Delft University of Technology

Containerization for High Performance Computing

Author:

Nienke Eijsvogel

Master of Engineering

in

Computer Science

November 23, 2022

Nienke Eijsvogel
Student Number: 5428599

Graduation committee:

Prof. Dr. Jan Rellermeyer
Dr. Lydia Chen
Dr. Matthias Moller

Abstract

Containerization, a lightweight form of virtualization, increasingly became more popular in the last decade. Containers can offer a level of isolation and privacy to the user, which are not always sought after. High performance computing workloads benefit from having a custom container filesystem, but would suffer from any overhead incurred by isolation. This research assesses overhead from Singularity containers for workloads that work on different levels of the memory hierarchy. Different configurations for number of processes and threading are tested for different types of benchmarks. No significant overhead was found which encouraged to test Singularity outside the high performance computing world. Singularity was implemented in Kubernetes, a dynamic container scheduler, by restructuring and rewriting the deprecated Singularity container runtime interface. The re-newed container runtime interface was implemented in Sykube, a local Kubernetes framework for Singularity. A Kafka streaming application was used as benchmark to assess latency in messaging and deployed in a local Docker container network and a local Singularity container network. The results of the local container networks were compared against the performance of the benchmark in Kubernetes. In the local setup Singularity outperformed Docker, while in Kubernetes setup Docker outperformed Singularity.

Contents

1 Introduction

Virtualization is a technique that aims to separate a service from the underlying physical form of that service. This can include virtualization of hardware, operating systems and storage devices. When computing environments are independent of physical infrastructure a level of flexibility is introduced that allows for multiple services to use the same infrastructure. Each virtual service can interact independently and run different applications or even operating systems, while sharing resources of a single host machine. Since Virtualization can provide a completely separate environment, it also introduces a level of isolation. This feature of virtualization can be useful to any user who values privacy and security. For example, it can provide users of a cluster to have an isolated view on their own file system, without the risk of any user intervening.

Virtualization can also provide cluster users with the flexibility to use custom libraries, without the need to alter the state of the cluster. Clusters are primarily overseen by an admin that controls the system, its state and its users. For security reasons users are often not allowed to make changes to the system, which results in limited flexibility with regards to application implementations. Implementations have to be compatible with the state of the cluster. This can strongly decrease efficiency and result in a sub-optimal workflow. There is also no guarantee that proper working code still work in the future as the state of the cluster evolves. Containerization, a lightweight form of virtualization, can soothe this problem. Containers can provide the user with a custom read only file system that provide an application all required dependencies. These container environments offer the reassurance that if an application runs properly, it will run properly every time it is deployed in this container environment.

Previous mentioned features of virtualization are isolation, privacy, security and flexibility. These properties however, are not always sought after by every user. Users who not seek any isolation or privacy but rather intend to achieve low latency on communication might have to debate on whether virtualization is beneficial. Examples of workloads that suffer from any latency and exert high demand on resources are advanced computing problems. Advanced computing problems as in Physics, Aerodynamics and other engineering fields are computational problems that require more computing power and resources than a single machine can offer. High performance computing is a field that works to support these advanced computing problems by aggregating computing power to deliver much higher performance. High performance computing is a field that would benefit from having flexibility of installment on a shared cluster, but can suffer from isolation overhead with its tightly coupled applications. Data intensive workloads can require a high degree of data exchange between processors and therefore inter container communication. This research aims to assess the effect containerization has on high performance computing and provide a clearer view on its trade offs. Are the benefits of containers worth any overhead it causes?

1.1 Problem statement

This research aims to assess the effect of containerization on High performance computing. To be able to discover any weak point, this research aims to extensively assess different type of workloads with various configurations. Only by extensively searching among multiple axes, a balanced conclusion can be conducted on the benefits and overhead of containerization. Heavy, data intensive, workloads are deployed that work on different levels of the memory hierarchy. To assess overhead of containerization related to the over-threading, configurations are run that force the applications to spend more time in the kernel space. This is effectuated by subscribing processes on a machine with more threads than the number of physical threads. Both over-threaded and non over-threaded implementations of the same workload are run. To assess any architecture specific overhead, all workloads are run various times with different configurations regarding the number of containers. All results are compared to bare metal implementations, which are a benchmark for any overhead incurred. Every configurations possible among the axes of memory hierarchy, architecture and threading is being made and tested. Meaning, every workload that works on a specific level of the memory hierarchy is being tested with a varying number of containers and with a varying level of over-threading. This research will be extended by testing Singularity, a high performance computing container framework, outside of the high performance world by implementing it in Kubernetes, a dynamic container scheduler. In addition the container runtime interface will be implemented in Sykube, a local Kubernetes Singularity container scheduler. The framework will be tested and performance will be compared to the industry standard container framework, Docker. To guide the process of researching the effect of containerization on High performance computing, the following three research questions are going to be answered.

- Research Question 1: Are there levels of the memory hierarchy for which containerization causes performance degradation?
- Research Question 2: How much overhead is related to architecture of container deployment or oversubscribing threads?
- Research Question 3: How does the High Performance Computing framework Singularity performs outside the high performance world?

1.2 Approach and Scope

To answer Research Question 1, applications of the High Performance Computing Challenge (HPCC) are used [1]. The Challenge is a benchmark suite that measures a range of memory access patterns among different levels of the memory hierarchy. The benchmarks applications assess performance of the system in Gflop/s, Gup/s, bandwith and latency. Since not only system performance is of interest in this research, run-time is added as a metric to all applications. Results of the bare metal implementation will be compared to the results of the container implementation, to determine any overhead.

To answer Research Question 2, the results of the previous mentioned HPCC are used as well. For every application, various configurations of the number of containers and processes are deployed to the cluster. For the not over-threaded implementation, the number of threads on a node has to be divisible by the number of containers scheduled, otherwise threads will be left idle or slightly oversubscribed. Different levels of over-threading are run which is effectuated by incrementing the processes on machine that all have the logical number of threads that are present on a node subscribed to them. Results of the bare metal implementation will be compared to the results of the container implementation, to determine any overhead.

To answer Research Question 3, Singularity will be tested in an environment that has different characteristics than traditional high performance computing environments. High performance computing workloads are inherently not dynamic and therefore implementing Singularity in a dynamic container scheduler as Kubernetes will serve as a good starting point for testing its generalizability. The Singularity container framework and container runtime interface will be implemented in Kubernetes and tested in a local Kubernetes setup. Since the Singularity container runtime interface and the local Kubernetes framework Sykube, are deprecated, both will have to be restructured and extended with new modules that enable them to work properly on newer Linux kernels. In addition to implementing Singularity in Kubernetes, Singularity will be tested with a workload that has different characteristics than traditional high performance computing workloads; An Apache Kafka streaming application is constructed that measures delay in messaging. The performance of the Kafka streaming application in Kubernetes with Singularity will be compared to the performance in Kubernetes with Docker, the industry standard container.

1.3 Thesis Outline and Contribution

The key contributions of this thesis are the following:

1 A comprehensive survey on relevant container frameworks for high performance computing and the underlying features that adhere to these frameworks.

2 An in-depth quantitative analysis on overhead incurred by deploying containers for different levels of the memory hierarchy. This in depth analysis is being done on the axes of memory hierarchy, container architecture and threading level.

3 Discussion of underlying features of container frameworks that can cause performance degradation by qualitative research and by deploying different container frameworks in a Kubernetes cluster.

4 New modules and extensions of the deprecated frameworks Sykube, a framework for a local Kubernetes cluster, and the Singularity container run-time interface that enable the frameworks to work for newer Linux Kernels and provide new features.

2 Background

This section starts with a brief history of containerization followed by an explanation of its key features and underlying techniques. These features and techniques form container frameworks and are affecting how these frameworks operate and therefore their performance. A key understanding of these underlying techniques is required to understand how they attribute to container frameworks and how they can cause weak points or overhead.

2.1 Hypervisor

Before container use increased in popularity, hypervisor virtualization, also known as virtual machines, was the norm. A Hypervisor is a piece of software that abstracts operating systems from the underlying hardware. Because the abstraction layer provides the user with an emulated version of the underlying hardware, the hardware can be shared among multiple virtual machines. Virtual machines are isolated from the host operating system and therefore provide an isolated environment to its user. Since virtual machines stack new operating systems on host machines, they demand a substantial amount of resources. Under certain circumstances, as when no other operating system than the host operating system is required or no strict security issues apply, the high demand on resources lacks substantiation and hardware virtualization might not be necessary.

2.2 Containerization

Moving one level up from hardware, containerization is a form of operating system level virtualization. The main idea of a container is that containers are standard units of software that packages up code and all its dependencies as an immutable file system effectuating that an application in this environment can run reliably from one computing environment to another. Containerization is a more lightweight form of virtualization since it does not require installment of a new operating system on the host machine. Containers run a user space on top of the host operating system kernel, requiring substantially less resources than virtual machines. When multiple containers are deployed, each container shares the operating system kernel with the other containers as normal processes would. The lightweight nature of containers is important with regard to the performance of high performance computing workloads, which can have a strong demand on resources itself. There are certain features containers can provide to the user;

- Scalability. The scalability feature accentuates the seamless way container clusters are able to scale up and down. The required time to deploy new instances on a running host is lower than instantiating a new operating system on virtualized hardware. Because of their lightweight nature and low instantiation time, containers are well scalable [2] [3]. High performance computing

benefits from quick instantiation of containers, however it does not require dynamic scheduling of new containers since the workload is typically set at the start. This feature is more of concern for any auto-scaling of services.

- Portability. The portability feature of containers ensures that containers can run reliable from one computing environment to another. This feature is of concern to high performance Computing, since it ensures the containerized workload will run properly on every shared cluster or machine, regardless of its state.
- Configurability. Containers offer a high level of configurability. The virtualization of the operating system can be configured allowing for custom container environments that serve different type of workloads. Containers can be regarderd as special type of processes with variability in isolation levels, features and privileges. Configurability is of interest to High performance computing since performance can be dependent on characteristics as resources and isolation.
- Isolation. As previously mentioned, virtualization can offer privacy and isolation [5] [6]. Isolation can affect performance of containerized workloads since workloads of high performance computing are commonly tightly coupled. All though dependent on the level of data locality, these workloads can require a high level of inter-container communication. The isolation property of containers is not a feature high performance computing typically gains from.

2.3 Operating systems

A container relies on the way operating systems isolate their resources and features to create its own environment during runtime. Since containers rely on the operating system, the characteristics of these kernels are important pillars for container frameworks. For containers to work properly, they have to be compatible with the low-level characteristics of the kernel. Containers can be build and deployed on all types of operating systems. Throughout the years various container frameworks have been introduced for different type of operating systems.

Oracle Solaris container framework [13] was the first official container release which combined system resource controls and boundary separation provided by zones, which were able to leverage features like snapshots and cloning from ZFS. LXC [15] is a container framework specifically designed for the Linux kernel and is not as lightweight as other container frameworks, but it resembles a virtual machine without the need of an operating system. An initiative has been proposed for a MacOS Containers to extend Containerd [14], a Linux container framework, to enable it to run natively on Macos [36]. Containerd is a container daemon that runs on Windows and Linux [14] and virtualizes Linux namespaces. Running Linux based containers on non-Linux operating systems, requires the presence of a Linux kernel on the host machine.

The Linux Kernel is also the base of Docker, one of the most widely adopted container frameworks [7]. Docker works as well on Windows and Macos, because the Docker packages for these operating systems contain a Linux kernel. Docker is a container framework adopted by big tech companies as Netflix, Spotify, Pinterest and Airbnb. The framework was for a while even hardcoded in Kubernetes, a container manager used by e.g. previous mentioned tech companies and developed by Google [12]. This emphasizes how dominant Docker is in the container landscape. However, Kubernetes announced to move away from the hardcoded Docker shim in December 2020 [37].

Kubernetes stated that containerization became an industry standard and therefore they added support for other container run-times. Docker has the benefit that it was already adopted by any company that wished to use Kubernetes up till December 2020. This does not necessarily mean Docker is the optimal container for the diverse types of workload run by previous mentioned tech companies and other companies. Kubernetes moving away from Docker insinuates that there is a demand for other frameworks with different characteristics than Docker. Singularity is a container framework which was constructed to accommodate high performance computing and can operate with a low level of privileges [8]. The Linux kernel is, in addition to Docker, also the base operating system of Singularity and therefore this research will narrow its scope to container frameworks that virtualize the Linux Kernel.

2.4 Related Work

There has been previous research into the usage of these frameworks for high performance computing to some extend and it is still an ongoing research topic [34] [35]. There has been work that assesses performance of single-container deployments and emphasize the possibility that deploying a high performance computing workload into a single Docker container can achieve near bare metal performance [24] [25]. These type of implementations provide a good set up for comparing hypervisor virtualization with containerization, but are somewhat restrictive in what they can say about container overhead for high performance computing in general. There recently, in 2022, has been research by Liu et al.[29] that has a more in-depth approach by also incorporating the container granularity. Liu et al. present an interesting approach where they schedule 32 processes per node, encapsulated in different numbers of containers varying from 1 to 32, incremented by multiplication with a factor 2 [29]. This research differs from Liu et al in the sense that workload is distributed among a different number of containers per host, but not in finer grained number of processes per container. This type of scheduling, without fine graining, produces results for a one on one comparison with the bare metal implementation. Liu et al. use metrics of the HPCC benchmark [1] which are used to assess the peak capabilities of a system in terms of gup/s, bandwith, flop/s. The peak values of these metrics describe a system and are determined by timing only smaller parts of the benchmarks that are relevant for the metrics. It is not ensured that these metrics are a one to one mapping to application performance and therefore container overhead. This research adds a runtime metric to have a well rounded base to do any claims up on about container overhead for high performance computing performance. In addition this research is run on up to 40 nodes opposed to 5 nodes in the research of Liu et al. Liu et al. have published another work where they test the time spent in MPI functions and assess the effect of oversubscribing cpu's

[33]. The technique they use differs from this research in the sense that they disable hyperthreading and oversubscribe on a process level by limiting the number of available cpu's to the mpi processes and they run their experiment on a single host. This research occupies all available cpu's, uses hyperthreading and oversubscribes threads on an application level and is run on a up to 40 node cluster. No other previous research has been found that assesses overhead from oversubscribing threads for high performance computing workloads in containerized environments.

2.5 Linux Kernel

Linux based container frameworks lend the way the Linux kernel isolates global system resources and use it as building blocks for a (secured) container environment. To have a clear understanding of the characteristics of different container frameworks, a low level understanding of the Linux kernel is required.

The Linux Kernel relies on the notion of namespaces. Namespaces are a feature of the Linux kernel that partitions kernel resources. A namespace wraps a global system resource in an abstraction that makes it appear to the processes within the namespace that they have their own isolated instance of the global resource. Namespaces can provide users of a shared cluster, commonly used in high performance computing, with their own private work space on the machine. Privacy is attained by ensuring that the private portion of a user space is only visible to its owner and no damage can be done to violate this private space. This form of privacy can be obtained by isolation which is feasible through namespaces. Namespaces have different features and effects, thus the use case of a container is decisive in which namespaces best be isolated. For example with a workflow that relies on communication between containers, isolating more than absolutely necessary can induce performance regressions while other tasks are better suited for isolation (e.g. database services or webservices). Different use-cases thus should be configured with the fitting namespaces.

2.5.1 Namespaces

User Namespace The user namespace allows processes to run with different user identifiers and/or privileges inside that namespace than are permitted outside. User namespace capabilities are not the same as capabilities on the host, but namespaced capabilities. The user namespace enables the Linux chroot() systemcall to change the root directory of a process and its children to a new location in the filesystem. The chroot() command was developed for Unix V7 in 1979 and was the beginning of process isolation and ultimately containers. In container context this means the root of a container has capabilities only within the container and across the range of User IDs that were mapped into the user namespace. For the user inside the container it means that the process has full privileges for operations inside the user namespace, but is unprivileged for operations outside the namespace.

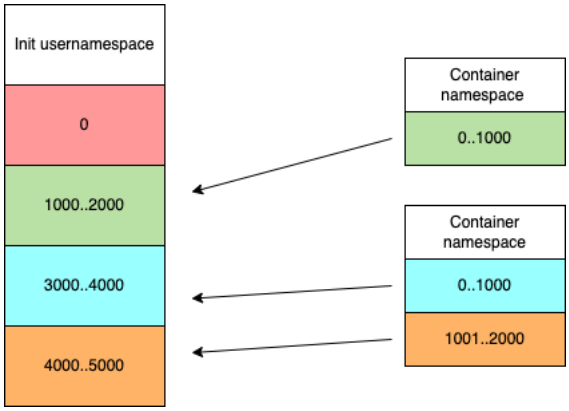

Figure 1: UID/GID Mapping

Fake root mode of a container allows an unprivileged user to run a container seemingly as a root user by leveraging user namespaces with user namespace UID/GID mapping. User namespace UID/GID mapping allows a user to act as a different UID/GID in the container context than they are on the host. A user can access a configured range of UIDs and GIDs in the container, which map back to a (possibly) unprivileged user UID or GID on the host. This enables a user to act as root (UID 0) inside a container with a level of privilege that only exist in the container environment, but with no privilege on the host.

UNIX Time-Sharing Namespace (UTS) Despite its name, the UTS namespace controls the hostname and the NIS domain. It can be used for setting the hostname and the domain which are visible to running processes in the same namespace. By assigning a container its own UTS namespace, the hostname for the process can be changed independently of the host or VM on which the process is running. Isolating the UTS namespace is required if one aims to construct a container network in which containers can access one another by their name.

Interprocess Communication (IPC) namespace . Interprocess communication provides seperation of named shared memory segments, message queues and semaphores. When the host's IPC namespace is shared with the container, it would allow processes within the container to see all the inter process communications on the host system. The IPC namespace is of interest for high performance computing since shared memory is the fastest interprocess communication mechanism and can be used in order to accelerate inter container communication. The operating system maps a memory segment in the address space of several processes to read and write in that memory segment, without calling operating system functions. For workloads that exchange large amounts of data, shared memory is superior to message-passing techniques like message queues, which require system calls for every data exchange.

Network namespace A network namespace is a logical copy of the network stack from the host system. The network namespace virtualizes the network stack and provides a virtual network interface. Every network namespace owns a private set of IP addresses, a routing table, socket listing, firewall, and other network-related resources. Virtualizing the network namespace allows for applications running in different containers to bind to the same port, while running on the same host. Firewall settings and iptabling are of interest for inter container communication when the network stack is isolated. When a network namespace is terminated, any virtual interfaces within it is terminated and any physical interfaces within it moved back to the initial network namespace.

Control group (Cgroup) namespace Cgroup namespaces are employed to control the system resources. The cgroups limits what resources (i.e Memory, CPU) are available to the group. Deciding whether this namespace is isolated or not within a container context, is very dependent on the use-case of the container. For data intensive workloads it can be best to not isolate any more than necessary on behalf of hardware to avoid performance degradation. Linkedin employees who investigated the effect of cgroups on Docker mentioned several issues regarding high performance [38].

