universität wien

MASTERARBEIT / MASTER'S THESIS

Titel der Masterarbeit / Title of the Master's Thesis

Containerized Data-Intensive Applications on Scientific Clusters

verfasst von / submitted by

Manfred Cerny-Käfer, BSc

angestrebter akademischer Grad / in partial fulfilment of the requirements for the degree of

Diplom-Ingenieur (Dipl.-Ing.)

Wien 2017 / Vienna 2017

Studienkennzahl lt. Studienblatt / degree programme code as it appears on the student record sheet:	A 066 940
Studienrichtung lt. Studienblatt / degree programme as it appears on the student record sheet:	Masterstudium Scientific Computing
Betreut von / Supervisor:	Univ.-Prof. Dr. Siegfried Benkner
Mitbetreut von / Co-Supervisor:	Dipl.-Ing. Yuriy Kaniovskyi, Bakk. techn.

Abstract

Container virtualization provides effective isolation for software installations and tasks with very low performance overhead compared to virtual machines.

Traditional HPC-oriented environments used by different scientific groups often need to support several frameworks and software stacks which creates considerable maintenance effort. The use of container virtualization enables fast deployment of new software stacks while avoiding any unwanted side effects affecting other software containers or natively installed software.

Many diverse frameworks have emerged for specific purposes like running scientific computations using MPI or processing of huge amounts of data with frameworks like Apache Spark and Apache Hadoop. For optimal resource utilization it is often desirable to run multiple frameworks in the same cluster, sometimes even simultaneously. The use of appropriate cluster and resource managers enables sharing of the cluster which allows sharing access to distributed datasets avoiding the need to replicate large datasets across different clusters. Hence, dynamic partitioning of the cluster can help to achieve high utilization and efficient data sharing. Combining resource managers with container virtualization potentially enables effective resource utilization of a compute cluster environment while reducing the maintenance effort.

In this work, native and containerized installations of Apache Spark in combination with diverse resource and cluster managers are set up and compared with regard to individual advantages and drawbacks, setup effort and difficulties and the usability from the end user perspective. Using different benchmarks, the runtime performance of selected setup combinations is compared to each other. Based on the experience, we describe which setups delivered the best runtime performance indicating eligibility for broader adoption in HPC and conclude with a discussion of encountered problems and lessons learned from the experiments.

Zusammenfassung

Container-Virtualisierung bietet effektive Isolation für Software-Installationen und die Ausführungsumgebung zur Laufzeit mit geringem Performance-Overhead im Vergleich zu virtuellen Maschinen.

Traditionelle Umgebungen im High-Performance-Computing, die von verschiedenen wissenschaftlichen Gruppen genutzt werden, benötigen oft Unterstützung für mehrere Frameworks und Software-Stacks, was erheblichen Wartungsaufwand verursacht. Die Verwendung von Container-Virtualisierung ermöglicht den schnellen Einsatz neuer Software-Stacks, während eine unerwünschte Beeinflussung anderer Installationen oder Software-Container vermieden wird.

Für bestimmte Zwecke, wie die Durchführung wissenschaftlicher Berechnungen unter Verwendung von MPI oder die Verarbeitung riesiger Datenmengen mit Frameworks wie Apache Spark oder Apache Hadoop, sind viele verschiedene Frameworks mit spezifischen Anwendungsbereichen entstanden. Um vorhandene Ressourcen optimal auszunützen, ist es oft erwünscht, mehrere Frameworks, manchmal auch simultan, im selben Cluster zu betreiben. Die Verwendung geeigneter Cluster- und Ressourcenmanager ermöglicht die gemeinsame Nutzung eines Clusters durch mehrere Applikation und den Zugriff auf verteilt gespeicherte Daten, ohne große Datenmengen aufwendig zwischen verschiedenen Clustern kopieren zu müssen. Die dadurch ermöglichte dynamische Partitionierung eines Clusters kann helfen, hohe Auslastung der Ressourcen und effizienten Zugriff auf verteilte Daten zu erreichen. Die Kombination von Ressourcenmanagern mit Container-Virtualisierung ermöglicht potentiell die effektive Auslastung der Ressourcen eines Clusters bei gleichzeitiger Reduktion des Wartungsaufwands.

In dieser Arbeit werden native und Container-basierte Installationen von Apache Spark in Kombination mit verschiedenen Cluster- und Ressourcenmanagern bezüglich ihrer individuellen Vor- und Nachteile, dem Installations- und Konfigurationsaufwand und etwaiger Schwierigkeiten sowie der Benutzerfreundlichkeit aus der Sicht des Anwenders verglichen. Mittels verschiedener Benchmarks wird das Laufzeitverhalten ausgewählter Kombinationen ermittelt. Basierend auf den Erfahrungen aus den Experimenten beschreiben wir die Varianten mit dem besten Laufzeitverhalten, die für breiteren Einsatz im High-Performance-Computing geeignet scheinen und schließen mit einer Diskussion der aufgetretenen Probleme und möglicher Lösungsansätze.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

The ongoing growth of scale combined with the need of energy efficiency in high performance computing (HPC) requires - besides energy-efficient hardware - optimal resource utilization.

Excessive energy consumption has become a first-class constraint in designing and deploying the next generation of supercomputers. The average power consumption of the top ten supercomputers in June 2017 was 7.9 MW [16], which is not only a major cost factor, but also corresponds to a significant amount of CO_2 emissions (depending on the means of electric power generation). Hence, energy efficiency has become a top concern for HPC systems. [52]

Many diverse frameworks have emerged for specific purposes like running scientific computations using message passing interface (MPI) or processing of huge amounts of data, e.g., with Apache Spark and Apache Hadoop. For optimal resource utilization it is often desirable to run multiple frameworks in the same cluster, using the best one for each application. Sharing the cluster also allows sharing access to distributed datasets avoiding the need to replicate large datasets across different clusters. Static partitioning of the cluster usually does neither achieve high utilization nor efficient data sharing. A resource manager that allows dynamic resource allocation for compatible frameworks potentially enables high cluster utilization and shared access to distributed data. [18]

Container virtualization provides effective isolation for software installations and tasks as well as enforcement of resource usage limitations to a very high degree with lower performance overhead than virtual machines. However, container virtualization always uses the host kernel to run the containerized processes.

Traditional HPC-oriented environments used by different scientific groups usually need to support several frameworks and software stacks which creates considerable maintenance effort and makes it hard and lengthy to install new software stacks or new versions of already installed software. It may also be impossible to install new software due to lacking availability for the currently used OS version or incompatibilities with existing and still needed versions of other software. The use of container virtualization enables fast deployment of new software while avoiding any unwanted side effects affecting other software containers or natively installed software. [33]

Resource managers provide abstractions of compute, memory and storage resources for single machines and entire clusters and enable the effective utilization of these resources for single applications as well as multiple applications in shared usage scenarios.

Combining resource managers with container virtualization potentially increases the effective resource utilization of a compute cluster environment.

1.1 Research Objective

The primary research objective is the exploration of the provisioning of containerized infrastructure for data-intensive applications (Apache Spark and the Hadoop Distributed File System are used for the purpose of the evaluation) utilizing Docker and different means of resource management like Apache Mesos, Apache YARN and Docker Swarm. Selected combinations of containerized and native software installations are compared in terms of performance and implications like setup and configuration effort for the resource provider and also for the consumers.

1.2 Chapter Overview

Chapter 2 presents related work including a short summary of the respective content.

Chapter 3 gives an overview of virtualization techniques and describes the motivation for virtualization and its benefits. The common virtualization techniques are covered and virtual machines are compared to lightweight virtualization with containers.

Chapter 4 presents a short overview of cluster and resource management with particular focus on the cluster management software and frameworks used for the experiments.

Chapter 5 briefly depicts the progress in various research fields leading to growing demand for large-scale computing and data-processing systems and gives an overview of Apache Spark, the large-scale data processing framework, that was used for the experiments and benchmarks.

Chapter 6 gives a detailed description of the setup of all required frameworks and tools.

Chapter 7 describes the experiments performed with Apache Spark using different setup and configuration options. The motivation, advantages and drawbacks for each setup are discussed elaborately and the experience including any obstacles encountered is illustrated.

Chapter 8 discusses the results of the runtime measurements of the conducted experiments in detail.

Chapter 9 presents two aspects relevant for data-intensive HPC applications as possible areas of future work.

Chapter 10 draws conclusions from the experiments and measurement results and mentions the lessons learned.

The appendices contain a summary of the used technology and tools and the Docker files to build the Docker images that were used for the experiments.

Chapter 2

Related Work

2.1 HPC vs. Big Data Infrastructure

An extension of the Pilot-Abstraction, that the authors have created to enable data-aware scheduling on the application-level, is described in [38]. Based on a comparison of the characteristics of HPC infrastructures and data storage for HPC with infrastucture built for Apache Hadoop and Apache Spark in conjunction with scheduling considering data-locality, the authors developed an extension of the Pilot-Abstraction to enable data-aware scheduling on the application-level. Their work is based on Pilot-Data [39], an extension of the Pilot-Job abstraction for supporting the management of data in conjunction with compute tasks. Pilot-Data separates logical data units from physical storage, providing the basis for efficient compute/data placement and scheduling [39]. In this paper, the authors show the use of Pilot-Abstraction as common, interoperable framework for HPC and the Apache big data stack to support a diverse set of workloads. [38]

An attempt to bridge the gap between HPC jobs running on HPC hardware and Hadoop jobs executing on commodity servers by allowing HPC and Hadoop jobs to co-exist on a single hardware facility is illustrated in [51]. There is substantial existing investment in HPC facilities present in many academic, research, and commercial institutions. With the emergence and increasing popularity of Apache Hadoop, a reliable, scalable and open-source framework for large-scale data processing running on cheaper commodity hardware clusters, the need to build and maintain a second parallel facility often arised. The authors developed a Hadoop YARN-based runtime system for MPJ Express software that allows executing parallel MPI-like Java applications on Hadoop clusters. [51]

2.2 HPC Batch Schedulers vs. Datacenter Resoure Managers

The job scheduler as operating system of the modern data center and the different characteristics (e.g., the duration of tasks, batch or immediate execution) of HPC and big data workloads are depicted in [45]. The authors analyze which features of schedulers are important for high performance data analysis jobs and compare the support for these features for a representative set of HPC and big data schedulers (Slurm, Grid Engine, Hadoop YARN, and Mesos). Their benchmark results show similar results for Slurm, Grid Engine, and Mesos with worse performance for YARN. The overall utilization could be further increased using a multilevel scheduling technique with the LLMapReduce tool. [45]

2.3 Virtualization and Containers in HPC

A novel approach for a compute and data-intensive pipeline framework is introduced in [34]. To optimize pipeline applications with respect to data and computational requirements of individual stages, the design of the framework supports the integration of a multitude of stage variants, encapsulating each pipeline stage and their runtime environment into containerized execution units. The containers allow, beside efficient resource utilization, supporting concurrent utilization of different implementation variants based on a multitude of programming languages and paradigms. [34]

The experiences introducing Docker to install new - and at that time incompatible - software on a scientific cluster are discussed in [33]. The authors argue, that Docker is ready for many workloads today, while some areas (e.g., RDMA interfaces) still need improvement, which are already targeted by community initiatives. [33]

Shifter, a virtualization environment developed by the National Energy Research Scientific Computing Center (NERSC) in Berkeley, that can transform user defined Docker and other virtualization images into its own environment for execution on

NERSC's Cray supercomputers is introduced in [32]. By applying additional checks on the image contents, the organization's high security standards are still met while the container environments run as virtualized jobs on the supercomputer. [32]

An open-source initiative to bring containers and reproducibility to scientific computing is presented in [36]. Its primary use case is to provide a secure means to capture and distribute software and compute environments. In contrast to Docker, Singularity does not require a daemon with root privilege to start containers. [36]

2.4 Benchmarking and Optimizing Data-Intensive Applications

A system that captures and optimizes large-scale machine learning applications for high-throughput training with a high-level API is introduced in [49]. KeystoneML is designed to run with large, distributed data sets on commodity clusters and constructs a distributed acyclic graph of tasks that are executed on an Apache Spark cluster. [49]

An approach to create job parallelism setting recommendations for Apache Spark jobs based on machine learning techniques is illustrated in [17]. The authors argue, that platforms like Apache Spark are complex and difficult to manage and there is a lack of tools to better understand and optimize such platforms. In that paper, a machine learning based method that uses statistical correlations to characterize and predict the effect of different parallelism settings on performance is proposed. These predictions are used to recommend optimal configuration parameters for task parallelization in big data workloads to users before launching their workloads in the cluster. In their evaluation, up to 51% performance gain could be achieved using the recommended parallelism settings. The model is also interpretable and can give users insight how different parameters affect the performance on their cluster. [17]

2.5 Scientific Cluster Hardware Power Measurement

A heterogeneous, high performance computing infrastructure that provides detailed power and energy-consumption data of its hardware components is presented in [52]. That system is equipped with Intel Xeon CPUs, Intel Many Integrated Cores (Xeon Phi), Nvidia GPUs, power-aware memory systems and hybrid storage with Hard Disk Drives (HDDs) and Solid State Disks (SSDs). Additional hardware-based power measurement adaptors and microcontrollers with custom firmware for data sampling were added to determine the power consumption of components without integrated power measurement capabilities. Using software to collect and aggregate the detailed power consumption data in conjunction with controlled job scheduling and exclusive cluster node usage, the system provides valuable support to research and development in energy-aware high performance computing and big data analytics. [52]

2.6 Frameworks

Mesos, a thin management layer that allows efficient resource sharing between diverse cluster computing frameworks is introduced in [18]. The main design elements of Mesos are a fine-grained sharing model at the task level and a distributed scheduling mechanism based on resource offers with delegation of scheduling decisions to application frameworks. Apache Spark is also shown to work efficiently and with performance gain with Mesos. [18]

A review of the key components, abstractions and features of Apache Spark as a unified engine for large-scale data analysis is presented in [48]. As Apache Spark is an open-source project with rapid adoption and development, it's hard for beginners to comprehend the full body of research behind it. To fill this gap, the authors provide real world use case scenarios, an overview of the Spark core and higher level libraries (e.g., MLlib for machine learning, GraphX for graph processing, Spark Streaming for streaming analysis, and Spark SQL for structured data processing), cluster managers and data sources. The core data abstraction, the Resilient Distributed Dataset (RDD) and also newer abstractions, like the DataFrame API and the Dataset API for structured and semi-structured data, are described in more detail. Separate chapters provide detailed overviews on Spark's MLlib and GraphX upper-level libraries and for Spark streaming. [48]

The motivation for creating the SparkBench benchmark suite and a characterization of the chosen benchmark workloads is given in [37]. Apache Spark has been increasingly adopted by industries in recent years for big data analysis with an active community developing a rich ecosystem around Spark. However, a Spark-specific benchmarking suite has been missing to guide the development and cluster deployment of Spark to better fit resource demands of user applications. In that paper, the authors present SparkBench, a benchmarking suite specific for Apache Spark, covering four main categories of applications, including machine learning, graph computation, SQL queries and streaming applications. [37]

How Mesos can be enriched with new resource policy capabilities as required for managing enterprise data centers is demonstrated in [1]. Over the last years, a number of cluster resource managers (e.g., Apache YARN, Google Borg and Omega, Apache Mesos, and IBM Platform EGO) have appeared, aimed at providing a uniform technology-neutral resource representation and management substrate. Apache Mesos is emerging as a leading open source resource management technology for server clusters. However, the authors argue, that the default Mesos allocation mechanism lacks a number of policy and tenancy capabilities, which are important in enterprise deployments. Hence, they developed an experimental prototype, integrating Mesos with the IBM EGO (enterprise grid orchestrator) technology and tested it with SparkBench workloads. [1]

Chapter 3

Virtualization

Virtualization provides an abstraction for the physical hardware and enables multiple operating systems and applications to run in isolation and concurrently on a single physical host machine [6].

The degree of abstraction can range from a highly specialized and constrained runtime environment to the complete emulation of a physical machine. Although virtualization is generally seen as an enabling technology for cloud computing, some of its benefits are also valuable for scientific computing environments.

3.1 Motivation and Benefits

The use of virtualization is motivated by some or all of the following benefits, depending on the domain (e.g., scientific computing or online transaction processing):

1. Energy saving and cost reduction through the consolidation of many - not fully utilized - physical servers into virtual servers running on fewer physical machines. This can also have the advantage of lower requirements for rack space and energy for cooling. [6], [40]
2. The provisioning of a virtual server usually requires less time than the setup of a physical machine. [40]
3. Horizontal scalability can be achieved by starting and stopping additional instances of a (suitable) application to dynamically adapt for changing demand. [40]

4. Vertical scalability is achieved by changing the virtualized resources assigned to the virtual machine. [40]
5. Reduction of a possible vendor lock-in. However, utilizing specialized hardware can be harder or even impossible in a virtualized system. [40]
6. Increased uptime can be achieved with virtualization as it is not necessary to hold a spare identical set of hardware, provided there are sufficient virtual resources available. Also disaster recovery can be faster and simpler for virtual machines. [40]
7. Applications and their runtime environments (e.g., libraries) can be isolated from each other. This is an important aspect as often different versions of libraries and utility programs cannot coexist without causing problems due to incompatibilities. Also the life of outdated applications that require an outdated runtime environment can be extended using virtualization. [40]

3.2 Types of Virtualization

To provide a runtime environment for an application, a wide range of virtualization techniques can be used that abstracts the underlying physical machine. Among the most commonly used techniques are full virtualization, full virtualization with paravirtualized drivers, paravirtualization and operating-system-level virtualization using containers.

3.2.1 Full Virtualization

Full virtualization (also known as hardware virtualization or hardware assisted virtualization) simulates sufficient hardware to run an unmodified guest operating system in isolation. It is required that the virtualized OS uses the same instruction set as the physical machine.

Operations within a virtual machine must be kept within that virtual machine and cannot be allowed to alter the state of another virtual machine or the hardware. Some machine instructions can be executed directly by the hardware, as their effects are entirely contained to the resources assigned to that virtual machine. Other instructions, that access or affect state outside the virtual machine, such as I/O operations, need to be intercepted and simulated by the hypervisor. [50]

An example for full virtualization with support for paravirtualized drivers is the Kernel-based Virtual Machine (KVM). [43]

The Xen project provides a hypervisor for full virtualization, full virtualization with paravirtualized drivers and also paravirtualization support. [44]

3.2.2 Full Virtualization with Paravirtualized Drivers

This type of virtualization is identical to hardware virtualization, but uses additional paravirtualized drivers for improved performance of the virtual machine.

3.2.3 Paravirtualization

In paravirtualization, a hypervisor provides a virtualization of the underlying physical machine, but in contrast to full virtualization, the guest operating system is modified and uses an interface provided by the hypervisor for operations that would have to be intercepted and simulated otherwise. Improved efficiency and performance can be achieved by communicating the intent of the guest operating system to the hypervisor. The guest OS is aware of running as guest in a virtual machine.

