BUILDING AN EDUCATIONAL COMPUTE CLUSTER

by

Samuel Howarth

A senior thesis submitted to the faculty of

Brigham Young University - Idaho

in partial fulfillment of the requirements for the degree of

Bachelor of Science

Department of Physics

Brigham Young University - Idaho

December 2025

BRIGHAM YOUNG UNIVERSITY - IDAHO

DEPARTMENT APPROVAL

of a senior thesis submitted by

Samuel Howarth

This thesis has been reviewed by the research committee, senior thesis coordinator, and department chair and has been found to be satisfactory.

Date	Joe Hill, Advisor
Date	David Oliphant, Senior Thesis Coordinator
Date	Todd Lines, Committee Member
Date	Matt Zachreson, Committee Member
Date	Evan Hansen, Chair

ABSTRACT

BUILDING AN EDUCATIONAL COMPUTE CLUSTER

Samuel Howarth

Department of Physics

Bachelor of Science

This thesis examines the historical development of parallel and high-performance computing, with particular emphasis on the educational and research value of compute clusters. A Beowulf-style cluster was constructed using Rocky Linux 9.6 to demonstrate practical principles of distributed computation. System deployment and automation were implemented through GRUB (Grand Unified Bootloader) menu configurations, PXE (Preboot Execution Environment) boot services, and Kickstart files, enabling fully unattended installations across multiple nodes. Configuration of cluster management tools, including SLURM (Simple Linux Utility for Resource Management), MUNGE (MUNGE Uid 'N Gid Emporium), and OpenMPI (Open Message Passing Interface), is documented in detail. System performance was evaluated using a parallelized benchmark program that approximates the value of π, demonstrating the cluster's computational efficiency, scalability, and reproducibility. The results

highlight the accessibility and pedagogical potential of open-source cluster computing for high-performance computing education.

ACKNOWLEDGMENTS

This project and thesis would not have been possible without the assistance and mentorship of Brother Joe Hill, my supervisor and first computational physics professor.

My heartfelt thanks go out to my own father Evan Howarth and my good friend Samuel King, who both provided insightful discussions and feedback in system architecture and project goals. Lastly, I would like to thank my mother Michelle and sister Ellen for the emotional support during this project and throughout my university career.

I cannot express enough gratitude to each one of you. Thank you.

Contents

List of Figures

Chapter 1

Background and History of Parallel and High-Performance Computing

Modern high-performance computing systems are the result of decades of architectural, technological, and conceptual evolution. Understanding how contemporary cluster-based systems emerged requires examining the historical constraints that shaped early computing, as well as the successive innovations that enabled parallelism at increasing scales. This chapter situates the cluster constructed for this project within that broader historical context, tracing the development of computing from early serial machines to vector processors, massively parallel systems, and ultimately to commodity-based clusters. By reviewing this progression, the chapter establishes the technical and conceptual foundations necessary for understanding the design choices and software architecture discussed in later chapters.

Parallel computing allows multiple processing units to work concurrently on parts of a problem, offering substantial increases in speed and scalability [**?**]. These ideas underpin the modern field of *high-performance computing* (HPC), which today supports advances in physics, chemistry, climate modeling, and artificial intelligence [**?**].

This chapter provides a historical overview of how computing evolved from serial machines to massively parallel systems, culminating in the development of accessible cluster-based computing for education and research [**?**].

1.1 The Beginning of Digital Computing

Since the earliest days of electronic computation, scientists and engineers have sought faster and more capable computing systems [1]. Traditional *serial* computers execute one instruction at a time, limiting the rate at which complex calculations can be performed [1]. As physical limits to processor clock speeds were approached in the early 2000s, improvements in raw performance slowed dramatically—a trend often described as the end of Dennard scaling and the slowdown of Moore's Law [1]. In response to these limitations, system designers increasingly turned toward architectures capable of performing many operations simultaneously, giving rise to the field of *parallel computing* [2].

Parallel computing allows multiple processing units to work concurrently on parts of a problem, offering substantial increases in performance and scalability [2,3]. These ideas underpin the modern field of *high-performance computing* (HPC), which today supports advances across a wide range of scientific disciplines [4]. This chapter provides a historical overview of how computing evolved from serial machines to massively parallel systems, culminating in the development of accessible cluster-based computing for education and research [5, 6].

The first generation of electronic computers, such as the ENIAC (1945) and UNIVAC I (1951), were serial machines capable of executing only one instruction at a time, reflecting the technological constraints of early electronic hardware [1]. While revolutionary for their time, these systems were limited by their sequential architec-

ture and by the physical constraints of early electronic components.

In the 1960s, researchers began exploring methods to increase computational throughput through simultaneous operations. One of the earliest large-scale attempts at parallel computing was the ILLIAC IV project, which employed a Single Instruction, Multiple Data (SIMD) architecture to apply the same operation to many data elements in parallel [7]. Around the same time, advances in processor architecture led to the development of vector machines, which achieved performance gains by operating on entire vectors of data rather than individual scalar values [2].

In 1966, Michael J. Flynn introduced a taxonomy for computer architectures based on the number of concurrent instruction and data streams: Single Instruction, Single Data (SISD), Single Instruction, Multiple Data (SIMD), Multiple Instruction, Single Data (MISD), and Multiple Instruction, Multiple Data (MIMD) [7]. Flynn's taxonomy remains a foundational framework for describing and categorizing parallel computing systems [1].

By the 1980s and 1990s, high-performance computing was dominated by large, specialized supercomputers designed for scientific and engineering applications [2]. Vector supercomputers extended earlier architectural concepts with multiple pipelines and shared-memory parallelism, enabling significant gains in sustained performance. During this period, researchers also pursued massively parallel processing (MPP) systems composed of many processors communicating through message passing, an approach that would later influence cluster-based computing models [2].

These developments laid the conceptual and architectural groundwork for modern cluster computing, in which commodity hardware and standardized software tools are used to construct scalable parallel systems [5].

1.2 Distributed and Cluster Computing

The 1990s saw the emergence of a transformative idea: rather than relying on specialized supercomputers, parallel computing could be achieved using networks of inexpensive, commodity hardware. This concept was realized through the development of *Beowulf clusters*, pioneered at NASA's Goddard Space Flight Center in 1994 [5]. A Beowulf cluster consisted of standard personal computers connected via Ethernet and running open-source software, typically Linux.

The Beowulf approach democratized access to high-performance computing, allowing universities, small research groups, and even individuals to assemble parallel systems at low cost. Around the same time, the standardization of the Message Passing Interface (MPI) in 1994 provided a portable and efficient means of inter-process communication, enabling programs to scale across distributed memory systems [3].

