View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by UNF Digital Commons

UNF Digital Commons

UNF Graduate Theses and Dissertations

Student Scholarship

2019

A method of evaluation of high-performance computing batch schedulers

Jeremy Stephen Futral
University of North Florida

Suggested Citation

Futral, Jeremy Stephen, "A method of evaluation of high-performance computing batch schedulers" (2019). *UNF Graduate Theses and Dissertations*. 869.
https://digitalcommons.unf.edu/etd/869

This Master's Thesis is brought to you for free and open access by the Student Scholarship at UNF Digital Commons. It has been accepted for inclusion in UNF Graduate Theses and Dissertations by an authorized administrator of UNF Digital Commons. For more information, please contact Digital Projects.
© 2019 All Rights Reserved

A METHOD OF EVALUATION OF HIGH-PERFORMANCE COMPUTING BATCH SCHEDULERS

by

Jeremy Futral

A thesis submitted to the
School of Computing
in partial fulfillment of the requirements for the degree of

Master of Science in Computer and Information Sciences

UNIVERSITY OF NORTH FLORIDA
SCHOOL OF COMPUTING

April, 2019

Copyright (C) 2018 by Jeremy Futral.

All rights reserved. Reproduction in whole or in part in any form requires the prior written permission of Jeremy Futral or designated representative.

The thesis "A Method of Evaluation of High-Performance Computing Batch Schedulers" submitted by Jeremy Futral in partial fulfillment of the requirements for the degree of Master of Science in Computer and Information Sciences has been

Approved by the thesis committee: Date

_____________________________ _______________

Eggen, Roger
Thesis Advisor and Chairperson

_____________________________ _______________

Ahuja, Sanjay

_____________________________ _______________

Liu, Xudong

Accepted for the School of Computing:

_____________________________ _______________

Elfayoumy, Sherif
Director of the School

Accepted for the College of Computing, Engineering, and Construction:

_____________________________ _______________

Klostermeyer, William
Interim Dean of the College

Accepted for the University:

_____________________________ _______________

Kantner, John
Dean of the Graduate School

ACKNOWLEDGEMENT

I wish to especially thank my wife, Cristen, for supporting me and pushing me to always be better than I think I am.

CONTENTS

List of Figures vii

List of Tables viii

Abstract ix

Chapter 1: Introduction 1

1.1 High-Performance Computing 1
1.2 High-Performance Computing Topologies 2

Chapter 2: Overview of Schedulers and Clusters 4

2.1 Overview of Beowulf 5
2.2 Overview of Portable Batch Scheduler Professional 6
2.3 Overview of Slurm Workload Manager 9
2.4 Overview of Kubernetes 11
2.5 Previous Work 13

Chapter 3: Methodology 16

3.1 Experimental Setup 19

Chapter 4: Results 22

Chapter 5: Future Work and Conclusion 29

5.1 Conclusion 30

Print Publications 31

Electronic Sources 31

Appendix A: Slurm Workload Manager Code Listing 35

Appendix B: Portable Batch Scheduler Professional Code Listing 37

Appendix C: Kubernetes Code Listing 39

Appendix D: Beowulf Code Listing 46

Vita 48

FIGURES

Figure 1: Portable Batch Scheduler Architecture 7

Figure 2: Portable Batch Scheduler Multiple Execution Host 8

Figure 3: Slurm Workload Manager Architecture 10

Figure 4: Kubernetes Master-Minion Architecture 12

Figure 5: 10th percentile Time-to-Spool for Class A Jobs 23

Figure 6: 90th percentile Time-to-Spool for Class C Jobs 23

Figure 7: 10th percentile Total-Time for Class A Jobs 26

Figure 8: 90th percentile Total-Time for Class C Jobs 27

TABLES

Table 1: Time-to-Spool (sec) 22
Table 2: RAM Usage 24
Table 3: Time-to-Process (sec) 25
Table 4: Total-Time (sec) 26

ABSTRACT

According to Sterling et al., a batch scheduler, also called workload management, is an application or set of services that provide a method to monitor and manage the flow of work through the system [Sterling01]. The purpose of this research was to develop a method to assess the execution speed of workloads that are submitted to a batch scheduler. While previous research exists, this research is different in that more complex jobs were devised that fully exercised the scheduler with established benchmarks. This research is important because the reduction of latency even if it is miniscule can lead to massive savings of electricity, time, and money over the long term. This is especially important in the era of green computing [Reuther18].