In 2005, the Virtual Machine Interface (VMI) was proposed by VMware as communication mechanism between guest OS and the hypervisor. The VMI enabled transparent paravirtualization with a single binary version of the guest OS which can either run on native hardware or on a hypervisor in paravirtualized mode. However, due to improved hardware virtualization support in x86 hardware, the VMI support was dropped from the Linux kernel and VMware products in 2011. [31]

In 2006, paravirt-ops (short pv-ops) - an alternative form of paravirtualization - emerged, which was initially developed by the Xen group. Pv-ops is a piece of Linux kernel infrastructure that allows a single compiled kernel binary that can either run native on bare hardware (or fully virtualized) or run paravirtualized on a suitable hypervisor (e.g., Xen). [50], [44]

3.2.4 Operating-System-Level Virtualization

In operating-system-level virtualization the kernel of an operating system allows multiple isolated user-space instances. These instances, called containers, software containers or jails, often look and feel like a real server from the point of view of its users. The kernel usually also provides resource-management features to limit the impact of one container's activity on other containers.

Although the containers are isolated, they always use the host system's kernel. Hence this approach is less flexible than hardware virtualization as only guest processes that are binary compatible to the host kernel can be run within containers. On the other hand, starting a new container is roughly equal to starting a new process on the host which is much faster than booting a complete operating system within a virtual machine. Also the performance overhead is very small as the processes within the container run natively on the host kernel without any hardware or software virtualization involved. [3]

3.2.5 Docker

Docker is a containerization platform with rapidly growing popularity using operating-system-level virtualization. A Docker container wraps a software in a filesystem that contains everything that the software needs to run (code, runtime, application libraries, system tools, and system libraries). Compare figure 3.1 on page 16 showing virtualization with virtual machines and figure 3.2 illustrating Docker containers.

Docker can be seen as an application delivery technology. The application data does not live in the container, it lives in a Docker volume, that can be shared between an arbitrary number of containers. The Docker containers are optimally stateless and immutable. Only the Docker volumes contain state and have to be backed up. In multitenancy environments when shared kernels can't be used or are undesired, virtual machines can be utilized to provide an extra layer of isolation compared to running containers on bare metal. [23]

Since Docker containers are very lightweight, developers can build stacks of Docker containers on their laptops that replicate a production environment. Then they can build and run their applications against that stack. Docker containers are also portable, they can be moved around. Another area with a lot of interest is called high capacity. While in traditional VMs the hypervisor often occupies about 10 to 15 percent of the host capacity, running Docker containers on a Docker host allows to run hyperscale numbers of containers as they sit right on top of the operating system and are very fast. Another interesting technology Docker relies on is the concept of copy-on-write which kernel developers call a union file system. The file system is built from layers of file systems. Every Docker container is built on top of a Docker image which is a prebaked file system that contains the libraries and binaries that are required to run the application. That Docker image is a read-only file system constructed of multiple layers, where each layer captures the changes from a specific operation (e.g., install PHP). These layers can be cached which makes rebuilding images extremely fast when layers can be reused. [3]

To isolate the processes within different containers from each other Docker relies on two pieces of kernel technology: namespaces and control groups (cgroups). Namespaces allow to basically build a box and assign resources like access to CPU, memory, a network or part of the file system. From inside that process namespace, any other processes outside of it are not visible. Control groups are designed to manage and constrain the resources available to a container, e.g., limit the amount of RAM and CPU cores or allow access to a network interface, in a fairly fine granularity. [3]

Containers also provide an isolated environment for network addresses and ports. As each container has its own virtual network interface, multiple processes running inside different containers can use the same port numbers. By default, the container does not expose network ports to the outside, but any port used inside the container can be mapped to an unused port on the host. [3]

3.2.6 Rkt Container Engine

Rkt [22] is an open-source implementation of the App Container Spec (appc) by CoreOs [21], which defines an image format (App Container Image) and a runtime environment for container virtualization. It was designed with strong focus on security from the beginning. Images can be fetched and run from an unprivileged user. There is no centralized daemon to manage containers, instead containers are directly launched from client commands, making it compatible with init systems such as systemd, upstart, and others. Rkt can also run Docker containers. [22]

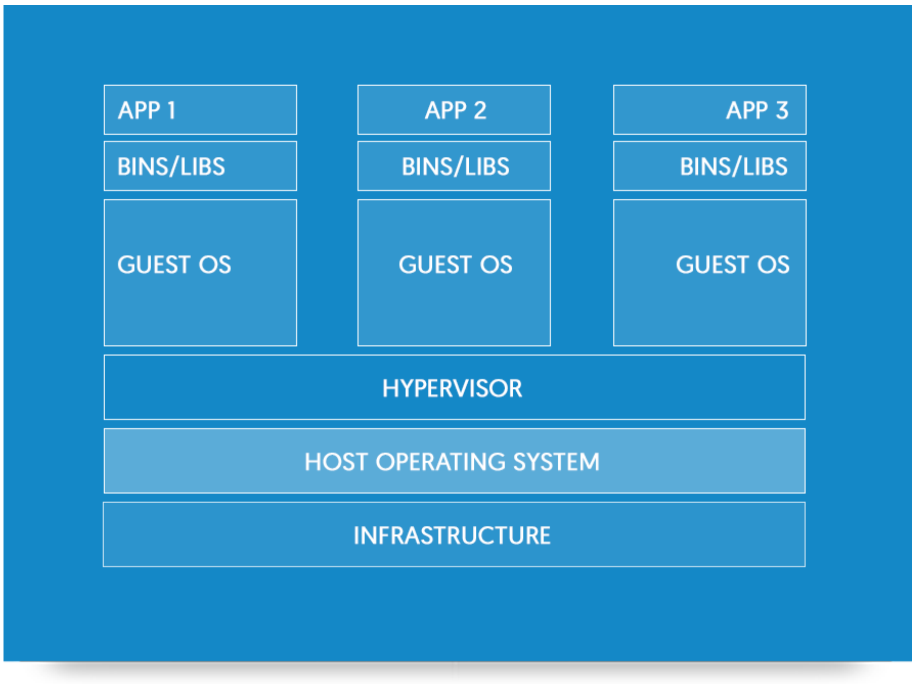

Figure 3.1: Virtual Machines, adopted from [24]

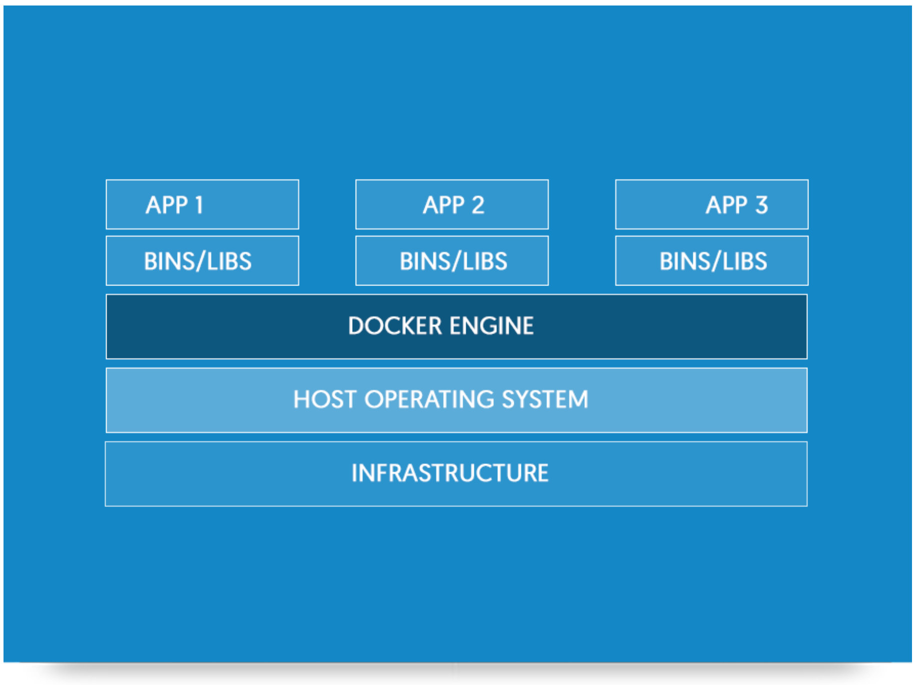

Figure 3.2: Docker Containers, adopted from [24]

3.3 Hardware Support for Virtualization

Although similar hardware support for virtualization was developed for different platforms, the discussion within this section is limited to the x86 platform and uses Intel's terminology for specific examples.

At the beginning, virtualization on the x86 platform was achieved using complex software techniques due to missing hardware virtualization support. Starting in 2005 Intel (and independently also AMD) developed processor extensions to improve the performance of virtualized x86 systems. These and later added extensions are part of "Intel virtualization" (VT-x). [7], [19]

The first generation of x86 hardware virtualization addressed privileged instructions.

In 2008, memory management unit (MMU) virtualization was added to the chipset to address the low performance of virtualized system memory. This extension is called Extended Page Table (EPT) which is Intel's implementation of Second Level Address Translation (SLAT), also known as nested paging. This avoids the overhead associated with software-managed shadow page tables. [7]

Starting with the Haswell microarchitecture (in 2013), Intel included hardware support for Virtual Machine Control Structure (VMCS) shadowing as a technology to accelerate nested virtualization. [7]

Today, advanced hardware support for different aspects of virtualization is available. CPU virtualization features enable live migration of virtual machines and efficient nested virtualization. Memory virtualization features include direct memory access (DMA) remapping and extended page tables. I/O virtualization features enable offloading of multi-core packet processing to network adapters. Virtual machines can also have full or shared assignment of graphics processing units (GPU). [7]

Chapter 4

Cluster and Resource Management

Many academic, research, and also commercial institutions have invested heavily in building HPC facilities for running scientific applications. Frameworks for processing large-scale data (e.g., Apache Hadoop, Apache Spark) have emerged in recent years and gained popularity as platform for executing even mission critical data processing applications. As these new frameworks typically run on relatively cheaper clusters built from commodity hardware, the situation arised to have to build and maintain two parallel facilities. To reduce the maintenance effort and also increase the utilization, it is desirable to be able to execute HPC jobs (e.g., based on MPI) and large-scale data processing applications (e.g., using Hadoop) on the same hardware cluster. [51]

This chapter gives a short overview of cluster and resource management with particular focus on the cluster management software and frameworks used for the experiments. Within this chapter, cluster management refers to the management of tasks or processes including monitoring and handling of failures, while resource management refers to the management of compute resources, task isolation and low-level constraint enforcement.

The provisioning and management of the cluster node machines is not discussed here.

4.1 HPC vs. Big Data Infrastructure

Traditionally, HPC environments have been designed to meet the compute demand of scientific applications and data was only a second order concern. Hence, HPC systems generally rely on fast interconnects between compute and storage and often use parallel file systems (e.g., Lustre or GPFS) on large, optimized storage clusters exposing a POSIX compliant interface connected via fast interconnects. This infrastructure is not optimal for data-intensive, I/O-bound workloads that require a high sequential read/write performance. [38]

Infrastructure for data-intensive workloads using the Apache Hadoop ecosystem or newer similar frameworks (e.g., Apache Spark) relies on data-locality. Frameworks like Hadoop co-locate compute and data, instead of moving the data to the compute resources. In addition to MapReduce, newer frameworks like Apache Spark also support memory-centric data processing and analytics, and machine learning. [38]

4.2 Cluster Management Overview

An application running on a compute cluster requires some management component to discover and access the cluster resources.

The job scheduler that decides which tasks are executed on which resources can be seen as the operating system of a data center. Job schedulers are a key part of modern computing infrastructure. The computing capabilities can only perform well if the job scheduler is effectively managing resources and jobs. [45]

Historically, in the domain of supercomputers, job schedulers were designed to run massive, long running computations over days and weeks. More recently, big data workloads have created a new class of computations consisting of many short computations taking seconds or minutes that process very large quantities of data. [45]

In the simplest case, the application or framework supplies its own standalone cluster manager and the worker components that need to run on each cluster node. Such frameworks (e.g., Apache Spark, Apache Hadoop) often also contain support for other cluster and resource managers (e.g., Apache Mesos). In that case the application can focus on the tasks that need to run and dynamically negotiate the required resources with the cluster and resource manager, which runs the worker processes on behalf of the application.

Resource managers provide abstractions of the resources for single machines and also for entire clusters. A resource manager offers an application programming interface (API), that is used by compatible client programs and frameworks to acquire resources and to run tasks that consume the assigned resources.

Cluster managers are responsible for running the required tasks and, in case of failures, also restarting lost tasks or services, if applicable. The task management is often (partially) delegated to application-specific task managers. [12], [10]

4.3 Standalone Cluster Management

For single-purpose clusters the standalone cluster manager is a simple and fast way to setup a cluster for a particular application or framework. The configuration is specific to the framework and usually includes a list of all cluster worker node machines and one or more dedicated master nodes. The list of worker nodes is often only required to remotely start or stop the worker processes on the worker nodes.

The worker processes usually register themselves with the master process on the active master node, hence it's generally possible to add nodes dynamically by starting new worker processes with the configuration required to contact the master.

The biggest disadvantage of standalone cluster managers is that generally only one particular application is supported. Hence, the simultaneous use of a cluster with dynamic resource sharing and different frameworks is normally not possible. Static partitioning of the cluster, where each node is reserved for a predefined framework, is easily achieved.

However, the simplicity of the setup and configuration of standalone cluster managers works well in combination with orchestration frameworks (e.g., Mesosphere [26], Docker swarm [24]), that can run services on a cluster. By providing suitable images, programs like Apache Spark can run as distributed services on a cluster. The service then often internally uses its own standalone cluster manager to utilize the resources that were assigned to the service by the external cluster manager.

4.4 General Cluster and Resource Management Software

Dedicated cluster and resource managers (e.g., Apache Mesos, YARN) offer a generic application programming interface (API) that can be used by many different guest frameworks simultaneously. [12], [10]

That enables dynamic and potentially more efficient sharing of cluster resources between various applications and frameworks, provided all these frameworks are compatible with the resource manager.

The various general cluster managers available provide different services, with stronger focus on the resource management aspect (e.g., Mesos, YARN) or the cluster management aspect (e.g., Docker swarm).

A resource manager generally uses some virtualization technology to isolate the client tasks from each other and to enforce resource utilization constraints.

4.5 Resource Abstractions

The resource abstractions common in resource managers generally include the number of CPU cores, the amount of memory (RAM), available storage capacity on hard disk drives (HDD) or solid state disks (SSD) and network interfaces and ports. Some frameworks also support the resource management for specialized hardware, like GPUs, which may require suitable plugins or the use of provided extension mechanisms.

4.6 Modeling Cluster Resources

For the simple use case, that all available resources of a host machine participating in a cluster should be offered as cluster resources, the automatic resource detection of the framework is usually sufficient and no machine specific resource configuration is required.

If needed, the available resources on a machine can also be specified manually using configuration files or command line parameters for the agents running on each worker node. [12], [10]

In addition to the automatic resource discovery, user-defined key-value pairs can be used for further distinction of the nodes or their specific properties.

4.7 Resource Scheduling

Efficient resource scheduling with a centralized scheduler is complex and challenging, especially when considering the diverse requirements for different client frameworks. Additionally, scalability and resilience are also hard to achieve with a complex centralized scheduler. An alternative to a centralized scheduler is delegating control over scheduling to the client frameworks. [18]

The resource negotiation comes in two primary flavors:

- pull-model (e.g., YARN): the client actively requests resources with specific properties which are then granted or refused by the resource manager
- push-model (e.g., Mesos): the resource manager actively makes resource offers and the client decides to accept or decline the offer based on its own requirements

4.8 Cluster and Resource Management Frameworks

This section contains an overview of three exemplary resource and cluster management frameworks, that were also used for the experiments in this work.

4.8.1 Apache Mesos

Apache Mesos is an open-source cluster management software. It tackles the problem to run multiple frameworks on a single cluster of (commodity) servers enabling dynamic sharing of cluster resources. Mesos's goals are

- high utilization of resources
- support for diverse frameworks

- scalability to 10,000's of nodes
- reliability in case of node failures

to improve cluster utilization and also possibly share data between frameworks. [12]

The main concepts are: [12]

- Mesos does only inter-framework scheduling via resource offers
- intra-framework scheduling is done by the framework's schedulers, which can account for specific requirements (e.g., data locality)
- push-model: Mesos makes resource offers, that clients can accept or decline
- task isolation using different containerizers (e.g., Docker or the Mesos containerizer using Linux containers)

Architecture

Mesos (cf. figure 4.1 on page 25) consists of two main components, a master daemon that manages the agent daemons which are running on each cluster node (agents were called slaves in early versions). Mesos frameworks run the actual tasks on the cluster nodes. In a high-availability setting one active master daemon and several standby master daemons running on different nodes coordinate the cluster state and master election using Apache Zookeeper [15]. [18], [12]

The master facilitates fine-grained resource (CPUs, GPUs, RAM, disk, and ports) sharing across frameworks by making resource offers. A resource offer contains a list of ⟨agent ID, resource1: amount1, resource2: amount2, ...⟩. How many resources each framework is offered is determined by the Mesos master according to a given policy, e.g., fair sharing or strict policy. The modular architecture of the master provides a plugin mechanism to add new allocation modules. [12]

Frameworks run on top of Mesos. Such a framework consists of a scheduler that registers with the master to be offered resources and executor processes that are launched on agent nodes to run the framework's tasks. The Mesos master determines how many resources are offered to each framework, and the framework schedulers select which of the offered resources are used. The framework passes a description of the tasks it wants to run to Mesos for the offered resources it accepts. Mesos, in turn, launches the tasks on the corresponding agent nodes. [12]

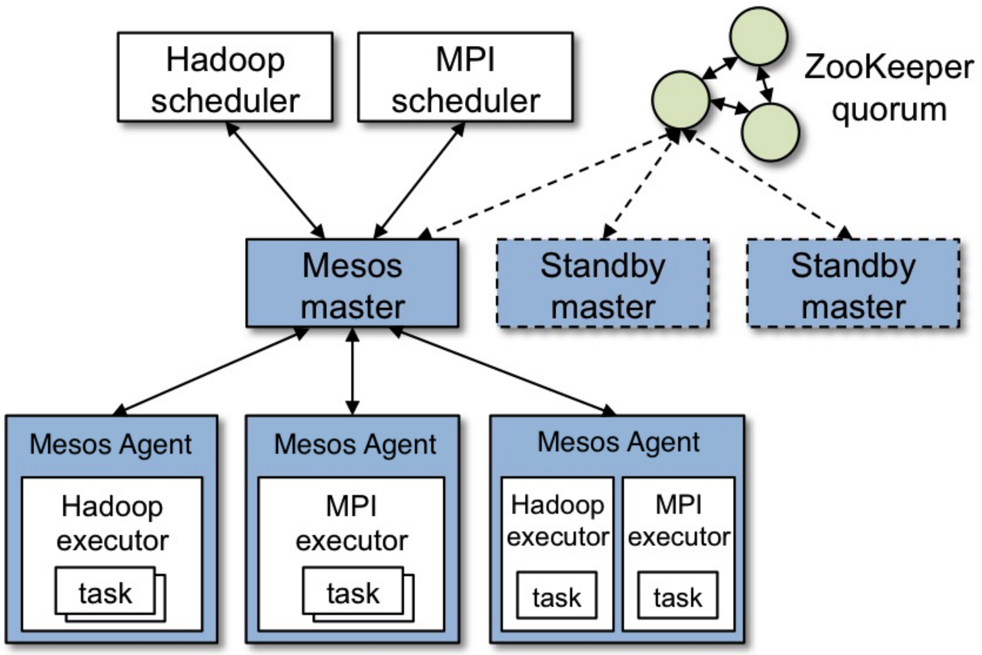

Figure 4.1: Mesos Architecture Overview, adopted from [18]

An example of how a frameworks gets scheduled to run a task is shown in figure 4.2 on page 26. [18]

1. Agent 1 reports to the master that it has 4 CPUs and 4 GB of memory free.
2. The allocation module of the master then decides that framework 1 should be offered all available resources. The master sends a resource offer, containing a description of these resources, to framework 1 in step (2).
3. In step (3), the framework's scheduler replies to the master with information about the tasks to run on the offered resources, in this case using ⟨2 CPUs; 1 GB RAM⟩ and ⟨1 CPU; 2 GB RAM⟩ for tasks 1 and 2.
4. In the last step (4) the master sends the tasks to agent 1, which allocates the correspondent resources to the framework's executor, which then launches the two tasks.