Open-source software became the backbone of modern cluster computing. Tools such as GCC (GNU Compiler Collection), MPICH (Message Passing Interface Chameleon), and later OpenMPI (Open Message Passing Interface), combined with Linux-based job schedulers like PBS (Portable Batch System) and SLURM (Simple Linux Utility for Resource Management), created an ecosystem that replicated many of the capabilities of large supercomputers using modest resources. Educational projects such as BYOC ("Build Your Own Cluster") exemplified this approach in teaching environments [8, 9].

The early 2000s marked a shift from specialized supercomputers toward parallelism at all scales. Processor manufacturers began integrating multiple cores onto single chips, making parallel computing a standard feature of consumer hardware. In 2006, NVIDIA introduced CUDA (Compute Unified Device Architecture), which enabled general-purpose programming of graphics processing units (GPUs), bringing massive

parallelism to scientific and engineering applications [10].

Contemporary high-performance systems combine CPUs, GPUs, and other accelerators in heterogeneous architectures. The achievement of petascale computing in 2008 with the *Roadrunner* system at Los Alamos National Laboratory, and exascale performance in 2022 with *Frontier* at Oak Ridge National Laboratory, represent milestones in computational capability [4]. These systems are complemented by the rise of cloud-based HPC services, allowing users to provision virtual supercomputers on demand.

Energy efficiency and scalability have become critical considerations in modern HPC design. The Green500 list now ranks supercomputers by performance per watt, reflecting the growing emphasis on sustainable computation. Open frameworks such as MPI and OpenMP continue to evolve to support increasingly complex hardware.

1.3 Applications of Cluster Computing in Academics

While national laboratories pursue exascale performance, smaller-scale clusters remain vital for education and research. University clusters allow students to develop practical skills in distributed computing, system administration, and scientific programming. By working with physical hardware and real schedulers, students gain an understanding of parallel job submission, process communication, and resource management that cannot be replicated in purely theoretical coursework.

Educational clusters such as Hope College's BYOC project and similar efforts using Raspberry Pi or x86 nodes demonstrate the accessibility of modern HPC techniques [6]. The compute cluster constructed for this thesis project follows in that tradition, providing an affordable, reproducible, and instructive system that bridges the gap between classroom theory and real-world computation.

Thus, the evolution of computing from serial to parallel systems reflects the continual pursuit of higher performance and scalability. From the earliest vector processors to modern heterogeneous clusters, the field of high-performance computing has advanced through a combination of hardware innovation and open collaboration. The rise of cluster computing—built on commodity hardware and open-source software—has made supercomputing accessible to researchers and students alike.

The following chapter introduces the theoretical principles and software paradigms that enable parallel systems to function effectively, forming the conceptual foundation for the design and implementation of the cluster built in this study.

Chapter 2

Cluster Software Foundations

High-performance computing (HPC) systems rely on collections of interconnected computers, known as *clusters*, to perform large-scale numerical or data-intensive computations efficiently. A cluster typically consists of multiple *compute nodes* managed by a central *head node* or controller. Each node runs a compatible operating system, shares access to a common file system, and communicates over a high-speed network. Software layers such as job schedulers and message passing libraries provide the framework that enables distributed tasks to be coordinated and executed in parallel.

This chapter provides an overview of the major software components that make up a modern Beowulf-style compute cluster. The focus is on the theoretical and architectural foundations that informed the implementation described in Chapter 3. The technologies discussed include the Linux operating system, workload managers such as SLURM, the Message Passing Interface (MPI), and supporting systems for automated deployment and node communication.

2.1 Linux in High-Performance Computing

Since the early 2000s, Linux has become the dominant operating system for high-performance and supercomputing environments. The reasons for its widespread adoption include open-source availability, modularity, scalability, and robust support for scientific and engineering workloads. Linux distributions can be customized to provide lightweight installations that maximize computational performance while minimizing overhead.

Rocky Linux, a downstream derivative of Red Hat Enterprise Linux (RHEL), is particularly suited to cluster environments because it maintains binary compatibility with enterprise-grade packages while remaining freely available. The stability and long-term support of Rocky Linux make it an ideal foundation for systems where reproducibility and consistency are essential. Chapter 3 details the specific installation and configuration of Rocky Linux 9.6 used in this project.

2.2 Job Scheduling and Resource Management

In a shared computing environment, user tasks must be efficiently allocated among available hardware resources. Job schedulers and resource managers automate this process, ensuring that computational workloads are executed fairly, efficiently, and without conflict. The scheduler maintains queues of user-submitted jobs, tracks node availability, and assigns resources according to policies such as fair-share scheduling, priority weighting, or backfilling.

The **Simple Linux Utility for Resource Management (SLURM)** is one of the most widely used open-source workload managers in HPC. SLURM operates through a distributed set of daemons: a central controller (`slurmctld`) that manages job queues and resource states, and node daemons (`slurmd`) that execute assigned

tasks. Users interact with the system through commands such as `sbatch`, `srun`, and `squeue`. SLURM's design supports scalability from small research clusters to some of the largest supercomputers in operation.

2.3 Parallel Programming with MPI

While a scheduler coordinates *when* jobs are run, the Message Passing Interface (MPI) defines *how* programs communicate across distributed memory systems. MPI provides standardized functions that allow independent processes to exchange data through explicit message passing. The model enables scalable parallelism by dividing computational tasks among multiple processors, each performing calculations on a subset of data.

Key MPI concepts include *ranks* (unique identifiers for each process), *communicators* (groups of processes that can communicate), and collective operations such as broadcast, scatter, and reduce. The flexibility of MPI allows developers to parallelize algorithms in both domain-decomposition and task-decomposition paradigms.

This project utilizes the **OpenMPI** implementation of the MPI standard, chosen for its portability, performance, and seamless integration with SLURM. OpenMPI supports a range of interconnects and can automatically detect and optimize communication paths across nodes. Chapter 4 presents benchmark results that demonstrate the scaling performance of the implemented system using OpenMPI.

2.4 Automated Deployment and Network Boot

Efficient cluster deployment requires the ability to install and configure many compute nodes consistently. Manual installation on each node is impractical and prone

to human error. Instead, automated installation mechanisms are used to provision operating systems and software over the network.

The **Preboot Execution Environment (PXE)** allows nodes to boot directly from a network interface without local storage media. When configured, a node requests boot instructions from a server via DHCP and downloads a bootloader and kernel image via TFTP or HTTP. PXE is commonly used in conjunction with **Kickstart** files, which contain predefined installation parameters such as partition layouts, package selections, and post-installation scripts. Together, PXE and Kickstart enable fully unattended and reproducible node provisioning.