PID namespace The PID namespace provides processes with an independent set of process IDs (PIDs) from other namespaces. PID namespaces allow functionalities such as suspending, resuming or migrating a set of processes. In terms of containerization this namespace allows for migrating the container to a new host while the processes inside the container maintain the same PIDs. the PID system works almost identically to that outside of the namespace. The process IDs inside the PID namespace start at 1 since they have their own process list, with the first process considered as the init process. The PID namespace ensures that the processes running inside a container are isolated from the host. With an isolated PID namespace, When you run a ps command inside a container, you only see the processes running inside the container and not on the host machine.

Mount namespace A mount namespace is the set of file system mounts that are visible to a process. Mounting itself is a process by which the operating system makes files and directories on a storage device available for users to access and can be seen as opening a file before reading/writing from it. The mount namespace is an important namespace for containerization because it allows for a container to have their own custom filesystem. The directory on which a filesystem is mounted is called the mount point and being a tree of directories, every filesystem has its own root directory. If a filesystem is present on disk, an empty directory in the tree is created which the filesystem should be mounted to before it can be used by other programs. Other filesystems can be mounted, either by the initialization scripts or directly by the users, on directories of already mounted filesystems. Container frameworks typically have the command line option to mount directories from the host into the container during initialization. A mounted filesystem is a child of the mounted filesystem to which the mount point directory belongs, in this case the host of the container.

2.6 Linux Filesystem

Until now it was discussed that namespaces have the ability to virtualize resources and limit visibility of files. On a file level there can also be restrictions; Linux files have features that describe who has which permissions over the file. Privileges and capabilities over files are of interest for containers because running containers requires some level of privilege escalation. Mounting a container filesystem, creation of namespaces and binding directories into containers are all actions that require a level of privilege. Since everything in the Linux kernel is considered a file, privileges are eventually granted on a file level which makes it worth to examine the Linux filesystem.

For Unix operating system and its derivatives such as Linux, everything is considered a file. A Unix file is an information container and is structured as a sequence of bytes. All files have file descriptors which contain information on the characteristics of the file. Particularly important in regards to containerization, are the access rights and file modes. File permissions on bind mounts are shared between the host and the containers. A file is marked by three types of flags:

- SUID: Default: A process executing a file keeps the UID of the process owner. If an executable file has a SUID flag set, the process gets the UID of the file owner.
- SGID: Default: A process executing a file keeps the user group ID of the process group. If an executable file has the SGID flag set, the process gets the user group ID of the file.
- Sticky: If an executable file has the sticky flag set, a request is made to the kernel to keep the program in memory after its execution terminates.

Setting up container environments requires system calls and functions which are considered privileged and require capabilities to execute. There are different ways to provide container frameworks with the required capabilities and privileges. Privilege levels can vary from all root access to only assigning capabilities to certain files. To run a container one of the following protocols has to be used:

- Root: Root user or sudo-granted users are capable of running containers.
- Root owned daemon process: A root owned daemon process manages containers. An IPC control socket is used for communication with this process. If trusted users are allowed to control the daemon they must acquire access to the control socket. This implementation caries the risk that user are given access to a control socket of a daemon process that is root owned.
- SetUID: A process with this capability can change its UID to any other UID. Binaries with the setUID bit enabled, are being executed as if they were running under the context of the root user. This enables normal (non-privileged) users to use special privileges, like opening sockets and isolating namespaces. A SetUID root program runs as root with all capabilities that come with root. If your container processes do not change UIDs/GIDs and are always running as the same UID they do not need this capability.

- User Namespace: Run a container with a limited set of privileged functions inside a user namespace. Privileges are only applicable within this namespace and not on the host.
- Capabilities: Manage privilege via capability sets. Non-privileged users can acquire privileges on a per file and process basis, which is the most fine-grained option out of the previous mentioned.

Certainly for high performance computing environments, the level of privilege that can be granted to users is a discussion point. High performance workloads are often being deployed on admin controlled shared clusters, on which root access is preferably not given to any user account. The risk is not necessarily in ill-intentioned users, but also malicious workloads that run inside a container. For high performance environments preferably one of the latter three previous mentioned protocols is used; setUID, User namespace or capabilities.

3 Frameworks

This section provides a background on container frameworks that are based on the Linux kernel and container schedulers. The Docker container framework and the Singularity container framework will be discussed. The Docker container is discussed since it is a framework widely adopted by tech companies and is very dominant in the container market. The Singularity framework is a framework specifically designed for High Performance Computing. The two container schedulers discussed are Slurm and Kubernetes. Slurm is a process and resource manager commonly used in high performance environments. Kubernetes is a dynamic container scheduler created by Google and deployed by various big tech companies.

3.1 Docker

The first container framework discussed is Docker, a lightweight and popular container framework [22]. Docker was created based on the idea of dotCloud which is a platform as a service (PaaS) for developers that enabled them to host, assemble and run their applications on the service. Developers were asking for the underlying technique and therefore Docker was created. Docker can provide virtualization of a single application and has a strong emphasis on enabling modularity. Modularity is the degree to which a system's components may be separated and recombined. This modularity translates into the use of microservices; Microservices are an architectural approach in which a single application is composed of many loosely coupled and independently deployable smaller components or services. Docker has a microservices-based architecture and enables easy linking of multiple containers together to create an application. Docker isolates the network stack and containers can communicate by container names. An essential principle of the microservice architecture is loose coupling. The micro service architecture makes it straightforward to elastically scale or update components of an application independently. Loosely coupled architectures are lean; With a single responsibility and without many dependencies. Communication should be lightweight as well and is often an HTTP resource API.

3.1.1 Images

Docker uses images which form a base for a containers' root filesystem. For Docker these images are not just one monolithic block, but are composed of multiple layers. Layering the filesystem creates a modular architecture that enables a more efficient workflow; For an update on an image, only a single layer has to be changed while the other layers can stay intact. Previously build layers can also be shared among images and if images are pulled, there will be no redundant copies of layers that are already present at destination. Each layer is a mapping with a command and this command is nothing but a file which will be stacked in to the image.

The container image layers are composed with the use of the Unionfs filesystem which gives a single coherent and unified view to files and directories of separate file-system. A storage driver handles the details about the way these layers interact with each other.

A different Linux distribution than the host OS can be used as the base of an image, which is commonly the first layer. Since all Linux distributions run the same Linux kernel and differ only in UserLand software, it is possible to simulate a different distribution environment by installing the required UserLand software and pretending it is another distribution. More specific, running a CentOS container on Ubuntu OS will mean that you will get the UserLand from CentOS, while still running the same kernel. The implication of this is, that while building the image, the installation repositories from package managers as apt-get and yum are from the specific base operating systems, in this case CentOS. After the base image layer, following layers can have varying purposes as installment of libraries, copying in local files or setting environment variables. These layers combined are able to compose a work-environment for a container that is customized to the workload it intends to carry. Docker images can be build locally or can be retrieved from the Docker hub, a public registry for Docker images.

3.1.2 Docker daemon

The Docker daemon runs as a service on the host operating system and can be seen as the executive force in the Docker framework. The root owned daemon does all the heavy lifting, listens for Docker API requests and manages Docker objects such as containers, images, volumes and networks. The daemon provides a place for shared state of all container objects and is able to manage resources as networks and volumes that may be shared between multiple containers. The Docker command line interface (CLI) and Docker daemon follow a client-server architecture; The Docker daemon runs as a background process that listens to requests on a unix domain socket (or IPC socket) at /var/run/docker.sock. The CLI checks if the syntax of the request to the daemon is correct and will create an API request. The unix domain socket Docker uses is root owned and anyone wishing to use Docker requires either root permission or Docker group membership. The Docker group is overseen by the administrator of a system and is intended for users who are not root, but want to make use of Docker. The security risk however is not mitigated by this group, because the attendees of this group still are given root permissions to use Docker they just do not have permission to run sudo commands for other purposes than Docker. The fact that the Docker daemon always runs as the root user has been a point of discussion with regard to security and is more importantly not always feasible for every user.

3.1.3 Docker runtime

. The Docker daemon itself does neither create nor run containers directly, but rather initiates their creation. Docker actually uses Containerd, another container framework, as its internal container runtime abstraction. Containerd pulls the container image from the docker registry and creates a

bundle based on the parameters provided by the Docker Daemon and the image. The default Docker (and Containerd) container runtime is Runc which is connected to Containerd via a shim (Fig.2). This shim is not only an entrypoint for Containerd but also a point of decoupling between the Docker daemon and actual running containers, which enables restarting the Daemon without affecting containers. Runc creates a new isolated container environment based on the given bundle and any meta data configuration files. The specified namespaces are the building blocks for the isolated environment and the image serves as the root file system of this environment. The namespaces Docker isolates by default are all namespaces except cgroup and user namespace:

- PID namespace for process isolation.
- NET namespace for managing network interfaces.
- IPC namespace for managing access to IPC resources.
- MNT namespace for managing filesystem mount points.
- UTS namespace for isolating kernel and version identifiers.

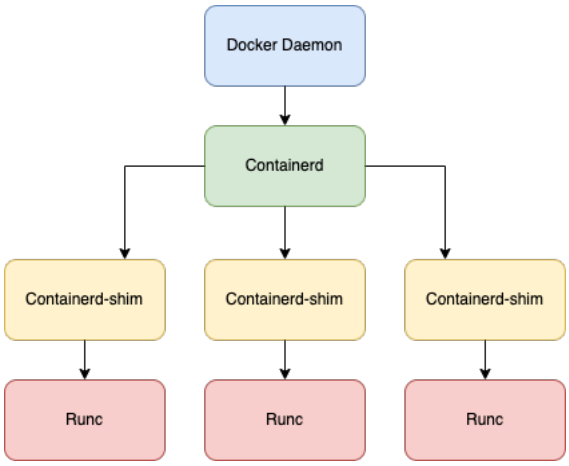

Figure 2: Docker Architecture

The isolation of namespaces in Docker also serves a security purpose; Since the Docker daemon is root owned, Docker containers could make changes to host namespace settings. Isolation of namespaces provides a barrier between the host and the container which reduce the security risk for the host. The UTS namespace is isolated because sharing the UTS namespace with the host provides full permission for each container to change the hostname of the host. The Docker documentation states that not isolating the UTS namespace is not in line with good security practice and should not be permitted.

Docker uses storage drivers not only to store image layers but also to store data during runtime in the writable layer of a container. The runtime environment can be seen as taking the filesystem that is created from the Image file and adding a writable layer. The container's writable layer does not persist after the container is deleted, but is suitable for storing short term data that is generated during runtime. It is preferred to write little data to a containers' writable layer and instead use Docker volumes to write data (Fig.3). Volumes are the favoured mechanism for persisting data generated by and used by Docker containers because bind mounts are dependent on the directory structure and operating systems of the host machine, while volumes are completely managed by Docker. Dockers philosophy is to have an environment that is only to a low extend integrated with the host and this includes how it persists data.

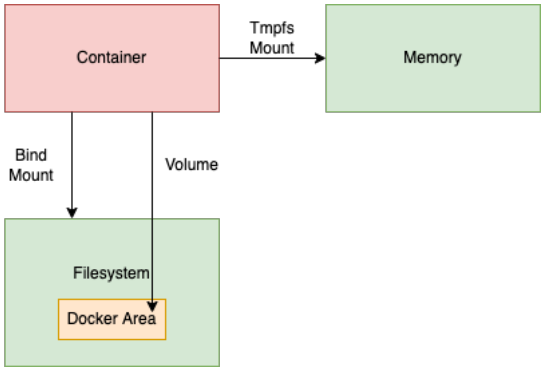

Figure 3: Docker volume mount architecture

3.2 Singularity

Singularity is designed for the field of high performance computing thus it deliberately works around root access requirements, which are often not feasible on a shared cluster. Singularity can be installed in any directory, but it must be ensured that the location you select supports programs running as SUID. Singularity containers are by default not isolated from the host filesystem. Singularity on purpose blurs the lines between the host and the container filesystem, enabling reading and writing of persistent data and easy leverage of hardware like GPUs and networks like Infiniband.

3.2.1 Singularity Image

Singularity uses a more simplified approach than Docker for building its images. Instead of a layered file-system, Singularity containers are stored in a single file which is recognizable by the .sif extension. This simplifies the container management lifecycle and facilitates features such as encryption and image signing to produce trusted containers. The execution of images is being done with use of the user privileges from the setUID functionality that allow to shortly change the UID to 0 (root) and do privileged system calls. The sif-file can be generated in various ways; Images can be build locally or pulled form hubs which serve as a warehouse for pre-generated images. Besides the Sylabs library, Docker images can also be pulled from the Docker hub, due to the Open Container Initiative that creates open industry standards around container formats and runtimes. After images from Docker are pulled the layers are being converted into a single file-system. The filesystem used by Singularity is a squashfs which is a compressed read-only filesystem for Linux.

3.2.2 Starter-Suid

The starter-SUID of Singularity is a root owned binary that starts the initialization of the container environment. During the building stage of the container root privileges are shortly granted to allow for privileged system calls and are dropped after this stage. The only code that escalates privilege within Singularity resides in the init function of the binary. Because starter-suid is a setuid binary, it starts execution as root and de-escalates privilege to the calling user's UID. Only when elevated privilege is required, for example when creating new namespaces, changing process capabilities or creating the root owned RPC process, does Singularity init re-escalate privilege. Once the privileged operation finishes, privilege immediately drops again.

3.2.3 Stage 1

The first process created is stage 1 which is mostly occupied with gathering configuration information and setting configuration options to be used later in execution. The root owned files that are gathered in this stage give the (unprivileged) user allowance to do certain privileged system calls, after which these privileges are immediately withdrawn. Stage 1 accesses the following resources:

- image.sif : User owned. A custom image file in SIF (Singularity Image Format) is a single executable file based container image.
- ecl.toml : Root owned. This file describes execution groups in which SIF files are checked for authorized loading/execution. The decision is made by validating both the location of the sif file in the file system and by checking against a list of signing entities.
- capability.json : Root owned. capability.json is the file maintained by Singularity where the capability commands create/delete entries accordingly.
- singularity.conf : Root owned. This is the global configuration file for Singularity. This file controls what the container is allowed to do on a particular host, and as a result this file must be owned by root.

Stage 1 begins with a SetContainerEnv() function that forwards any environment variable set for the container environment. Subsequently the init() function starts which will set a process running as [U=3000,P=2000] temporarily to [U=0,P=2000], allowing the execution of privileged systemcalls. During stage 1 privileged calls are the starter initialization, trying to load the overlay kernel module and checking if we are running as setUID. The starter configuration is initialized in shared memory to later share with child processes. After the privileged initialization part of the code is executed, a priv_drop() function is called that drops all root privileges and returns the process to its previous user id.

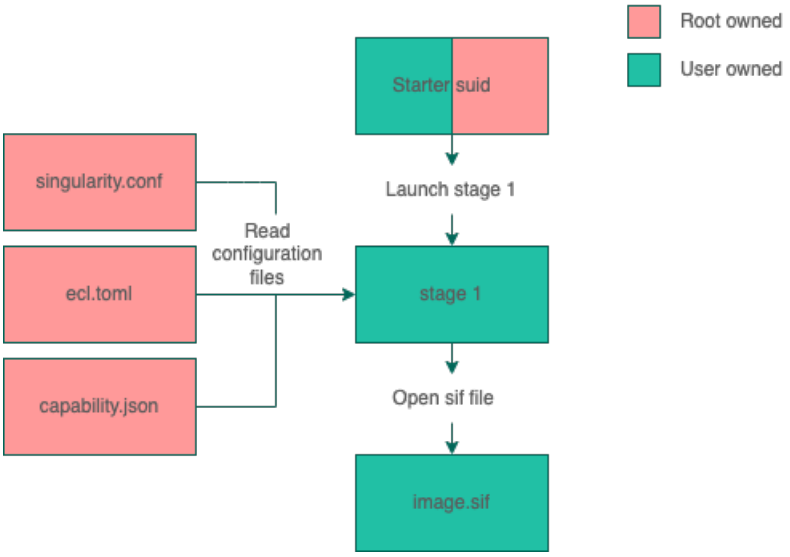

Figure 4: Stage 1 Singularity

3.2.4 Stage 2

Stage 2 is occupied with the execution of the action script that describes what should be done in the newly created container environment. After stage 1 finishes stage 2 starts with creating 2 socketpairs; A socketpair for master communication channel and a RPC socketpair for communication between stage 2 and the gRPC server. The container context, is in case of singularity the gRPC server which serves runtime engine requests via socket communication with the Master. The process where the gRPC server resides is root owned.

A function priv_escalate() is called that again sets the UID temporarily to 0 to allow privileged system calls. When the system calls during runtime of Singularity are being traced it is evident that the only namespace that is isolated is the mount namespace. An unshare(clone_newns) call is executed that unshares the mount namespace so that the calling process has a private copy of its namespace which is not shared with any other process. In privileged mode, first the filesystem UID is changed to the UID that the container runs in on the host, meaning the unprivileged UID. A mount namespace is created in ns/mnt after which a function apply_privileges() sets capabilities for this namespace. Privileges are then dropped after which the UID returns to its previous UID. An addRootfsMount() function adds the container image as the root filesystem in a read only mode, a requirement for container images. Other required directories as /sys, /proc, /dev are mounted into the container as well as the home directory and user defined mount binds provided through the command line. Stage 2 waits for the gRPC server to finish executing any instructions it receives.

After stage 2 has finished, a container environment with a custom file system runs in a way that resembles how a normal process would run on the host. No isolation of the cgroup and network namespace allows singularity to incorporate network drivers as infiniband and make use of shared memory communication layers. No isolation of the UTS namespace results in a localhost.domain entry in the /etc/hostname file. No isolation of the IPC namespace results in all containers being able to take note of all of the IPC on the host system, including of other containers. Since by default various host directories are bind mounted into the container, building a Singularity container in the /home directory, will give access to all files in this directory as well as its sub directories. The lines between the host filesystem and the container filesystem are intentionally blurred, which enables a Singularity container to run with strong resemblance to a normal process on the host, only with a custom filesystem.

3.3 Slurm

The Simple Linux Utility for Resource Management (Slurm) is an open source, fault-tolerant, and highly scalable cluster management and job scheduling system for large and small Linux clusters [9]. The scheduler contains an overview of the system resources and schedules processes or containers accordingly. Slurm is not primarily constructed to schedule containers but is a more general resource manager. The Slurm scheduler has three key jobs; The scheduler assigns resources for a period of time to anyone requesting to execute a job. This is done by slurmctld, the central management daemon of Slurm. Second, it provides a framework for starting jobs and monitoring their workflow on the set of allocated nodes. Finally, it arbitrates contention for resources by managing a queue of pending work. Slurm is commonly used in the high performance computing world to schedule distributed workflows. Slurm is a quite static scheduler and not particularly fit to dynamically (re)start containers. If the workload does not require dynamic schedulling, Slurm will suffice.