As 1 CPU and 1 GB of RAM are still free on agent 1, the allocation module can offer them to framework 2. When tasks finish and new resources become free, this resource offer process is repeated.

Mesos agents can run with minimal configuration and use automatic resource detection in this case. For precise control over the resources offered by Mesos it is

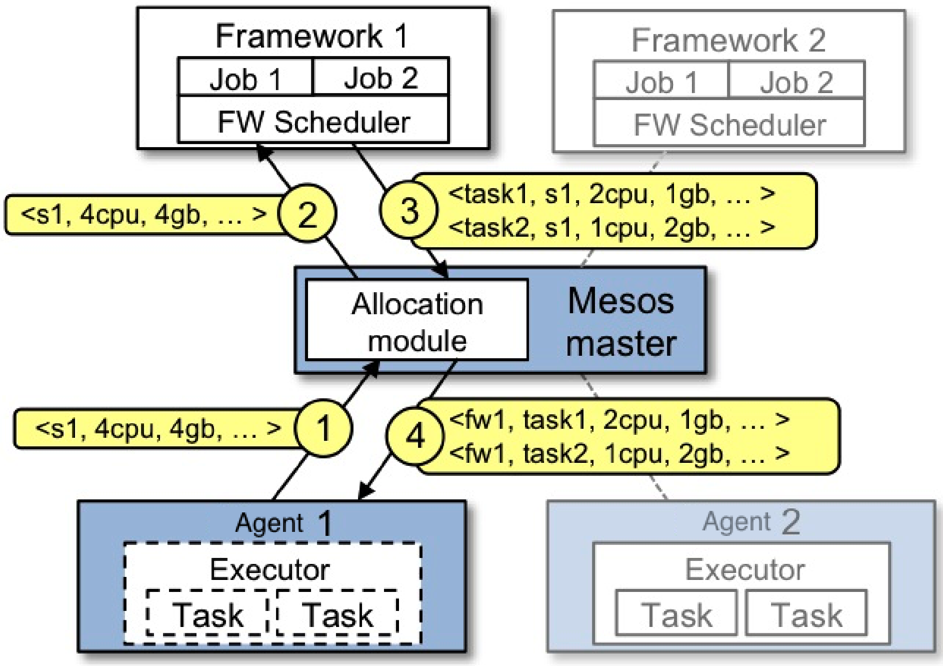

Figure 4.2: Mesos Architecture Example, adopted from [18]

possible to define a set of key-value pairs for each node that consists of predefined properties like CPUs, GPUs, RAM, disk, ports and also user defined properties. [12]

4.8.2 Apache Hadoop YARN

YARN is the resource management component of the Apache Hadoop map-reduce framework.

The fundamental principle of YARN is the separation of resource management and job scheduling into separate daemons. There is a global `ResourceManager` (RM) and a per-application `ApplicationMaster` (AM). [10]

Refer to figure 4.3 on page 28 for an architecture overview of YARN. The `ResourceManager` is the single authority that arbitrates resources among the applications in the system. The `NodeManager` agent runs on each machine and is responsible for containers, monitoring their resource usage (CPU, RAM, disk, and network) and reporting the same to the `ResourceManager`. The `ApplicationMaster` is an application-specific library tasked with negotiating resources from the `ResourceManager` and working with the `NodeManagers` to execute and monitor the tasks. [10]

The `ResourceManager` has two main components, the `Scheduler` and the `ApplicationsManager`. The `Scheduler` is responsible for allocating resources to the various running applications taking into account familiar constraints of capacities, queues, etc. The `Scheduler` does not perform monitoring or tracking of the status of the application and it does not offer guarantees about restarting failed tasks either due to application failure or hardware failures. The scheduling function of the `Scheduler` is solely based on the resource requirements of the application using the abstract notion of a resource container which incorporates elements such as CPU, RAM, disk space, network, etc. The `Scheduler` has a pluggable policy which performs the partitioning of the cluster resources among the various queues, applications, etc. [10]

The `ApplicationsManager` accepts job submissions, negotiates the first container for executing the application specific `ApplicationMaster` and provides the service for restarting the `ApplicationMaster` container on failure. The per-application `ApplicationMaster` negotiates appropriate resource containers from the scheduler, tracking their status and monitoring for progress. [10]

YARN also offers resource reservation via the `ReservationSystem`, that allows users to specify a profile of resources over time and temporal constraints and reserve resources to ensure the predictable execution of important jobs. [10]

YARN uses a pull-model for resource negotiation, where the client (e.g., an `ApplicationMaster`) requests resources specified in terms of CPU, RAM, disk space, network, etc.

4.8.3 Docker Swarm

A Docker swarm is a cluster of Docker engines that are typically distributed over multiple hosts. The docker engines participating in a swarm are running in swarm mode. The swarm provides cluster management and orchestration features including scaling, replication, rolling updates and load balancing for services, which are deployed in the swarm. For services, that are externally accessible, Docker swarm provides ingress load balancing. It is also possible to create private overlay networks that are automatically extended to all nodes running a service using that network. [24]

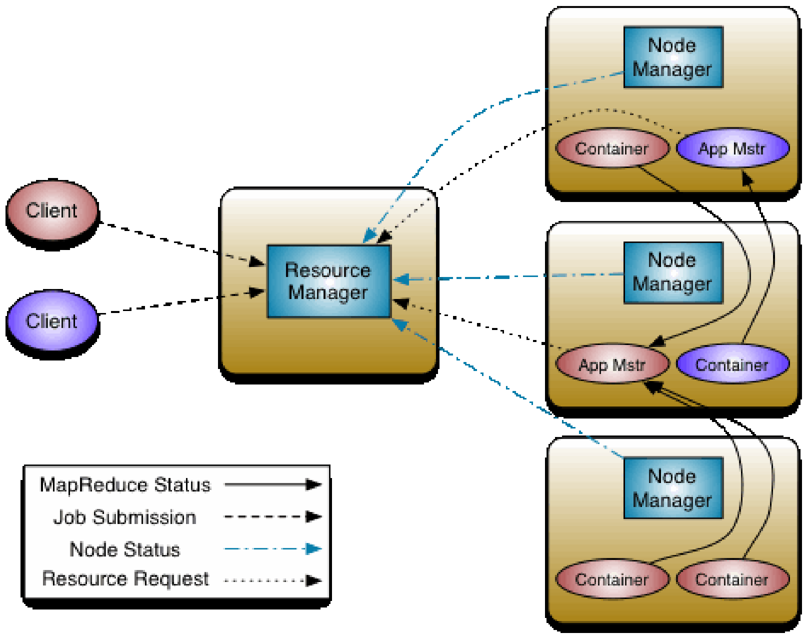

Figure 4.3: Hadoop YARN Architecture, adopted from [10]

Chapter 5

Data-Intensive Applications Running on Scientific Clusters

In recent years, the progress of contemporary collaborations in various research fields led to high demand for large-scale computing and data-processing systems. Many domains, traditionally utilizing high-performance computing for simulation of complex formal models, began to expand towards data-intensive computing, as the available data volumes outgrow computing capabilites. [34]

While there are many different approaches to tackle data-intensive HPC problems, the rest of this chapter puts the focus on Apache Spark [13], a general purpose large-scale data processing framework, that was used for the experiments and benchmarks in this work.

For machine learning applications, there also is (among others) the KeystoneML [2] framework, that provides a higher level API to describe a machine learning pipeline that is then optimized and runs on Apache Spark. [49]

Apache Hadoop [10] is a general purpose map-reduce framework that also provides a distributed file system, the Hadoop distributed file system (HDFS), which was used for storing the test data. Since the Hadoop map-reduce framework was not used at all, it is not described here.

5.1 Apache Spark

Apache Spark is a general engine for large-scale data processing. Spark supports applications written in Java, Scala, Python and R. The execution engine is based on a directed acyclic graph (DAG) and supports acyclic data flow and in-memory computing with programmatic cache control. Interactive shells providing a read-eval-print-loop (REPL) are offered for Scala, Python and R. Spark runs on Hadoop, Mesos, and standalone, and can also be deployed in the cloud. [13]

Spark can run in batch mode or streaming mode (using micro-batches).

The core data abstraction in Spark is the Resilient Distributed Dataset (RDD), which is a fault-tolerant, read-only, partitioned collection of records, providing the base for designing scalable data algorithms and pipelines. Additional dataset representations were developed and introduced with later releases, e.g., the DataFrame API and the Dataset API. [48]

With its programmatic cache control, Spark is especially well suited for workloads that require iterative processing with several passes over the same dataset (e.g. for machine learning algorithms). [48]

Apache Spark was initially started by Matei Zaharia at UC Berkeley's AMPLab in 2009 and open sourced in 2010 under a BSD license. [48]

5.1.1 Spark Architecture

A Spark application runs as an independent set of processes on a cluster, coordinated by the `SparkContext` object in the main program (called the driver program). [13]

To run on a cluster, the `SparkContext` can connect to several types of cluster managers (Spark standalone cluster manager, Mesos or YARN), which allocate resources across applications. Spark then acquires executors on nodes in the cluster, which are processes that run computations, and read and store data for the application. Next, Spark sends the application code (supplied as JAR or Python files passed to the `SparkContext`) to the executors. Finally, the `SparkContext` sends tasks to the executors to run. [13]

See figure 5.1 on page 31 for a cluster component overview.

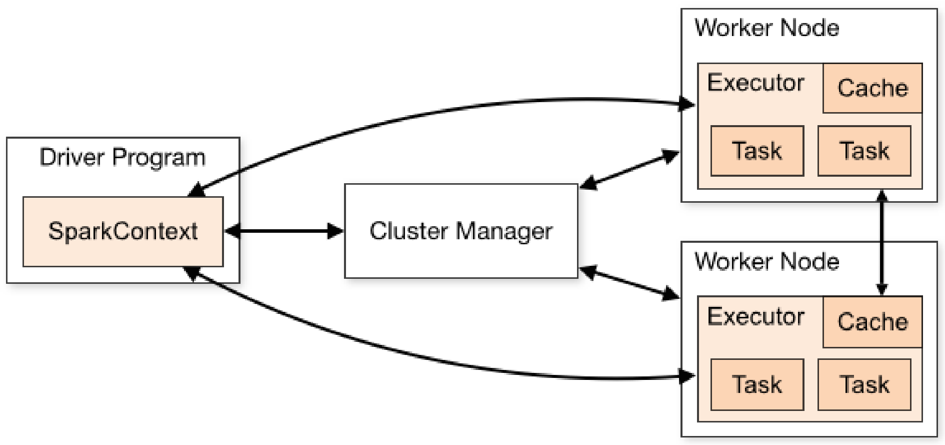

Figure 5.1: Spark Cluster Components, adopted from [13]

5.1.2 Submitting Applications to Spark

Spark provides a `spark-submit` script to launch applications on a cluster. That script allows to specify the cluster manager and other application specific options, which can also be specified programmatically within the application. The application can be submitted with two different deploy modes, **client** and **cluster** mode.

Client Deploy Mode

It's a common deployment strategy to submit the applications from a gateway machine which is physically co-located with the cluster worker machines. That machine can also be the master node in a cluster using the Spark standalone cluster manager. The client deploy mode is appropriate in this setup and the driver is launched directly within the `spark-submit` process, which acts as a client to the cluster. The standard input and output streams are attached to the console, hence the client mode is especially suitable for applications with user interaction, like the REPL (e.g., the Spark shell). [13]

A fast, low-latency and reliable network connection between the machine, where the application is submitted, and the cluster node is essential in client deploy mode. [13]

Cluster Deploy Mode

If the application is submitted from a machine far from the cluster worker machines, it is common to use cluster deploy mode to minimize network latency between the drivers and the executors. [13]

In cluster deploy mode the driver runs on any of the cluster worker nodes and thus cannot interact with the user. The application needs to read and write all data and results using an external data storage (e.g., HDFS).

Chapter 6

Preparing for the Experiments

This chapter gives a detailed description of the setup of all frameworks and tools required for the experiments.

The cluster consists of only four identical nodes, thus the high availability setup was not used.

6.1 Test Environment

The `nova` scientific cluster, that was used to perform the experiments, consists of four identical machines with two AMD Opteron(tm) Processor 6172 CPUs (in total 24 cores) and 48 GB of RAM each running CentOS Linux release 7.3.1611 (Core) at the time of the experiments. Only the front end machine (`nova`) is connected to the internet, while all four cluster nodes (`nova, nova-compute01, nova-compute02, nova-compute03`) are connected to a separate high-speed (Infiniband) LAN.

The cluster topology is shown in figure 6.1 on page 34.

6.2 Infrastructure Preparation

This chapter describes the preparations regarding the installation and configuration of the software required for all of the experiments. Additional software and configuration for individual experiments is described in the respective section.

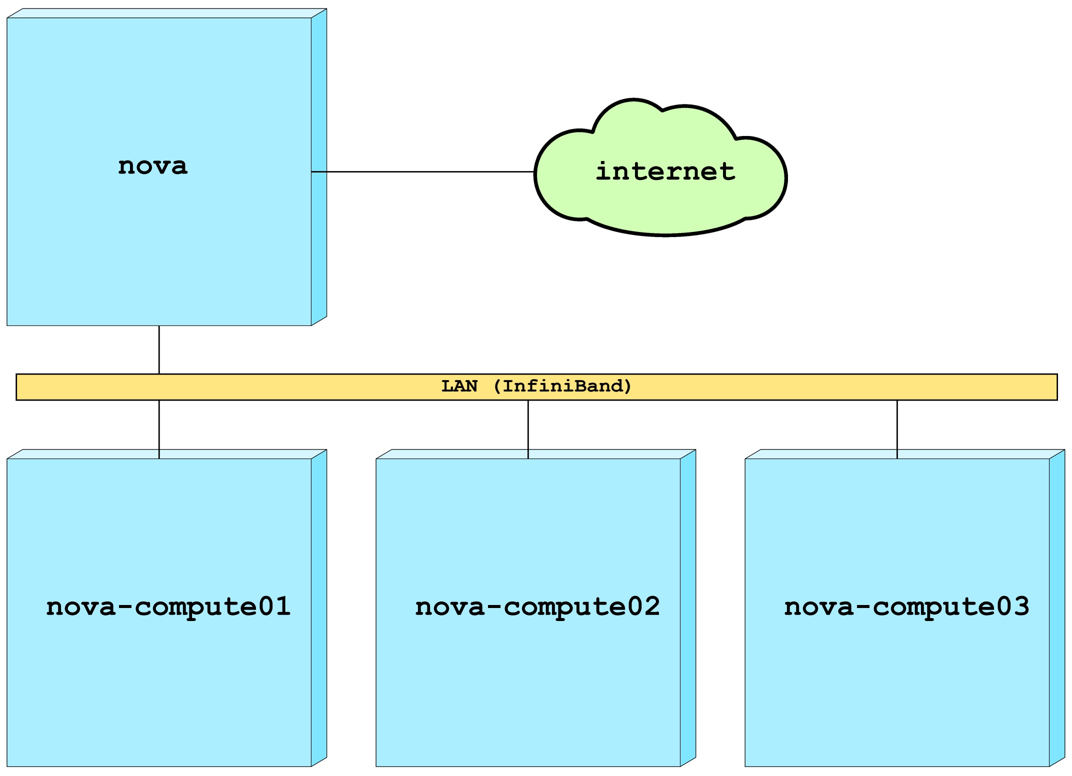

Figure 6.1: Nova Cluster Topology Overview

To minimize the maintenance overhead, whenever possible, software was installed into a shared file system mounted on each cluster node. This includes the Java Virtual Machine (JVM), Apache Hadoop, and Apache Spark.

Only Apache Mesos was compiled from source using the shared filesystem and installed locally on each node.

6.2.1 Apache Hadoop Installation and Configuration

In addition to the already existing, older JVM, a current version (Oracle Java SE Development Kit 8u112) was installed into a directory in the shared filesystem and the `JAVA_HOME` environment variable was updated accordingly for the relevant user accounts.

Apache Hadoop (version 2.7.3, later updated to 2.8.0) was installed into a shared directory as described on the project homepage [10] with its own user `hadoop` and configured to run the Hadoop NameNode on the primary node and a Hadoop DataNode on each cluster node.

The included Hadoop startup scripts require the setup of a login without password on every cluster node to be able to start and stop the Hadoop master, worker, NameNode, and DataNode processes, which was enabled using SSH keys.

Hadoop is used primarily to run the Hadoop distributed file system (HDFS) and also the YARN resource manager for some setup variants.

The HDFS cluster was started and stopped using the supplied shell scripts (`$HADOOP_HOME/sbin/start-dfs.sh` and `$HADOOP_HOME/sbin/stop-dfs.sh`).

6.2.2 Apache Spark Installation

Apache Spark (version 2.0.2 with the Hadoop 2.7 client libraries) was installed into a directory in the shared file system as described on the project homepage [13] with its own user `spark`. The initial configuration used the Spark standalone cluster manager on the primary node (`nova`) and a Spark worker on all four cluster nodes.

The included Spark startup scripts require the setup of a login without password on every cluster node to be able to start and stop the Spark master and agent processes, which was enabled using SSH keys.

The specific configurations for different cluster managers are described in the respective sections.

6.2.3 Docker Installation

The already installed outdated version of Docker (1.10.3) was removed and a current version (1.12.5, later upgraded to 1.13.0 and then to 17.04.0-ce) was installed according to the instructions for CentOS 7 on the Docker web page [24].

6.2.4 Private Docker Registry Installation

A private (insecure) Docker registry was installed on `nova` according to the description on the Docker web page [24]. Since the registry is publicly available as Docker image on the Docker hub, it was simply started as Docker container with the registry service port bound to the local area network interface for the cluster nodes, which is not accessible from the internet. The data folder from the container was mapped to a directory in the local filesystem on the host.

To make the private registry accessible from Docker, the `--insecure-registry nova:5000` option needs to be added to the Docker service configuration file `/etc/systemd/system/docker.service.d/docker.conf` on each cluster node and after reloading the service configuration, the Docker daemon must be restarted.