This approach greatly simplifies maintenance and scalability. As discussed in Chapter 3, the head node in this project hosts PXE and HTTP services that automate installation of all compute nodes, ensuring a consistent software environment across the cluster.

2.5 Secure Communication and Shared Filesystems

Communication between nodes in a cluster must be both secure and efficient. The **Secure Shell (SSH)** protocol provides encrypted remote login and command execution capabilities, enabling passwordless communication through key-based authentication. SSH is also used by job schedulers to launch and monitor processes across nodes.

In addition to secure communication, a shared filesystem is essential for simplifying data access and management. The **Network File System (NFS)** protocol allows directories on the head node to be exported and mounted by compute nodes as if they were part of the local filesystem. This shared storage model enables all nodes to access the same executables, datasets, and output files without redundant copies.

2.6 Summary

This chapter has introduced the foundational software and networking technologies that underpin the construction and operation of a Beowulf-style compute cluster. The concepts of Linux-based system architecture, workload scheduling, parallel computation, automated deployment, and secure communication form the basis of scalable high-performance computing systems.

The next chapter, Chapter 3, describes the practical implementation of these technologies in the development of a new compute cluster. Each configuration step, from PXE network setup to SLURM and OpenMPI integration, is documented in detail to provide a reproducible framework for future educational and research use.

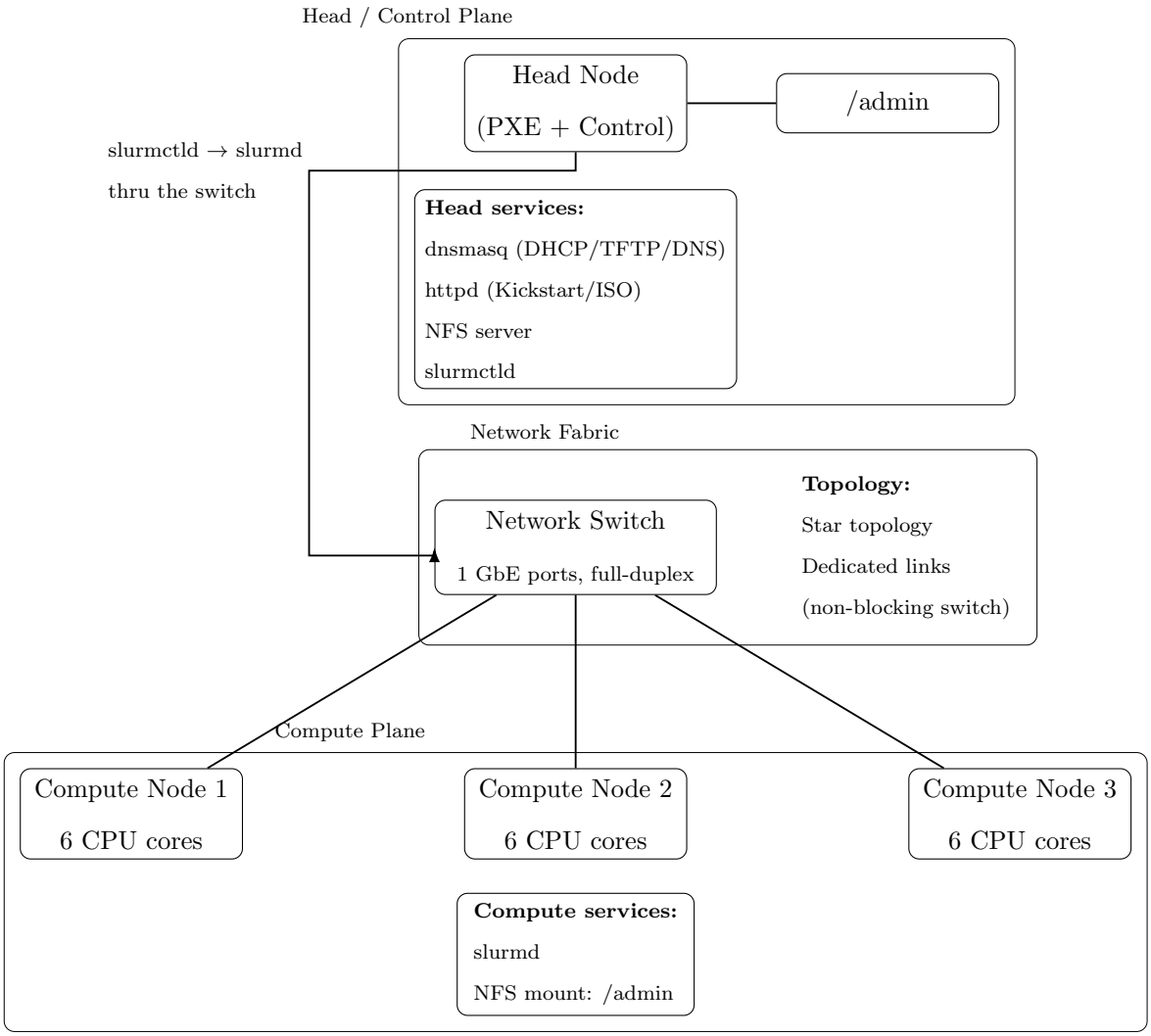

Figure 2.1 Cluster architecture showing network topology, storage layout with NFS mounts, and service roles.

Chapter 3

The Construction of a New Cluster

This chapter serves as a more detailed record of the steps taken to build and configure the compute cluster used in this project, code is included in the appendix. The goal was not only to create a functional high-performance computing environment, but also to document the process in sufficient detail that the cluster could be reconstructed or expanded by future students. The major steps included making the head node self-reinstallable, setting up network boot (PXE) services, configuring secure communication and shared storage, and deploying the workload manager and message-passing tools necessary for parallel computation.

3.1 Head Node Configuration and Self-Reinstallation

The first step in building the cluster was preparing the head node to serve as both a management server and a PXE boot host for future compute nodes. The head node was installed with Rocky Linux 9.6, using a primary NVMe disk (`nvme0`) partitioned into two parts: one for the operating system and a second partition mounted at `/admin`. The `/admin` directory was used to store ISO images, kickstart configuration

files, and related resources necessary for automated reinstallations.

A key design goal was to make the head node itself self-reinstallable. By configuring a local PXE service and maintaining a kickstart file tailored for the head node, the system could be fully reinstalled from its own resources. This design ensures that the entire cluster can be rebuilt from scratch without external installation media, simplifying maintenance and reproducibility.

Starting with a USB drive loaded with the unpackaged ISO of the desired Linux distribution, *Rocky 9.6* in this case, insert the USB and complete a first install of the new operating system on the head node. This first installation will generate a blank kickstart file at `/root/anaconda-ks.cfg`. Inside this kickstart file we can add instructions to add the `/admin` partition and its file structure. Save this kickstart in a safe location, I used a second USB thumb drive to ensure the kickstart would not be overwritten. Once this is done we can perform a new installation of the operating system, using the newly modified kickstart to enact this new partitioning plan. This is the Second installation.