3.4 Kubernetes

Containers with their lightweight features are able to quickly instantiate and carry various types of workloads. Kubernetes is a container scheduler that caters to these features. Kubernetes is an open-source system for automating deployment, management, and scaling of containerized workloads [12]. Kubernetes provides its user with a dynamic scheduling service that can seamingless scale any service that is scheduled to it.

A pod is the basic object of Kubernetes which is the smallest component of a Kubernetes cluster, but can contain multiple containers. Just like Singularity, Kubernetes makes use of a gRPC server for constructing a client server architecture between pods. Pods manage most namespaces of the containers. Within pods network, UTS, IPC and cgroup are not isolated between containers. Kubernetes provides every pod its own cluster-private IP address and therefore there is no need to explicitly create links between pods or map container ports to host ports (Fig.5). Once a Pod is scheduled

to a node, the Kubelet on the node creates a new cgroup namespace for the Pod. The pod which contains the containers can be seen as an abstraction on top of an abstraction. The containers create their own environment which allows for communication via the container runtime interface. The pod can be seen as an abstraction on top of this that also has its own environment and communication protocol. Together they provide the user with a dynamic scheduling environment in which one can easily instantiate, update or destroy services and pods.

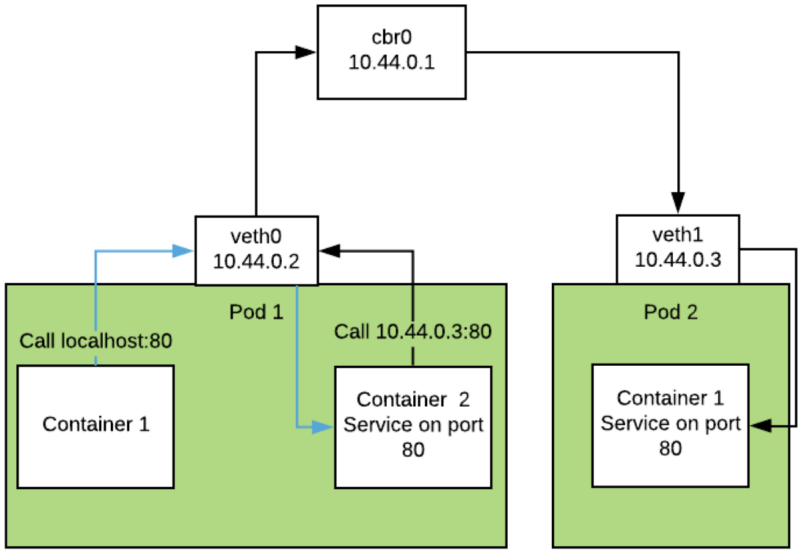

Figure 5: Pod network

The Kubelet is the primary node agent that runs on each node, which is responsible for managing the deployment of pods to Kubernetes nodes. The Kubelet communicates with containers via the container-runtime interface which is a plugin interface that enables the Kubelet to use a wide variety of container runtimes. During instantiating, the Kubelet is configured with a container endpoint and a container endpoint string. The string can have the value 'remote' or 'Docker' and the endpoint is a socket for container communication, which combined define where to pull images and where to post Kubelet requests. When a container on a node has to be destroyed, the Kubelet acting as a client sends a message to the gRPC server running on the node's container runtime interface instance. Subsequently the container runtime interface interacts with the container runtime engine installed on the worker node.

A Service is an abstraction which defines a logical set of Pods and a policy by which to access them. This can be referred to as a micro-service, which is an abstract way to expose an application running on a set of Pods as a network service. Services in Kubernetes maintain a clear endpoint for pods which remain the same, even when the pods are relocated to other nodes. While a service is the interface of a micro-service, the deployment is what keeps the actual underlying pods running. The deployment is what describes the state of the pods and updates them accordingly if required. As pods are an abstraction layer on top of containers, a deployment is an abstraction on top of pods and

a service is an abstraction on top of deployments. Together they provide the user with a dynamic scheduling environment in which containers can be easily instantiated, updated or destroyed.

3.5 Framework discussion

The container frameworks Singularity and Docker and the scheduling frameworks Slurm and Kubernetes have been described. Four frameworks developed for different use cases and with different strong- and weak-points. Kubernetes and Docker receive a lot of attention from the industry and greatly contribute to the wide adoption of containers on a larger scale. Slurm is a more general resource scheduler not optimized for container usage, but a well known framework in the high performance computing world. Singularity is developed to be operative in environments where only a limited amount of privilege is obtainable, often seen in high performance computing. This section will discuss which frameworks, not only on a privilege level, will best be suited for high performance computing and in this research.

Docker is mimicking a virtual machine on a software level, while Singularity can be regarded as a process with its own filesystem. Docker has a daemon process that runs in the background listening to requests on /var/run/docker.sock while Singularity uses no such socket and is started by its binary, located in the PATH environment variable. Docker isolates more namespaces compared to Singularity and by default has no connection to the host filesystem. Singularity favors isolation over integration which a suitable choice for high performance computing with data intensive workloads, with a high level of inter process communication.

During the building of Docker containers, the root owned control socket is passed through to a container and the control of this root owned socket is what makes Docker vulnerable for security issues. Anyone who accesses a container running as root, is able to start undesirable processes in it. For this reason, Docker is not installed on the cluster available in this research, the DAS-5 cluster. Singularity on the other hand follows the paradigma 'untrusted users running untrusted containers'. Neither the user scheduling a Singularity container nor the workload should be able to do any harm to a system. Singularity works with setUID which escalates privileges to root shortly; A technique that still caries some risk because the container is still given a UID=0 for a short time frame. A risk that system admins have to weigh, but can be regarded as acceptable since privileges are dropped without opportunity to regain. Sylabs is working towards instantiation of containers with capabilities, which allows for finer grained capability control and will still support all of Singularity's features.

What is interesting is how the two frameworks approach security issues in a different manner. Docker virtualizes the UTS namespace because otherwise the container would be able to change the UTS configuration of the host, since Docker runs as root. Docker has a security issue with its root setup and therefore isolates the container more from the host to introduce the idea of more security. Singularity does not have this issue since the container does not have root privileges, meaning Singularity could never make changes to the host UTS namespace.

Kubernetes is a dynamic scheduling service which introduces different abstractions, as pods, services and deployments, that provide the user with an environment that can be easily customized and updated. For high performance computing, which often contains static workloads, the Kubernetes framework has a redundant richness in features. A high performance computing workload can be scheduled not requiring any services and deployments and no duplication of the workload. The Slurm scheduler has some similar components as Kubernetes in terms of resource management, but it lacks the abstractions Kubernetes adds to the process in addition to the root privilege issue.

After evaluating the two container frameworks and two container schedulers and bearing in mind the restrictions of resources available, Singularity and Slurm will be used in the testing set up for containers and high performance workload. Not only in terms of security issues but also the minimal level of namespace isolation, makes Singularity is a better choice for shared high performance computing clusters. Less isolation also enables a transparant integration of message passing libraries for communication between tightly coupled processes of high performance computing. While Docker is a framework with an impressive user community, its features seem unfit for high performance computing. Kubernetes has a redundant richness in features for the static workload of high performance computing and also has privilege issues on a shared cluster, while the Slurm scheduler will suffice and is available on the Das-5 cluster.

4 Benchmarks

This section discusses different benchmark applications that pressure different levels of the memory hierarchy and will serve as test benchmarks for bare metal and containerized workloads. The benchmarks are originally measuring the maximum number of glop/s, gup/s and network bandwith to quantify the maximum performance of a supercomputer or cluster and therefore only smaller parts of the benchmarks that are relevant for these metrics are timed. This research is not aimed at quantifying the maximum performance of a system, but more aimed at quantifying overhead for containerized applications in its entirety. Measured peak performance is therefore not conclusive enough for bare metal and container performance comparison. In this research also a runtime metric is added to each benchmark.

4.1 Linpack

High performance Linpack (HPL) is a floating-point benchmark that solves a dense system of linear equations in parallel. The benchmark is solved in double precision (64 bits) arithmetic on distributed-memory computers. HPL is a compute intensive and a highly parallel process. The processors cache is utilized up to the maximum limit, though the HPL benchmark itself may not be considered as a memory intensive benchmark.

$$Ax = b$$

Here A is a given n × n matrix and b is a given n-vector. The dense linear system is solved for the unknown n-vector x. The size of the matrix is defined in the input file as well as the dimensions of the process-grid P and Q. The process grid defines the MPI Ranks (P*Q).

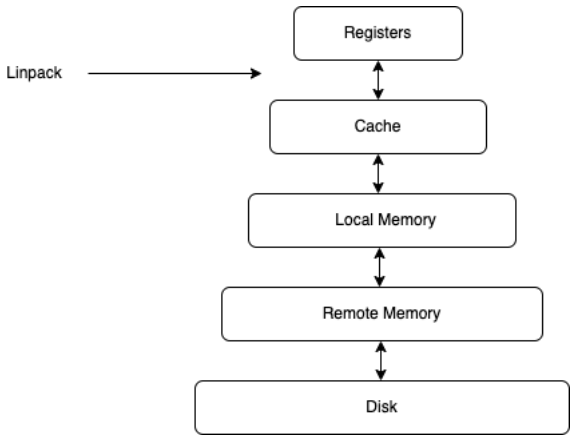

Figure 6: Linpack Memory Hierarchy impact

HPL is a cache intensive application. The HPL benchmark mostly stresses the register and cache levels in the memory hierarchy (Fig. 6). For HPL the dominant cost is CPU-related because computation has higher complexity order than communication: O(n3) versus O(n2) [10]. Previous research found that computational intensive jobs, either running on CPU or GPU, have small overhead for Docker containers [11]. Following these results it is not expected that Singularity will cause substantial overhead for cache intensive applications either.

4.2 Stream

The Stream benchmark is a benchmark for measuring sustainable memory bandwidth. This benchmark does not have a MPI implementation but only a Star and Single implementation. All three implementations do use the MPI library, but no substantial messaging between the separate processes. The single implementation schedules the whole workload on one cpu while the star-implementation can use multithreading.

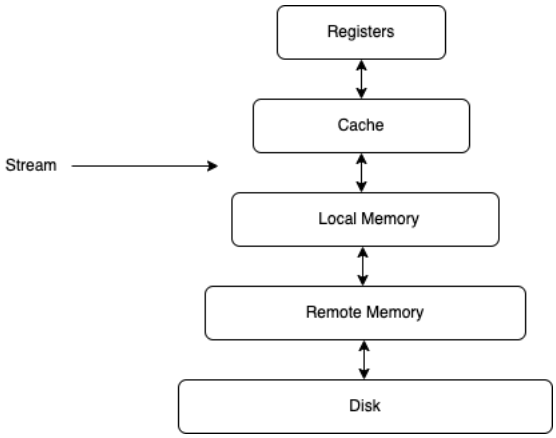

Figure 7: STREAM Memory Hierarchy impact

Stream is a benchmark that measures sustainable RAM memory bandwith which is located in the higher levels of the memory hierarchy (Fig. 7). Stream uses four simple vector kernels: Copy, Scale, Add and Triad and reports the corresponding computing rate in MB/s. Since the aim of this benchmark is not to assess any network performance, a MPI implementation would be redundant.

4.3 Parallel Matrix Transpose

The Ptrans benchmark performs a parallel matrix transpose. On a uni-processor a matrix transposition does not require matrix data to be transposed in physical memory. It can be transposed by exchanging row and column indices. In a distributed memory environment, as on a cluster, memory locations can not be simply interchanged between rows and columns. It should be noted that the performance of Ptrans can be dependable on the configuration of the processes grid. The performance of the benchmark is at best when the numbers of communicating pairs are at minimum. For example, for a matrix of 9x9, 3x3 processes grid has 3 communicating pairs (2-4, 3-7 and 6-8). However, a 1x9 processes grid has 36 communicating pairs. According to HPCC benchmark rules, only one configuration of a processes grid should be used for the entire benchmarks suite. If the process grid variables P and Q are relatively prime, the matrix transpose algorithm involves complete exchange communication [16]. If communication causes overhead for container implementations, different process grid configurations can cause fluctuating results.

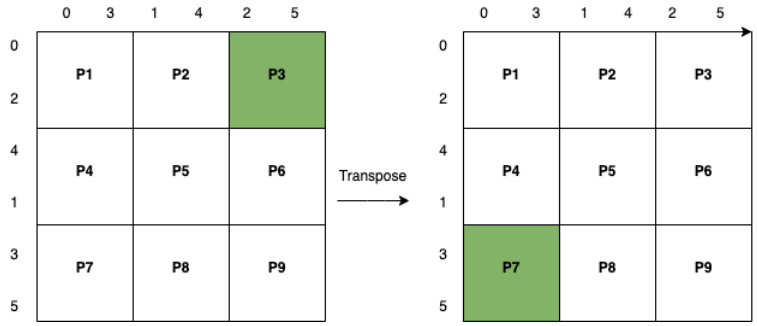

Figure 8: P=3,Q=3 Ptrans Processors, drawn from [16]

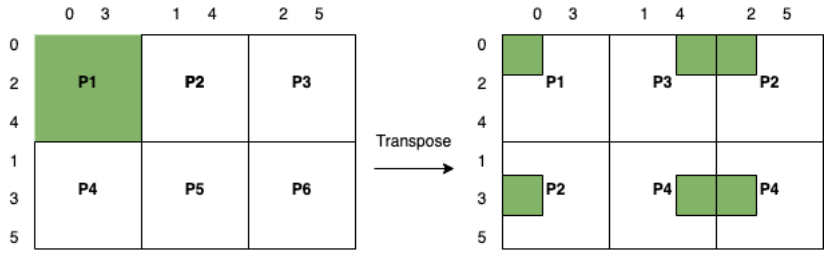

Figure 9: P=3,Q=2 Ptrans Processors, drawn from [16]

Ptrans measures communications where pairs of processors exchange messages of significant size simultaneously. This benchmarks assesses the total communication capacity of the system interconnect. Two random distributed matrices A and B of size m-by-m are created and subsequently A' + B is computed.

$$A \leftarrow A^t + B$$

Ptrans benchmark makes use of a block-cyclic distribution for which the variable 'NB' is set to define the block size for the data distribution in the input file for the benchmarks. For every process scheduled first the master thread is active which performs the block-cyclic division of the matrix parts. Current computer architectures own hierarchical memories in which access to data in upper levels of the memory hierarchy, namely registers, cache and local memory, are faster than access to lower levels, shared or disk memory. To exploit this hierarchy, block-partitioned algorithms are preferred for dense linear algebra systems, where operations are performed on sub-matrices rather than individual elements.

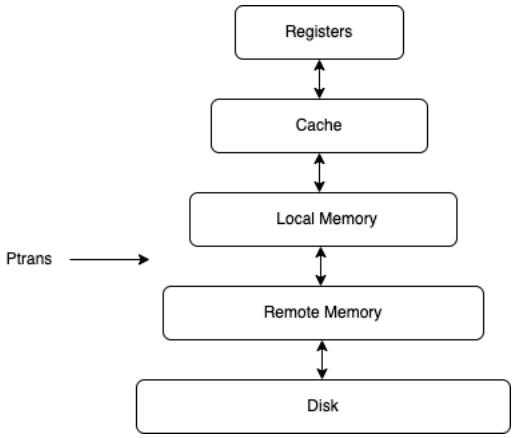

Figure 10: PTRANS Memory Hierarchy impact

This benchmark ultimately stresses the interconnect and the result is returned in gigabytes per second (Fig.10). The transfer rate of a communication mechanism is the amount of data that can be sent per unit time. The data transfer rate (in gbyte/s) is calculated by dividing the size of n 2 matrix entries by the time it took to perform the transpose and generation time of the matrix is not included in this rate.

4.4 Random Ring Latency Bandwith Benchmark

The Latency Bandwith benchmark measures the communication pattern with a parallel all-processes-in-a-ring architecture to assess the bandwith of a network (Fig.11). All processes are arranged in a ring topology and each process sends and receives a message from its left and its right neighbor in parallel. The ring is the geometric mean of the bandwidth of ten different randomly chosen process orderings in the ring. With this type of parallel communication, the bandwidth is defined as total amount of message data divided by the number of processes and the maximum time needed for all processes. The operation count is linearly dependent on the number of processors in the tested system

and the time the tests take depends on the parameters of the tested network. This benchmark takes into account interprocess communication as well so different process configurations have an effect on the benchmark results.

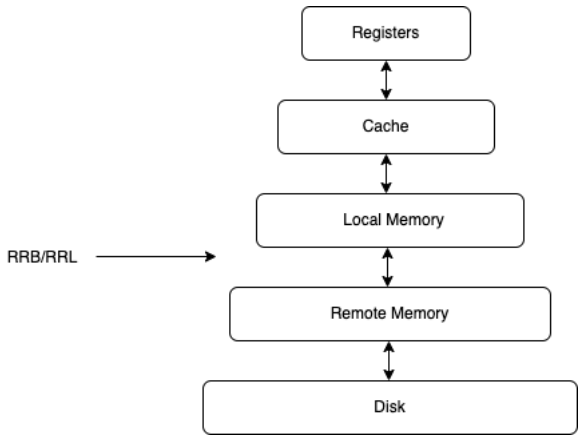

Figure 11: RandomRing Bandwith and Latency Memory Hierarchy impact

4.5 FFT

Fast fourier transform computes the discrete fourier transform (DFT) of a sequence or its inverse (IDFT). DFT converts a finite sequence of equally-spaced samples of a function into a same-length sequence of equally-spaced samples of the discrete-time Fourier transform (DTFT), which is a complex-valued function of frequency. FFT is mainly used for audio and acoustic measurement. A signal is converted into spectral components and provides frequency information about the signal. The benchmark measures the floating point rate of execution of double precision complex one-dimensional discrete fourier transform. The formula for FFT used is:

$$Z_k \leftarrow \sum_{j}^{m} z_j e^{-2\pi i \frac{jk}{m}}; \quad 1 \leq k \leq m$$

Figure 12

FFT can be described as a memory-bound application that will not correspond strong to an increase in the number of cpu cores [17]. While compute-bound application such as matrix multiplication gain performance from an increased number of computational units, memory-bound applications such as the Fourier transform (FFT) have not benefited as much. Large Fourier transforms do not use the cache hierarchy and bandwidth to main memory efficiently due to the non-unit stride access

patterns inherent to the algorithm [17]. This makes the task of hiding the latency of strided memory access patterns when accessing main memory difficult. This benchmark puts most pressure on the network and is aimed to assess latency in messaging and bandwith of a system. Both the local memory bandwidth and the network bandwidth of a cluster. A sequential implementation is mainly pressuring memory IO while a parallel implementation is mainly pressuring the network [18]. The FFT benchmark is intended to to pressure the mid and lower levels of the memory hierarchy (Fig.13).