6.2.5 Apache Mesos Installation

Apache Mesos is not available as pre-built binary package. The complete source code and all other files required to build it are available as download from the project web page [12].

The Mesos source package (version 1.1.0) was downloaded and extracted into a folder on the shared file system. The binaries were built following the instructions given on the Mesos web page [12] for CentOS 7.1. All required dependencies were installed using the `yum` package manager on each cluster node.

After successfully running the tests, Mesos was installed on each node using "`make install`". That command installs the mesos binary files into `/usr/local/bin` and the libraries into `/usr/local/lib`.

Initially, that library path was part of the library search path only on the primary cluster node. The resulting problem surfaced with the following error message each time a worker task was started on any other cluster node than the primary one: `mesos-executor: error while loading shared libraries: libmesos-1.1.0.so: cannot open shared object file: No such file or directory`

Hence, it was essential to create the necessary configuration file and to update the library search path on each node with the commands shown in listing 6.1.

Listing 6.1: Expand the Library Search Path

```
echo "/usr/local/lib" > /etc/ld.so.conf.d/mesos.conf
ldconfig -v
```

Running the Mesos master and agents with their own non-root user (`mesos`) did not work as intended, since other non-root users (e.g., `spark`) were unable to submit tasks to the cluster. The Mesos documentation [12] also states, that, to enable multiple Unix users to submit to the same cluster, the Mesos agents need to run as `root`.

Thus, for all experiments the Mesos master and agents were run as `root`.

To prevent the guest processes running on the cluster resources to also run as `root`, the parameter `switch_user=true` can be set for the agents. However, then the Mesos slaves still created the sandbox folders with owner `root` with exclusive access permission for the owner, which caused the guests running as non-root (e.g., `spark`) to fail immediately with permission denied error for the sandbox directory.

Hence, the parameter `switch_user=false` had to be used for all experiments, creating a less secure setup.

Mesos provides shell scripts to start and stop the Mesos master and agents on all cluster nodes which need to be run as `root` on the master node:

- `/usr/local/sbin/mesos-start-cluster.sh`
- `/usr/local/sbin/mesos-stop-cluster.sh`

For this to work, the IP addresses of the master and agent nodes need to be added to the respective configuration file:

- `/usr/local/etc/mesos/masters` contains the IP address of the master node
- `/usr/local/etc/mesos/slaves` contains the IP addresses of all cluster nodes

Each Mesos worker requires its own IP address as environment variable. To avoid having to set the correct IP address on each node, an auto-discovery command was specified as follows: `MESOS_ip_discovery_command="hostname -i"`

The Mesos containerizer and Docker containers were used for the experiments, which was set with the `containerizers=docker,mesos` parameter.

See listings 6.2 and 6.3 on page 38 for the Mesos configuration files, which reside in the directory `/usr/local/etc/mesos`.

Listing 6.2: Mesos Master Configuration File: `mesos-master-env.sh`

```
export MESOS_ip=10.0.0.1

export MESOS_log_dir=/var/log/mesos
export MESOS_work_dir=/var/lib/mesos
```

Listing 6.3: Mesos Agent Configuration File: `mesos-agent-env.sh`

```
export MESOS_ip_discovery_command="hostname -i"
export MESOS_master=10.0.0.1:5050

export MESOS_switch_user=false
export MESOS_containerizers=docker,mesos

export MESOS_work_dir=/var/lib/mesos
export MESOS_log_dir=/var/log/mesos
```

6.3 Benchmarks

The following benchmarks were executed for each setup:

1. Pi approximation using a Monte Carlo method
2. K-Means clustering benchmark from Spark-Bench [5]
3. Terasort benchmark from Spark-Bench [5]

All benchmarks were run in Spark client mode on the primary cluster node (`nova`).

6.3.1 Pi Approximation

The pi approximation is part of Apache Spark's examples and was run for three different numbers of partitions (1,000, 5,000, and 10,000), which is specified as the last command line parameter in listing 6.4 on page 39. For each partition 100,000 samples are calculated to estimate the value of pi.

The parameter (`--master`) for the cluster manager is dependent on the installation and deployment mode variant.

Listing 6.4 shows the command to submit the pi approximation for the Spark standalone cluster manager, which is accessible at `spark://nova:7077`.

Listing 6.4: Start Pi Approximation Using Spark Standalone Cluster Manager

```
$SPARK_HOME/bin/spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master spark://nova:7077 \
  --executor-memory 2G \
  $SPARK_HOME/examples/jars/spark-examples_2.11-2.0.2.jar \
  1000
```

6.3.2 Spark-Bench

Spark-Bench is an open-source benchmarking suite specific for Apache Spark. It comprises a comprehensive set of workloads currently supported by Apache Spark. It also includes a data generator that allows users to generate input data of arbitrary size. [37]

Spark-Bench lists the following use cases: [5]

1. Enables quantitative comparison for Apache Spark system optimizations such as caching policy, memory management optimization and scheduling policy optimization.
2. Provides quantitative comparison for different platforms and hardware cluster setups such as Google cloud and Amazon cloud.
3. Offers insights and guidance for cluster sizing and provision and also helps to identify bottleneck resources.
4. Allows in-depth study of the performance implications of various aspects including workload characterization, configuration parameter impact, scalability, and fault tolerance behavior of an Apache Spark system.

6.3.3 Spark-Bench Installation and Setup

Spark-Bench was installed by cloning the public Git repository with the command "`git clone https://github.com/SparkTC/spark-bench.git`" and following the instructions given on the project page [5], which included locally building one missing dependency (`wikixmlj`, cloned from `https://github.com/delip/wikixmlj.git`). Both projects can be built using Apache Maven [11], which was already installed on `nova`, with the command "`mvn clean install`".

However, the Spark version (1.6.1), Hadoop version (2.3.0), and Scala version (2.10.5) originally used in Spark-Bench was outdated. Hence, the `pom.xml` file was edited to update the Spark version to 2.0.2, Hadoop version to 2.7.3 and the Scala version to 2.11.8. This produced some compile errors, which all had the same original cause: `object Logging is not a member of package org.apache.spark`. That object in the Spark source code was moved to a private package in Spark version 2.0.0. The offending statements, which only printed some debug logs, were simply removed to fix the compile errors.

The included shell scripts require the `bc` calculator utility to be installed, which was done using the `yum` package manager.

6.3.4 Spark-Bench Benchmark Configuration

The base directory for the Spark-Bench test data was created in the HDFS using the command "`hdfs dfs -mkdir /SparkBench`".

The following modifications were made in the main configuration file `${SPARK_BENCH_HOME}/conf/env.sh`:

- `master=nova`
- `HDFS_URL="hdfs://${master}:9000"`

For both benchmarks - described in the following two sections - the number of partitions of the input data was chosen to be close to a multiple of the overall number of CPU cores (which is 96). Especially for long running tasks, to avoid having a large number of CPU cores waiting for few tasks to finish, the number of partitions should be close to, but not just slightly greater than a multiple of the overall number of CPU cores. The chosen number of partitions was 270 for large and medium size data sets and 90 for small data sets. As Spark can efficiently support tasks as short as 200 ms [13], it is reasonable to increase the level of parallelism for the larger data sets to keep each task's input set relatively small.

Even if this choice possibly may not yield optimal performance for the given hardware configuration, the most important factor was to provide a working configuration that remained unchanged for the experiments to produce comparable results.

K-Means Benchmark Configurations

The K-Means benchmark was run with three configurations which differ in the number of points in the input data (1 million, 5 million, and 10 million points) and in the number of partitions (90 or 270 partitions).

The input data was generated for each configuration with the command "`${SPARK_BENCH_HOME}/KMeans/bin/gen_data.sh`".

Each benchmark run was started with the command "`${SPARK_BENCH_HOME}/KMeans/bin/run.sh`".

For the large input data set the content of the configuration file `${SPARK_BENCH_HOME}/KMeans/conf/env.sh` is shown in listing 6.5.

Listing 6.5: K-Means Benchmark Configuration File

```
NUM_OF_POINTS=10000000
NUM_OF_CLUSTERS=100
DIMENSIONS=20
SCALING=0.6
NUM_OF_PARTITIONS=270
MAX_ITERATION=10
NUM_RUN=1
NUM_TRIALS=10
```

For the medium sized input data set the line with the `NUM_OF_POINTS` was changed according to:

```
NUM_OF_POINTS=5000000
```

And for the small input data two lines in the configuration file were changed according to:

```
NUM_OF_POINTS=1000000
NUM_OF_PARTITIONS=90
```

Terasort Benchmark Configurations

The Terasort benchmark was run with three configurations which differ in the number of elements (5 million, 10 million, and 50 million) in the input data.

The input data was generated for each configuration with the command "`${SPARK_BENCH_HOME}/Terasort/bin/gen_data.sh`".

Each benchmark run was started with the command "`${SPARK_BENCH_HOME}/Terasort/bin/run.sh`".

For the large input data set (roughly 5 GB) the content of the configuration file `${SPARK_BENCH_HOME}/Terasort/conf/env.sh` looks as shown in listing 6.6:

Listing 6.6: Terasort Benchmark Configuration File

```
NUM_OF_RECORDS=50000000
NUM_OF_PARTITIONS=270
MAX_ITERATION=12
NUM_TRIALS=10
```

For the medium sized input data set (approximately 1 GB) the line with the `NUM_OF_RECORDS` was changed to:

```
NUM_OF_RECORDS=10000000
```

And for the small input data (approximately 500 MB) the configuration file was changed according to:

```
NUM_OF_RECORDS=5000000
```

Chapter 7

Running the Experiments

This chapter describes the experiments performed with Apache Spark using different setup and configuration options. The motivation, advantages, and drawbacks for each setup are discussed elaborately and the experience is described including any obstacles encountered.

7.1 Native Apache Spark

This section describes the experiments performed with the native installation of Apache Spark using different resource and cluster management strategies.

7.1.1 Native Apache Spark with Standalone Cluster Manager

This setup uses the native Spark installation with the standalone cluster manager and will be referred to as “Spark standalone” configuration. The input and output data is stored in the HDFS provided by the native Hadoop installation.

The deployment overview and runtime communication pattern is shown in figure 7.1 on page 44.

See figure 7.2 on page 45 for the HDFS payload and control data flows.

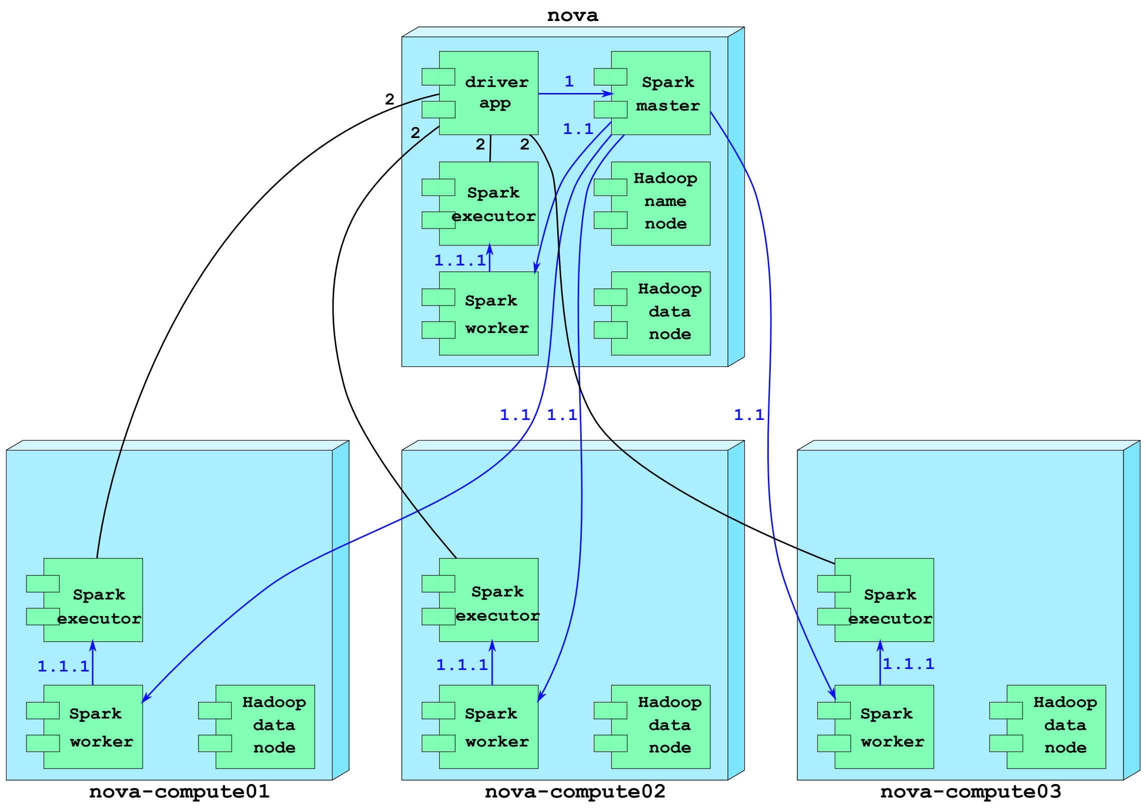

Figure 7.1: Communication Pattern for Native Apache Spark with Standalone Cluster Manager. 1. The driver program asks the Spark master for resources. 1.1 The Spark master contacts the Spark worker processes on the cluster nodes. 1.1.1 The Spark workers start the executor processes which then connect to the driver and regularly send heartbeat messages. 2. The driver program sends tasks to the executors and collects the result data.

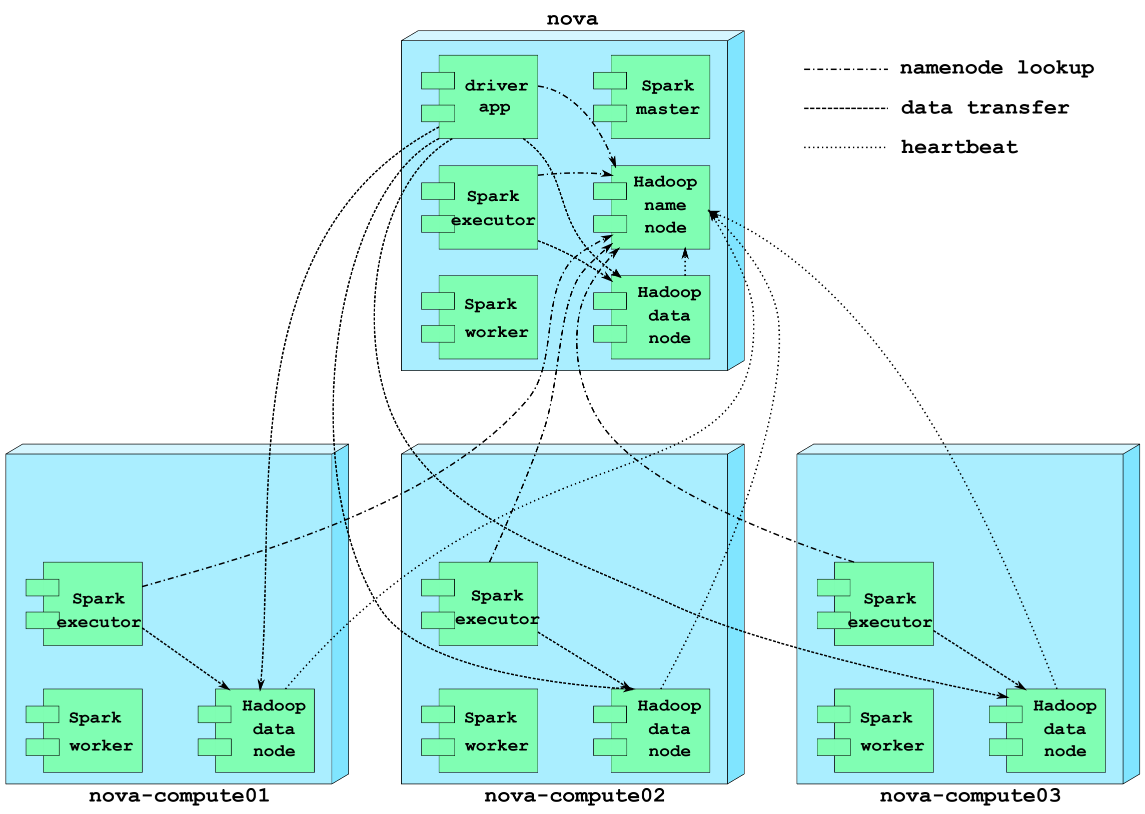

Figure 7.2: HDFS Access for Native Apache Spark with Standalone Cluster Manager. The Spark executor's (potential) access to Hadoop data nodes on other cluster nodes is omitted. Also not shown is the communication between Hadoop data nodes for data replication.

Motivation

The native Apache Spark and Hadoop installation is the simplest and most straight forward way to install the software.

This setup with statically defined cluster resources does not add any overhead and is expected to yield the best runtime performance for exclusive cluster usage. With the Spark master and the Spark workers already running, the job startup latency should also be minimal. Hence the benchmark results from this setup will define the baseline for performance comparisons.

Drawbacks

The standalone cluster manager has a simple first in - first out (FIFO) scheduler, that may not work as well as the other cluster managers supported by Spark (Mesos and YARN) when several jobs run in parallel.

The native installation can require some work for setup and upgrades to newer versions, especially when customizing is required or additional software dependencies need to be maintained on the machines. Running multiple versions simultaneously requires even more maintenance overhead, particularly when different versions of the same programs and libraries are needed. The use of a configuration management tool like Puppet [28] or Chef [20] can mitigate part of that maintenance overhead, but also requires substantial up-front time investment.

Running the Experiments

First, the distributed file system HDFS was started as described in section 6.2.1 on page 34 and the Spark cluster was started using the provided shell script `$SPARK_HOME/sbin/start-all.sh`. After the experiments were finished, the Spark cluster was stopped with the shell script `$SPARK_HOME/sbin/stop-all.sh`.

Next, the pi approximation benchmark was run as described in section 6.3.1 with the Spark master parameter set for the standalone cluster manager (`--master spark://nova:7077`).

Then, the Spark-bench configuration parameter for the Spark master in the primary configuration file was changed to `SPARK_MASTER=spark://${master}:7077`.

Both benchmarks (K-Means and Terasort) were run as described in section 6.3.4 on page 40.

Experience

Running the benchmarks was simple and straightforward and did not cause any problems.

7.1.2 Native Apache Spark with YARN

This setup uses the native Spark installation with the YARN resource manager from Hadoop. The input and output data is stored in the HDFS provided by the native Hadoop installation.

Motivation

Running Spark with YARN as resource manager can be useful if Spark is deployed on an existing Hadoop cluster, that is used to run map-reduce jobs. In that case YARN is already configured and available and can easily be used as cluster manager for Spark. The pull-model, where the client actively requests resources from the cluster manager, should work well with the desired data locality for Spark jobs when the data is stored in the HDFS on the same cluster nodes.

Drawbacks

Running Spark on YARN requires a binary distribution of Spark which is built with YARN support (e.g., a version with built-in Hadoop support) [13].

Running the Experiments

The experiments did not run successfully.