After the second installation is completed, the ISO files and the kickstart can be copied into the `/admin` partition at `/admin/iso/rockyLinux/Rocky-9.6/` and `/admin/ks/headnode/ks.cfg` respectively. Additionally, now a new separate kickstart file containing instructions to set up a compute node can be created at

`/admin/ks/computenode/ks.cfg`.

This new kickstart does not need any specific instructions for drive partitioning, and won't require any modification at this time.

To make the head node self-reinstallable, a custom GRUB (Grand Unified Bootloader) menu entry was created that launches the Rocky Linux installer directly from the local ISO and kickstart files stored under `/admin`. This approach allows the system to be rebuilt automatically without external media.

The GRUB configuration file is typically found at `/etc/grub.d/40_custom`. Open and add the following code to `/etc/grub.d/40_custom` to add a new boot entry.

```
menuentry 'Reinstall Head Node (Automated)' {
    set root=(hd0,1)
    linuxefi /admin/rocky9/images/pxeboot/vmlinuz \
        inst.stage2=file:///admin/rocky9 \
        inst.ks=file:///admin/kickstarts/headnode.ks
    initrdefi /admin/rocky9/images/pxeboot/initrd.img
}
```

The configuration points the installer to the kernel (vmlinuz) and initial ramdisk (initrd.img) from the ISO, and specifies the kickstart file and repository path. Save the GRUB configuration with the following command:

```
sudo grub2-mkconfig -o /boot/efi/EFI/rocky/grub.cfg
```

Now when the headnode reboots this option will appear as one of the boot options, thus allowing the computer to return to the conditions specified by the kickstart whenever necessary.

3.2 PXE Boot and Network Installation Service

A critical component of the cluster design was establishing a PXE (Preboot Execution Environment) service that allowed both the head node and all future compute nodes to install Rocky Linux automatically over the network. This setup ensures that the cluster can be rebuilt or expanded quickly without requiring manual intervention or physical installation media.

3.2.1 PXE Overview

PXE booting enables computers to load a boot image directly from the network instead of a local disk. The process requires three main services:

1. **DHCP** to assign IP addresses and tell clients where to find the bootloader.
2. **TFTP** to provide the bootloader and kernel/initrd images.
3. **HTTP** (or NFS) to serve the installation files and kickstart configuration.

For this cluster, these services were consolidated on the head node using a minimal `dnsmasq` configuration for DHCP and TFTP, and `httpd` (Apache) for hosting the installation files.

3.2.2 Directory Structure

All PXE-related files were stored on the head node's `/admin` partition, organized as follows:

```
/admin/
 |-- rocky9/                # ISO contents copied here
 |-- kickstarts/            # Kickstart files (e.g., headnode.ks, compute.ks)
```

The Rocky Linux ISO was mounted and its contents copied into `/admin/rocky9/`, allowing the installer to fetch required packages over HTTP. The kickstart files were placed in `/admin/kickstarts/` so they could be accessed via links such as:

```
http://headnode/admin/kickstarts/compute.ks
```

The EFI bootloader files and GRUB configuration used for PXE booting were stored in a separate directory at `/root/tftpboot`, which served as the TFTP root. In this project UEFI GRUB

3.2.3 Configuring `dnsmasq`

A minimal `dnsmasq` configuration provided both DHCP and TFTP functionality. The configuration file `/etc/dnsmasq.conf` included only the essential directives:

```
interface=enp1s0
dhcp-range=192.168.1.100,192.168.1.200,12h
dhcp-boot=grubx64.efi
enable-tftp
tftp-root=/root/tftpboot
log-queries
```

This setup assigns IP addresses to PXE clients on the private cluster network and points them to the EFI bootloader served from the TFTP root.

The bootloader files (`grubx64.efi`, `shimx64.efi`, and `grub.cfg`) were placed in `/root/tftpboot/`. These were copied from the Rocky Linux ISO's EFI directory, ensuring compatibility with UEFI-based compute nodes.

3.2.4 Configuring `httpd`

The `httpd` web server was installed and configured to serve files from `/admin`. In `/etc/httpd/conf.d/admin.conf`, the directory was exposed via:

```
Alias /admin/ "/admin/"
<Directory "/admin/">
    Options Indexes FollowSymLinks
    AllowOverride None
    Require all granted
</Directory>
```

After restarting `httpd`, both the ISO contents and kickstart files became accessible over the internal network.

3.2.5 GRUB Boot Configuration

For UEFI clients, the PXE boot process used GRUB EFI instead of the legacy PXELINUX loader. The configuration file `/root/tftpboot/grub.cfg` defined the menu entries and installation parameters:

```
menuentry 'Rocky Linux 9.6 Automated Install' {
    linuxefi /rocky9/images/pxeboot/vmlinuz \
        inst.ks=http://headnode/admin/kickstarts/compute.ks \
        inst.repo=http://headnode/admin/rocky9/
    initrdefi /rocky9/images/pxeboot/initrd.img
}
```

When a compute node booted over the network, GRUB loaded the kernel and initrd files from the TFTP server and then retrieved the installation and configuration data over HTTP.

3.2.6 Testing and Verification

After enabling and starting both `dnsmasq` and `httpd`, a test PXE boot was performed using the head node itself. The system successfully booted via the network, fetched its kickstart file, and reinstalled automatically, confirming that the PXE service was correctly configured. Once validated, the same setup was used to provision additional compute nodes, allowing fully unattended installations consistent with the head node environment.

3.3 Secure Shell (SSH) Configuration

After node installation, passwordless SSH was configured to allow secure and frictionless communication between the head node and compute nodes. SSH key pairs were generated on the head node and distributed to all nodes, allowing commands and file transfers to occur without manual password entry. This step is essential for cluster management tasks and for SLURM job scheduling, which relies on seamless remote execution.

3.4 Network File System (NFS) Setup

A shared file system was established using the Network File System (NFS) protocol to provide each node with access to common files such as software packages, user directories, and administrative scripts. Centralizing these resources on the head node simplifies management and ensures consistency across the cluster.

The head node exported the `/admin` directory, which contained installation resources, and a dedicated `/shared` directory for user data. The export configuration file `/etc/exports` included the following lines:

```
/admin  *(ro,sync,no_root_squash)
/shared *(rw,sync,no_root_squash)
```

These entries make both directories accessible to all nodes on the internal network. The `sync` option ensures that file writes are committed to disk before the server replies, providing data consistency. The `no_root_squash` directive allows administrative scripts executed by root on compute nodes to retain their privileges when accessing shared directories, which is important for automated provisioning.