The methodology used to assess these schedulers involved the execution of custom automation scripts. These custom scripts were developed as part of this research to automatically submit custom jobs to the schedulers, take measurements, and record the results.

There were multiple experiments conducted throughout the course of the research. These experiments were designed to apply the methodology and assess the execution speed of a small selection of batch schedulers. Due to time constraints, the research was limited to four schedulers.

The measurements that were taken during the experiments were wall time, RAM usage, and CPU usage. These measurements captured the utilization of system resources of each of the schedulers. The custom scripts were executed using, 1, 2, and 4 servers to determine how well a scheduler scales with network growth. The experiments were conducted on local school resources. All hardware was similar and was co-located within the same data-center. While the schedulers that were investigated as part of the experiments are agnostic to whether the system is grid, cluster, or super-computer; the investigation was limited to a cluster architecture.

Chapter 1

INTRODUCTION

This research is comprised of several chapters. This first chapter discusses high-performance computing, and its various topologies. The second chapter introduces the clusters and schedulers that were evaluated during the experimentation phase of the research. Chapter 2 also discusses previous works that were reviewed that relate to this research. Chapter 3 discusses the methodology used to assess the clusters and schedulers that were presented in chapter 2. Chapter 4 discusses the results obtained from the experiments. Chapter 5 draws conclusions and identifies future work that may be derived from this research.

1.1 High-Performance Computing

High-performance computing is the use of parallel processing for running advanced application programs efficiently, reliably and quickly [Yang13]. According to Rouse, a high-performance computer can be composed of nearly anything, from commodity hardware, to individual user PCs spread across the globe, to a large single super computer in a single data-center, or even to a collection of virtual machines in the cloud [Rouse07].

Today's high-performance computing applications require parallel processing [AWS15]. This is accomplished either by deploying grids or clusters of standard servers and central processing units in a scale-out manner, or by creating specialized servers and systems with unusually high numbers of cores, large amounts of total memory, or high throughput network connectivity between the servers, and from servers to high-performance storage [AWS15].

Originally as late as 2007, the most common users of high-performance computing systems were scientific researchers, engineers and academic institutions. Some government agencies, particularly the military, also rely on high-performance computing for complex applications [Rouse07]. However, most high-performance computing as of 2017 is done in the commercial sector, in fields such as aerospace, automotive, semiconductor design, large equipment design and manufacturing, energy exploration, and financial computing [AWS15].

1.2 High-Performance Computing Topologies

As high-performance computing has increased in popularity so has its applications and its various forms. Now there are three specific types of high-performance computing topologies: cluster, grid, and super-computer. Clusters are connected on a local area network , implemented on commodity hardware, and optimized for throughput and low

latency services [Kaur14]. Grid systems are geographically dispersed, are dynamically sized, and implemented on any kind of compute resource [Kaur14]. Grids may not be dedicated. Super-computer systems are a single computer with many dedicated resources.

These topologies all support different parallel programming paradigms. These paradigms are addressed as part of Flynn's taxonomy. Flynn's taxonomy consists of four types of computer systems. These are Single Instruction and Single Data (SISD), Single Instruction and Multiple Data (SIMD), Multiple Instruction and Single Data (MISD), and Multiple Instruction and Multiple Data (MIMD) [Flynn66]. All four are considered examples of parallel computing [Haase99].

Chapter 2

Overview of Schedulers and Clusters

A batch scheduler maximizes the assignment of resources to jobs [Sterling01]. Essentially the batch scheduler assigns work to resources based upon their availability, their current load, and reassigns work based upon any changing conditions. One can also write jobs that can be simple or complex shell scripts that are submitted to the batch scheduler to be executed.

