	0	1
	5	3	5	4	5	4	3	4	0	1	0	0
	4	4	4	3	5	3	1	3	1	2	2	2
	3	3	3	3	3	3	3	3	0	3	0	0
	2	1	2	2	1	0	1	1	0	0	0	0
	1	1	4	2	1	1	2	2	0	0	0	0
	4	4	4	2	2	3	3	3	0	2	0	0
	1	1	4	0	2	0	1	4	3	0	1	0
	3	3	4	1	2	2	3	3	0	0	0	0
17/18	5	5	5	4	5	2	4	2	0	1	1	2
	4	3	3	2	2	2	3	2	0	0	0	1
	1	1	3	0	1	1	1	1	0	0	0	0
	0	1	4	1	4	1	3	3	0	0	0	0
	0	0	1	0	1	0	2	1	0	0	0	0
	3	3	5	3	3	2	0	1	1	1	1	1
	3	3	5	3	3	2	1	4	1	1	1	1
	4	4	5	4	4	4	2	2	1	1	1	1
	2	1	3	1	2	0	1	1	0	0	0	0
	1	1	1	0	0	0	3	0	0	0	0	0
	0	1	1	0	1	0	1	0	0	0	0	0
	3	0	0	0	1	1	3	3	1	0	0	0
	2	1	2	0	1	0	2	2	0	0	0	0

Table C.9: Skills Audit Results - VCC Start

Cohort	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12
16/17	3	2	4	3	3	3	3	2	2	2	2	2
	5	5	5	4	3	3	3	3	1	1	0	1
	5	5	5	4	5	4	5	4	4	5	3	2
	5	5	5	4	4	3	2	4	4	3	3	3
	4	5	5	5	4	3	3	2	4	3	2	2
	3	2	4	2	4	3	2	4	3	2	2	2
	5	5	5	4	4	3	2	4	4	3	3	3
	4	4	4	4	4	4	5	3	4	4	4	5
	4	4	4	4	4	4	3	3	3	3	2	0
	3	3	3	3	3	3	1	1	1	2	0	0
	5	3	5	4	4	4	4	3	2	2	2	2
	4	4	4	3	2	2	3	1	1	1	0	0
	1	2	4	1	2	1	2	2	1	1	1	2
	4	4	4	4	4	4	3	3	3	3	3	3
	5	4	5	4	4	4	4	5	3	3	3	2
	3	3	5	3	3	3	3	4	2	3	1	1
	5	4	5	4	4	2	2	2	2	1	1	2
	5	4	5	3	4	4	3	2	2	1	2	2
	4	4	5	5	5	3	4	2	3	2	3	3
	3	3	4	3	1	1	3	3	2	2	3	3
	3	3	3	3	3	2	2	0	1	1	1	1
	5	3	4	3	3	3	2	3	3	3	3	3
	3	4	4	4	3	3	3	4	3	3	3	3
	3	3	4	3	0	3	2	1	2	2	2	3
	5	4	5	4	4	3	3	4	3	4	3	3
	3	3	4	3	3	2	1	0	0	0	0	1
17/18	1	3	5	3	3	1	1	0	2	2	2	2
	4	3	4	4	4	3	4	3	4	3	3	3
	2	2	2	1	1	2	1	2	1	2	2	2
	2	2	3	2	4	3	3	2	4	3	2	3
	3	3	4	4	4	4	4	4	3	3	3	3
	4	3	5	5	5	3	4	4	3	3	2	1
	1	2	2	3	4	3	3	2	2	2	0	0
	4	1	2	2	3	1	4	1	1	1	1	1
	2	3	4	2	3	3	3	2	3	3	1	3
	5	4	5	4	4	4	4	3	3	3	3	3
	1	3	3	2	2	1	2	2	1	1	1	1
	1	2	3	2	2	1	2	1	1	2	1	1

Table C.10: Skills Audit Results - VCC End

Cohort	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12
16/17	4	4	4	3	3	3	2	4	1	2	1	1
	3	4	4	4	4	2	3	4	3	3	3	3
	4	4	4	3	2	1	3	1	0	1	1	1
	4	4	4	4	4	4	5	3	3	4	2	3
	2	2	5	5	4	1	3	2	1	1	1	1
	3	3	4	3	2	1	3	3	0	0	0	0
	3	2	4	4	3	4	1	4	0	0	0	0
	5	4	5	3	3	2	1	1	0	0	0	0
	3	3	3	3	0	3	1	3	0	0	0	0
	3	3	4	3	3	2	1	3	0	0	0	0
	3	3	4	3	3	3	1	3	0	0	0	0

Table C.11: Skills Audit Results - OSCAR Start

Cohort	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12
16/17	4	3	4	4	3	3	2	3	4	3	3	3
	4	4	4	3	3	3	3	3	4	3	3	3
	3	3	4	4	2	4	0	4	2	2	2	2
	4	4	4	4	3	2	4	3	3	2	2	0
	4	4	4	4	4	4	4	4	4	4	4	4
	5	5	5	4	4	3	3	3	0	2	3	2
	4	4	4	4	4	4	3	5	3	3	3	3
	4	4	5	3	4	2	2	3	2	3	1	3

Table C.12: Skills Audit Results - OSCAR End