The primary problem was the configured memory limit of 2 GB per task for YARN which is exceeded by the effective memory requirement of the executors used for the benchmarks.

My colleague and second supervisor for this work was simultaneously using the `nova` cluster for his work on another project which required the YARN resource manager. He upgraded Hadoop to version 2.8.0 because of new features that he required.

The installed Spark version also contained the Hadoop libraries for version 2.7. This might have caused problems, which could have been solved with a newer version of Spark. However, this was not a good option since the benchmark experiments for all other configuration were already finished with the exact same Spark version (2.0.2).

To avoid interfering with the work of my colleague at that time, I decided not to change the YARN configuration. Due to time constraints, the work on this configuration was not resumed later.

7.1.3 Native Apache Spark with Mesos

This setup uses the native Spark installation with Mesos as cluster manager and is referred to as "Spark with Mesos" configuration. The input and output data is stored in the HDFS provided by the native Hadoop installation (the Hadoop nodes are not managed by Mesos).

See figure 7.3 on page 49 for the deployment diagram including the communication pattern.

Motivation

Using Mesos as cluster manager enables dynamic partitioning of the cluster resources between Spark and other frameworks and also between multiple instances of Spark.

Drawbacks

Running Spark with Mesos requires the Spark binary distribution to be available on all nodes (creating some overhead if it is extracted from the archive into the sandbox each time). Alternatively, Spark can also be installed in the same location on all Mesos worker nodes. As Spark was already installed in a shared directory, which is mounted in the same location on all nodes, the Mesos parameter `spark.mesos.executor.home` was used to point to the Spark home directory (the actual value of `$SPARK_HOME`).

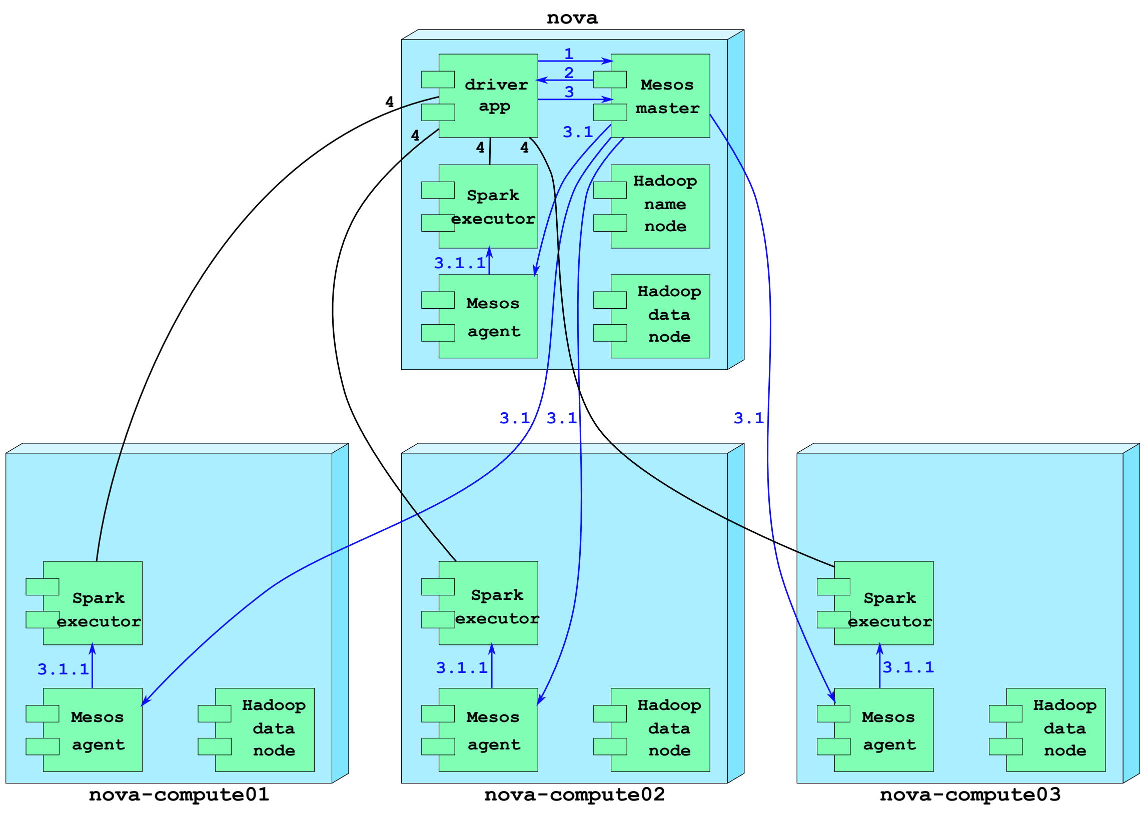

Figure 7.3: Communication Pattern for Native Apache Spark with Mesos Cluster Manager. 1. The driver program registers itself as framework with the Mesos master. 2. The Mesos master makes resource offers to the driver program. 3. The driver program asks the Mesos master to run Spark executors on the accepted resources. 3.1 The Mesos master contacts the Mesos agent processes on the cluster nodes. 3.1.1 The Mesos agents start the Spark executor processes which then connect to the driver and regularly send heartbeat messages. 4. The driver program sends tasks to the executors and collects the result data.

Running the Experiments

First, the distributed file system HDFS was started as described in section 6.2.1 on page 34.

Next, the Mesos cluster manager was started as described in section 6.2.5 on page 36.

Then, the pi approximation was run as described in section 6.3.1 with the Spark master parameter set for the Mesos resource manager (`--master mesos://nova:5050`).

The Spark-bench configuration parameter for the Spark master in the primary configuration file was also changed to `SPARK_MASTER=mesos://${master}:5050`. Both benchmarks (K-Means and Terasort) were run as described in section 6.3.4 on page 40.

Experience

Getting the tests to run required some extra work and research. With the initial Mesos setup, which included running the Mesos master and agents with the `mesos` user, any user without `root` privilege (e.g., `spark`) cannot submit tasks to the cluster manager. To overcome this, the Mesos master and agents were started as `root`. Then, all submitted tasks failed immediately. The reason - permission denied error - was found in the `stdout` and `stderr` stream contents logged in the sandbox directories of the tasks on the used worker nodes. The Mesos agents created the sandbox directories as `root` with exclusive access for the owner, hence the `spark` user, that was supposed to run the executor binary, could not access its sandbox directory. Using the `root` user to run the tasks solved that problem. However, regarding security this is not an optimal solution.

7.2 Containerized Apache Spark

This section describes the experiments performed with different containerized installations of Apache Spark using miscellaneous cluster managers.

The majority of the experiments uses host networking, where the containerized application uses the host's network stack, but there is one setup that uses a private Docker overlay network.

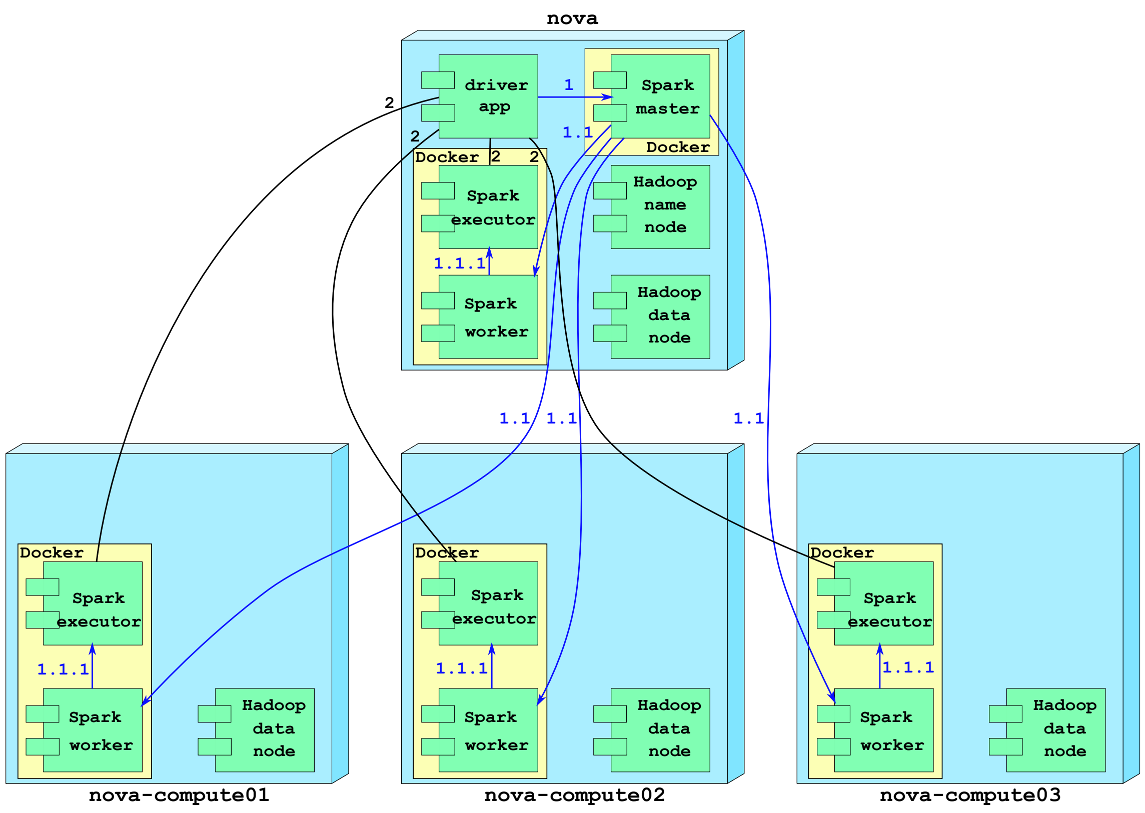

Figure 7.4: Communication Pattern for Containerized Apache Spark with Standalone Cluster Manager Using Host Networking. 1. The driver program asks the Spark master for resources. 1.1 The Spark master contacts the Spark worker processes on the cluster nodes. 1.1.1 The Spark workers start the executor processes which then connect to the driver and regularly send heartbeat messages. 2. The driver program sends tasks to the executors and collects the result data.

7.2.1 Containerized Apache Spark with Standalone Cluster Manager using Host Networking

In this setup Spark is running within manually managed Docker containers on each node using host networking, that means the container uses the host's network stack. This setup is referred to as "Spark standalone with Docker" configuration. The input and output data is stored in the HDFS provided by the native Hadoop installation.

See figure 7.4 for the deployment overview and the communication pattern.

Motivation

This setup offers the advantages of containerized software while simultaneously avoiding the overhead of a bridged network for the containers. Access to local network resources (also on the same host) is provided without further configuration. The runtime characteristics can be expected to be similar to the native installation.

Drawbacks

The application directly uses the host's network ports and the application's configuration may need to be changed to utilize other than the application standard ports. Without additional configuration, only one instance can be run at a time due to network port collisions. The containerization could possibly cause some noticeable performance degradation.

Running the Experiments

Refer to sections B.1 and B.2 (page 81) for the creation of the required Docker images.

First, the distributed file system HDFS was started as described in section 6.2.1 on page 34.

Next, the Spark master was started on the primary node as shown in listing 7.1.

Then, the Spark workers were started on each cluster node as shown in listing 7.2 on page 53.

Listing 7.1: Start Containerized Spark Master with Host Networking

```
docker run -it --rm \
  --name=sparkmaster \
  --network=host \
  -e SPARK_MASTER_HOST=10.0.0.1 \
  nova:5000/mck/spark:2.0.2 /bin/bash
# inside the container (ctrl-c to stop):
/usr/local/spark-2.0.2/sbin/start-master.sh
```

Listing 7.2: Start Containerized Spark Worker with Host Networking

```
docker run -it --rm \
  --name=sparkworker \
  --network=host \
  -e SPARK_MASTER_HOST=10.0.0.1 \
  nova:5000/mck/spark:2.0.2 /bin/bash
# inside the container (ctrl-c to stop):
/usr/local/spark-2.0.2/sbin/start-slave.sh nova:7077
```

The pi approximation benchmark was run as described in section 6.3.1 on page 38 with the Spark master parameter set for the standalone cluster manager (`--master spark://nova:7077`).

The Spark-bench configuration parameter for the Spark master in the primary configuration file was also changed to `SPARK_MASTER=spark://${master}:7077`. Both benchmarks (K-Means and Terasort) were run as described in section 6.3.4 on page 40.

Experience

Running the benchmarks was simple and straightforward. From the user's point of view, there is no difference between the native Spark installation and this setup. There was no noticeable performance degradation compared to the native installation.

7.2.2 Apache Spark with Standalone Cluster Manager in Docker Swarm

In this setup Docker is run in swarm mode on each cluster node with all nodes attached to a single Docker swarm. Apache Spark and Hadoop (providing the HDFS) run within Docker containers managed by Docker swarm with each container connected to the same overlay network.

See figure 7.5 on page 54 for the network topology.

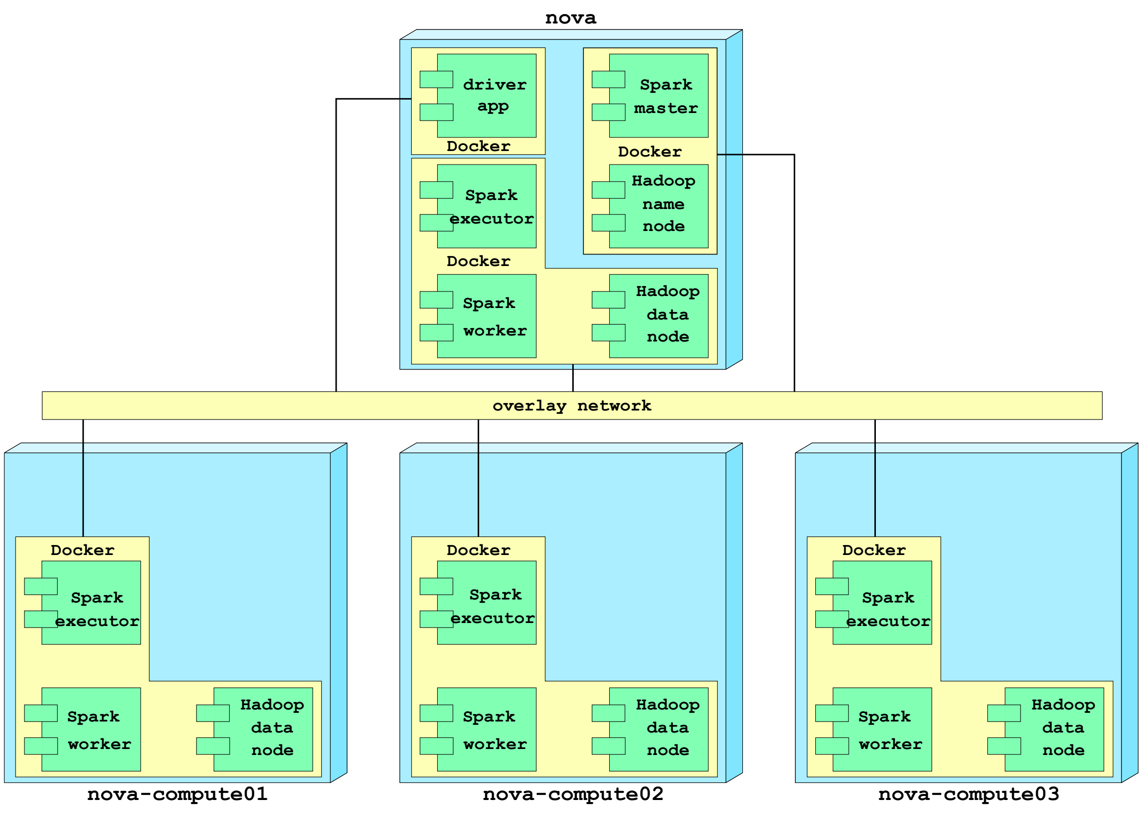

Figure 7.5: Network Topology for Containerized Apache Spark with Standalone Cluster Manager Using Docker Swarm Mode

Motivation

This setup offers the advantages of containerized software combined with simple cluster management provided by Docker swarm. The number of Spark worker instances can be scaled even while the application is running (if the application, like Spark, can adapt to node failures and newly started nodes).

It is possible to publish specific ports of the application, e.g., to submit new tasks or to access the Spark web UI. Alternatively, without publishing any ports, the Spark cluster can be used from other Docker containers connected to the same overlay network.

Drawbacks

For processes within a Docker container, that does not use host networking, it is difficult and cumbersome to enable access to network ports on the local host. For that reason the HDFS was also run within a Docker container without using external data storage, which is not suitable for long-term data storage. Docker volumes can be used to overcome that problem, but require additional configuration.

Running the Experiments

First, a Docker swarm was created on the primary cluster node and the other cluster nodes were then added to that swarm as described in the Docker documentation [24]. Additionally, the primary node was tagged with `node.role = manager`, to ensure that the Spark master will run on the primary node when started with suitable arguments (cf. listing 7.5 on page 57).

The singularities/spark image [47], which can be found on Docker Hub (`https://hub.docker.com/`), was used as it conveniently also contains Hadoop. To prevent multiple image downloads, the image was copied to the local private repository as shown in listing 7.3 on page 56.

Listing 7.3: Copy Singularities Image from Docker Hub to Private Registry

```
# pull image from Docker hub
docker pull singularities/spark:2.0
# tag with name for private registry
docker tag singularities/spark:2.0 nova:5000/spark:2.0
# push image to private registry
docker push nova:5000/spark:2.0
```

An overlay network, which is required to connect the containers running on different nodes, is created as shown in listing 7.4. The network must be created with the `--attachable` option to be accessible by Docker containers that are not managed by Docker swarm.

Listing 7.4: Create an Attachable Overlay Network

```
docker network create \
  --attachable \
  --driver overlay \
  --subnet 10.0.9.0/24 \
  nova-net
```

The single instance of the Spark master is created with the command shown in listing 7.5, and multiple instances of the Spark worker are created as indicated in listing 7.6 on page 57. A Docker version of at least 1.13.0 is required as the `--hostname` parameter was added with this version. That parameter is required to specify a hostname for a node, such that other nodes can resolve that name to the actual IP address within the overlay network.

Listing 7.5: Create Spark Master Service

```
docker service create \
  --name=sparkmaster \
  --hostname=sparkmaster \
  --network=nova-net \
  --publish 8080:8080 \
  --env HDFS_USER=hadoop \
  --constraint 'node.role == manager' \
  --replicas 1 \
  nova:5000/spark:2.0 master
```

Listing 7.6: Create Spark Worker Service

```
docker service create \
  --name sparkworker \
  --network nova-net \
  --mode replicated \
  --replicas 4 \
  --env HDFS_USER=hadoop \
  nova:5000/spark:2.0 worker sparkmaster
```

Experience

The setup basically worked with Docker version 1.13.0. However, only some of the published ports were actually accessible from the primary cluster node. Hence, it was impossible to access the Spark master and to submit tasks. To overcome this limitation, the Spark-Bench benchmark suite was packaged and run as Docker container (see section B.3 on page 83), which was attached to the overlay-network `nova-net`. Preliminary tests without recorded timing looked fine. Also scaling the number of Spark worker instances up and down using the tools for Docker swarm worked well.