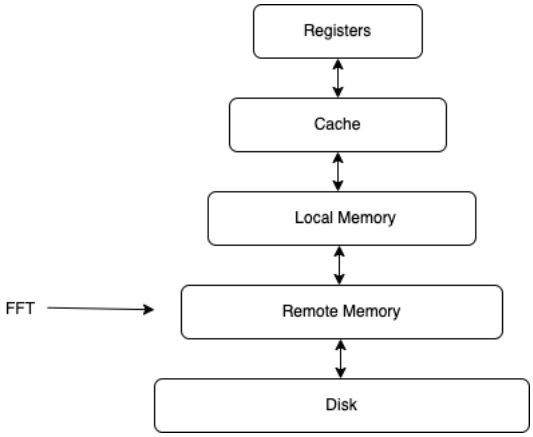

Figure 13: FFT Memory Hierarchy impact

4.6 Random Memory Access

The Random Access tests the speed at which a machine can update the elements of a table spread across global system memory (Fig.14). Random memory performance often maps directly to the application performance. A small percentage of random memory accesses (cache misses) in an application can significantly affect the overall performance of that application. Giga updates per Second (gup/s) is a measurement of how frequently a computer can issue updates to randomly generated locations. The RandomAccess benchmark stresses the latency and especially bandwidth capabilities of a machine. The gup/s of a system is computed by the amount of memory locations that can be updated randomly in one second, divided by 1 billion (1e9). The term 'Random' here does not refer to RAM memory but means there is little to no relation between the current and next address to be updated. A table is constructed and divided among processors and processes. Subsequently an address stream is generated for every process. The value at that address is read and modified by an integer operation; Add, And, Or or Xor with a literal value. The updated value is written back to memory. Every cpu in the system operates on its own address stream.

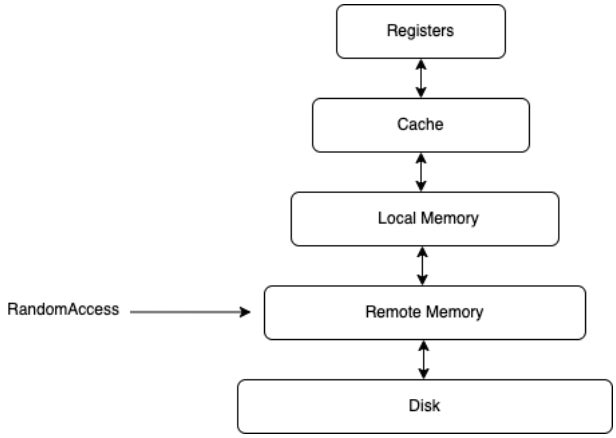

Figure 14: RandomAccess Memory Hierarchy impact

5 Singularity High Performance Cluster

This section will describe the setup and specifications of the Singularity and bare metal implementations of the hpcc benchmarks. The experiment is set up on the DAS-5 cluster of the TU Delft. DAS-5 is a cluster of 68 machines with dual 8-core CPUs connected with InfiniBand FDR links. Infiniband is a network commenly used in high performance computing. InfiniBand sends data in serial and can carry multiple channels of data at the same time in a multiplexing signal. The network has a point-to-point switched io fabric architecture designed to increase the communication speed between cpu's and devices within servers and subsystems located throughout a network. Singularity does not isolate the network and hardware and can incorporate Infinband with ease. Slurm is used as the process and container scheduler and the DAS-5 cluster has 45 nodes available for Slurm usage.

5.1 Singularity and MPI

The hpcc benchmarks and commonly High performance applications use the message passing interface (MPI) for communication between the tightly coupled processes. Since Singularity isolates no network namespace or cgroup, the MPI library can be used in an identical manner as in bare metal set up. Sylabs mentions two ways to deploy MPI; a hybrid approach or a bind approach [19]. The only requirements is that all MPI containers specifically have to live in the same user namespace as the host. MPI commands as `mpirun` and `mpiexec` can be used as they normally would in bare metal setup. The specific steps are as following:

- The `mpirun` command is called by the the resource manager or by the user from shell
- OpenMPI then calls the process management daemon; the run-time layer or Open Run-Time Environment (ORTED)
- The ORTED process launches the Singularity container requested by the mpirun command
- Singularity builds the container and namespace environment
- Singularity launches the MPI application within the container
- The MPI application launches and loads required OpenMPI libraries
- The Open MPI libraries connect back to the ORTED process via the Process Management Interface (PMI)
- Processes within the container run as they would run directly on the host.

The Hybrid Approach. The hybrid approach uses the host installed MPI and also the MPI library installed in the container filesystem. The launcher commands called on the host are from the host MPI while inside the container environment the MPI from the container filesystem is used.

The Bind Approach. The bind approach only uses the MPI version on the host. This is called the Bind model since it requires to bind mount the MPI version available on the host into the container. For this approach mounting user-specified files, which is sometimes disabled by system administrators for operational reasons, should be allowed.

The hybrid approach was first used in this experiment; The same MPI version 4.0.2 was installed in the container environment as on the host. The disadvantage of this implementation is that if the container filesystem and therefore MPI is not build on the same machine that the container is run on, it is not an optimized version for the specific machine. Building a Singularity image requires 'sudo' privileges which are not feasible on the DAS-5 cluster. The image is therefore build remote and pulled to the DAS-5 cluster. The hybrid approach can therefore result in a sub-optimal performance of the MPI benchmarks. The dynamic linkloader showed different dependencies, as expected, since both libraries were compiled and run in different environments. This is a side-effect of the hybrid approach that can skew results.

To obtain results that would not be skewed in any way, the host MPI was binded into the container. To exactly mimic the host environment, the MPI directory and recursively all files of all dependent objects were bind mounted into the container. The main reason why dynamic linking is usually not preferable for containers is that the system om which the executable is build can be different from the system on which the application is run. Even for similar Linux distributions as Ubuntu and Debian this can cause issues with naming of files and libraries. This is why executables are preferably build in the same container image as where they run. In this research the same executable was used for the bare metal as the container setup, to limit the possibility of any causes that could skew results. The bind approach sidelines the portability feature of containers. However, for this specific use-case it is required to obtain results that serve as a proper base for comparison of bare metal and container performance and are not skewed in any way.

5.2 MPI Communication Layers

The Openib btl layer is used for infiniband communication between processes. The vader BTL is a low-latency, high-bandwidth mechanism for transferring data between two processes on the same node via shared memory. The vader BTL layer is supported by the Open Portable Access Layer (OPAL) which enables sharing memory between processes on the same server. Since no isolation of network or cgroups is being done by Singularity, all MPI communication layers can be used without any issue.

5.3 Runtime

In addition to the benchmark metrics that measure the peak in processing or the highest bandwith use of a system, a script that measures the run-time of an application is added. Run-time is an important measure to assess overhead incured by containers opposed to bare metal. Runtime is measured by adding a `MPI_Wtime()` functions before starting a benchmark and after a benchmarks returns and subtracting the two time points.

5.4 Benchmark Input Configuration

The configurations for the benchmark are given in a input file named 'hpccinf.txt'. Most important for the experiment are the following parameters:

- Ns : The problem size of the matrix. The matrix will have dimensions : problem size x problem size.
- NBs : The number of block sizes that one wants to use.
- Ps : Number of grid rows
- Qs : Number of grid columns

The variables P an Q together account for the process grid size to which the workload is divided. The data is distributed into a two dimensional P-by-Q grid of processes according to the block-cyclic scheme to ensure the scalability of the algorithm as well as proper load balance. The n-by-n+1 coefficient matrix is logically partitioned into nb-by-nb blocks, that are cyclically divided onto the P-by-Q process grid. The input tuning file states that for two possible PxQ settings of 1x6 and 2x8:

"If one was starting xhpl on more than 16 nodes, say 52, only 6 would be used for the first grid (1x6) and then 16 (2x8) would be used for the second grid. The fact that you started the MPI job on 52 nodes, will not make HPL use all of them. In this example, only 16 would be used." This is an important notion for the settings of the experiment because it implicates that the process grid dimensions should be at least as large as the number of nodes that are used. If the process grid has more processes than the number of MPI processes scheduled, P and Q are refactored to match the number of MPI processes and also the problem size is refactored. The PxQ grid-size set is for the Ptrans and HPL benchmarks while the other applications receive an input based on the grid-size. The benchmark is designed to exploit resources at its maximum and therefore also scales up problem size to what it believes is appropriate for the number of processes. To ensure the same problem size for all experiments, for comparison reasons, the process grid should match the number of MPI processes. Since the dimension of the process grid can affect communication and results, the process grid is aimed

to be as square as possible for the given input of nodes and processes. A function is implemented in the `init script` that adheres to this. The Ns variable describes the input size for Ptrans. A Ns input of 600.000 accounts for a 300.000 x 300.000 matrix. The other benchmarks receive other forms of input, for which the size is derived from the Ns variable. Linpack is run with a different input configuration file than the other benchmarks, since the application has different memory requirements. Linpack showed memory errors for problem sizes, which the other benchmarks could still handle.

5.5 Thread and Node Configuration

The init script can run subsequent number of nodes and processes. The script takes in the following arguments:

- start node range
- end node range (exclusive)
- start number of process per node
- end number of process per node (exclusive)

There is a correction being made on the number of processes per node; The nodes of the DAS 5 cluster are dual 8 core. A single process or container is only able to use 8 of the 16 cores of a node and therefore 16 of 32 threads. The correction made is that if you mean to assign a single process occupation to a node, 2 processes/containers are scheduled. A single process/container scheduled would leave half of the node idle. Problem size, the input matrix dimension, will correlate with increase of the number of nodes.

The multi-thread library used is Openmp which is aware of the number of threads owned by the node which it is scheduled on, but is not aware of any other processes being scheduled on the same node. This default way of operating is used for the overthreading implementation; For every new process scheduled 16 threads, the number of threads Openmp identified on the node, are being used. For the non-overthreaded implementation, processes are gradually increased and the number of threads are divided equally among the number of processes per node. For the no overthreading implementation only configurations for which the following equation, for which NP is the number of processes scheduled per node, holds are considered:

$$32 mod NP = 0$$

Other configurations would slightly over subscribe threads or use less threads than available. Configurations of 2, 4, 8 and 16 processes per node satisfied this equation. An environment variable

for the number of threads is set in the Slurm scheduling script. The over threaded implementation is run in configurations of 2 to 8 processes per node. Higher configurations would cause too much performance degradation due to too heavily over subscription of threads.

- Overthreading: for every process added 16 more threads are used.
- No overthreading: the number of threads available are divided over the number of processes.

For all benchmarks Multi-threading is used, except the RandomAccess benchmark which only has a single-thread implementation. Overthreading is used for every benchmark except RandomAcess and Linpack. Overthreading the register and cache intensive Linpack benchmark causes extreme strong performance degradation and was therefore not included. The number of nodes used for the benchmarks was subjective to the number of nodes on which Slurm is installed and to the activity of other users. Due to different configurations, repetition and signficicant problem sizes, runtime could take up to several days. Benchmarks that pressure the network, Ptrans, Random Memory Access, Random Ring Latency Bandwith and FFT, were run on 40 nodes with a problemsize of 600.000. To enable other user activity on the cluster, cache intensive Linpack was run on 25 nodes with a problemsize of 200.000. Stream that measures the local memory bandwith has a workload that can not be distributed and is therefore run on 1 node. The blocksize NBs is set to 1000, which enables an even division of the workload.

6 Results

This section covers the results from the 5 benchmark applications; Linpack, Ptrans, Random Memory Access, Stream and the Bandwith Latency benchmarks. The official benchmark results and runtime results will be discussed. To be able to answer the 3 research questions, results will be discussed for different levels of threading and for different architecture configurations. Also an analysis will be done that aims to reason about the benchmark implementations and their optimal process configuration. For convenience the word 'processes' covers both container processes as well as bare metal processes, in case no other specification is made. To determine whether results are statistically significantly different, a T-test with a p value 0.05 is used that tests whether the runtime is lower for bare metal- than for container- implementation and tests whether the hpcc benchmark results are higher for bare metal- than for container- implementation. A p-value near the left tail of the test (<0.05) meaning the runtime or hpcc metric distribution indeed follows the relation as stated in the null hypothesis, but with a neglectible difference. A p-value near the right tail of the test (>0.95) meaning the runtime or hpcc metric distribution does not follow the relation as stated in the null hypothesis, but with a neglectible difference.

6.1 Linpack

The Linpack benchmark that performed the floating point operations on a dense linear system shows stable results for every repeated run. The Linpack benchmark was run on 25 nodes with a problem size of 200.000. For the no overthreading configuration, The MPI implementation and Container implementations show close to equal run-time results (Fig. 15). One process or container per core is the optimal configuration in terms of run-time. Linpack is stated to be perfectly parallel and can fully utilize the cpu space. For a compute intensive application that benefits from efficient cache and register-use, it appears that dividing the workload over multiple smaller processes adheres to the efficiency of the benchmark. The workload of the master thread is also parallelized by adding more processes, which can cause a decrease in runtime.

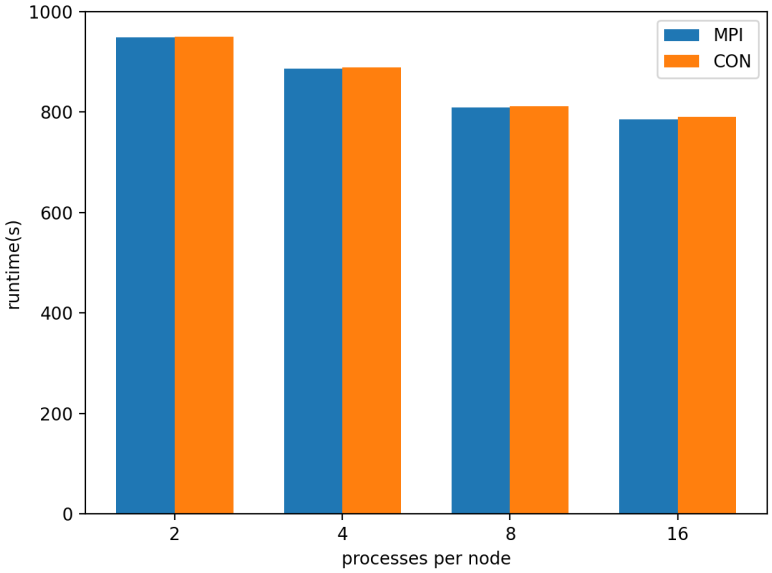

Figure 15: No overthreading: NBs 200.000, Nodes 25

The gflop/s of a system that the Linpack benchmark aims to approximate, is almost perfectly linearly correlated with runtime (Fig. 17). This should be taken into account when interpreting the actual meaning of this gflop metric. Runtime shows correlation with the number of processes scheduled per node. This can be seen in Fig. 15 by the gradual (linear) decrease of runtime. As expected there is also correlation between gflop/s and processes per node (Fig. 16). The 3 metrics all approximate a linear correlation. These results implicate that the gflops measure, even for a perfectly parallel application as Linpack, is actually reflecting a suboptimal configured application instead of being a system metric. For the gflop metric to have meaning with regards to the system, any bottleneck within an application should be removed. The gflop metric, as the runtime metric, shows near equal results for mpi and container implementation.

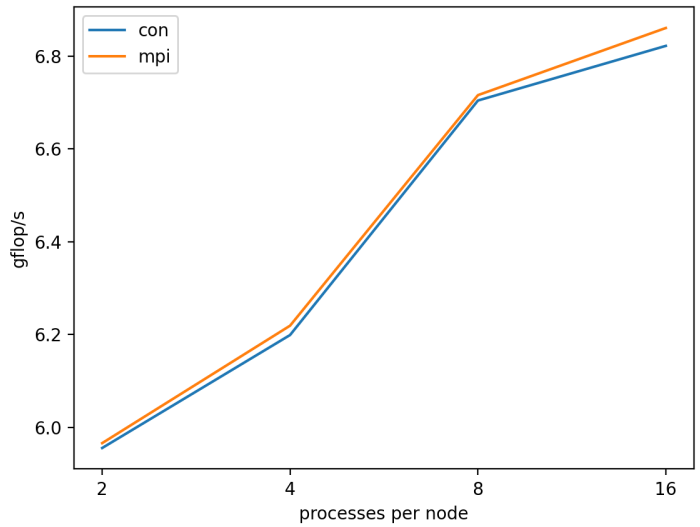

Figure 16: No overthreading: NBs 200.000, Nodes 25

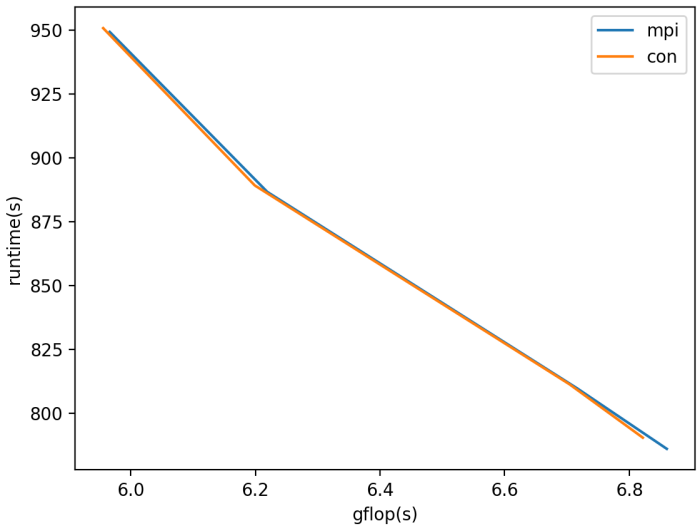

Figure 17: No overthreading: NBs 200.000, Nodes 25

The null hypothesis being tested are container runtime is larger than mpi runtime and container gflop/s is lower than mpi gflop/s (Tab.1). As expected, a higher runtime results in a lower gflop metric meaning if mpi runtime is slightly lower than container runtime, mpi gflop is slightly higher than container gflop. With p-values below 0.05, runtime and gflop distributions of mpi and container implementation are not significantly different. For a compute intensive application as Linpack that is cache and register intensive, deploying Singularity seems to have little to no effect on performance.

T-test	2 proc	6 proc	8 proc	16 proc
Con Runtime > Mpi Runtime	p=0.0101	p=0.0184	p=0.0202	p=0.0304
T-test	2 proc	6 proc	8 proc	16 proc
Con gflops < Mpi gflops	p=0.0095	p=0.0116	p=0.0215	p=0.0337

Table 1: T-test HPL, p=0.05

6.2 Stream

The benchmark Stream measures sustainable memory bandwith which is located between local memory and cache. Only an overthreaded configuration is run, since workload can not be divided among multiple processes but only replicated, meaning scheduling more processes on a machine is only doubling the workload. The mpi implementation of Stream is useful for profiling the entire sustainable memory bandwith of distributed system and finding any nodes that are lagging. Since this research is not aimed at specifying features of a particular system but on assessing overhead of containerization, Stream is run on one and the same node.