Due to time constraints and the large number of available batch schedulers, this research focused on only four specific schedulers. These were specifically chosen in that they are free, open source, prolific, easily obtained, and have a unique architecture. It is the opinion of this research that these specific architectures presented are also representative of the wider landscape of batch schedulers available.

The batch scheduler software suites that were chosen were the default Linux job scheduler running on a Beowulf cluster, Portable Batch Scheduler Professional, Slurm Workload Manager, and Kubernetes. Slurm Workload Manager and Portable Batch Scheduler run on many of the top 500 scientific, academic, and industrial systems. Kubernetes runs

on many corporate networks and cloud-based systems. Beowulf runs on many academic and hobbyist systems. So, it is appropriate to study these systems in depth.

2.1 Overview Beowulf Cluster

Beowulf clusters are mostly found in academic [Becker97] and hobbyist settings [Brown04]. Beowulf clusters are typically loosely coupled compute resources that reside on dissimilar or in some cases commodity hardware. While there is not a standard definition of what constitutes a Beowulf cluster, they typically have a message passing interface package, a patched Linux kernel to take advantage of universal process IDs, and also patched for Distributed Inter-Process Communication (DIPC) [Becker97]. The one thing that is common among Beowulf clusters is the use of the Linux operating system and sharing of a home folder via the Network File System (NFS) [Sterling01].

Beowulf clusters do not have a specific batch scheduler. Beowulf clusters are not defined by their batch scheduling architecture; however, it is not uncommon to encounter Beowulf clusters with various types of batch scheduling software pre-installed. A few examples are Condor, Maui, Portable Batch Scheduler Professional, or even Slurm Workload Manage in some unusual cases. Since the Beowulf cluster’s scheduler in the experiments is the default Linux scheduler, it functioned as the control for the experiments that were conducted during this research.

Typically, Beowulf implements only the kernel's default scheduler on a per-server basis. The software engineer needs to design with this constraint in mind. When a Beowulf cluster does not have a batch scheduler, which is often the case, all resources are managed directly by the operating system and application. This means that compute resources are maintained independently on each node by the node's local Linux kernel. The implication for this is that all resource calculation must be performed ahead of time and maintained independently by the developer.

A major benefit to Beowulf clusters is that a Linux capable system with a shared NFS partition is the only hard requirement [Sterling01]. This enables Beowulf clusters to be made from nearly any spare compute resources including Raspberry Pis [Vaughan-Nichols17].

2.2 Overview of Portable Batch Scheduler Professional

The first batch scheduler that is included in this research is Portable Batch Scheduler Professional. According to the manual PBS is a distributed workload management system which manages and monitors the computational workload on a set of one or more computers [Altair18].