After updating the exports file, the NFS service was enabled and started:

```
sudo systemctl enable nfs-server
sudo systemctl start nfs-server
exportfs -rav
```

Each compute node mounted the shared directories automatically at boot time. The following entries were added to `/etc/fstab`:

```
headnode:/admin  /mnt/admin  nfs  defaults  0 0
headnode:/shared /shared     nfs  defaults  0 0
```

With NFS configured, all nodes in the cluster shared a common software and data environment. This arrangement allowed the system administrator to install software, distribute scripts, and store results in a unified workspace, facilitating collaboration and maintenance.

3.5 SLURM, MUNGE, and OpenMPI Configuration

With the hardware and basic networking infrastructure in place, the next major step in constructing the cluster was configuring the software stack responsible for authentication, workload management, and parallel execution. This stack consisted of MUNGE for node authentication, SLURM for job scheduling and resource allocation, and OpenMPI for distributed parallel computation.

3.5.1 MUNGE Authentication

MUNGE (MUNGE Uid 'N' Gid Emporium) was used to provide lightweight, credential-based authentication between cluster nodes. Rather than relying on centralized user

credentials or external directory services, MUNGE uses symmetric key authentication to verify that messages originate from trusted nodes within the cluster.

A single MUNGE key was generated on the head node and securely distributed to all compute nodes. This key was stored at `/etc/munge/munge.key` with strict ownership and permission settings, ensuring that only the `munge` user could access it. Once the key was synchronized across all nodes, the MUNGE daemon was enabled and started system-wide.

This approach ensured that SLURM daemons could authenticate with one another while keeping the authentication mechanism simple, reproducible, and well-suited to a small instructional cluster.

3.5.2 SLURM Workload Manager

SLURM served as the cluster's primary workload manager. It was responsible for job submission, scheduling, node allocation, and resource tracking. The head node was designated as the SLURM controller, running the `slurmctld` daemon, while each compute node ran a `slurmd` daemon responsible for executing assigned tasks.

The core SLURM configuration was defined in the `slurm.conf` file, a complete version of which is included in Appendix B. Key configuration concepts are summarized below.

Control Machine

The `ControlMachine` parameter in `slurm.conf` specifies the hostname of the node responsible for managing the cluster. In this deployment, the head node was designated as the control machine. All scheduling decisions, job state tracking, and node management operations originated from this node.

Node Definitions

Each compute node was explicitly defined in the configuration file using a `NodeName` entry. These definitions included the node hostname, number of CPU cores, and current operational state. Explicit node definitions allowed SLURM to accurately track available resources and assign jobs accordingly.

Nodes were grouped logically using naming patterns, enabling the configuration to scale cleanly if additional compute nodes were added in the future.

Partition Definitions

SLURM partitions were used to organize available compute resources into schedulable groups. For this cluster, a single default partition was defined that included all compute nodes. This partition served as the primary execution environment for user jobs.

Partition-level configuration allowed limits to be placed on runtime, node usage, and access control, providing flexibility for future expansion or instructional use cases.

3.5.3 OpenMPI Integration

OpenMPI provided support for parallel execution using the Message Passing Interface (MPI) standard. SLURM's native integration with MPI allowed users to launch distributed programs using either `srun` or `mpirun`, with SLURM automatically handling process placement and resource allocation.

This integration simplified the user workflow by eliminating the need for manual hostfiles and ensuring that MPI processes were launched only on nodes allocated by the scheduler.

3.5.4 System Integration Summary

Together, MUNGE, SLURM, and OpenMPI formed the functional core of the cluster. MUNGE ensured secure inter-node authentication, SLURM provided structured access to shared compute resources, and OpenMPI enabled scalable parallel execution. This software stack allowed multiple users to submit and execute parallel jobs efficiently while maintaining centralized control and reproducibility.

3.6 Testing and Validation

After completing the PXE, SSH, and NFS configurations, a validation process was performed to ensure that all cluster components operated as intended. The following criteria were tested:

1. **PXE Boot Functionality:** Each node was tested to confirm successful network boot and automated installation.
2. **Network Connectivity:** All nodes were verified to have IP connectivity and name resolution within the cluster network.
3. **Passwordless SSH:** Secure, key-based authentication was confirmed between the head node and compute nodes.
4. **NFS Mounts:** Shared directories were confirmed to mount correctly and remain accessible after reboots.
5. **SLURM Integration:** A basic MPI job was submitted through `sbatch` and completed successfully across multiple nodes.

The successful completion of these tests demonstrated that the cluster could reliably deploy, communicate, and execute distributed computations. The configura-

tion achieved the project's goal of providing a reproducible and educational high-performance computing platform.

3.7 Future Work

While the current configuration achieves a functional and scalable compute cluster, several extensions and improvements are possible:

- **Monitoring and Logging:** Deploying a monitoring suite such as `Ganglia`, `Grafana`, or `Prometheus` would enable real-time performance tracking and resource visualization.
- **RAID Storage:** Refinement of the networked drives that the nodes read from and write to with a RAID setup will protect the cluster from data loss in the event of a failure in the storage drives
- **Containerized Workflows:** Adding support for container runtimes such as Singularity or Apptainer would allow users to run isolated, reproducible software environments.
- **Job Benchmarking and Optimization:** Continued performance testing could identify bottlenecks and guide optimizations in scheduling and I/O handling.

Together, these enhancements would strengthen the cluster's usability, scalability, and long-term maintainability, paving the way for future student research and teaching applications.

Chapter 4

Benchmarking and Performance Results

To evaluate the performance and scalability of the compute cluster constructed for this project, a series of benchmark tests were conducted using distributed-memory parallel computation. The tests were designed to assess both computational throughput and communication efficiency when executing parallel workloads under the SLURM workload manager.

The benchmark program computed the numerical value of π using a parallel implementation of the midpoint integration method. This approach provided a simple but effective test of the cluster's performance, as it involved minimal communication between processes and allowed for controlled scaling across multiple nodes.

All parallel programs were compiled using `mpicc` and executed through SLURM batch scripts using `srun` or `mpirun`. Each job recorded its total runtime, number of cores utilized, and the computed value of π to a shared output file. The full source code is included in Appendix A.