Stream is scheduled with a single processes of which the threads gets incremented with 16 for every new iteration and the input size of the array is 60.000. The graphs of the four kernels that were run were Add, Copy, Scale and Triad (Fig. 18, Fig. 19, Fig. 20, Fig. 21) are from a single run as well as runtime in Fig. 22. The graphs emphasize the randomness that occurs when running Stream and that supremacy of one implementation over the other is not conclusive. The first configuration of 16 threads is occupying all available threads while the subsequent configurations are over subscribing the machine. The benefit applications can have from oversubscribing threads when waiting for other processes and communication, does not apply to the Stream benchmark since no division of the workload is possible. Oversubscribing threads by 16 threads shows to not be beneficial for either of the kernels since the peak memory bandwith lowers (Fig. 18, Fig. 19, Fig. 20, Fig. 21) while runtime increases (Fig. 22). The mpi and container implementation show overall close results and the T-test shows no sigfnicant differences between the mpi and container implementations (Tab. 2, Tab. 3, Tab. 4, Tab. 6, Tab. 5). There is no reason to assume for an application as Stream that works on the local memory level of the memory hierarchy, Singularity causes performance degradation.

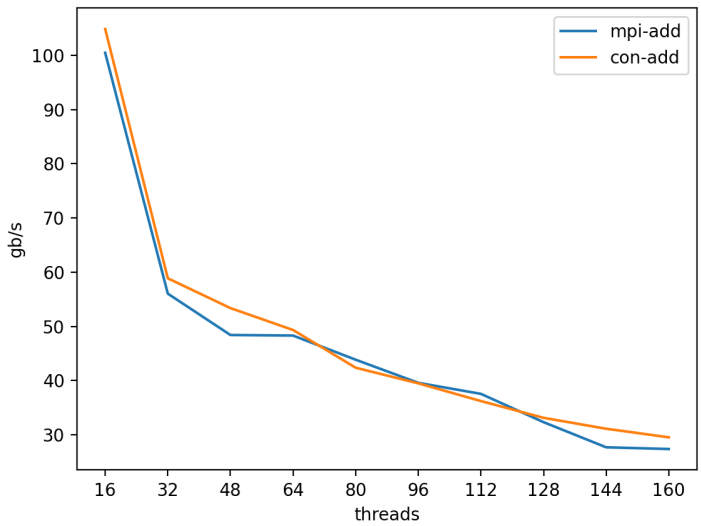

Figure 18: STREAM Add Kernel

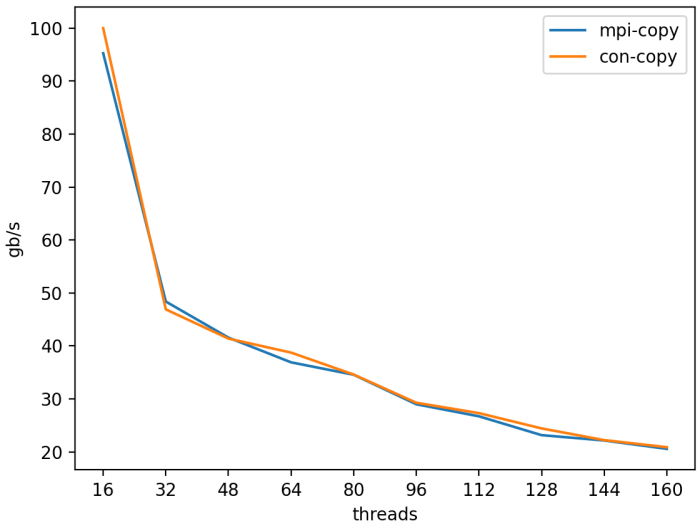

Figure 19: STREAM Copy Kernel

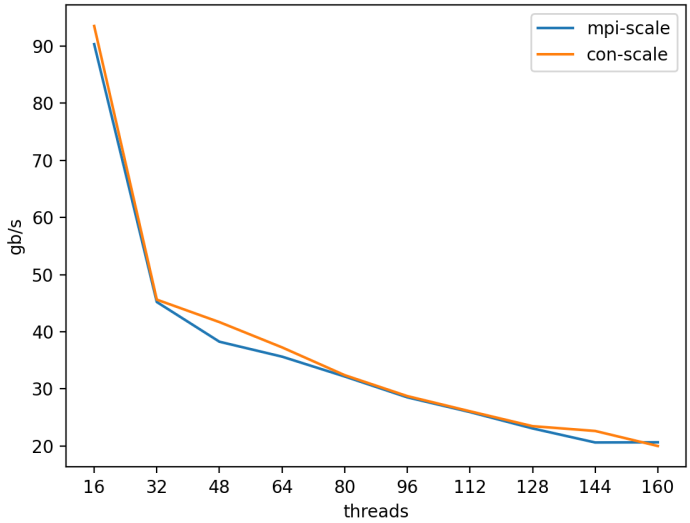

Figure 20: STREAM Scale Kernel

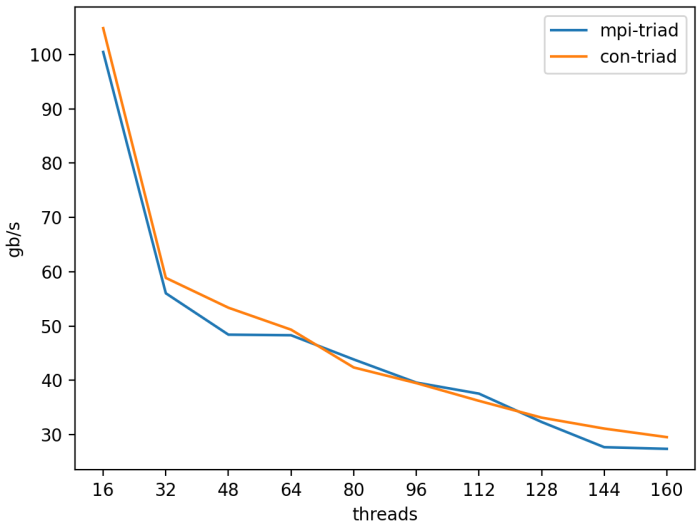

Figure 21: STREAM Triad Kernel

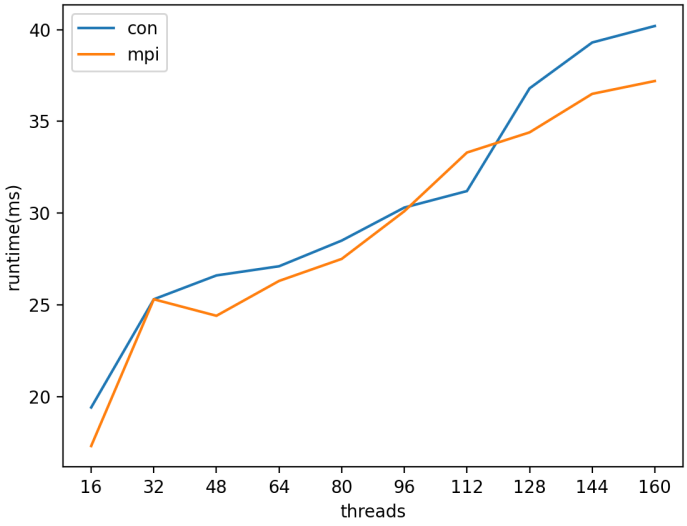

Figure 22: STREAM Runtime

T-test	2 proc	4 proc	6 proc	8 proc
Con Runtime > Mpi Runtime	p=0.0131	p=0.0392	p=0.0312	p=0.0403
	10 proc	12 proc	14 proc	16 proc
Con Runtime > Mpi Runtime	p=0.0021	p=0.9911	p=0.0237	p=0.9845

Table 2: T-test Stream Runtime, p=0.05

T-test	2 proc	4 proc	6 proc	8 proc
Con Add > Mpi Add	p=0.0321	p=0.0089	p=0.0221	p=0.0019
	10 proc	12 proc	14 proc	16 proc
Con Add > Mpi Add	p=0.0263	p=0.9721	p=0.0126	p=0.0248

Table 3: T-test Stream Add, p=0.05

T-test	2 proc	4 proc	6 proc	8 proc
Con Copy > Mpi Copy	p=0.9927	p=0.0256	p=0.0241	p=0.0018
	10 proc	12 proc	14 proc	16 proc
Con Copy > Mpi Copy	p=0.9887	p=0.0011	p=0.0133	p=0.0444

Table 4: T-test Stream Copy, p=0.05

T-test	2 proc	4 proc	6 proc	8 proc
Con Scale > Mpi Scale	p=0.997	p=0.0148	p=0.0391	p=0.0227
	10 proc	12 proc	14 proc	16 proc
Con Scale > Mpi Scale	p=0.0007	p=0.0022	p=0.9969	p=0.0021

Table 5: T-test Stream Scale, p=0.05

T-test	2 proc	4 proc	6 proc	8 proc
Con Triad > Mpi Triad	p=0.0222	p=0.0208	p=0.0012	p=0.0373
	10 proc	12 proc	14 proc	16 proc
Con Triad > Mpi Triad	p=0.9910	p=0.0116	p=0.9933	p=0.0401

Table 6: T-test Stream Triad, p=0.05

6.3 Ptrans

The Ptrans benchmark measures the inter process communication speed of the system. The runtime results for the no over-threading configuration show a strong decrease in runtime for more than 2 process per node (Fig. 23). The mpi configuration shows overall a slightly faster runtime than container configuration, but only marginal for the optimal configuration of one process per 2 cores i.e. 8 processes per node. Since the master thread of a process can only communicate with one node at a time, processes can be stuck in blocking communication. An improvement in runtime can be explained by the diminished workload of the master thread and less blocked waiting time. More processes with smaller workloads can have handle the smaller `MPI_Sendrecv` messages faster and leave their blocked stage.

The Ptrans benchmark determines flops based on a timer that starts after the matrix is generated. The function `sltimer` is started right before the transpose function and stopped immediately after. The timer does not incorporates generation time of the matrix or any other function. This timer function differs from the timer function used for Fig. 23 and Fig. 25. The runtime timer function measure the entire runtime of the benchmark and does not isolates the transpose part. There were slight fluctuations in the difference in runtime between containerized and bare metal form for different runs and neither configuration showed a clear preference for one implementation over the other. The run in Fig. 23 shows the mpi implementation has a lower runtime for 2 and 16 processes per node but the container implementation a slight lower runtime for 4 processes per node. The optimal configuration of 16 processes per node show near equal results. This optimal configuration is stating that every core should have their own process or container. The terra flop performance of the benchmark shows the highest rate at a configuration of 8 processes per node (Fig. 24). A slight lower runtime for the 16 processes per node in Fig. 23 indicates that the transpose of the matrix is executed faster with 8 processes but the generation of the matrix is executed faster with 16 processes. Following the T-test results (Tab. 7), there is no reason to assume Singularity causes significant performance degradation for the not overthreaded implementation of a communication intensive, parallel matrix transpose.

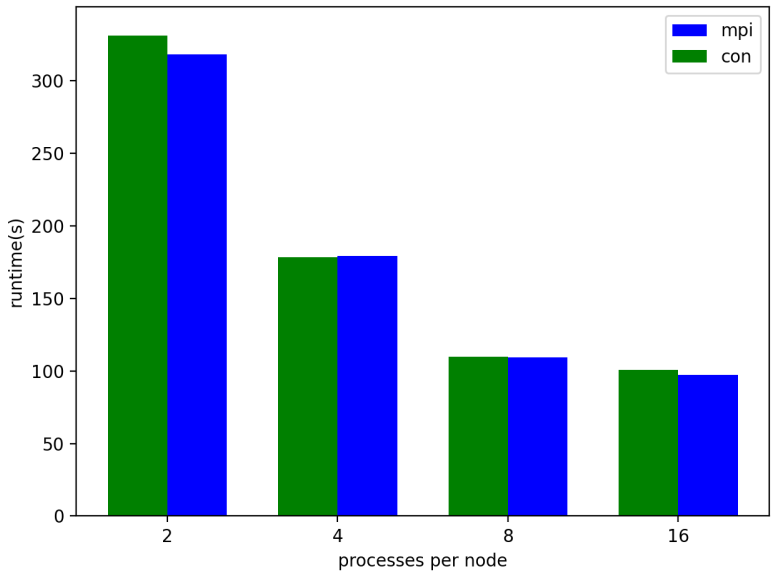

Figure 23: No overthreading: NBs 600.000, Nodes 40

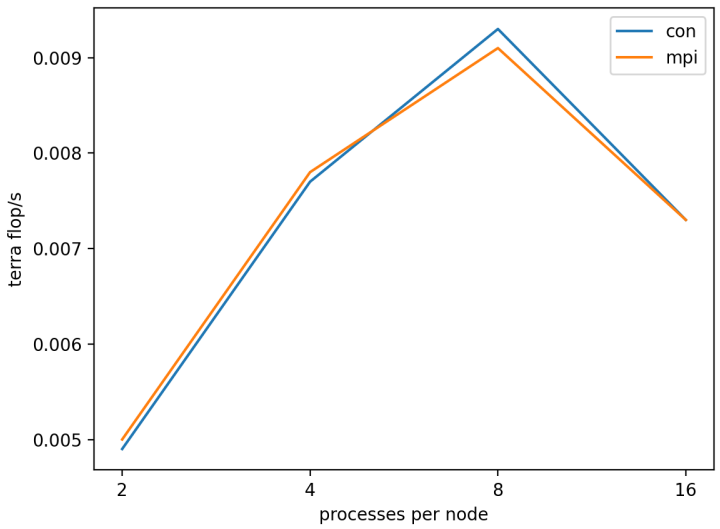

Figure 24: No overthreading: NBs 600.000, Nodes 40

T-test	2 proc	6 proc	8 proc	16 proc
Con Runtime > Mpi Runtime	p=0.0230	p=0.0175	p=0.0011	p=0.9969
T-test	2 proc	6 proc	8 proc	16 proc
Con terraflops < Mpi terraflops	p=0.0167	p=0.0229	p=0.0010	p=0.0023

Table 7: T-test PTRANS not-overthreaded, p=0.05

The overthreaded implementation shows an optimal configuration of 2 processes per node for the runtime metric (Fig. 25). Results show near equal runtime for the container and mpi implementation for all different configurations. Overthreading seems to be disadvantageous for the performance of the benchmark. The terra flop performance of the transpose part of the benchmark peaks at 8 processes per node which is a strongly overthreaded implementation (Fig. 26). The sudden drop at 6 processes can be caused by a higher transpose time. As discussed the configuration of the process grid can have a strong effect on the amount of communication required. The terra flop performance of 8 processes per node can be influenced by factors as overthreading being beneficial for the transpose part which uses blocking mpi communcation or by the amount of communication required for the transpose. For overthreaded as not overthreaded implementation on a 40 node cluster, no significant difference was found. With a p value of 0.05, the null hypothesis Container Runtime is higher than Mpi Runtime and Con terraflops is lower than Mpi terraflops would both be rejected. Following these results there is no reason to assume Singularity causes significant performance degradation for a communication intensive, parallel matrix transpose.

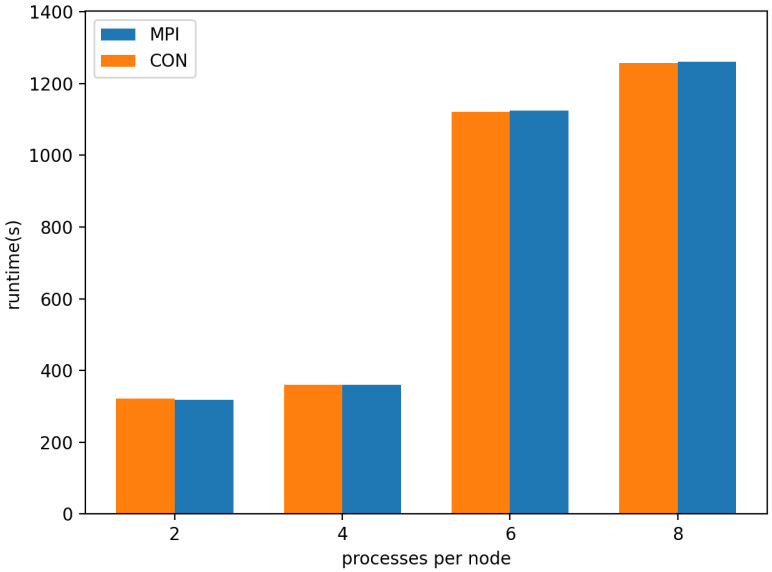

Figure 25: Overthreading: NBs 600.000, Nodes 40

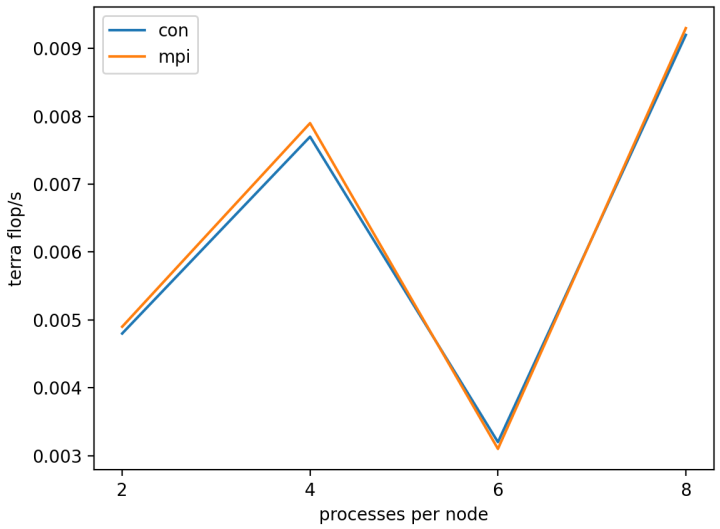

Figure 26: Overthreading: NBs 600.000, Nodes 40

T-test	2 proc	4 proc	6 proc	8 proc
Con Runtime > Mpi Runtime	p=0.0210	p=0.0165	p=0.0119	p=0.0193
T-test	2 proc	4 proc	6 proc	8 proc
Con terraflops < Mpi terraflops	p=0.0118	p=0.0277	p=0.0163	p=0.0180

Table 8: T-test PTRANS overthreaded, p=0.05

6.4 RandomAccess

The Random Access benchmark does not have a multithreaded implementation and is only able to be run with a single thread. The input size is derived from the NBs 600.000 that is provided to Linpack.

$$totalMaintablesize = 2^{38} = 274877906944$$

RandomAccess uses mostly small sized non-blocking communication. RandomAccess accesses data from all the processes but has the advantage to exploit shared memory benefits from processes sharing the same node. The benchmark was run multiple times and the random pattern in fluctuating differences in performance between mpi and container implementation were seen in every run. the results captured in Fig. 27 and Fig. 28 are from a single run to emphasize this. This type of results are in line with the strong randomness that is inherent to this benchmark. Memory can be accessed all over the cluster and can be local, shared or on another node. Also the number of mpi invocations made to a specific node can strongly fluctuate. It is most notable that although the benchmark is single thread it does not react extremely strong to an increase in processes and threads (Fig. 27). The optimal configuration is 6 processes per node, but run-time difference is only slight compared to 4 processes per node. The restricted number of processes that perform optimal seem to indicate the io bound nature of this benchmark. The slight lower gup/s performance for the configurations of 2 and 4 mpi processes per node are in line with the higher runtime seen for these configurations. For the runtime performance as well as for the best gup/s performance a configuration of 6 processes per node is optimal.