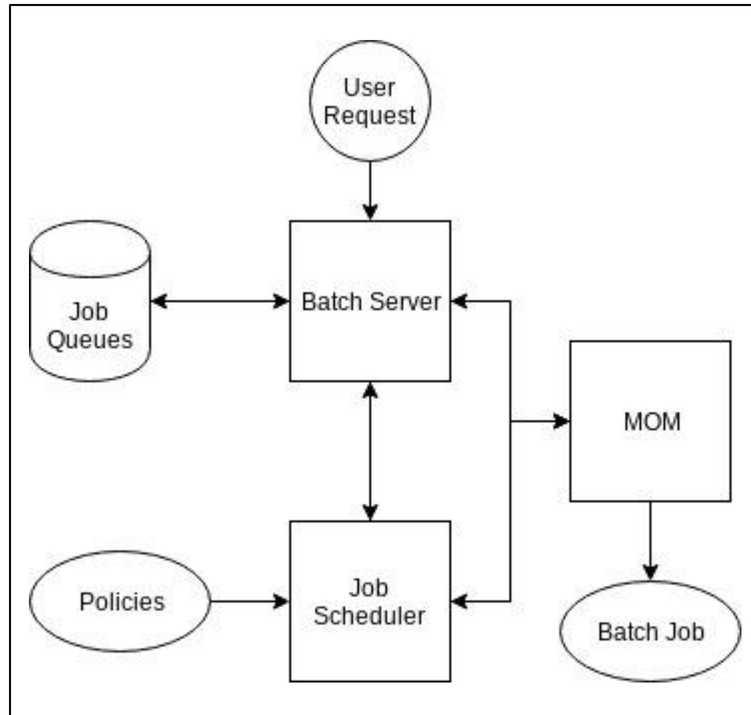

Figure 1: Portable Batch Scheduler Architecture [HPC2N17]

In Portable Batch Scheduler Professional (PBSPro) the software suite consists of a Batch Server daemon, a Job Scheduler daemon, and a job executor also known as a Machine Oriented Mini-server or MOM [HPC2N17]. The high-level architecture is illustrated in Figure 1. The Batch Server daemon is where users submit their job requests to be processed [HPC2N17]. Typically, client software is loaded on user workstations and specialized software is utilized to send commands to the Batch Server that can either schedule or modify jobs. These jobs are held in queues on the Batch Server until the resources that are required for them to execute becomes available.

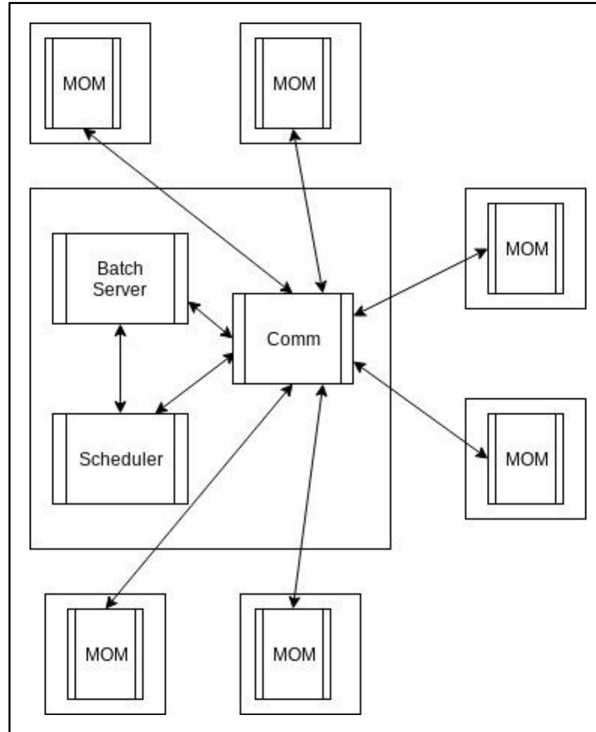

Figure 2: Portable Batch Scheduler Multiple Execution Host [Altair18]

The Job Scheduler daemon communicates with each of the job executors (or MOMs) on the different nodes [Sterling01]. The Job Scheduler determines the state of the node and if new resources are available for the MOM to begin execution of a new job for that node. It also communicates with the Job Scheduler daemon to determine if any new jobs are available for execution on the collection of nodes. It is important to note that the Job Scheduler daemon does not necessarily exist on the same server as the Batch Server. In the case of the experiments, it will be co-located to reduce any latency.

The architecture that was explored as part of this research was the Multiple Execution Systems as illustrated in Figure 2. In this configuration, the job executor daemons are installed on the worker nodes. The controller node contains a Batch Server, a Job Scheduler daemon, and a communications daemon. The worker nodes communicate with the hosted communication daemon which proxies the messages and either routes them to the Batch Server, the Job Scheduler daemon, or other worker nodes. One important aspect is that the scheduler and server daemon are backed by a database. The database maintains the job queues and all accounting information that is accessed by the Job Scheduler daemon. Currently, as of 2018 this database is PostgreSQL 9.2.