Prior to running the measurements, Docker was updated to version 17.04.0-ce. After that update, the Spark cluster within the Docker containers did not work any more. The Spark workers, which were not residing on the primary node, could not connect to the Spark master. Although the hostname mapping can be specified explicitly using the `--host` parameter (e.g., `--host sparkmaster:10.0.9.2`) when creating the `sparkworker` service, the problem persisted. In effect, there was a single node cluster instead of a four node cluster.

For that reason, there are no performance measurements for this setup.

7.2.3 Containerized Apache Spark with Mesos

In this setup Apache Mesos is used as resource manager for the Spark worker instances running in Docker containers. The Spark binaries are provided as Docker image (see section B.4 on page 85). This setup is referred to as "Spark with Mesos and Docker" configuration.

The distributed filesystem HDFS is provided by the native Apache Hadoop installation, which is not managed by Mesos.

See figure 7.6 on page 59 for the deployment overview and communication patterns.

Motivation

This setup offers the advantages of containerized software combined with flexible cluster management provided by Mesos. Diverse frameworks or different versions of the same framework (e.g., Spark) with specific configurations and dependencies can simply be provided using Docker images and can be run simultaneously with dynamic cluster resource sharing.

Drawbacks

Mesos uses host networking for the Docker containers, at least in version 1.1.0, which was used for the experiments. If the managed software requires fixed network ports, this may require additional configuration to avoid port collisions. In most cases, like for Apache Spark, this in not an issue, as the executor processes only need to connect to the process that is running the SparkContext, which does not require any fixed ports on the worker nodes and simply uses ephemeral ports.

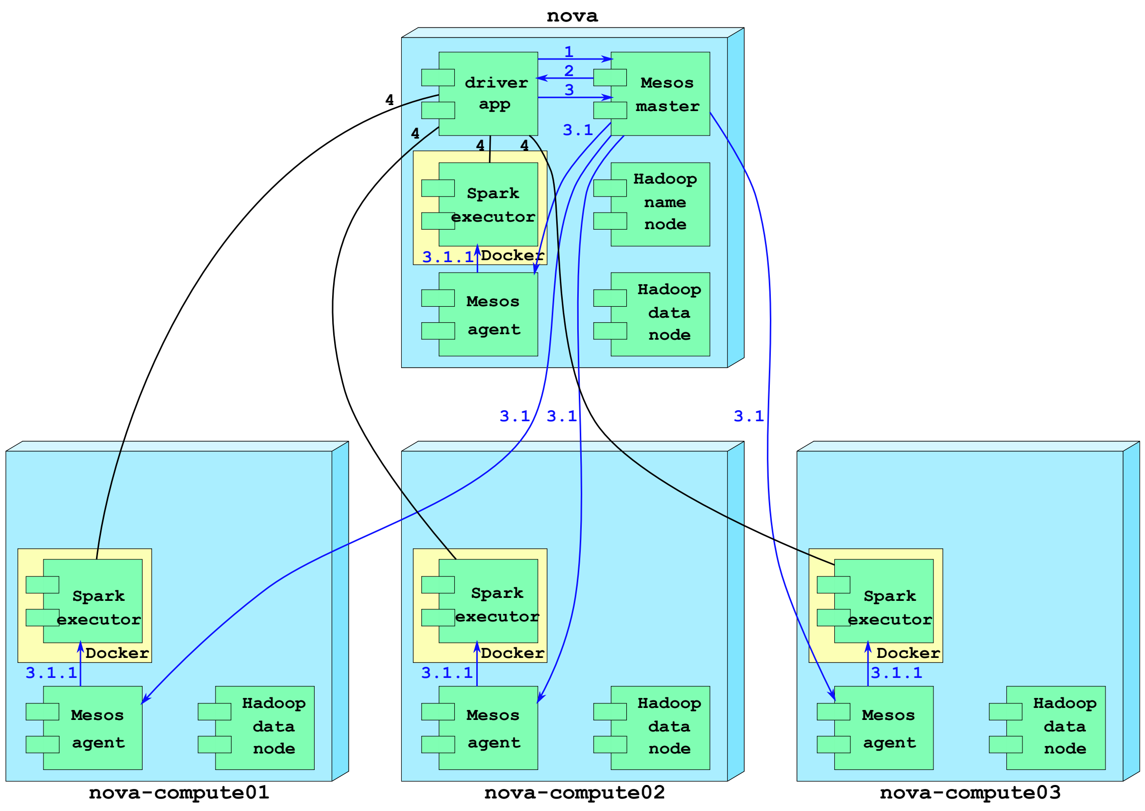

Figure 7.6: Deployment Overview and Communication Pattern for Containerized Apache Spark with Mesos. 1. The driver program registers itself with the Mesos master. 2. The Mesos master makes resource offers. 3. The driver program accepts all or a part of the offered resources. 3.1. The Mesos master instructs the Mesos agents to start Spark executors on the accepted resources. 3.1.1 The Mesos agents start Spark executors using the specified Docker image. 4. The Spark executors connect to the driver which then sends tasks to the executors and collects the result data.

Running the Experiments

Refer to section B.4 on page 85 for the creation of the required Spark-for-Mesos Docker image.

First, the Spark configuration was adapted to use the Docker image and the Spark installation directory within the Docker image according to listing 7.7. Note: The Docker image name must be specified completely with its version (e.g., `latest`).

Listing 7.7: Spark Configuration for Mesos with Docker: `spark-defaults.conf`

```
spark.mesos.executor.home              /opt/spark
spark.mesos.executor.docker.image ↩
   ↪ nova:5000/mck/spark-mesos-docker:latest
```

Next, the distributed file system HDFS was started as described in section 6.2.1 on page 34.

Then, the Mesos cluster manager was started as described in section 6.2.5 on page 36.

The Spark application can be run or submitted, respectively, from within a Docker container as well as from a compatible and properly configured native installation.

All benchmarks with timing were run using the native Spark installation.

As first test for running Spark within a Docker container, a container was started as shown in listing 7.8 running the Spark shell, which was working as expected.

Note: It is necessary to use host network mode or additional network configuration for the Docker container to access the Mesos master on the local machine.

Listing 7.8: Start Spark Shell in Docker Container with Spark-for-Mesos Docker Image

```
docker run -it --rm \
  --network host \
  -e SPARK_MASTER="mesos://10.0.0.1:5050" \
  -e SPARK_IMAGE="nova:5000/mck/spark-mesos-docker:latest" \
  nova:5000/mck/spark-mesos-docker \
  /opt/spark/bin/spark-shell
```

After setting the Spark master parameter for Mesos as cluster manager (`--master` `mesos://nova:5050`), the pi approximation was started as described in section 6.3.1 on page 38.

The Spark-bench configuration parameter for the Spark master in the primary configuration file was also changed to `SPARK_MASTER=mesos://${master}:5050`. Both benchmarks (K-Means and Terasort) were run as described in section 6.3.4 on page 40.

Experience

Initially, the executors on the worker nodes could not be started and were killed prematurely by Mesos. The saved output from the standard error stream `stderr` on the worker nodes revealed that the download and extraction of the required Docker image took too long. The problem was fixed by manually pulling the images on each worker node using the command "`docker pull nova:5000/mck/spark-mesos-docker:latest`". Alternatively, the executor registration timeout value could be increased.

Then, the pi approximation ran without any problems.

However, the K-Means clustering benchmark exhibited problems with the medium and large input data size. The measured time showed large variations between subsequent runs and the log from the benchmark program contained warnings, reporting lost tasks due to executors being stopped because of executor heartbeat timeouts. Although the affected executors were restarted subsequently, there was a significant delay due to the default heartbeat timeout of 180 s after which the executor is restarted and the lost tasks are rescheduled (on the same node or on another one). The original problem that caused the heartbeat timeouts could not be determined. It was verified, that the Docker container was not stopped by the Docker daemon due to an internal error or exceeding its memory limit. Another warning in the log stating a timeout waiting for an executor to terminate within 10 s might indicate a problem in the executor process itself. However, the observed warning could also be caused by a network operation, that times out after an extended period, making the process seemingly unresponsive. See section 8.2 on page 65 for a detailed discussion of the problems and the probable cause.

The Terasort benchmark also showed problems similar to the K-Means benchmark. While some stages of the job (e.g., stage 1) ran fine with high CPU utilization in approximately the same time as with the “Spark with Mesos” configuration, other stages (e.g., stage 2) took much longer with very low CPU utilization.

Chapter 8

Benchmark Results

In this chapter the results of the runtime measurements of the conducted experiments are discussed in detail.

The complete aggregated timing data can be found in section C on page 88ff.

All the figures in this section use the minimum time value for a data series corresponding to a single configuration and problem size. For the cases, where the figures would significantly change if another value (e.g., the mean value) was used, this is discussed in detail in the text.

8.1 Pi Approximation Benchmark Results

See figure 8.1 on page 64 for the absolute wall-clock times for each configuration and problem size, which is specified as the number of partitions. Figure 8.2 on page 65 shows the times relative to the "Spark standalone" configuration and the same number of partitions.

Surprisingly, the fastest setup for the pi approximation was the configuration with Spark using the standalone cluster manager running in Docker, but this configuration was just slightly faster (up to 0.5 s) than the native Spark installation with the standalone cluster manager. Access to the locally stored Docker image being faster than the file access in the shared file system, where the native Spark installation and the JVM were located, seems the most probable explanation for this observation.

The configurations using Mesos as cluster manager show an overhead, compared

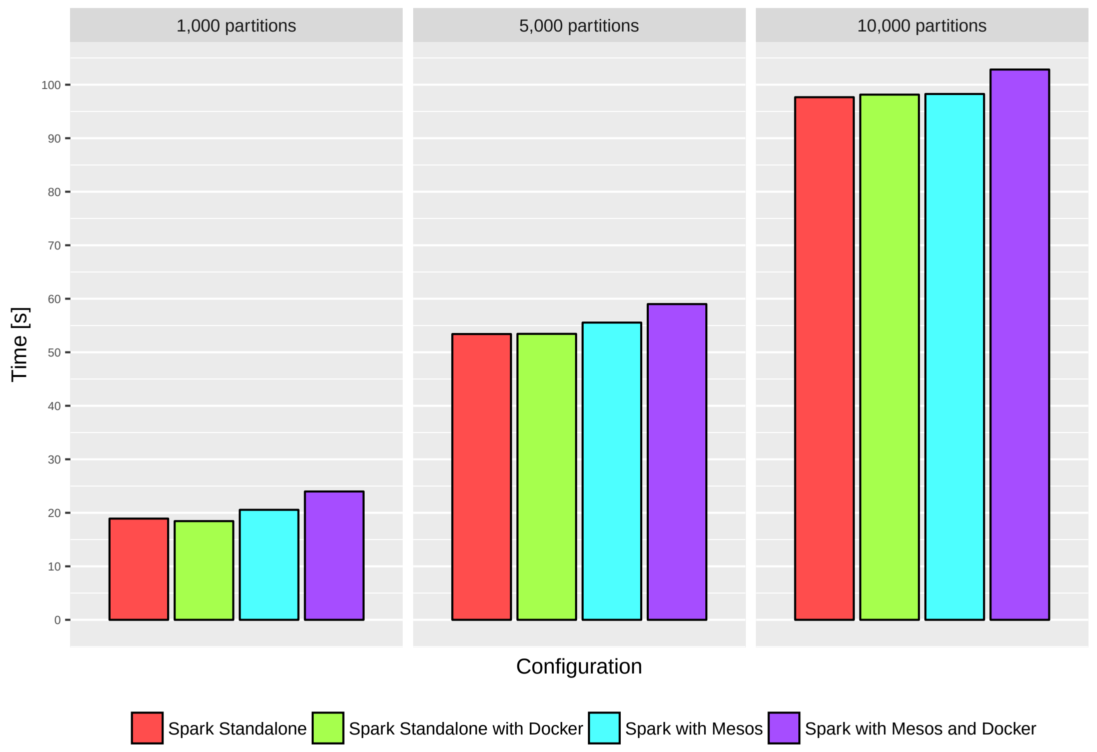

Figure 8.1: Absolute Time Comparison for the Pi Approximation Benchmark

to the standalone cluster manager, which is roughly independent of the number of partitions. The containerized configuration using Mesos and Docker exhibits slightly higher overheads. The start of the Spark executor processes on demand works similar in all configurations. Hence, it's unclear what causes the overhead in the configurations using Mesos. However, the “Spark standalone with Docker” configuration, that uses Docker in a similar way, also using host networking, performs roughly as well as the “Spark standalone” configuration.

Although this approximately constant time overhead is noticeable for short overall run times, the relative overhead diminishes with longer run time and already drops below 5 % for an overall runtime of approximately 100 s. For long running tasks, this overhead may be negligible.

Aside from the different startup overheads, the concrete choice of the cluster manager and using containerization or not seems to be insignificant for a workload similar to the pi approximation, which is CPU intensive with little inter-node communication.

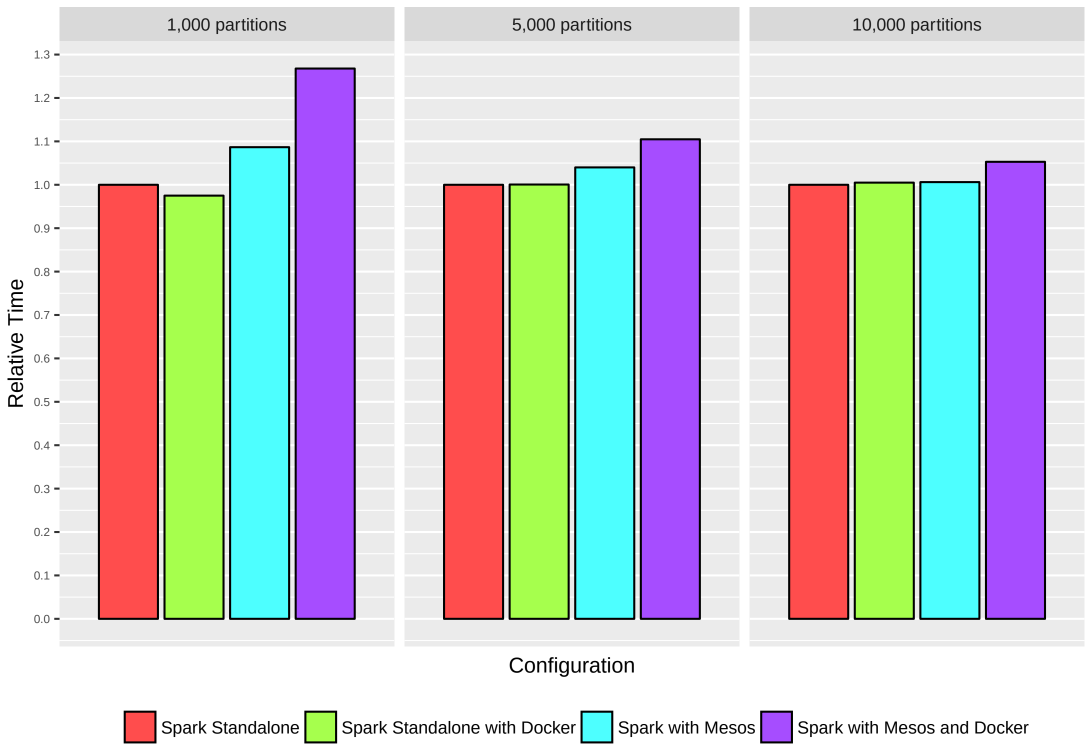

Figure 8.2: Relative Time Comparison for the Pi Approximation Benchmark

8.2 K-Means Clustering Benchmark Results

See figure 8.3 on page 67 for the absolute wall-clock times for each configuration and problem size, which is specified as the number of points. Figure 8.4 on page 67 shows the times relative to the “Spark standalone” configuration and the same number of points.

For the small and medium sized problem (1 million and 5 million points) the fastest configuration for the k-means clustering was “Spark standalone”, while “Spark standalone with Docker” and “Spark with Mesos” exhibited a similar overhead being approximately 4 % to 5 % slower.

However, for the large sized problem (10 million points) the configurations “Spark standalone with Docker” and “Spark with Mesos” both performed significantly better than the “Spark standalone” configuration, showing speedups of 16 % and 19 %, respectively.

On the other hand, the “Spark with Mesos and Docker” configuration always performed significantly worse, with an overhead of 45 % to 55 %, relative to the “Spark standalone" configuration.

For the large problem size, the standard deviation of the time measurements was 136 s (with minimum 191.2 s and maximum 694.6 s) for the "Spark with Mesos and Docker" configuration, but only approximately 22 s for the other configurations. The extremely high maximum value was caused by Spark executors being killed and restarted on behalf of the `SparkContext` due to communication timeouts. It is unclear why the Spark executors did not finish their work. The CPU utilization of the executor dropped for a longer period on the node where an executor was subsequently restarted, which indicates that it was not the CPU workload that took too long. That also rules out extensive garbage collection in the JVM due to insufficient memory limits. The problem also occurred on different worker nodes and it could be verified that the Docker containers of the affected Spark executors were not stopped by the Docker daemon due to an internal error or exceeding its memory limit.

A possible explanation for the problem, that was not noticed at the time of the experiments, is the Linux kernel version mismatch between the host machine running kernel version 3.10.0-514.16.1.el7.x86_64 and the image, which is based on Ubuntu 17.04 using a 4.10 kernel. Although different kernel versions for the host and the container generally are fine, that may have caused network performance problems in this case. The problem seems to be related to an open issue in Docker with host kernel version 3.10 (cf. [46]).

For this mixed workload, which involves both high CPU utilization and network communication, the "Spark with Mesos" and "Spark standalone with Docker" configurations are the best choice for the large problem size.

8.3 Terasort Benchmark Results

See figure 8.5 on page 68 for the absolute wall-clock times for each configuration and problem size, specified as the number of points (or elements). Figure 8.6 on page 69 shows the times relative to the "Spark standalone" configuration and the same number of points.