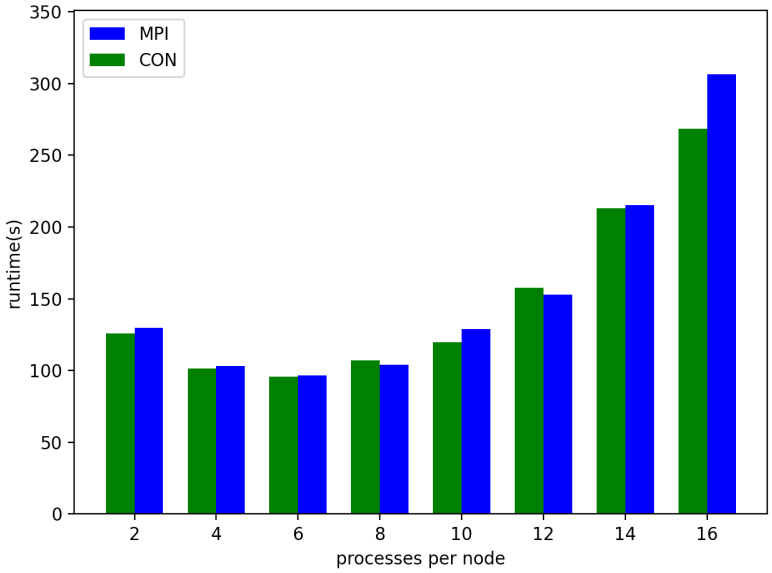

Figure 27: Single threaded: Input Size 274877906944 , Nodes 40

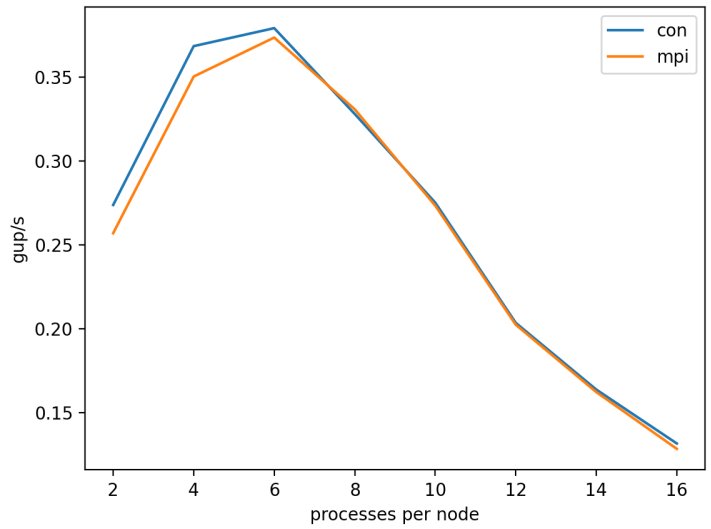

Figure 28: Single threaded: Input Size 274877906944 , Nodes 40

A repeated number of runs evens out the fluctuations seen in Fig. 27 and Fig. 28 and results in insignificant differences for all process configurations on both the gup/s metric as well as the runtime. Following the T-test results in table 9, there is no reason to assume containerization causes performance degradation for a random memory access application with a high number of MPI invocations with random memory access.

T-test	2 proc	4 proc	6 proc	8 proc
Con Runtime > Mpi Runtime	p=0.0118	p=0.9654	p=0.9891	p=0.0239
	10 proc	12 proc	14 proc	16 proc
Con Runtime > Mpi Runtime	p=0.0300	p=0.0162	p=0.9912	p=0.9784
	2 proc	4 proc	6 proc	8 proc
Con gups < Mpi gups	p=0.0218	p=0.0006	p=0.9900	p=0.0033
	10 proc	12 proc	14 proc	16 proc
Con gups < Mpi gups	p=0.0118	p=0.0115	p=0.9611	p=0.0221

Table 9: T-test RandomAccess not-overthreaded, p=0.05

6.5 Fast Fourier Transform

The FFT benchmark measures the number of floating point operations a system can handle. The input size N is derived from the input file configuration:

$$N = 34359738368$$

FFT uses MPI_AllReduce and MPI_Alltoall as a communication pattern to transfer large data messages. FFT performance of MPI and container implementations show near similar runtime results for the optimal configuration of 4 processes per core (Fig. 29). Terra flop/s is at its highest at 4 processes per node, which matches the optimal runtime configuration (Fig. 30). However Terra flop/s for a configuration of 6 processes are lower than for the configuration of 2 processes which does not hold for their runtime performance. For the terraflop/s a differentiaton is being made on computing and generation time of the data and therefore the timer functions of Fig. 30 and Fig. 29 differ. The results indicate that 8 processes per node results in a lower runtime for generating the data, but 2 processes results in lower runtime for the computing part of the FFT. Following the results of the T-test there is no reason to assume containerization causes performance degradation for a not overthreaded implementation of FFT (Tab. 10).

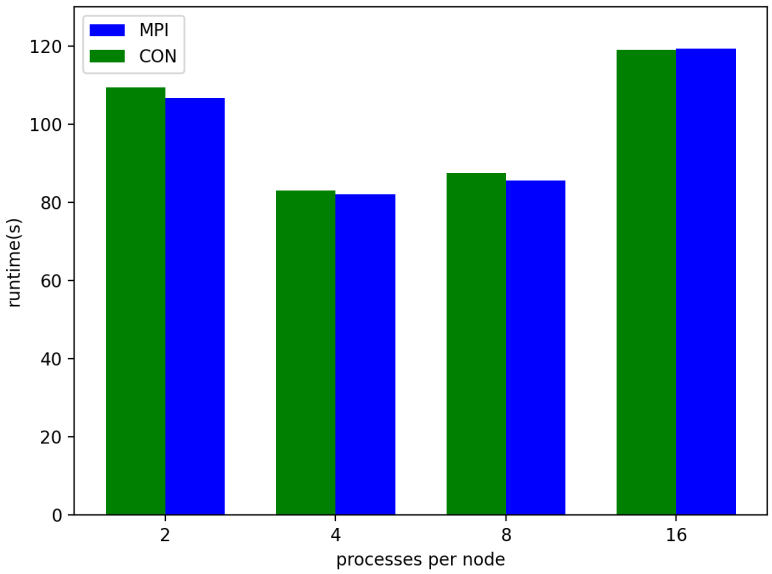

Figure 29: No overthreading: N=34359738368, Nodes 40

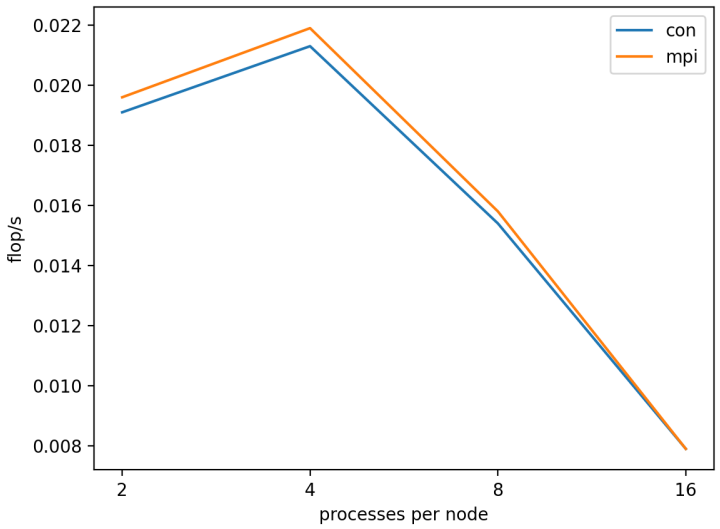

Figure 30: No overthreading: N=34359738368, Nodes 40

T-test	2 proc	6 proc	8 proc	16 proc
Con Runtime > Mpi Runtime	p=0.0110	p=0.0126	p=0.0271	p=0.9990
T-test	2 proc	6 proc	8 proc	16 proc
Con terraflops < Mpi terraflops	p=0.0143	p=0.0162	p=0.0011	p=0.9808

Table 10: T-test FFT not-overthreaded, p=0.05

The overthreaded implementation of FFT shows an optimal runtime result for 6 processes per node (Fig. 31). This runtime result is lower compared to the lowest not overthreaded runtime result (Fig. 29) and indicates that the overall performance benefits from overthreading. The peak performance of flop/s is measured at 4 processes (Fig. 32), which does not match the optimal runtime configuration. The results of the T-test indicate that containerization causes no overhead for an overthreaded implementation of FFT (Tab. 11).

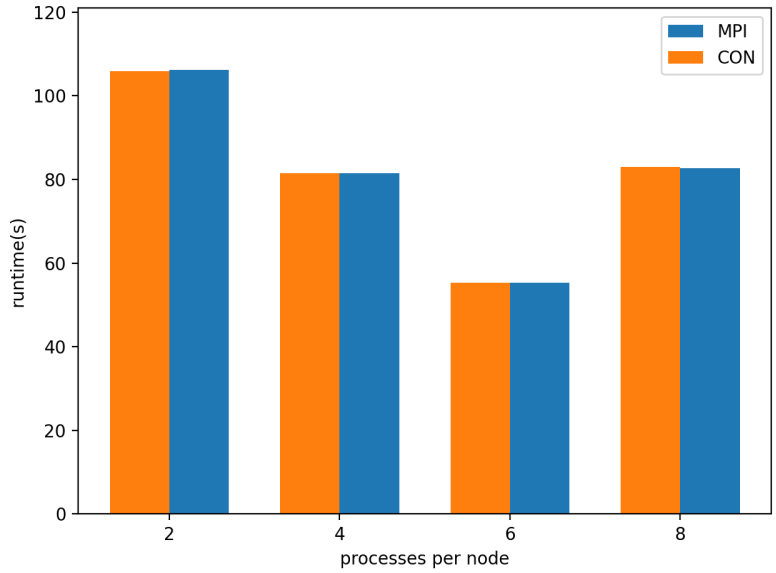

Figure 31: Overthreading: N=34359738368, Nodes 40

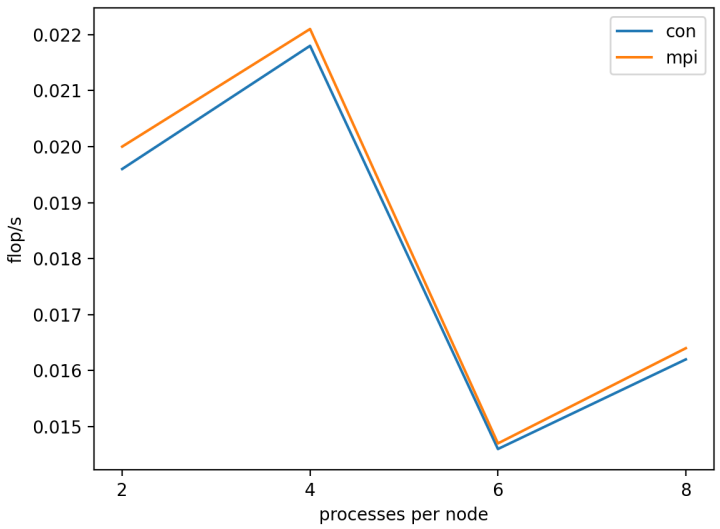

Figure 32: Overthreading: N=34359738368, Nodes 40

T-test	2 proc	4 proc	6 proc	8 proc
Con Runtime > Mpi Runtime	p=0.0033	p=0.0049	p=0.9760	p=0.0044
T-test	2 proc	4 proc	6 proc	8 proc
Con terraflops < Mpi terraflops	p=0.0218	p=0.0305	p=0.0002	p=0.0032

Table 11: T-test FFT overthreaded, p=0.05

6.6 Bandwith/Latency Benchmarks

The bandwith and Latency benchmark assesses the bandwith and latency of a system by examining a random ring communication pattern. The lowest runtime for the not overthreaded implementation is measured at 4 processes per node (Fig. 33). Most notable is that the communication latency is higher for this configuration (Fig. 34) while the bandwith is lower (Fig. 35). The inconsistency in this benchmark is that inter-process communication is also included. This effects both the bandwith and latency results which inversely follow the same slope. No notable or signficiant differences were found between container and bare metal implementation for the not-overthreaded implementation (Tab. 12).

The overthreaded implementation shows an optimal result of 6 processes per node. Over threading is preferred over not overthreading when it comes to runtime results (Fig. 33, Fig. 36). The bandwith and latency decrease and increase, as the number of processes increases, equal to the not overthreaded implementation. Results for containerized and bare metal form are close to equal which is shown by the T-test results 12. Following these results there is no reason to assume containerization causes overhead for the random ring bandwith latency application.

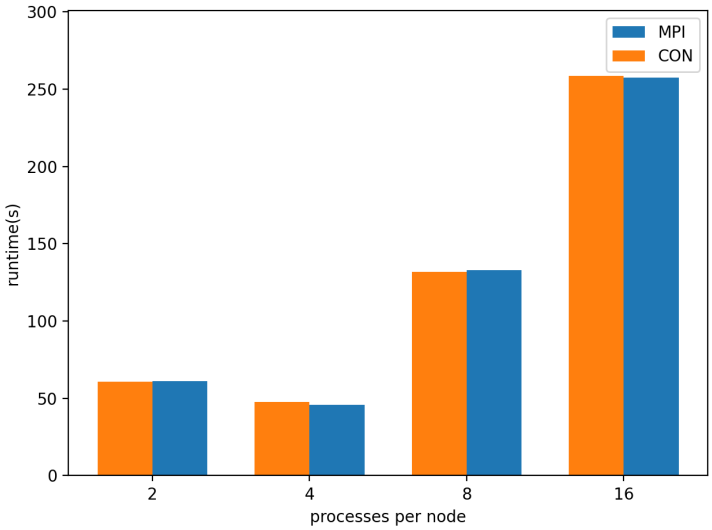

Figure 33: No overthreading: Runtime, NBs 600.000, Nodes 40

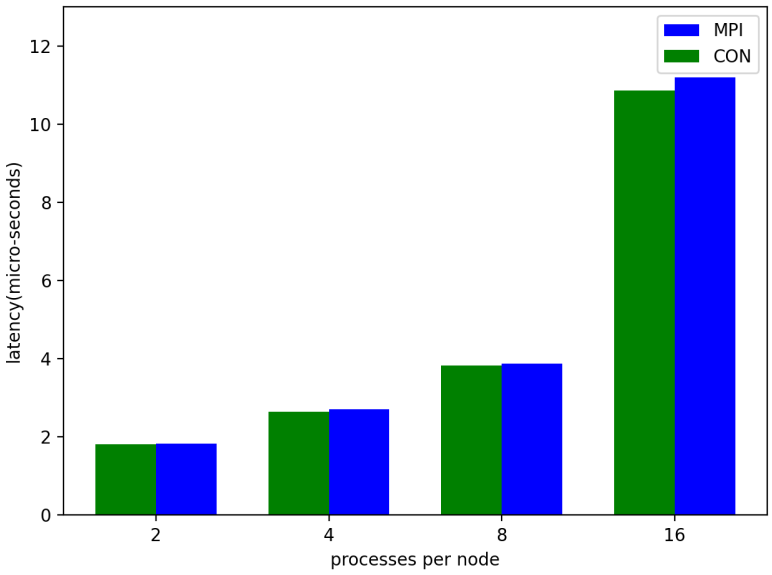

Figure 34: No overthreading: Communication Latency, NBs 600.000, Nodes 40

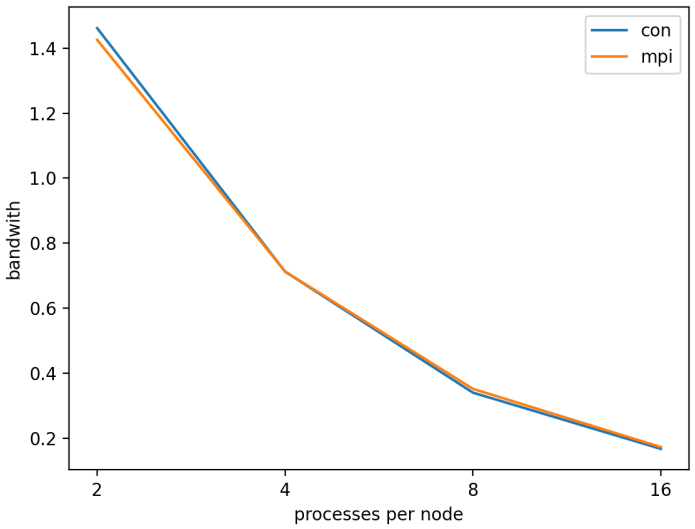

Figure 35: No overthreading: Bandwith, NBs 600.000, Nodes 40

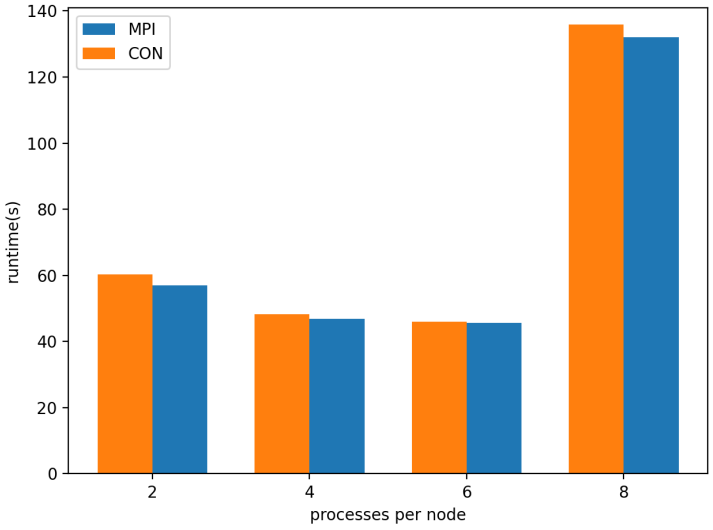

Figure 36: Overthreading: NBs 600.000, Nodes 40

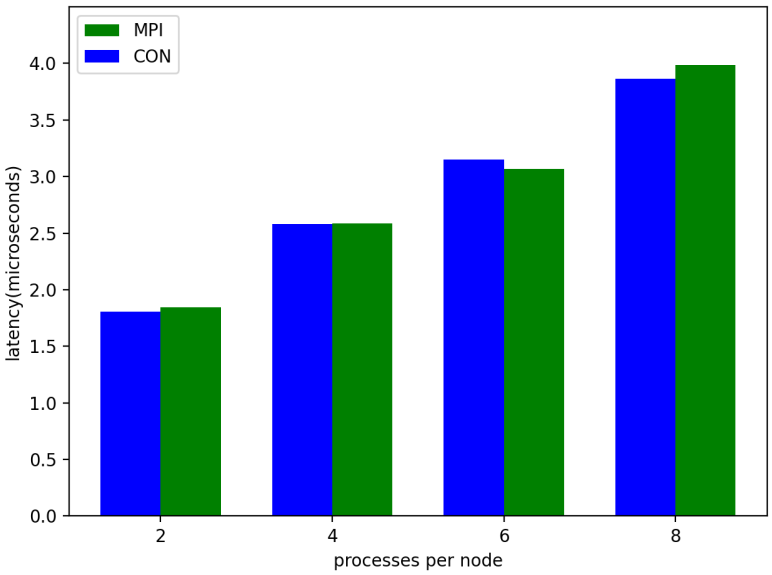

Figure 37: Overthreading: NBs 600.000, Nodes 40

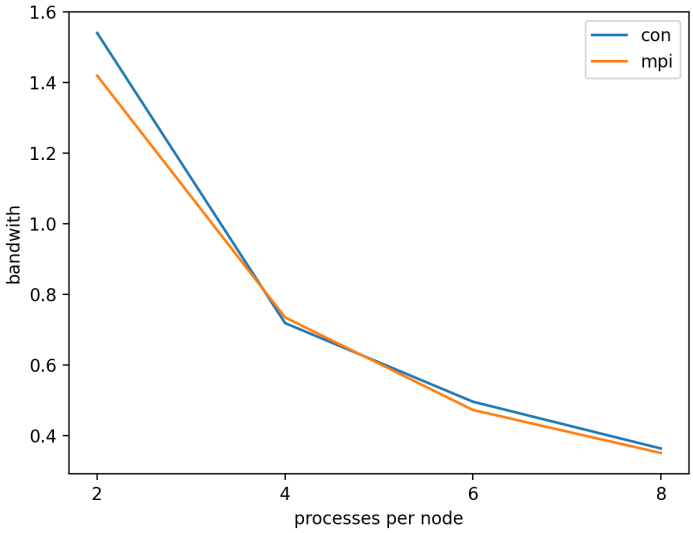

Figure 38: No overthreading: Bandwith, NBs 600.000, Nodes 40

T-test	2 proc	4 proc	8 proc	16 proc
Con Runtime > Mpi Runtime Latency	p=0.0290	p=0.0142	p=0.9929	p=0.0036
	2 proc	4 proc	8 proc	16 proc
Con Runtime > Mpi Runtime Bandwith	p=0.0019	p=0.0003	p=0.0029	p=0.9999
	2 proc	4 proc	6 proc	8 proc
Con latency < Mpi latency	p=0.0280	p=0.0021	p=0.9991	p=0.0042
	2 proc	4 proc	8 proc	16 proc
Con bandwith < Mpi bandwith	p=0.9980	p=0.0341	p=0.022	p=0.9984