2.3 Overview of Slurm Workload Manager

On the November 2013 Top500 list, five of the ten top systems use Slurm including the number one system [Slurm13A]. These five systems alone contain over 5.7 million cores [Slurm13A]. The Slurm architecture consists of a primary job controller daemon (SlurmCTLD) which issues commands to daemons (SlurmD) on the worker machines as illustrated in figure 3. The architecture also optionally consists of an accounting database and additional job controller daemons. The database and additional controller daemons interface with the primary job controller daemon to provide highly available backups.

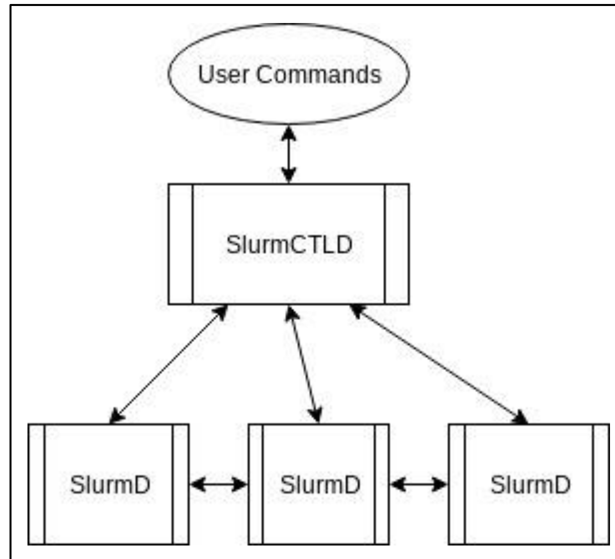

Figure 3: Slurm Workload Manager Architecture [Slurm13B]

The job controller daemons track user submitted jobs and submits them to the primary Slurm daemon for scheduling once compute resources are available. Slurm also provides a suite of command line applications. These can be run to interact with the Slurm daemon and the job controller daemons to schedule jobs and control their behavior [Slurm13B].

The Slurm daemons are responsible for utilizing compute resources as they become available and are exhausted. The Slurm daemon allocates resources based on a partition scheme. In Slurm, a partition is where certain compute resources have been allocated and reserved for various jobs to ensure they are always available for those job sets.

According to Namur et al., Slurm workload manager has the ability to run jobs in one of four methods: multi-process, multi-threaded, data-centric, and master-worker paradigm [Namur17]. Multi-process applications can be of any of the various MPI variants that are available, such as OpenMPI. Multi-threaded applications can be implemented with either p-Threads or OpenMP, which use a shared memory model. Data-centric models rely on the problem being embarrassingly parallel. In embarrassingly parallel problems, data can be easily split among multiple instances and processed independently without communication [Neiswanger15]. In a master-worker application, the master can implement any combination of the earlier described methods. Additionally, the master dispatches work to the workers and then accumulates the results.

2.4 Overview of Kubernetes

According to Kubernetes et al., Kubernetes is an open-source system for automating deployment, scaling, and management of containerized applications [Kubernetes17]. Kubernetes leverages a technology known as containerization. In containerization, a moderate portion of the operating system is loaded with a target application as a single process in memory. This is in contrast to traditional computing where many applications are housed on a single operating system and share the same user space.

Containerization gives applications the ability to be deployed with the operating system of their choosing using the tools and libraries already available. These containers are described using a file called a Dockerfile, which is essentially a recipe of how to configure the operating system and application in memory. This allows developers more freedom to write custom applications without having to worry about their target environment. Containerization also keeps the applications in a pristine environment each time they are launched. The container is destroyed, and all its resources are released on application termination.