The benchmark evaluated the integral

$$\pi = \int_0^1 \frac{4}{1+x^2}\,dx, \tag{4.1}$$

using the **midpoint rule** for numerical integration. The integrand was defined as

$$f(x) = \frac{4}{1+x^2}, \tag{4.2}$$

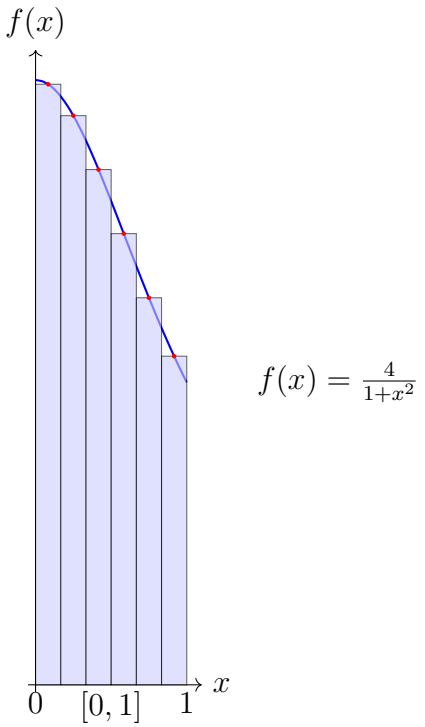

Figure 4.1 Midpoint-rule approximation of $\int_0^1 \frac{4}{1+x^2}\,dx$ using subintervals of width h.

and the integral was approximated by dividing the interval $[0,1]$ into $N = 10^9$ equal subintervals of width $h = 1/N$. The midpoint rule gives

$$\pi \approx h \sum_{i=0}^{N-1} f\left(\left(i + \frac{1}{2}\right) h\right). \tag{4.3}$$

This method is well suited to parallelization because each term in the summation can be computed independently.

4.1 Parallelizing the Task

The computation was parallelized using the Message Passing Interface (MPI). The total range of integration was divided evenly among p MPI processes. Process r, where $r \in [0, p-1]$, computed the local partial sum

$$S_r = \sum_{i=\frac{rN}{p}}^{\frac{(r+1)N}{p}-1} f\left(\left(i+\frac{1}{2}\right)h\right), \tag{4.4}$$

and the global sum was recovered through a reduction operation:

$$\pi \approx h \sum_{r=0}^{p-1} S_r. \tag{4.5}$$

Each process computed its local contribution independently and only communicated during the final reduction stage. The `MPI_Reduce()` function was used to combine the results, with the root process (rank 0) writing the final value of π, along with timing information, to an output file.

The elapsed time for each run was measured using the `MPI_Wtime()` function, providing accurate wall-clock timing for the entire computation.

All benchmarks were executed under the SLURM workload manager. Batch scripts defined the resource allocation parameters (nodes, tasks per node, and wall time limits) and executed the MPI program using `srun`. Typical configurations included 1, 2, 4, 5, and 6 cores, distributed across one or more nodes. Do note that because `MPI_Reduce()` is only called once at the end, this benchmark is not a good measure of network communication speeds which can be a significant bottleneck for

more complex parallel computing tasks, where in order to account for boundary conditions `MPI_Reduce()` must be called after every iteration.

Each job produced an output file containing:

- Number of cores used
- Computed value of π
- Total execution time (in seconds)

An example of a minimal SLURM submission script is shown below:

```
#!/bin/bash
#SBATCH --job-name=pi_benchmark
#SBATCH --output=pi_%j.out
#SBATCH --ntasks=8
#SBATCH --time=00:10:00
#SBATCH --partition=standard

module load openmpi
srun ./pi_mpi
```

Listing 4.1 Example SLURM batch script for benchmarking runs.

4.2 Performance and Scaling Results

The total runtime $T(p)$ was recorded for varying numbers of cores. The **speedup** $S(p)$ and **parallel efficiency** $E(p)$ were calculated as

$$S(p) = \frac{T(1)}{T(p)}, \qquad E(p) = \frac{S(p)}{p}. \tag{4.6}$$

An ideal speedup would follow $S(p) = p$, corresponding to perfect linear scaling. In practice, deviations from this ideal behavior arise due to communication overhead, memory access latency, and synchronization costs between MPI processes.

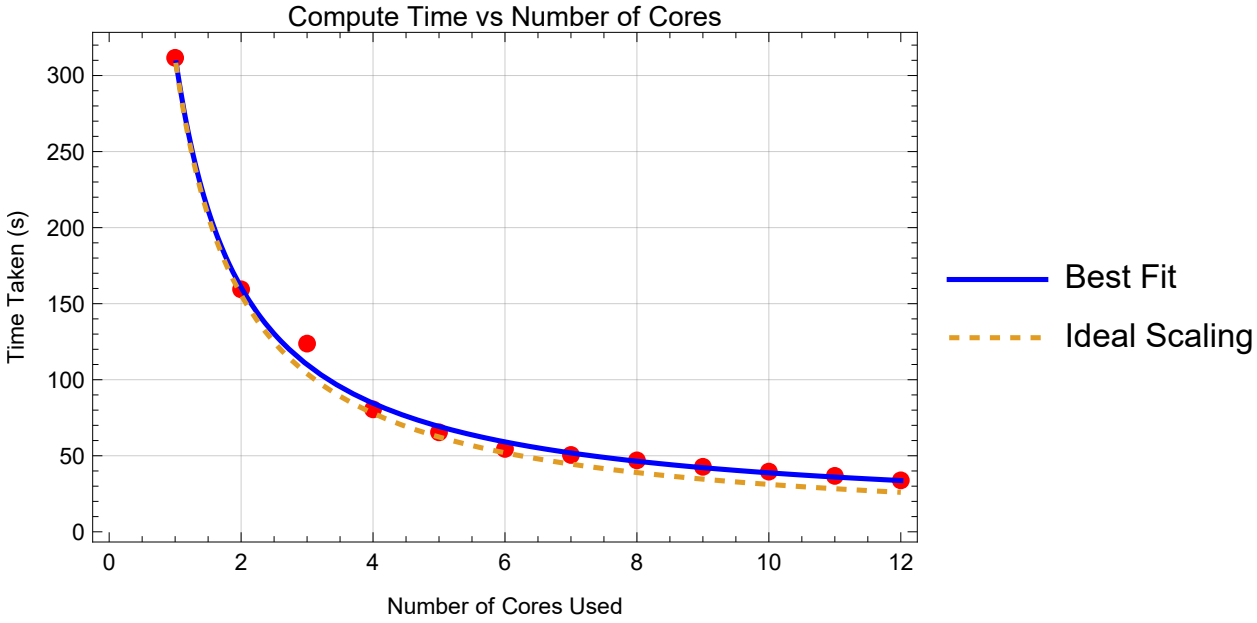

Figure 4.2 Measured execution time as a function of the number of cores used. The curve demonstrates strong scaling for small process counts, with increasing communication overhead at higher core counts. The dashed line represents an ideal compute time with no delays from communication.