Table 12: T-test Bandwith/Latency not-overthreaded, p=0.05

T-test	2 proc	4 proc	8 proc	16 proc
Con Runtime > Mpi Runtime Latency	p=0.0182	p=0.0302	p=0.0211	p=0.0099
	2 proc	4 proc	8 proc	16 proc
Con Runtime > Mpi Runtime Bandwith	p=0.0269	p=0.9861	p=0.0022	p=0.9900
	2 proc	4 proc	6 proc	8 proc
Con latency < Mpi latency	p=0.0091	p=0.0012	p=0.0393	p=0.0193
	2 proc	4 proc	8 proc	16 proc
Con bandwith < Mpi bandwith	p=0.9981	p=0.0325	p=0.0117	p=0.9666

Table 13: T-test Bandwith/Latency overthreaded, p=0.05

7 Singularity in Kubernetes

The results that where produced during the first experiment that showed low to zero overhead sparked the curiosity to test Singularity outside the high performance computing world. There were no signs of existent overhead when Singularity was tested with different HPC workloads, which encourages to test how well Singularity performs in a different setting with a different type of workload. In order to assess how Singularitys performance generalizes to other type of workloads, Singularity will be implemented in Kubernetes. A scheduler as Kubernetes is particulary well fit for dynamic workloads, which high performance workloads are inherently not. The distributed streaming framework Apache Kafka will be used as a workload with different characteristics than traditional high performance workloads. Apache Kafka is a created by Linkedin and the problem they originally set out to solve was low-latency ingestion of large amounts of event data from the LinkedIn website [21]. A Kafka pipeline is constructed and deployed in Singularity and Docker, with Kubernetes as a container scheduler. Since Docker is the industry standard and was integrated in Kubernetes it will serve as a benchmark for the performance of Singularity.

Sylabs had put some effort in 2019 into producing a container run-time interface [40] and a local Kubernetes setup named Sykube [39], which is the equivalent to Dockers Minikube [31]. Minikube is a local Kubernetes setup that uses two containers that serve as nodes, in which Kubernetes and Docker are installed. The containers will host the Kubelet and instantiate containers and pods within the container environment itself. Sykube uses the same architecture as Minikube, but with Singularity containers. Sylabs stated that they are looking for a new home for Sykube and while they recon the interest in using Singularity with Kubernetes, their time and effort will go into optimizing Singularity. Since Sylabs last commits in 2019 to their container runtime interface and Sykube, the container market has changed. The latter years the container market has gotten more attention, became more mature and most importantly, Docker and its shim were removed from the Kubernetes source code. Rewriting and restructuring the Singularity-cri and Sykube is a reasoned next step. The second part of this research aims to make Singularity more widely applicable by integrating the framework with Kubernetes and assess its performance outside high performance computing.

Besides the curiosity to assess how well Singularity performs with different workloads, Singularity its security benefits be shown to advantage even more in a Kubernetes setup. The security risk that Kubernetes carries is that infected workloads who can get control over a root owned container socket, ultimately could control the Kubelet and schedule unwanted new workloads. The increase in cybercrime has been alarming with a reported global rise of 50% between 2018 and 2020, costing the global economy just under 1 trilion USD in 2020 [41]. With the rapid increase of adoption of containers and container schedulers, the liability risk of any security issues these frameworks carry increases as well. A characteristic of the container frameworks that carries substantial risk are the image hubs that offer a off the shelf marketplace for applications. Pre-build filesystems of other users or companies can be infected and while layers with statements as pip and apt-get can be traced, files copied in by the builder can not. Singularity will significantly reduce security risks since unsecure workloads can not escalate any privilege or get control over the Kubelet. Users who did not have the option to deploy Kubernetes because of security risks will now have the option to deploy Kubernetes with Singularity.

Having a broader range of container frameworks to choose among for different type of workloads or domain restrictions, as in HPC environments, will only adhere to the quality of workflow. The movement away from the docker shim by Kubernetes was the first step in this process and integration of Singularity will be a reasoned next step. Additionally, integrating a container framework with Kubernetes is an enabler of the adoption of a container framework on a wider scale. Container frameworks that are well integrated with Kubernetes have the advantage that any line of business or research wishing to use Kubernetes are restricted to containers with a proper container runtime interface.

7.1 Sykube

Sykube is a local setup for Kubernetes with Singularity and sets up 2 Singularity containers with Kubernetes installed, one acting as master and one acting as a worker node. Requirement is only having Socat and Singularity >=3.2 installed. Building and running their image locally, as well as pulling it from the Singularity registry do not work. User community discussions on Singularity-cri and Sykube commonly mention inability to deploy any of the frameworks. First an overview of the Sykube definition file is being made and all its implications. Subsequently an overview of the Sykube executable is being made.

7.1.1 Sykube Image

The Sykube github repository contains an image definition file that can be build locally or can be pulled from the Sylabs library and a executable that will be copied in `/usr/local/bin` after running the image. The definition file has an Ubuntu:xenial (16.04) base image which uses a Linux kernel v4.4. The definition file creates directories and sets environment paths after which the Sykube executable is copied into the image. The proper libraries as Kubernetes, Singularity, Singularity-cri and other dependencies are installed and importantly the supervisor library is installed. Supervisor is a core component in enabling running Kubernetes and a container inside a container. In a host setup the Kubelet as well as the Singularity container runtime interface run as a Systemd service. Systemd is a process manager and handles the management of services like reaping, restarting, and shutting down. The issue with running Systemd inside a container is that Systemd does not run as chroot, meaning starting a Systemd service inside a Singularity container is not feasible. Supervisord does not require root access as Systemd and is able to run services within a container environment. Three Supervisord files are created; A Kubelet service, a Singularity-cri service and a Kube-proxy service to access a dashboard. Kube-proxy is a network proxy that runs on each node in your cluster, implementing part of the Kubernetes Service concept and maintaining network rules on nodes. These network rules allow network communication to your Pods from network sessions inside or outside of your cluster. Kube-proxy uses the operating system packet filtering layer if there is one and it's available, otherwise Kube-proxy forwards the traffic itself. The Kubelet service is executing a script where Kubelet configurations are being set use the Singularity socket for communication and the Kubelet is started. Other Kubelet configurations files are a ClusterRoleBinding file which assigns an adminrole

in the cluster and a Kubeadm file which specifies the Singularity socket for communication and the podsubnet. A CNI loopback bridge and a `/etc/hosts` file, which contains information how to translate container hostnames to IP-addresses, are written. During runtime of the container a RAM tmpfs is mounted and the mount point is set to `-make rshared` to ensure the mount can be replicated and containers can be instantiated inside the container as well. At last the Supervisord services are started and the Sykube executable will be copied into the `/usr/local/bin` directory on the host which ends the build of the image.

7.1.2 Sykube Executable

The Sykube executable takes in arguments from the command line and is used to instantiate the Sykube cluster on the host. The sykube init command starts with configuring the host with proper networking rules and other host configurations required for running Kubernetes. Iptables are configured which are used to set up, maintain, and inspect the tables of IP packet filter rules in the Linux kernel which is required for Kubernetes communication. A Container Network Interface (CNI) file is written which contains the specifications on how the Iptables should be configured to allow communication on the subnet.

A Singularity instance is started named `sykube-master` which is configured with a persistent overlay directory which allows to overlay a writable file system on an immutable read-only container for the illusion of read-write access. Some required directories are mount binded into the container as the cgroup directory which is used by Kubernetes to instantiate pods. Furthermore the instance is configured with network configurations and a `-keep-privs` flag that let the root user keep privileges in the container. The `/etc/hosts` file within the container is extended with the container name and ip address, hence when Kubernetes instantiates it can recognize the Singularity container in which it is encapsulated as a proper node. The Kubeadm init command is run in the Singularity container which then starts the Kubernetes master node with previous set configurations. After instantiation the Kubernetes master node is running and able to be configured by the `kubectl apply` command. The worker node is instantiated in the same manner as the master node and joined to the Kubernetes cluster by the join token produced by the master node.

Running the Sykube image with Singularity updated to newer versions varying from 3.5.0 till 3.10.0 had no positive impact on newer operating systems as Ubuntu 20.04 and Centos 9. Previous mentioned versions of Singularity do run properly without the intervening of the container runtime interface on these operating systems. Testing Kubernetes with Docker as container runtime causes no problems and runs properly. The Singularity container runtime interface is, as Sykube, deprecated and can not connect to the Kubelet when run on the host, without the Sykube setup. Incompatability between the host kernel and Sykube seems to be in the Container runtime interface, Singularity-cri.

7.2 Singularity-cri

The Container Runtime Interface of Singularity enables the communication between the Kubelet and the container runtimes. Container runtime interfaces are able to interpet incoming messaging from the Kubernetes API and translate messages to a format the installed container runtime can interpet. In the case of Sykube, the Singularity-cri Supervisord service was up and running as well as the Kubelet Supervisord service. Logs of the Kubelet do not always give full statements in logs or proper direction to underlying causes of errors, requiring tracing of the error with kernel logs. Both supervisord services, Sycri as well as the Kubelet were straced by adding a strace argument directly in the Supervisord files.

A blocking mount syscall showed the mount of a squashfs was invalid due to an unrecognized argument. Error terms from the Kernel can be non conclusive and an eval argument will not be sufficient to point out the cause of the issue. The incompatability is assessed further with dmesg, a Linux utility that displays kernel-related messages retrieved from the kernel ring buffer. The dmesg kernel logs showed an invalid error term was given to a squashfs mount and pointed to a wrongly configured mount syscall of an SIF image. Examining the system calls that were made by Singularity-cri, it became evident that these system calls were not compatible with the new Linux Mount api that evolved between Linux Kernel 4 and 5. To enable Singularity-cri and Sykube to run properly on newer Linux kernels and operating systems, all compatibility issues between the Singularity-cri source code and Linux Kernel version 5+ api have to be resolved. Examining the code base of the Singularity-cri, it became evident that runtime interface is not just an interface that translates requests and pass them to the container runtime executable. The container runtime interface also incorporates part of the implementation and execution phase of request that come in. Interestingly enough, the Singularity version on the host as well as the version installed in the Sykube container image have restricted impact on the runtime activity of Sykube or the Singularity-cri. The initial part of building and mounting sif files is actually being executed by the Singularity-cri source code itself and not by the installed Singularity executable.

The problem with the Singularity version that is incorporated in Singularity-cri, is that it refers to an older singularity version. This version includes a syscall that is still configured for the mount API from kernel version 3 and 4. Ubuntu version 18.04 and up, Centos 8 and up and most newer operating systems use Linux Kernel 5. Since the Singularity-cri is, as discussed, not just an interface but includes the implementation part, it therefore contains function calls to Singularity version 3.3.0. These function calls are provided with arguments by the Singularity-cri, but are entirely tuned to the Singularity 3.3.0 version. The deceiving part of this is that the Singularity version on the host as well as the Singularity version installed in the Sykube.def have no influence over this. Solving this issue requires, besides updating dependencies and versions, rewriting the source code and re-designing the behaviour of Singularity-cri to make it compatible with newer Linux kernel versions.

- The Singularity-cri `pkg/server/device/device.go` file will be rewritten to incorporate nvidia device plugins that work for newer linux kernels.
- The `pkg/network/network.go` file is extended with two new functions that add context to networking functions.
- `pkg/image/image.go` file a new function call is introduced that also incorporates a context and adds the option to verify every non-signature partition of an images.
- A `Systemd service` file is written and added to the `README.md` file to run the container runtime executable as a service.
- A `statement` for the Kubelet that directly incorporates the Singularity socket and the podsubnet is added to the executable.

As the rewritten and restructured Singularity-cri is now compatible with newer version of the Linux kernel, it will be included in Sykube. The Sykube executable in `/usr/local/bin` is extended with new modules and features. After addition of the following features Sykube did run without issues and proper on Linux Kernel 5 operating systems.

7.3 Kafka

Kafka is a distributed data store optimized for ingesting and processing streaming data in real-time. The main idea is that its messaging system is captured as a distributed commit log. Kafka its logs are a sequence of records with append-only entries (Fig. 39). The Kafka frameworks consist of 4 main components which all have their own specifc role and together construct a fault tolerant streaming framework.

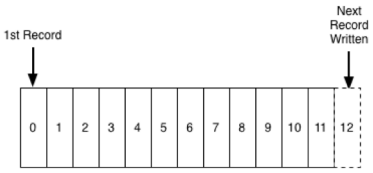

Figure 39: log use

- *Zookeeper*: Zookeeper is the manager that is primarily occupied with tracking the status of nodes in the Kafka cluster and maintaining a list of Kafka topics and messages. Zookeeper managers maintain an in-memory image of state, along with a transaction log and snapshots in a persistent state store.

- *Producer*: The producer creates new messages and writes them to the appropriate Kafka broker. Producers can be seen as the starting point of the Kafka streaming framework. Producers send the data in a standardized binary message format. The data is written directly to disk by the Brokers in the same format, without any modification.
- *Broker*: The Kafka broker handles all requests from all clients, both producers and consumers as well as metadata. It also manages replication of data. The replication factor can be set based on the risk of losing the data. Kafka breaks topic logs of producers up into partitions. Each partition is an ordered, immutable sequence of messages written to log.
- *Consumer*: Subsequently the logs are read from Disk by a consumer, an application that reads records from Kafka brokers. The logs are read in the same unmodified binary data format. The broker exposes the logs to consumers and consumers can build applications around it. The consumer is an event loop that calls the application code to process incoming message. Kafka also has a connect API that is written on top of the consumer API to read data into data sinks as Hadoop or a database (Fig. 40).

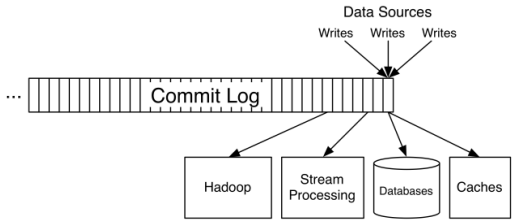

Figure 40: Log use

7.4 Kafka Application

For this experiment a Kafka streaming application was constructed that measures the time-difference between the point in time a message is produced by the producer and the point in time the message is received and read by the consumer. A single zookeeper manager is used to from the confluentinc/cp-zookeeper:3.3.0-1 Docker image. A single broker is used from the Docker confluentinc/cp-kafka:4.1.2-2 image. A producer container and a consumer container are created from a custom images that take as argument which container framework is used, which defines the server to bootstrap to. The producer creates a message and produces this to a topic that is bind to a bootstrap server. The message contains a time-stamp that is generated the moment the message is produced. Delay is calculated by generating a time-stamp the moment a message is read inside the consumer container and subtracting this timestamp from the producer timestamp which is tagged to the message.

7.5 Local Container Network

First, a local setup with no Kubernetes scheduling is deployed. A local cluster is set up in which the containers directly communicate to one another with no scheduler in between. For this experiment the resources of Google Cloud Platform were used. A Google cloud EC2 instance with 8 gb of memory and 8 cores was instantiated with Ubuntu 20.04 as operating system and 20gb persisted disk. The instance was furthermore configured with allowance of HTTP trafficking and with IP forwarding enabled.

This setup serves as a benchmark for the Kubernetes experiment. Singularity and Docker are both installed on the instance. The same Kafka and zookeeper images are used for the Docker and Singularity implementation. Singularity pulls the Docker image and stores it locally. Subsequently Singularity converts the Docker image to a Singularity image-file. The Kafka producer generates batches of messages. Every batch contains 100 messages which are send every second. The Kafka consumer reads messages and calculates the delay. After receiving 10.000 messages the average delay is written as output. The streaming application was run 20 times for both instances to ensure valid results with all runs stopping after 60.000 messages. The local setup is run by starting 4 containers: A Zookeeper container, a Kafka Broker container, a Consumer container and a Producer container. The Network namespace is by default shared with the host meaning Singularity communicates through localhost. The Docker implementation is configured by default with isolation of the network stack and uses the Kafka network for communication between containers. Containers were addressed by their name and port. The application is run for 60.000 messages after which it terminates. For every 10.000 messages the average delay is outputted. Meaning, the delay is specific per fraction of 10.000 messages. The longest delay is measured for the first 10.000 messages for both implementations (Fig. 41). As discussed, Docker by default has stronger isolation properties than Singularity and most importantly for streaming, virtualizes the network stack. Average delay per 10.000 messages is lower for Singularity compared to Docker after 20 repeated runs (Fig. 41).