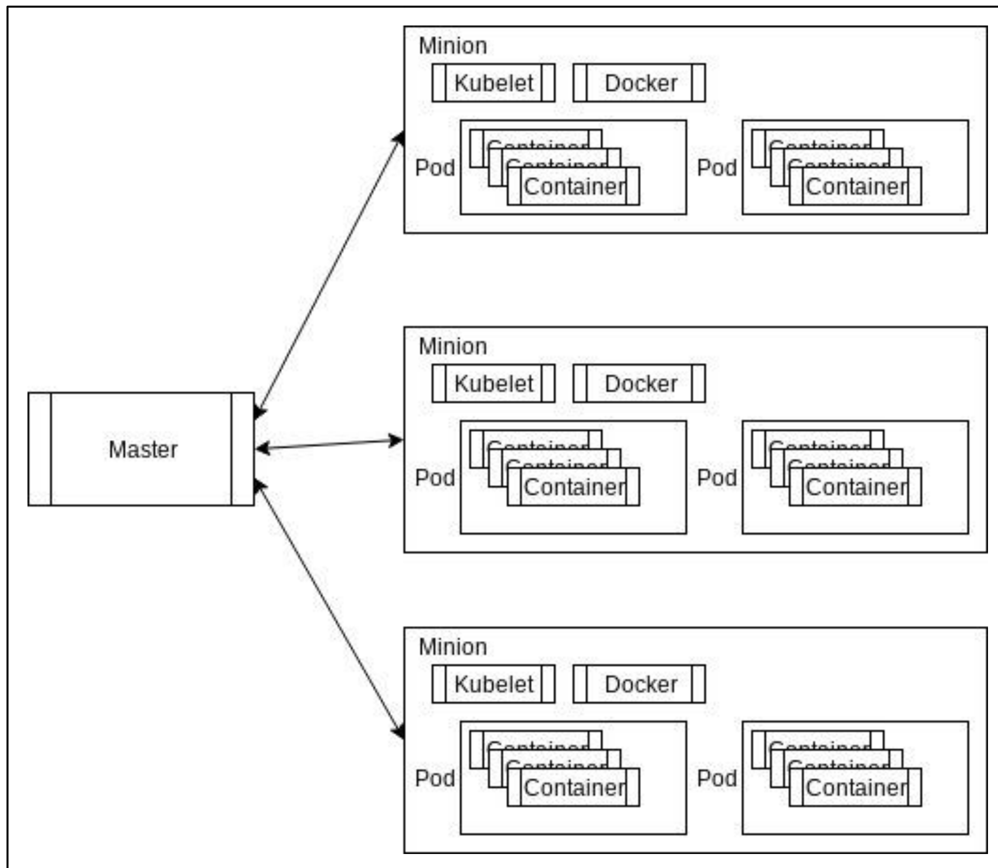

Figure 4: Kubernetes Master-Minion Architecture [Gupta15]

Kubernetes has a discrete architecture made of master nodes that manages minion nodes, as show in figure 4. Each of the minion nodes implements a Docker daemon and a Kubelet daemon that maintains the various container images in memory. These are organized into logical partitions called pods. Work is then distributed amongst the pods per application.

The master architecture can be either a single master node which maintains all the core components or a collection of master nodes with the various components spread across the master nodes. The master node contains the master Kubernetes daemon. This communicates with the minions, a batch scheduler, a user authorization component for managing system user access to the master controller, a RESTful API for remote management, and an information daemon that maintains the status of the minion machines. All of these components are controlled via user command line from a remote workstation or a dedicated server with the components supplied.

2.5 Previous Work

After extensive searching of the University's and other online sources, there were no articles that could be identified that clearly demonstrated benchmark comparisons of the

performance of any schedulers directly head to head. Instead, a selection of articles with intersecting benchmarks and technologies are presented for review.

The first article reviewed was a comparison of four different schedulers that are similar to what this research compares. The researchers Reuther et al., used Slurm Workload Manager, Grid Engine, Apache Hadoop Yarn, and Mesos [Reuther18]. The treatment was very thorough and many of the conclusions that the researchers arrived at were similar in terms of time-to-spool jobs. The problem is that the benchmark they used does not fully exercise the cluster. All jobs that were submitted were sleep jobs of varying lengths. It is the opinion of this research that the reason that sleep jobs are not a sufficient method of measurement is that as the batch job script increases in in length and computational complexity it will increase the time-to-spool.