The total runtime $T(p)$ was recorded for varying numbers of cores. The **speedup** $S(p)$ and **parallel efficiency** $E(p)$ were calculated as

$$S(p) = \frac{T(1)}{T(p)}, \qquad E(p) = \frac{S(p)}{p}. \tag{4.7}$$

An ideal speedup would follow $S(p) = p$, corresponding to perfect linear scaling. In practice, deviations from this ideal behavior arise due to communication overhead, memory access latency, and synchronization costs between MPI processes.

The results show that the cluster achieves near-linear speedup for up to several cores, confirming efficient CPU utilization and minimal network bottlenecks. Beyond

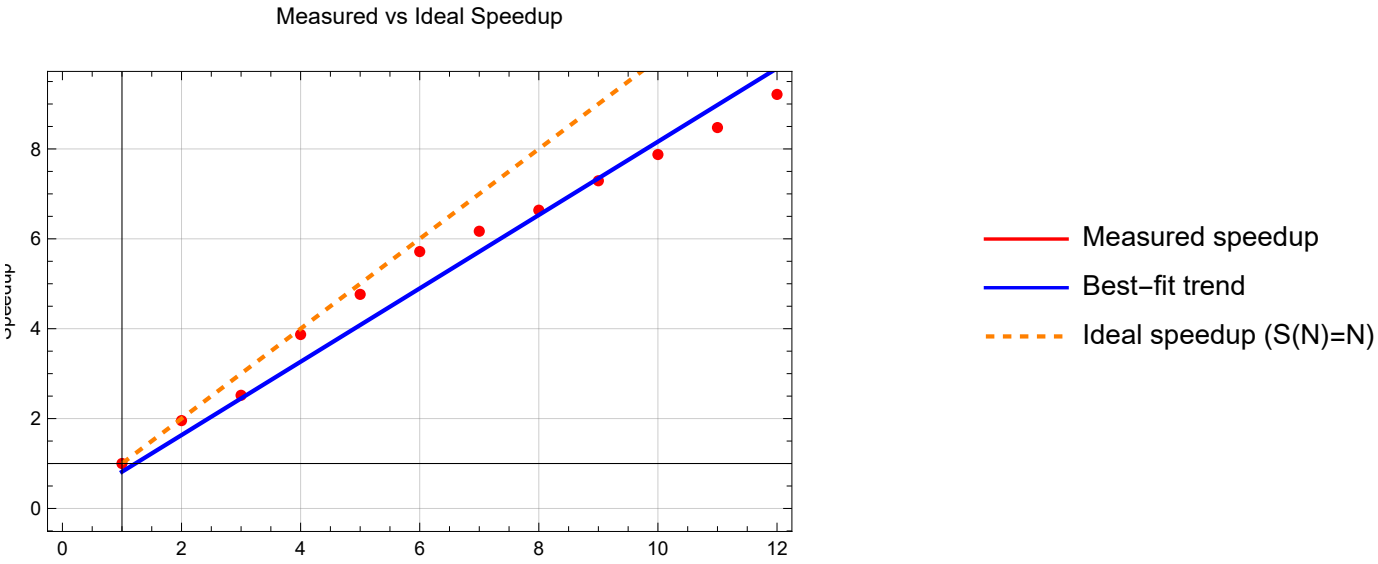

Figure 4.3 Measured versus ideal speedup. Ideal linear scaling is represented by the dashed line, while the measured curve shows realistic performance limited by inter-process communication and synchronization.

that range, speedup begins to level off as the cost of inter-process communication increases relative to computation time.

4.3 System Stability and Reproducibility

Each benchmark test was repeated multiple times to verify consistency. SLURM logs confirmed correct participation of all nodes and no job failures. The system demonstrated stable operation across repeated tests, validating the reliability of both the PXE-based deployment and the SLURM configuration.

The benchmarking results confirmed that the cluster supports efficient distributed-memory parallel computation using MPI. The midpoint integration benchmark provided a simple yet effective measure of performance, showing strong scaling across multiple cores and nodes. These tests also verified the proper configuration of SLURM, MPI communication, and file output systems.

The complete C source code used in this benchmark, along with sample SLURM batch scripts, is included in Appendix A.

Chapter 5

Conclusion

The primary goal of this project was to design, build, and validate a functional high-performance computing (HPC) cluster using recycled hardware, modern Linux tooling, and automated deployment techniques. By leveraging Rocky Linux 9.6 as the operating system and PXE boot for node provisioning, the cluster was configured to allow rapid rebuilds and easy expansion. The head node was designed to be self-reinstallable using local ISO and kickstart files, ensuring that both maintenance and system recovery can be performed with minimal manual intervention.

The configuration of SLURM as the job scheduler, MUNGE for authentication, and OpenMPI for parallel computation demonstrated that the cluster could effectively manage multi-node, multi-core workloads. Benchmarking with a simple numerical integration program showed near-linear scaling for moderate core counts, confirming that both the hardware and software environment were capable of supporting distributed computation. Deviations from ideal performance at higher core counts were consistent with expected communication overhead and limitations inherent in parallel processing, providing a realistic demonstration of HPC concepts for educational purposes.

Several key takeaways emerged from this project. First, careful planning of the head node and shared administration directories simplified the deployment and expansion of compute nodes. Centralizing ISO images, kickstart files, and PXE boot configurations allowed for reproducible system builds, reducing errors and manual effort. Second, integrating industry-standard tools such as SLURM and OpenMPI provided a practical environment for testing parallel programs, while exposing students to real-world HPC software paradigms. Finally, leveraging inexpensive or recycled hardware showed that functional and scalable compute clusters can be built without prohibitive cost, highlighting the accessibility of high-performance computing in an educational setting.

Beyond technical accomplishments, this project underscored the importance of documentation, maintainability, and modular design. Detailed records of system setup, configuration, and benchmarking ensure that future students and researchers can reproduce or expand the cluster with confidence. The combination of automated provisioning, networked file systems, and secure communication provides a robust framework that supports both learning and research, reinforcing the educational value of hands-on HPC experience. Ultimately, the cluster constructed in this project represents not only a computational resource but also a teaching tool that embodies the principles of scalability, automation, and reproducibility in high-performance computing.