Regardless of the problem size, the fastest configurations are "Spark standalone with Docker" and "Spark with Mesos" with roughly up to 20 % (only 10 % when

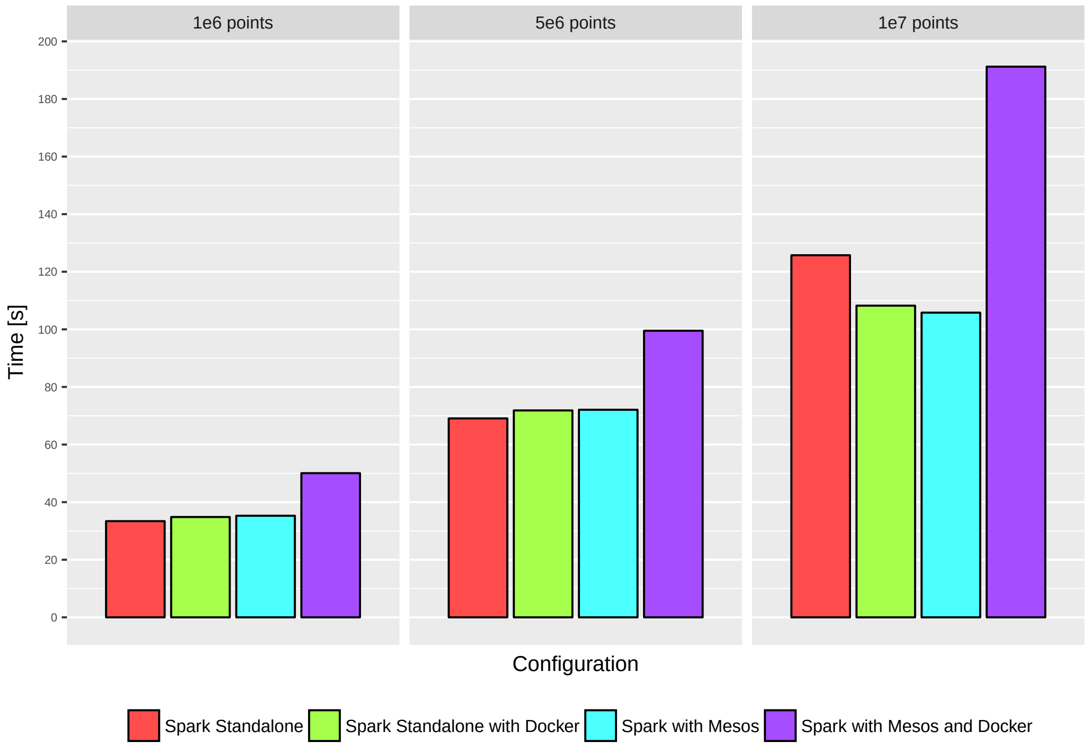

Figure 8.3: Absolute Time Comparison for the K-Means Clustering Benchmark

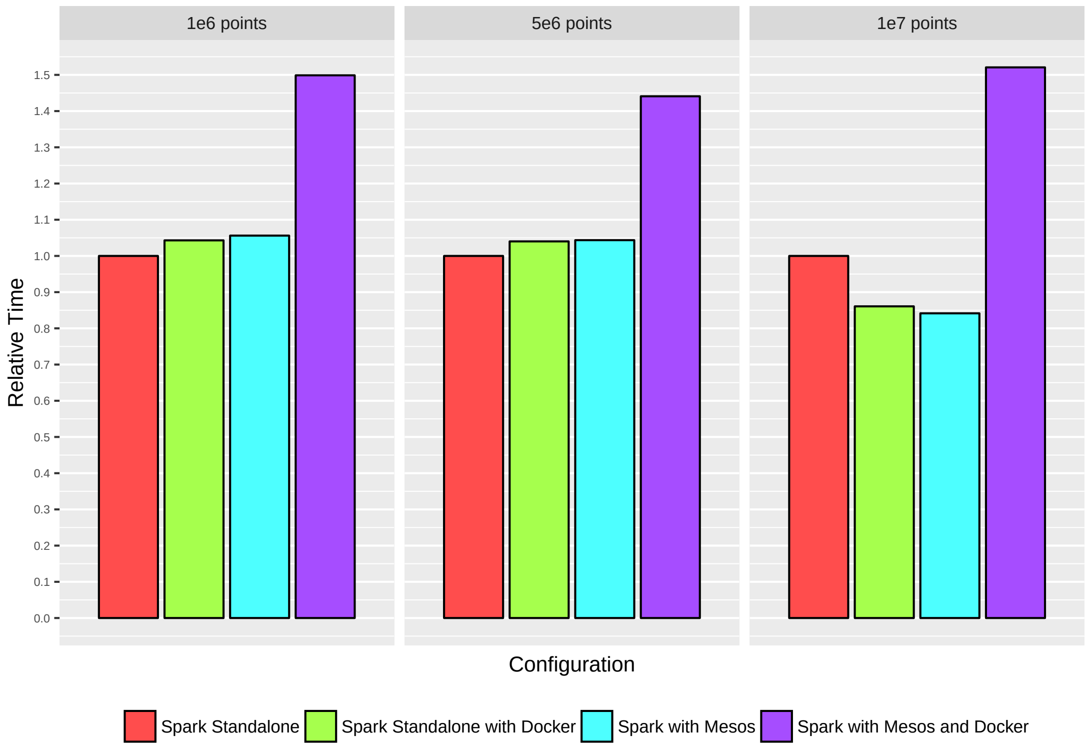

Figure 8.4: Relative Time Comparison for the K-Means Clustering Benchmark

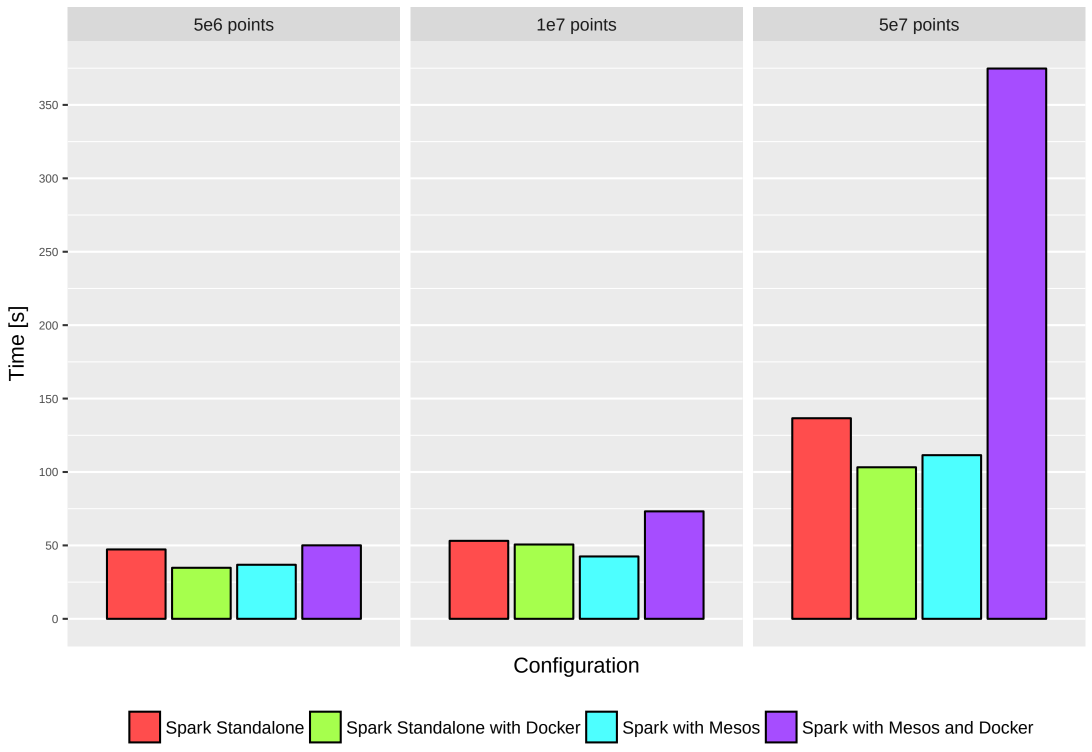

Figure 8.5: Absolute Time Comparison for the Terasort Benchmark

considering the average run time) speedup relative to the “Spark standalone” configuration.

The “Spark with Mesos and Docker” configuration was always slower with the overhead increasing with the problem size from 6 % to 274 %.

The problem also showed in the run times of the different stages of the computation, e.g., stage 1 took approximately the same time and high CPU utilization for all configurations. However, stage 2 took much longer with the “Spark with Mesos and Docker” configuration while the CPU utilization dropped from a high level (averaged over all 24 CPU cores per node) to only 2 CPU cores actually being used. No executor processes were stopped in this case. This observation suggests a problem in the inter-node (and possibly also inter-process) communication of the executors and the driver program hosting the `SparkContext`. The reason for this problem could not be determined. However, as already stated in section 8.2, the issue was probably caused by a problem related to the Linux kernel version mismatch between the host machine and the Docker image (cf. [46]).

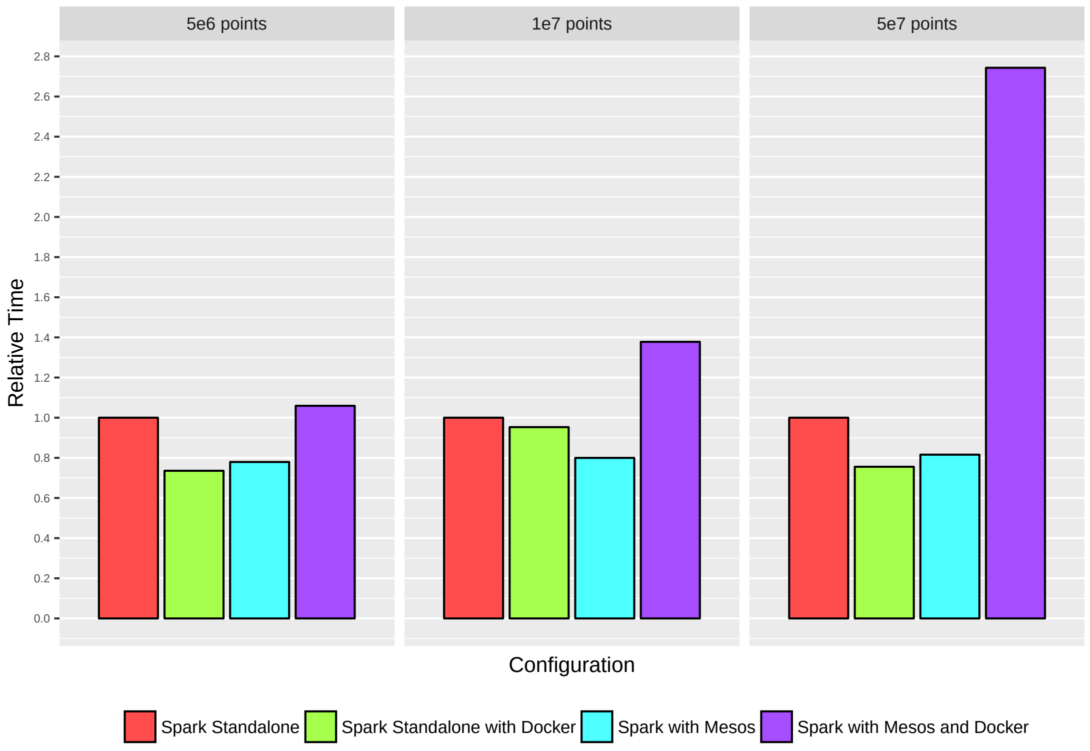

Figure 8.6: Relative Time Comparison for the Terasort Benchmark

Chapter 9

Future Development

Based on the good performance seen in the experiments using Docker, further steps towards a more complete containerized environment seem to be promising.

In the following section, two aspects relevant for data-intensive HPC applications are presented as possible areas of future work.

9.1 Sharing Cluster Resources Between Diverse Scientific Workloads

It is desirable to be able to also have distributed scientific applications based on other programming models participate in a cluster with resource sharing.

An important model for HPC is MPI, and while the Mesos documentation states running an MPI application as use case example [12], it's currently hard to find examples or open source projects with recent development activity that integrate MPI applications with Mesos as cluster manager. Nevertheless, efforts should be made to run an MPI application on a Mesos cluster as the sole application using the cluster and also with dynamic cluster sharing, even if the number of MPI processes may need to remain fixed for the time that the MPI application runs.

9.2 Persistent Storage

For the experiments described in this work, the long-term persistent storage (in the Hadoop distributed file system) was set up natively on the cluster machines. One reason for the native installation was the complex setup for consistent local storage when using Docker in swarm mode or with Apache Mesos as resource manager.

The primary commercial contributor to the development of Apache Mesos is the company Mesosphere [26], which also builds the Datacenter Operating System (DC/OS) [25] that utilizes Mesos as resource management framework, together with the container orchestration platform Marathon. Marathon [35] is a Mesos meta framework for container orchestration, that can start other frameworks such as Apache Storm [14] and ensures they survive machine failures.

DC/OS is Open Source Software that is available as community-supported free product and also as an enterprise product with commercial support by Mesosphere [27] and can be installed on a single machine, in the local cluster or data center or in the cloud using different cloud providers.

A product like DC/OS, even in the free community-supported version, provides useful tools to set up frameworks like Apache Hadoop, Apache Spark as well as many other distributed software products that can be run within a container and supports dynamic assignment of cluster resources to enable effective resource utilization. Such a product can simplify the scheduling and dynamic cluster partitioning for multiple cluster guest frameworks with different resource requirements and load profiles, which also scientific applications may benefit from.

Using local storage features of Mesos, Marathon also provides different options for storage of persistent data for stateful applications. As an example a MySQL database instance can use a local persistent volume and Marathon ensures that the corresponding container is pinned to that node and will reuse the local storage when the container is restarted, hence preventing data loss. These mechanisms allow a simplified setup of distributed data stores (e.g., Apache Cassandra [9]) and file systems (like the Hadoop distributed file system) for data-intensive applications. Other storage options include the use of remote block storage devices that prevent data loss in case of a permanent node failure.

An alternative option to explore is OpenShift by Red Hat® [30] which includes container orchestration using the open source software Kubernetes [4] which is based on Google's experience running massive data centers. OpenShift is the commercial product based on the open source community project OpenShift Origin [29].

Chapter 10

Conclusion and Lessons Learned

10.1 Conclusion

The experiments show that operating-system-level virtualization has reached a level of efficiency that is very close to running an application natively on a machine.

For Apache Spark, the distributed big data application framework used in the experiments, the natively running application and the version running in Docker containers with host networking deliver nearly identical performance. For some workloads and problem sizes the containerized version was even running up to 20 % faster.

Also Apache Spark running natively with Apache Mesos as resource manager performed very similar to the natively running version with the standalone cluster manager. The Mesos containerizer, which basically uses the same Linux kernel features (e.g., namespaces and control groups) as the Docker daemon for the task isolation shows a similarly small runtime overhead.

Apache Spark using Apache Mesos as resource manager using the Docker containerizer did not work satisfyingly as it showed poor performance or even severe problems, like workers being killed and restarted due to communication timeouts probably caused by network related problems. The cause of the problems could not be verified without a doubt due to time constraints, but the most compelling explanation seems to be a Docker issue [46] in conjunction with a specific Linux kernel version mismatch, which was unintentionally introduced by building the respective Docker image based on Ubuntu 17.04 using a kernel version 4.10. The host machines are running with an older kernel version (3.10.0-514.16.1.el7.x86_64) and the problem seems closely related to an open issue in Docker with host kernel version 3.10 (cf. [46]).

Taking the good performance of Spark running in Docker containers into account, provided that a Linux kernel version compatible to the host is used in the image, the performance of Spark with Mesos and Docker can be expected to be equal to Spark running natively and using Apache Mesos as resource manager. However, this expectation is yet to be verified.

Regarding the user experience there is no significant difference between running in Docker containers and the native installation with the standalone cluster manager or with Mesos as resource manager.

From an administrative point of view, deploying a pre-tested, complete software stack within a single container image without any side effects on other software running on the host, can tremendously speed up the setup of new software.

While for our experiments only the moderately restrictive default security options of Docker were used, for production systems it is advisable to apply more restrictive policies, e.g., by using custom AppArmor [42] profiles, as any user defined image should be treated like untrusted software.

10.2 Lessons Learned

10.2.1 Docker Releases

The development pace for Docker is quite fast, releasing stable versions with new features every three months. However, sometimes working things break with new releases, as was the case with private overlay networks. While the Spark cluster with the standalone cluster manager was working well in Docker swarm mode with Docker version 1.13.0, the worker processes residing on another host could not connect to the Spark master any more with Docker version 17.04.0-ce.

10.2.2 Docker and Linux Kernel Versions

Although it is generally not required to have identical Linux kernel versions on the host machine and inside the Docker images, it is preferable to use a recent kernel on the host to minimize the chances of version related problems, which may emerge as subtle or even severe effects. If it is not possible to upgrade the host machine to use a more recent kernel, an alternative solution may be to base the Docker image on a system image with the same or a similar Linux kernel version.

10.2.3 Security

While security regarding protecting the cluster from threats originating from user-supplied software and images was not within the focus of the experiments, this is generally an important concern.

The problems we faced trying to run the Mesos agents as non-`root` user (cf. section 6.2.5 on page 36) showed that we need to expect further configuration effort for the setup of a sufficiently secure environment.

For running Docker images it may be advisable to create a suitable custom default AppArmor [42] profile or even specific profiles for each containerized application.

To provide further protection against privilege escalation from Docker containers internally running as `root` user, user id remapping to an unprivileged or even non-existent user on the host system should be considered. [24]

Appendices

A Summary of Used Technology and Tools

This section gives a brief summary of the technology and tools used for the experiments.

Apache Spark [13], [48] as generic framework to build data-intensive applications is used to run several benchmark applications with different setup configurations. The Spark installation also contains the simplest benchmark used in the experiments (the pi approximation) as example for creating Spark applications.

From Apache Hadoop [10] the Hadoop Distributed File System (HDFS) was used to store the benchmark test data, and YARN was utilized as resource manager for one of the setups. The Hadoop MapReduce framework itself was not used in any way.

The K-Means and Terasort benchmarks to assess the runtime performance of the different setups were provided by the Spark-Bench [5], [37] benchmark suite, which is available as source code (written in Java and Scala). The project was slightly adapted to use a more recent Spark version (2.0.2) and built with the Apache Maven [11] build tool.

The common runtime for all Java virtual machine based programs (Hadoop, Spark, Spark-Bench, Maven) is the Java virtual machine contained in the Oracle Java Development Kit [8].

Docker [24] was used to build the Docker images and to run Docker containers from these images. The private Docker registry was installed to store the private Docker images and make them accessible to all cluster nodes.

Apache Mesos [12], [18] served as cluster manager for some of the setup variants.

Summary of applications, tools and frameworks:

- Apache Hadoop and Apache Hadoop YARN [10]
- Apache Maven [11]
- Apache Mesos [12], [18]
- Apache Spark [13], [48]
- Docker and Docker registry [24]
- Oracle Java SE Development Kit 8 [8]
- Spark-Bench benchmark suite [5], [37]

B Building the Docker Images

B.1 Building the Base Docker Image

The base Docker image is based on the official CentOS image and contains the Oracle Java SDK. The CentOS image version matches the version running on the host machine. That Docker image was built with the command "`docker build -t nova:5000/mck/base .`" from the directory containing the Dockerfile shown in listing B.1 on page 82.

B.2 Building the Spark Docker Image

The Spark image is based on the base Docker image and additionally contains Apache Spark.

That Docker image was built with the command "`docker build -t nova:5000/mck/spark:2.0.2 .`" from the directory containing the Dockerfile shown in listing B.2 on page 83. The creation of that Docker image was inspired by the singularities/spark image [47] which can be found on Docker Hub (`https://hub.docker.com`).