According to Sakar et al., the researchers were employed by Tata Steel in Jamshedpur, India [Sakar12]. In the article, they wrote PBS batch jobs for a cluster, known as Reynolds. The batch jobs would then execute their own benchmarks on varying numbers of nodes. For their benchmark, they used OpenFoam which would then simulate various scenarios which were designed to exercise the system. The researchers unfortunately did not provide the code for the OpenFoam benchmarks and also no other batch scheduler schedulers were evaluated.

Another relevant benchmark paper is from Madani et al, whose comparison is of MPI specifications: MVAPICH2 and Intel MPI [Madani11]. They performed tests by varying the message package size that was communicated between nodes. They then performed these tests on both MVAPICH2 and Intel MPI frameworks and recorded the results. The paper did not include any indication that a batch scheduler was used; however, this research is relevant in that the NASA Parallel Benchmarks are MPI based.

In terms of heterogeneous processors, one can look to Soner et al [Soner 12]. In this article, the authors devise a new type of scheduler to be used in conjunction with the Slurm Workload Manager [Soner12]. This scheduler is capable of differentiating between GPU and CPU cores. According to the authors of the article, some jobs are ill suited for GPU processing time and should be exclusively scheduled on CPU cores. This article also delves into the best way to schedule these resources and ensure maximum utilization.

Docker container technology has also started to be utilized recently in conjunction with high-performance computing and can be illustrated in [Alfonso18]. In this article, the authors introduced and evaluated a tool called Elastic Cluster for Docker or EC4Docker. Its goal is to automate the deployment of Docker containers that are preconfigured with a batch scheduler and libraries associated with High-Performance computing. Instead of Kubernetes, they use Docker's competing product Swarm.

Chapter 3

METHODOLOGY

Two kernels, Class A and Class C, of the Embarrassingly Parallel (EP) of the NASA Parallel Benchmark suite were used to test the systems. Class A was used to simulate short running tasks and Class C was used to simulate long running tasks. To simulate complicated workloads, several different automation and batch job scripts were written as part of this research. These scripts were used to execute both classes of tasks many times and on various number of nodes

For this research, four pairs of scripts, describe previously, were executed with various parameters and measurements were taken. Each pair of scripts were structured identically, except for some minor changes to accommodate the scheduler being tested. The pair of scripts consisted of an automation script which then submitted a batch job script to the batch scheduler to be executed.

The automation script initializes resource monitoring and records the current time immediately before submitting the batch job script. Once the batch job script is submitted to the job scheduler, the time is recorded again upon execution by the executor

service on a remote node. It then begins execution of several NASA Parallel Benchmark programs. Once the benchmark programs complete, the batch job script records the time a final time.

According to NASA, NASA Parallel Benchmarks are a small set of programs designed to help evaluate performance [NPB18]. The benchmarks are derived from computational fluid dynamics (CFD) applications and consist of five kernels and three pseudo-applications. NASA Parallel Benchmark 3.1.1 provides three programming models that can be leveraged. OpenMPI, OpenMP, and Serial. The OpenMPI variant of NPB 3.1.1 was chosen instead of OpenMP and serial since it leverages the cluster in its entirety. OpenMP was not chosen since it does not support the cluster architecture that was chosen for this research [Eijkhout11]. OpenMPI is an open source Message Passing Interface implementation that is developed and maintained by a consortium of academic, research, and industry partners and is used for High-Performance Computing [OpenMPI18].

The time between when the automation script records the time initially and when the batch job script records the second time is the time-to-spool. The time-to-spool metric represents the amount of time it takes for the batch scheduler to completely pre-process the batch job script sent from the command line. The batch scheduler then begins execution of the batch job script itself on the worker nodes. The amount of time recorded between the second and final time, after the benchmark programs complete, is the time-

to-process. This time span represents the amount of time it takes for the batch to execute the script itself.

The pair of scripts are customized with three parameters. The first parameter is the number of benchmark programs that the batch job script will execute. This will give a good sample of more complicated batch job scripts. As the length and computational complexity of the script increases, the performance should degrade amongst the different schedulers. The second parameter is the test number. This parameter is for informational purposes only and tags the file names with a number that can be used to serialize the tests for easy extraction later. The third parameter is the specific benchmark program that will be executed multiple times during the batch job script. In our tests this was either NASA Parallel Benchmark Class A or Class C.