Appendix A

Benchmark Program Source Code

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

double f(double x) {
    return 4.0 / (1.0 + x*x);
}

int main(int argc, char* argv[]) {
    int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
```

```
const long long N = 1000000000; // Number of intervals (1
    billion)
double h = 1.0 / (double)N;
double local_sum = 0.0, pi = 0.0;

// Split work across ranks
long long start = (N / size) * rank;
long long end   = (rank == size-1) ? N : start + (N / size
    );

double t0 = MPI_Wtime();

for (long long i = start; i < end; i++) {
    double x = h * ((double)i + 0.5);
    local_sum += f(x);
}

local_sum *= h;

// Reduce all local sums to rank 0
MPI_Reduce(&local_sum, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
    MPI_COMM_WORLD);

double t1 = MPI_Wtime();

// Only rank 0 writes results
if (rank == 0) {
```

```
        FILE *fp = fopen("/home/mpiuser/pi_result.txt", "w");
        if (fp == NULL) {
            fprintf(stderr, "Error␣opening␣file␣for␣writing\n"
               );
            MPI_Abort(MPI_COMM_WORLD, 1);
        }
        fprintf(fp, "Number␣of␣cores␣used:␣%d\n", size);
        fprintf(fp, "Computed␣value␣of␣pi:␣%.15f\n", pi);
        fprintf(fp, "Time␣taken␣(s):␣%.6f\n", t1 - t0);
        fclose(fp);
        printf("Computation␣complete.␣Results␣saved␣to␣/home/
           mpiuser/pi_result.txt\n");
    }

    MPI_Finalize();
    return 0;
}
```

Listing A.1 MPI benchmark program used for numerical estimation of π.

Appendix B

SLURM Configuration File

Appendix C

Headnode Kickstart File

```
# Generated by Anaconda 34.25.5.17
# Generated by pykickstart v3.32
#version=RHEL9

# Graphical install is optional
#graphical

%addon com_redhat_kdump --enable --reserve-mb='auto'
%end

# Keyboard layout and system language
keyboard --xlayouts='us'
lang en_US.UTF-8

# Package group
%packages
@core
```

```
@^graphical-server-environment

# cluster management and job scheduling
munge
munge-libs
munge-devel
slurm
slurm-slurmd
slurm-slurmctld

# MPI packages
openmpi
openmpi-devel

#networking and time synchroization
nfs-utils
chrony

# PXE services
dnsmasq
tftp-server
httpd

# other useful utilities
vim
wget
curl
```

```
bash-completion


%end


# Run the Setup Agent on first boot
firstboot --enable


# PRE: Dynamically set up /admin partition depending on
   whether "ADMIN" partition already exists
%pre
#!/bin/sh
admin_dev=$(blkid -L ADMIN)


if [ -n "$admin_dev" ]; then
    partname=$(basename "$admin_dev")
    echo "part␣/admin␣--fstype=xfs␣--noformat␣--onpart=${
       partname}" > /tmp/admin-ks-partition.ks
else
    echo "part␣/admin␣--fstype=xfs␣--size=20480␣--ondisk=
       nvme0n1␣--label=ADMIN" > /tmp/admin-ks-partition.ks
fi
%end


# Disk and partition layout
ignoredisk --only-use=nvme0n1
clearpart --all --initlabel --disklabel=gpt --drives=nvme0n1
bootloader --boot-drive=nvme0n1
```

```
part /boot/efi --fstype=efi --size=600 --fsoptions="umask
   =0077,shortname=winnt" --ondisk=nvme0n1
part /boot     --fstype=xfs --size=1024 --ondisk=nvme0n1
%include /tmp/admin-ks-partition.ks
part /         --fstype=xfs --size=81920 --grow --ondisk=
   nvme0n1

network --bootproto=static --ip=192.168.100.1 --netmask
   =255.255.255.0 --gateway=192.168.100.1 --device=eno1 --
   activate --hostname=headnode.localdomain

# Timezone and root password
timezone America/New_York --utc
rootpw --plaintext TempP1234!

# immediately after install we want to run all the scripts to
   setup our installed packages
%post --log=/root/kickstart-post.log

# enabled and start chronyd
systemctl enable chronyd
systemctl start chronyd

# Munge setup
if [ ! -f /etc/munge/munge.key ]; then
        /usr/sbin/create-munge-key
```

```
fi
chown munge:munge /etc/munge/munge.key
chmod 400 /etc/munge/munge.key
systemctl enable munge
systemctl start munge

# Slurm services
systemctl enable slurmctld
systemctl start slurmctld
systemctl enable slurmd
systemctl start slurmd

# enable and start dnsmasq for DHCP/TFTP
systemctl enable dnsmasq
systemctl start dnsmasq

systemctl enable tftp.socket
systemctl start tftp.socket

systemctl enable httpd
systemctl start httpd

# adjust firewall rules for all services
firewall-cmd --permanent --add-service=munge
firewall-cmd --permanent --add-service=slurmctld
firewall-cmd --permanent --add-service=slurmd
firewall-cmd --permanent --add-service=dnsmasq
```

```
firewall-cmd --permanent --add-service=tftp
firewall-cmd --permanent --add-service=http
firewall-cmd --reload


%end
# POST (non-chroot): Add GRUB menu entry for PXE reinstall
%post --nochroot
cat << 'EOF' >> /mnt/sysimage/boot/grub2/custom.cfg
menuentry 'Install␣Rocky␣9.6␣from␣/Admin' {
    linuxefi /images/pxeboot/vmlinuz inst.repo=http
       ://192.168.100.1/admin/iso/rockyLinux/Rocky9.6/ inst.ks
       =http://192.168.100.1/admin/ks/headnode/ks.cfg
    initrdefi /images/pxeboot/initrd.img
}
EOF
%end
```

Listing C.1 Kickstart file used for the headnode in this project.

Bibliography

[1] David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, 5th edition, 2017.

[2] Jack Dongarra, W. Orser, M. Rosing, and R. Smith. A history of high performance computers. *Communications of the ACM*, 33(9):47–58, 1990.

[3] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press, 3rd edition, 2014.

[4] Jack Dongarra, Pete Beckman, Omar Aaziz, et al. Report on the achievement of the first exascale supercomputer. Technical report, U.S. Department of Energy, Exascale Computing Project, 2022.

[5] Thomas Sterling, Daniel Savarese, John E. Becker, John E. Dorband, Udaya Ranawake, and Charles V. Packer. Beowulf: A parallel workstation for scientific computation. In *Proceedings of the International Conference on Parallel Processing*, pages 11–14, 1995.

[6] Brian Barrett, Jason Cope, Aaron Fahey, Jared Kienzle, and John Muehlbauer. Raspberry pi clusters for teaching parallel computing. *Computing in Science & Engineering*, 21(5):82–88, 2019.

[7] Michael J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, 1966.

[8] Nathan R. Vance, Michael L. Poublon, and William F. Polik. Byoc: Build your own cluster, part i – design. *Faculty Publications*, (1439), 2016. Paper 1439.

[9] Nathan R. Vance, Michael L. Poublon, and William F. Polik. Byoc: Build your own cluster, part ii – installation. *Faculty Publications*, (1438), 2016. Paper 1438.

[10] David B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 4th edition, 2021.