Listing B.1: Dockerfile for the Base Docker Image with Java SDK

```
FROM centos:7.3.1611

RUN yum install -y curl nc; yum update -y;  yum clean all

ENV JDK_VERSION 8u121
ENV JDK_BUILD_VERSION b13
RUN \
   curl -LO \
   "http://download.oracle.com/otn-pub/java/jdk/\
$JDK_VERSION-$JDK_BUILD_VERSION/\
e9e7ea248e2c4826b92b3f075a80e441/\
jdk-$JDK_VERSION-linux-x64.rpm" \
   -H 'Cookie: oraclelicense=accept-securebackup-cookie' && \
   rpm -i jdk-$JDK_VERSION-linux-x64.rpm; \
   rm -f jdk-$JDK_VERSION-linux-x64.rpm; \
   yum clean all

ENV JAVA_HOME /usr/java/default
```

Listing B.2: Dockerfile for Apache Spark Based on the Base Docker Image

```
FROM nova:5000/mck/base

ENV SPARK_VERSION=2.0.2
ENV SPARK_HOME=/usr/local/spark-$SPARK_VERSION

# Install Apache Spark
RUN mkdir -p "${SPARK_HOME}" \
  && export ARCHIVE=spark-$SPARK_VERSION-bin-hadoop2.7.tgz \
  && export \
      DL_PATH=apache/spark/spark-$SPARK_VERSION/$ARCHIVE \
  && curl -sSL https://mirrors.ocf.berkeley.edu/$DL_PATH | \
      tar -xz -C $SPARK_HOME --strip-components 1 \
  && rm -rf $ARCHIVE

ENV PATH=$PATH:$SPARK_HOME/bin

# Ports
EXPOSE 7077 8080 8081

# Fix environment for all users
RUN echo 'export SPARK_HOME=$SPARK_HOME' >> /etc/bashrc \
  && echo 'export PATH=$PATH:$SPARK_HOME/bin'>> /etc/bashrc
```

B.3 Building the Spark-Bench Docker Image

The Spark-Bench Docker image is based on the Spark Docker image and adds all files from the previously built Spark-Bench benchmark suite (see chapter 6.3.3 on page 40). Some configuration files need to be adapted, which is done when the image is created.

The Docker image was built with the command “`docker build -t nova:5000 /spark-bench .`” from the directory containing the Dockerfile shown in listing B.3 on page 84.

Listing B.3: Dockerfile for the Spark-Bench Docker Image

```
# based on the Spark Docker image
FROM nova:5000/mck/spark:2.0.2

# install the bc utility
RUN apt-get update && apt-get install -y bc vim \
  && apt-get clean

# configure Hadoop file system access
RUN echo '<configuration><property>\
<name>fs.defaultFS</name>\
<value>hdfs://sparkmaster:8020</value>\
</property></configuration>' \
> $HADOOP_CONF_DIR/core-site.xml

# create directory and set access rights
ENV SPARKBENCH=/spark-bench
RUN mkdir ${SPARKBENCH} && chmod 777 ${SPARKBENCH}

# create hadoop user
RUN \
 adduser --no-create-home --disabled-login \
 --gecos "" --uid 1000 hadoop

# copy spark-bench files and modify configuration
COPY . ${SPARKBENCH}/
RUN \
 sed 's/nova/sparkmaster/ ; s/:9000/:8020/' \
 -i ${SPARKBENCH}/conf/env.sh && \
 sed 's/ssh ${master} "date␣+%F-%T"/date +%F-%T/' \
 -i ${SPARKBENCH}/bin/funcs.sh
RUN chown -R hadoop ${SPARKBENCH}/*

USER hadoop
WORKDIR ${SPARKBENCH}
```

B.4 Building the Spark-for-Mesos Docker Image

The Docker image containing Apache Spark for use with Apache Mesos was inspired by the work of Bernardo Gomez Palacio showing how to build a single node Spark cluster with Mesos [41].

This image is based on a current Ubuntu Docker image. The required Mesos libraries and their dependencies (i.e., `libcrypto.so.10`, `libcurl.so.4`, `libmesos.so`, `libsasl2.so.3`, `libssh2.so.1`, `libssl.so.10`, `libsvn_delta-1.so.0`, and `libsvn_subr-1.so.0`) were copied from the local Mesos installation and the `/usr/lib64` directory, due to the fact that no Mesos build is available in any public repository.

The Docker image was built with the command "`docker build -t nova:5000/mck /spark-mesos-docker .`" from the directory containing the Dockerfile shown in listing B.4.

The Spark configuration files `spark-defaults.conf` and `spark-env.sh` are shown in listings B.5 and B.6 on page 87.

Listing B.4: Dockerfile for the Spark-for-Mesos Docker Image

```
FROM ubuntu

# install utilities
RUN apt-get update && \
 apt-get install -y curl software-properties-common && \
 apt-get clean

# install latest Oracle Java 8 JDK and clean up
RUN \
 echo oracle-java8-installer \
  shared/accepted-oracle-license-v1-1\
  select true | debconf-set-selections && \
 add-apt-repository -y ppa:webupd8team/java && \
 apt-get update && \
 apt-get install -y oracle-java8-installer && \
 rm -rf /var/lib/apt/lists/* && \
 rm -rf /var/cache/oracle-jdk8-installer
```

```

# Define commonly used JAVA_HOME variable
ENV JAVA_HOME /usr/lib/jvm/java-8-oracle

# get Apache Spark 2.0.2
RUN \
 curl http://d3kbcqa49mib13.cloudfront.net/\
  spark-2.0.2-bin-hadoop2.7.tgz | tar -xzC /opt && \
 mv /opt/spark* /opt/spark

# copy local spark config and scripts
COPY spark-conf/* /opt/spark/conf/
COPY scripts /scripts

# add required mesos library and dependencies
COPY lib/* /usr/lib/
RUN chmod +x /scripts/*

ENV SPARK_HOME /opt/spark

# install other Mesos dependencies
RUN apt-get update && \
 apt-get install -y build-essential python-dev \
  python-virtualenv libcurl4-nss-dev libsasl2-dev \
  libsasl2-modules maven libapr1-dev libsvn-dev \
  zlib1g-dev &&
 apt-get clean

# set entrypoint to the start script
ENTRYPOINT ["/scripts/run.sh"]
```

Listing B.5: Spark Configuration for the Spark-for-Mesos Docker Image: `spark-defaults.conf`

```
spark.master                              SPARK_MASTER
spark.mesos.mesosExecutor.cores           MESOS_EXECUTOR_CORE
spark.mesos.executor.docker.image         SPARK_IMAGE
spark.mesos.executor.home                 /opt/spark
spark.driver.host                         CURRENT_IP
spark.executor.extraClassPath             /opt/spark/custom/lib/*
spark.driver.extraClassPath               /opt/spark/custom/lib/*
```

Listing B.6: Spark Configuration for the Spark-for-Mesos Docker Image: `spark-env.sh`

```
#!/usr/bin/env bash
export MESOS_NATIVE_JAVA_LIBRARY=\
${MESOS_NATIVE_JAVA_LIBRARY:-/usr/lib/libmesos.so}
export SPARK_LOCAL_IP=${SPARK_LOCAL_IP:-"127.0.0.1"}
export SPARK_PUBLIC_DNS=${SPARK_PUBLIC_DNS:-"127.0.0.1"}
```

C Timing Measurement Data

This section contains the aggregated timing data from the experiments. Each scenario is identified by the configuration used to run Apache Spark (Spark standalone cluster manager, Spark standalone cluster manager with Docker, Spark with Mesos, and Spark with Mesos and Docker) and the respective problem size. The data contains the aggregated timing data in seconds given as minimum, average and maximum wall-clock time and the standard deviation (SD). The number of runs is given in the column denoted with **count**.

Table 1 shows the pi approximation timing data for the different configurations and problem sizes given as the number of partitions.

Table 2 on page 89 presents the K-Means clustering timing data for the different configurations and input data sizes given as the number of points.

Table 3 on page 89 presents the Terasort timing data for the different configurations and input data sizes given as the number of elements.

Spark configuration	partitions	min time [s]	avg time [s]	max time [s]	SD time [s]	count
Standalone	1,000	18.9	19.5	20.1	0.4	10
Standalone with Docker	1,000	18.4	19.1	19.7	0.4	10
Mesos	1,000	20.6	20.9	21.3	0.2	10
Mesos and Docker	1,000	24.0	24.5	24.8	0.3	10
Standalone	5,000	53.4	55.3	57.2	1.3	10
Standalone with Docker	5,000	53.4	55.4	57.8	1.5	10
Mesos	5,000	55.5	57.3	58.2	0.8	10
Mesos and Docker	5,000	59.0	60.1	62.0	0.9	10
Standalone	10,000	97.7	101.1	103.5	2.1	10
Standalone with Docker	10,000	98.1	100.9	103.8	2.0	10
Mesos	10,000	98.3	102.5	107.0	2.4	10
Mesos and Docker	10,000	102.8	106.0	108.8	1.8	10

Table 1: Pi Approximation Benchmark Timing Data

Spark configuration	points	min time [s]	avg time [s]	max time [s]	SD time [s]	count
Standalone	1×10^6	33.4	34.2	37.2	1.1	10
Standalone with Docker	1×10^6	34.8	36.4	39.2	1.3	10
Mesos	1×10^6	35.3	37.7	39.4	1.6	10
Mesos and Docker	1×10^6	50.1	51.7	53.3	1.1	10
Standalone	5×10^6	69.1	105.4	187.7	43.2	10
Standalone with Docker	5×10^6	71.8	83.1	107.7	12.0	10
Mesos	5×10^6	72.1	80.6	93.1	6.9	10
Mesos and Docker	5×10^6	99.5	107.8	136.4	10.9	10
Standalone	1×10^7	125.7	157.5	190.2	22.8	12
Standalone with Docker	1×10^7	108.2	135.0	178.4	19.0	10
Mesos	1×10^7	105.8	133.8	176.8	23.0	10
Mesos and Docker	1×10^7	191.2	282.6	694.6	136.4	13

Table 2: K-Means Clustering Benchmark Timing Data

Spark configuration	elements	min time [s]	avg time [s]	max time [s]	SD time [s]	count
Standalone	5×10^6	47.2	49.0	51.6	1.3	10
Standalone with Docker	5×10^6	34.8	45.9	53.8	7.3	10
Mesos	5×10^6	36.8	44.2	53.4	6.1	10
Mesos and Docker	5×10^6	50.0	54.8	63.1	5.3	10
Standalone	1×10^7	53.1	59.1	62.8	3.0	10
Standalone with Docker	1×10^7	50.6	60.5	67.3	5.9	10
Mesos	1×10^7	42.5	54.4	69.3	8.5	10
Mesos and Docker	1×10^7	73.2	88.0	99.6	8.2	10
Standalone	5×10^7	136.6	154.6	167.2	9.8	10
Standalone with Docker	5×10^7	103.2	134.4	165.9	20.6	10
Mesos	5×10^7	111.5	125.9	135.6	9.1	10
Mesos and Docker	5×10^7	374.8	473.4	567.6	64.6	10

Table 3: Terasort Benchmark Timing Data

Bibliography

[1] Abed Abu-Dbai, David Breitgand, Gidon Gershinsky, Alex Glikson, and Khalid Ahmed. Enterprise resource management in mesos clusters. In *Proceedings of the 9th ACM International on Systems and Storage Conference*, page 17. ACM, 2016.

[2] UC Berkeley AMPLab. Keystoneml main page. `http://keystone-ml.org/`, 2017. Accessed: 2017-10-14.

[3] Charles Anderson. Docker. *IEEE Software*, 32(3), 2015.

[4] The Kubernetes Authors. Kubernetes production-grade container orchestration. `https://kubernetes.io`, 2017. Accessed: 2017-05-27.

[5] The Spark Bench authors. Benchmark suite for apache spark. `https://github.com/SparkTC/spark-bench`, 2016. Accessed: 2016-12-28.

[6] Mariana Carroll, Paula Kotzé, and Alta Van der Merwe. Secure virtualization: benefits, risks and constraints. 2011.

[7] Intel Corporation. Intel® main page. `http://www.intel.com`, 2016. Accessed: 2016-11-23.

[8] Oracle Corporation. Oracle main page. `http://www.oracle.com`, 2017. Accessed: 2017-09-23.

[9] The Apache Software Foundation. Apache cassandra. `http://cassandra.apache.org`, 2017. Accessed: 2017-05-27.

[10] The Apache Software Foundation. Apache hadoop. `http://hadoop.apache.org`, 2017. Accessed: 2017-05-27.

[11] The Apache Software Foundation. Apache maven. https://maven.apache.org, 2017. Accessed: 2017-06-24.

[12] The Apache Software Foundation. Apache mesos. http://mesos.apache.org, 2017. Accessed: 2017-05-27.

[13] The Apache Software Foundation. Apache spark. https://spark.apache.org, 2017. Accessed: 2017-05-27.

[14] The Apache Software Foundation. Apache storm. http://storm.apache.org, 2017. Accessed: 2017-05-27.

[15] The Apache Software Foundation. Apache zookeeper. https://zookeeper.apache.org, 2017. Accessed: 2017-09-10.

[16] Prometeus GmbH. Top500 main page. https://www.top500.org/, 2017. Accessed: 2017-10-14.

[17] Álvaro Brandón Hernández, María S Perez, Smrati Gupta, and Victor Muntés-Mulero. Using machine learning to optimize parallelism in big data applications. *Future Generation Computer Systems*, 2017.

[18] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, volume 11, pages 22–22, 2011.

[19] Advanced Micro Devices Inc. Amd main page. http://www.amd.com, 2016. Accessed: 2016-11-23.

[20] Chef Software Inc. Chef configuration management. https://www.chef.io, 2017. Accessed: 2017-06-22.

[21] CoreOS Inc. App container spec. https://coreos.com/rkt/docs/latest/app-container.html, 2017. Accessed: 2017-06-24.

[22] CoreOS Inc. rkt - the pod-native container engine. https://github.com/rkt/rkt, 2017. Accessed: 2017-06-24.

[23] Docker Inc. Docker for the virtualization admin. E-book received per e-mail after registration at https://goto.docker.com/docker-for-the-virtualization-admin.html.

[24] Docker Inc. Docker. https://www.docker.com, 2017. Accessed: 2017-05-27.

[25] Mesosphere Inc. The definitive platform for modern apps dc/os. https://dcos.io, 2017. Accessed: 2017-05-27.

[26] Mesosphere Inc. Mesosphere. https://mesosphere.com, 2017. Accessed: 2017-05-27.

[27] Mesosphere Inc. Mesosphere enterprise dc/os. https://mesosphere.com/product, 2017. Accessed: 2017-05-27.

[28] Puppet Inc. Puppet configuration management. https://puppet.com, 2017. Accessed: 2017-06-22.

[29] Red Hat Inc. Openshift origin the open source container application platform. https://www.openshift.org, 2017. Accessed: 2017-05-27.

[30] Red Hat Inc. Red hat openshift. https://www.openshift.com, 2017. Accessed: 2017-05-27.

[31] VMware Inc. Vmware main page. https://www.vmware.com, 2016. Accessed: 2016-11-23.

[32] Douglas M Jacobsen and Richard Shane Canon. Contain this, unleashing docker for hpc. *Proceedings of the Cray User Group*, 2015.

[33] Spencer Julian, Michael Shuey, and Seth Cook. Containers in research: Initial experiences with lightweight infrastructure. In *Proceedings of the XSEDE16 Conference on Diversity, Big Data, and Science at Scale*, XSEDE16, pages 25:1–25:6, New York, NY, USA, 2016. ACM.

[34] Yuriy Kaniovskyi, Martin Koehler, and Siegfried Benkner. A containerized analytics framework for data and compute-intensive pipeline applications. In *Proceedings of the 4th Algorithms and Systems on MapReduce and Beyond*, BeyondMR'17, pages 6:1–6:10, New York, NY, USA, 2017. ACM.

[35] Tobias Knaup and Florian Leibert. Marathon. https://github.com/mesosphere/marathon, 2017. Accessed: 2017-05-27.

[36] Gregory Kurtzer, Vanessa Sochat, and Michael Bauer. Singularity: Scientific containers for mobility of compute. *PLoS One, May 2017, Vol.12(5)*, 2017.

[37] Min Li, Jian Tan, Yandong Wang, Li Zhang, and Valentina Salapura. Spark-bench: a comprehensive benchmarking suite for in memory data analytic platform spark. In *Proceedings of the 12th ACM International Conference on Computing Frontiers*, page 53. ACM, 2015.

[38] Andre Luckow, Pradeep Mantha, and Shantenu Jha. Pilot-abstraction: A valid abstraction for data-intensive applications on hpc, hadoop and cloud infrastructures? *arXiv preprint arXiv:1501.05041*, 2015.

[39] Andre Luckow, Mark Santcroos, Ashley Zebrowski, and Shantenu Jha. Pilot-data: an abstraction for distributed data. *Journal of Parallel and Distributed Computing*, 79:16–30, 2015.

[40] David Marshall. Top 10 benefits of server virtualization. `http://www.infoworld.com/article/2621446/server-virtualization/server-virtualization-top-10-benefits-of-server-virtualization.html`, 2011. Accessed: 2017-02-10.

[41] Bernardo Gomez Palacio. Running your spark job executors in docker containers. `https://github.com/berngp/mesos-spark-docker`, 2015. Accessed: 2017-05-02.

[42] AppArmor Project. Apparmor main page. `http://wiki.apparmor.net`, 2017. Accessed: 2017-09-23.

[43] KVM Project. Kvm main page. `https://www.linux-kvm.org/page/Main_Page`, 2016. Accessed: 2016-11-23.

[44] Xen Project. Xen project main page. `https://xenproject.org`, 2016. Accessed: 2016-11-23.

[45] Albert Reuther, Chansup Byun, William Arcand, David Bestor, Bill Bergeron, Matthew Hubbell, Michael Jones, Peter Michaleas, Andrew Prout, Antonio Rosa, et al. Scalable system scheduling for hpc and big data. *arXiv preprint arXiv:1705.03102*, 2017.

[46] Ed Sabol. Slow networking inside containers (tried host and bridge). `https://github.com/moby/moby/issues/30801`, 2017. Accessed: 2017-09-07.

[47] Jhoel Salas. Singularities spark docker image. `https://hub.docker.com/r/singularities/spark`, 2016. Accessed: 2017-01-14.

[48] Salman Salloum, Ruslan Dautov, Xiaojun Chen, Patrick Xiaogang Peng, and Joshua Zhexue Huang. Big data analytics on apache spark. *International Journal of Data Science and Analytics*, pages 1–20, 2016.

[49] Evan R Sparks, Shivaram Venkataraman, Tomer Kaftan, Michael J Franklin, and Benjamin Recht. Keystoneml: Optimizing pipelines for large-scale advanced analytics. In *Data Engineering (ICDE), 2017 IEEE 33rd International Conference on*, pages 535–546. IEEE, 2017.

[50] Leendert van Doorn. Hardware virtualization trends. `https://www.usenix.org/legacy/event/vee06/full_papers/vandoorn-keynote.pdf`, 2006. Accessed: 2016-11-23.

[51] Hamza Zafar, Farrukh Aftab Khan, Bryan Carpenter, Aamir Shafi, and Asad Waqar Malik. Mpj express meets yarn: towards java hpc on hadoop systems. *Procedia Computer Science*, 51:2678–2682, 2015.

[52] Ziliang Zong, Rong Ge, and Qijun Gu. Marcher: A heterogeneous system supporting energy-aware high performance computing and big data analytics. *Big Data Research*, 8:27–38, 2017.