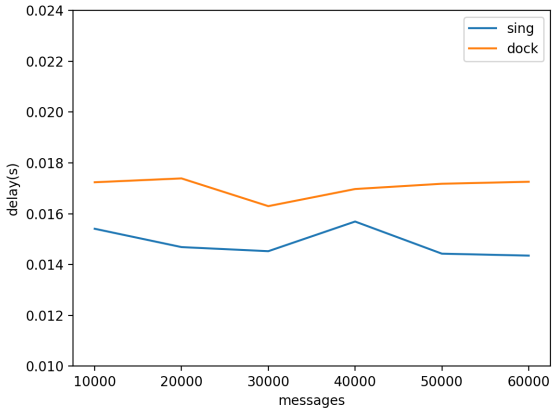

Figure 41: Local setup Kafka messaging delay

7.6 Kubernetes container network

For this experiment the resources of Google Cloud Platform were used. The same EC2 instance with 8 gb of memory, 8 cpu's and Ubuntu 20.04 with 20gb persisted disk is used as in the local setup. Sykube and Minikube are instantiated by their init commands and both start one master node and one worker. A namespace is created of which the purpose in Kubernetes can be compared to the role of a namespace in the Linux kernel; Namespaces provide isolation and access control, so that each microservice can control the degree to which other services interact with it. Namespaces also allow for different teams or users to work on their own isolated partition of a Kubernetes cluster. A pod is scheduled with a zookeeper manager, a Kafka broker, a producer and a consumer. Just as for the local setup, both implementations use the same images. The Minikube setup outperforms the Sykube setup in terms of delay in messaging for 20 repeated runs (Fig. 42).

The important difference between the local container network and the Kubernetes cluster, is that Pods are the entities that mainly provide isolation of resources by Linux namespaces. In local set up this is managed on a container level, while with Kubernetes the Pod takes over some of the isolation responsibilities from the containers. Relatively, Kubernetes adds more isolation to Singularity than to Docker. Docker itself provides network isolation which gives all Docker containers a unique ip address. If a Pod hosts Docker containers, network responsibilities are taken care off on a pod level and Docker containers do not have their own unique ip address. Singularity does not provide network isolation but when deployed in a Pod, a network isolation layer will be added by the Pod. Therefore, Singularity and Docker containers will have the same level of network isolation in the Kubernetes cluster while this will not hold for the local setup. The namespaces that are taken care off on a pod level are IPC namespace, Network namespace, UTS namespace and cgroups. Therefore these namespaces do not provide a more isolated environment for Docker. The container frameworks have equal isolation properties on behalf of these namespaces. The abstraction layer of Kubernetes therefore makes the Singularity implementation end up with approximately the same level of isolation as the Docker implementation. Docker remains a very well developed framework with a lightweight container runtime.

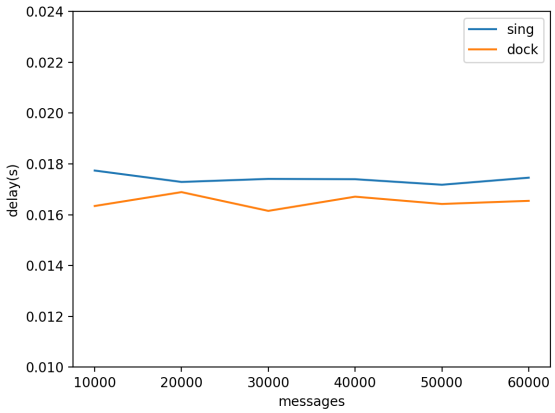

Figure 42: Kubernetes setup Kafka messaging delay

8 Conclusion

The first part of this research was attributed to assessing the effect of containerization on high performance computing. High performance computing consists of tightly coupled processes that require communication and integration with low latency. Different benchmark applications that work on different level of the memory hierarchy were tested in setups with different numbers of processes. The experiments were as well run in over-threaded form to test the cost of thread switching in a container environment. The hybrid setup, that scheduled a container for every MPI process scheduled, created an identical setup for both bare metal and container implementation. What became evident during this research, is that it is a fairly sensitive process to mimic bare metal performance inside a container. Installing same versions of libraries that are used on the host, inside the container environment showed to be insufficient. The environment in which libraries are compiled and run in can include libraries which are included and end up effecting performance. Only by tracing these dependencies an exact mimicked environment can be obtained that can serve as a proper base for comparison. Scheduling hpcc benchmarks in Singularity resulted in conclusive results: Little to no overhead was caused by container isolation, for every level of the memory hierarchy. Singularity can be described as a process with its own filesystem and a very low level of process isolation. The overthreaded implementation as well caused little to no overhead. For neither of the levels of the memory hierarchy that these benchmarks work on, overhead was detected when deployed with Singularity (Fig. 43).

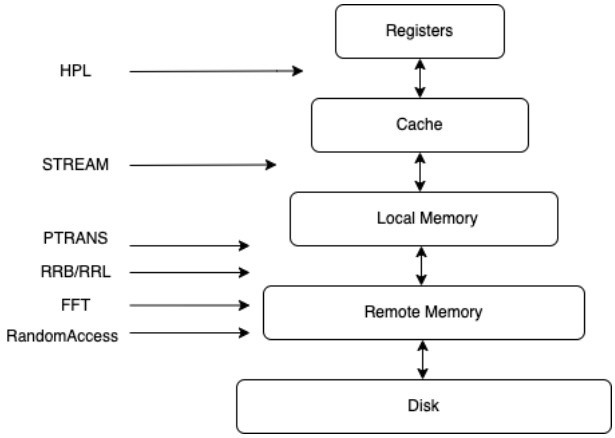

Figure 43: Benchmarks Impact on the Memory Hierarchy

The second part of this research was aimed at testing Singularity outside the high performance computing world. To have a more clear view on what the effect of isolation is on performance, both Docker and Singularity were tested in the same setup. Local, with no intervenience of Kubernetes, Singularity was at advantage. Singularity's slogan 'Integration over isolation' underlines the aim to isolate at a bare minimum. Docker is a very well maintained framework that is more aimed at supporting self sufficient, loosely coupled applications. The results of the Docker and Singularity setups show that the frameworks differences in characteristics indeed result in different performance.

Dockers more isolated environment seems to have an effect on performance in a local streaming set up. Singularity performed slightly better in terms of delay between message produced and message consumed. The second setup was aimed at examining perfomance of Singularity in a local Kubernetes cluster and comparing its results to Docker. To enable Singularity to work in an identical way as Dockers Minikube, the container runtime interface was restructured and new modules were implemented. Kubernetes infrastructure in a way has a similar goal as container frameworks but more on a scheduler level: Both create separate environments and provide communication points. The Kubernetes layers seem to have a more disadvantageous effect on Singularity's performance, because Kubernetes has a stronger isolating effect on Singularity compared to Docker. Following these results it can be concluded that Docker in Minikube setup outperforms Singularity in Sykube setup. In a local setup, Singularity performs better than Docker, with Singularity lacking IPC, UTS and Network isolation. After extensive assessment of the results of the prior mentioned tests, the three subquestions that guided this research will be answered.

- Research Question 1: *Are there levels of the memory hierarchy for which containerization causes performance degradation?* The overhead incurred by containerization for high performance workloads and Singularity is non significant with a p value of 0.05, for every level of the memory hierarchy. No significant disadvantageous effects were found for either of the benchmark applications when deployed in Singularity. The results of the Kubernetes setup and the local container setup however indicate that the UTS, IPC and Network namespaces cause an effect on performance for streaming application Kafka. When Singularity is tested in Kubernetes, which has a stronger isolating effect on Singularity containers compared to Docker, the framework no longer out performs Docker. These results indicate that isolating certain namespaces can introduce delay in messaging. Conclusively, no significant overhead was found due to isolation when using Singularity with the hpcc benchmarks, for neither levels of the memory hierarchy. Subjective to its degree, isolation does have an effect on performance and therefore the statement that containerization in general will not cause overhead will not hold.

- Research Question 2: *How much overhead is related to the architecture of container deployment or oversubscribing threads?* The hpcc benchmarks were run with various configurations. Every application showed different optimal configurations, both in terms of runtime and benchmark results. The optimal configuration per benchmark for the hpcc metric and runtime were not always conclusive. The optimal architecture for the container implementation was similar to the architecture of the mpi implementation. This underlines the thought that the optimal configuration is dependent on the application characteristics and not on using a container environment. Conclusively, there is an optimal architecture to deploy containers, but this is strongly subjective to the type of application. No specific overhead is found related to container architecture. Overthreaded container implementations do not show significant different results compared to the mpi implementations. Conclusively, the more isolated environment of a container does not causes performance degradation from overthreading .

- Research Question 3: *How does the High Performance Computing framework Singularity performs outside the high performance world?* Singularity was tested outside the High Performance Computing world. A different type of workload was used; a Kafka streaming application. Both a local setup as well as an implementation in Kubernetes, a dynamic container scheduler, were

implemented. The delay in messaging was compared to the same implementation with Docker. Singularity with a limited amount of isolation compared to Docker, performed better in local setup. In Kubernetes setup, the virtualization layers of the Kubernetes framework provided Singularity and Docker with a more equal level of isolation. In Kubernetes setup Docker outperformed Singularity in terms of delay in messaging. Although Docker showed to be slightly more performative with regards to supporting a Kubernetes Kafka cluster, Singularity with its minimal level of isolation seems to be performative outside of the high performance world as well. Singularity has the strong point that it lacks security issues, which makes it a fair choice in Kubernetes clusters.

While there is a broader range of container frameworks to choose from, a well documented, dominant framework as Docker has the benefit that it is already in use by many companies. Familiarity with a framework and a user friendly documentation is a force that should not be underestimated. Now containers are being used at a larger scale, advanced container schedulers as Kubernetes are becoming more of a necessity than an option. Proper container runtime integration with Kubernetes is becoming an enabler of capturing a market share, when Kubernetes is becoming an industry standard that to boot shows an inclination towards rootless implementations. A highly developed and mature container market where different types of container frameworks can be depict from for every use case, without excludement because of poorly developed runtime interfaces, will only adhere to quality of workflows.

Although the Kafka application showed slightly less delay in Minikube, Sykube and the Singularity-cri are interesting frameworks for the future with regards to the limited security risks. With the rise of cybercrime and Kubernetes being a framework that has the ability to control a cluster, security risks should not be taken lightly. One of the main advantages of Singularity is for it to run as non-root, which remains a relevant feature also for Kubernetes clusters. Running Kubernetes node components as a non-root user is a feature that is included in newer Kubernetes versions which implicates a market demand [32]. Singularity is a good candidate for Kubernetes clusters which vows for a well maintained Singularity-cri and Sykube. The fact that Singularity shows close to non existent overhead is promising for container usage in high performance computing.

References

[1] Luszczek, Piotr Dongarra, Jack Koester, David Rabenseifner, Rolf Lucas, Bob Kepner, Jeremy McCalpin, John Bailey, David Takahashi, Daisuke. (2004). Introduction to the HPC Challenge Benchmark Suite.

[2] Horizontal and Vertical Scaling of Container-Based Applications Using Reinforcement Learning July 2019 DOI:10.1109/CLOUD.2019.00061 Conference: 2019 IEEE 12th International Conference on Cloud Computing

[3] Perri, Damiano and Simonetti, Marco and Tasso, Sergio and Ragni, Federico and Gervasi, Osvaldo, 2021, Implementing a scalable and elastic computing environment based on Cloud Containers

[4] Bovet, Daniel and Cesati, Marco, Understanding The Linux Kernel, 2005, Oreilly Associates Inc

[5] O. I. Alqaisi, M. S. Haq and A. S. Tosun, "Security of Containerized Computer Vision Applications," 2022 2nd International Conference on Computing and Information Technology (ICCIT), 2022, pp. 115-120, doi: 10.1109/ICCIT52419.2022.9711600.

[6] S. Kehrer, F. Riebandt and W. Blochinger, "Container-Based Module Isolation for Cloud Services," 2019 IEEE International Conference on Service-Oriented System Engineering (SOSE), 2019, pp. 177-17709, doi: 10.1109/SOSE.2019.00032.

[7] Boettiger, C. (2015). An introduction to Docker for reproducible research. ACM SIGOPS Operating Systems Review, 49(1), 71-79.

[8] Kurtzer GM, Sochat V, Bauer MW. Singularity: Scientific containers for mobility of compute. PLoS One. 2017 May doi: 10.1371/journal.pone.0177459.

[9] Yoo, A.B., Jette, M.A., Grondona, M. (2003). SLURM: Simple Linux Utility for Resource Management. In: Feitelson, D., Rudolph, L., Schwiegelshohn, U. (eds) Job Scheduling Strategies for Parallel Processing. JSSPP 2003. Lecture Notes in Computer Science, vol 2862. Springer, Berlin, Heidelberg. https://doi.org/10.1007/10968987_3

[10] J. Dongarra, The LINPACK benchmark: An explanation in Supercomputing, vol. 297, Springer, Berlin, Heidelberg, 1988, pp. 456-474

[11] Performance Evaluation of Deep Learning Tools in Docker Containers August 2017 DOI:10.1109/BIGCOM.2017.32 Conference: 2017 3rd International Conference on Big Data Computing and Communications (BIGCOM) Project: Performance Evaluation of Deep Learning Tools in Docker Containers

[12] “Kubernetes,” Kubernetes. Available: https://kubernetes.io/. [Accessed: July 2022]

[13] Price, Daniel Tucker, Andrew. (2004). Solaris Zones: Operating System Support for Consolidating Commercial Workloads.. 241-254.

[14] Crosby, M. (2017). What is containerd? https: https://blog.mobyproject.org/

[15] S., Senthil. (2017). Practical LXC and LXD: Linux Containers for Virtualization and Orchestration. 10.1007/978-1-4842-3024-4.

[16] Jaeyoung Choi, Jack J. Dongarra, David W. Walker, Parallel matrix transpose algorithms on distributed memory concurrent computers, Parallel Computing, Volume 21, Issue 9, 1995, Pages 1387-1405, ISSN 0167-8191, https://doi.org/10.1016/0167-8191(95)00016-H.

[17] D. T. Popovici, T. M. Low and F. Franchetti, "Large Bandwidth-Efficient FFTs on Multicore and Multi-socket Systems," 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2018, pp. 379-388, doi: 10.1109/IPDPS.2018.00048.

[18] Jeremy Kepner. 2009. Parallel MATLAB for Multicore and Multinode Computers (1st. ed.). Society for Industrial and Applied Mathematics, USA.

[19] Sylabs userguide Singularity, https://docs.sylabs.io/guides/3.7/user-guide/mpi.html

[20] Calico CNI Kubernetes bridge https://github.com/projectcalico/calico

[21] Tanvir Ahmed, Kafka's origin story at LinkedIn, https://www.linkedin.com/pulse/kafkas-origin-story-linkedin-tanvir-ahmed/

[22] M. T. Chung, N. Quang-Hung, M. Nguyen and N. Thoai, "Using Docker in high performance computing applications," 2016 IEEE Sixth International Conference on Communications and Electronics (ICCE), 2016, pp. 52-57, doi: 10.1109/CCE.2016.7562612.

[23] Saha, P., Beltre, A., Uminski, P., Govindaraju, M.: Evaluation of Docker Containers for Scientific Workloads in the Cloud. In: Proceedings of Practice and Experience on Advanced Research Computing (PEARC18). ACM, New York (2018). https://doi.org/ 10.1145/3219104.3229280

[24] Zhang, J., Lu, X., Panda, D.K.: Performance characterization of hypervisor-and container-based virtualization for HPC on SRIOV enabled infiniband clusters. In: Proceedings of 30th International on Parallel and Distributed Processing Symposium pp. 1777–1784. IEEE (2016). https://doi.org/10.1109/ IPDPSW.2016.178

[25] 9. Zhang, J., Lu, X., Panda, D.K.: Is singularity-based container technology ready for running MPI applications on HPC clouds? In: Proceedings of 10th International Conference on Utility and Cloud Computing (UCC17), pp. 151–160. ACM, New York (2017). https://doi.org/10.1145/3147213.3147231

[26] Saha, P., Beltre, A., Govindaraju, M.: Scylla: a mesos framework for container based MPI jobs. CoRR (2019). arXiv:1905.08386

[27] Beltre, A.M., Saha, P., Govindaraju, M., Younge, A., Grant, R.E.: Enabling HPC workloads on cloud infrastructure using Kubernetes container orchestration mechanisms. In: Proceedings of CANOPIE-HPC 2019: 1st International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC, pp. 11–20 (2019). https://doi.org/10.1109/CANOPIEHPC49598.2019.00007

[28] Analysis of linux os security tools for packet filtering and processing Translation of Title Linux OS paketų filtravimo ir apdorojimo saugumo priemonų analizė Authors Melkov, Dmitrij ; Paulikas, Šarūnas DOI 10.3846/mla.2021.15180

[29] Performance characterization of containerization for HPC workloads on InfiniBand clusters: an empirical study Peini Liu, Jordi Guitart

[30] Jha DN, Garg S, Jayaraman PP, Buyya R, Li Z, Morgan G, Ranjan R (2019) A study on the evaluation of HPC microservices in containerized environment. Concurr Comput. https://doi.org/10.1002/cpe.5323

[31] Gutierrez, Felipe. (2021). Minikube. DOI: 10.1007/978-1-4842-7460-6_12.

[32] Kubernetes guide - Running Kubernetes Node Components as a Non-root User https://kubernetes.io/docs/tasks/administer-cluster/kubelet-in-userns/

[33] Liu, P., Guitart, J. Performance comparison of multi-container deployment schemes for HPC workloads: an empirical study. https://doi.org/10.1007/s11227-020-03518-1

[34] M. T. Chung, N. Quang-Hung, M. -T. Nguyen and N. Thoai, "Using Docker in high performance computing applications," 2016 IEEE Sixth International Conference on Communications and Electronics (ICCE), 2016, pp. 52-57, doi: 10.1109/CCE.2016.7562612.

[35] Martin, J.P., Kandasamy, A. Chandrasekaran, K. Exploring the support for high performance applications in the container runtime environment. Hum. Cent. Comput. Inf. Sci. 8, 1 (2018). https://doi.org/10.1186/s13673-017-0124-3

[36] MacOS Containers Initiative https://macoscontainers.org/

[37] Kubernetes blog https://kubernetes.io/blog/2020/12/02/dockershim-faq/

[38] Zhenyun Zhuang, Cuong Tran, J. Weng, H. Ramachandra and B. Sridharan, "Taming memory related performance pitfalls in linux Cgroups," 2017 International Conference on Computing, Networking and Communications (ICNC), 2017, pp. 531-535, doi: 10.1109/ICCNC.2017.7876184.

[39] Singularity Sykube implementation - Sylabs https://github.com/sylabs/sykube.

[40] Singularity container runtime Interface implementation - Sylabs https://github.com/sylabs/singularity-cri.

[41] Cremer, F., Sheehan, B., Fortmann, M. et al. Cyber risk and cybersecurity: a systematic review of data availability. Geneva Pap Risk Insur Issues Pract 47, 698–736 (2022). https://doi.org/10.1057/s41288-022-00266-6