Since the various batch systems pre-process the batch job scripts and look in the comments for additional parameters, the scripts were not parameterized for the number of nodes. The number of nodes that the jobs required were adjusted manually before run-time.

3.1 Experimental Setup

The servers in this experimental setup were all virtualized instances that resided on VMWare hosts. The front-end server that was used as the batch server for the experiments was provisioned with 8GB of RAM and 4 vCPU cores. Each of the worker nodes were provisioned with 4 GB of RAM and 1 vCPU core each.

To ensure that the clock was synchronized for the timed portion of tests all server clocks were synchronized using the Network Time Protocol (NTP) with NTP United States pool servers. These NTP servers ensured that the clock drift between the workers and front-end server was minimal and within 100 milliseconds. In addition to NTP, all servers ran OpenSSH_7.4p1 and OpenMPI 3.1.1 with parameters'--enable-openib-rdmacm --with-slurm --with-tm=/opt/pbs'. Once everything was built and installed, four experiments were then conducted.

The first experiment was executed using a standard Beowulf cluster. No special software or daemons were installed except for OpenSSH daemons to facilitate communication to the nodes for benchmark execution. Even though the Beowulf cluster does not include a standardized scheduler, it has been included to serve as the baseline. The other clusters will be compared against this baseline.

The second experiment was conducted with Portable Batch Scheduler Professional 18.1.2. The front-end server hosted the Batch Server (pbs_server), the Job Schedulers (pbs_sched), and the Communication Daemon (pbs_comm). It also was backed by PostgreSQL 9.2.23 as its job scheduling queue. The worker nodes each host a Job Executor Daemon (pbs_mom).

The third experiment was conducted with the Slurm Workload Manager Scheduler 17.11.18. The front-end server hosted the Slurm Controller Daemon (SlurmCTLD) and the workers hosted the Slurm Worker Daemons (SlurmD). In addition to the Slurm Daemons, the Munge Daemons were also started to provide authentication between nodes in the Slurm cluster.

The final experiment was the Kubernetes cluster. All nodes in the Kubernetes cluster hosted both the Docker Daemon and the Kubelet Daemons. The Kubelet Daemon on the front-end node hosted the pods etcd, kube-apiserver, kube-controller, kube-proxy, and weave-net. The worker nodes Kubelet Daemon hosted kube-proxy, coredns, and weave-net. During the experiment the nodes also hosted a set of custom daemon pods to support the benchmark programs. The daemon pods were specifically written and designed to contain the NASA Parallel Benchmark programs, OpenMPI libraries, and OpenSSH.

To deploy the daemon pods, a request was submitted to the master node to provision the worker pods on the worker nodes. Once the request was submitted, the batch job would then be submitted to the cluster. The batch job would then provision the controller node. The controller node would test that the worker pods were available and begin running the batch shell script provided. This would then run the requested jobs on the worker pods.

Each of the described setups were then tested using the batch and automation scripts. The scripts were executed with 10, 20, 30, 40, 50, and 100 benchmark programs per batch job with Class A and again with Class C embarrassingly parallel benchmark program variants.

Chapter 4

RESULTS

In terms of time-to-spool jobs, Beowulf outperformed all of the schedulers. This can be seen in table 1, figure 5, and figure 6. Kubernetes did not perform well in terms of startup time as can be seen in the previously mentioned tables and figures. The reason that Kubernetes did not perform well was that the worker pods had to be first provisioned before a controller pod could be provisioned via the batch job. The batch job then had to perform a DNS lookup of the worker pods and then it would be forced to wait till the worker pods were available.

TIME-TO-SPOOL (MS)	**CLASS A**	**CLASS C**
KUBE	1814.5	2023.3
PBS	191.7	292
SLURM	269.1	724.3
BEOWULF	180	190

Table 1: Time-to-Spool (ms)

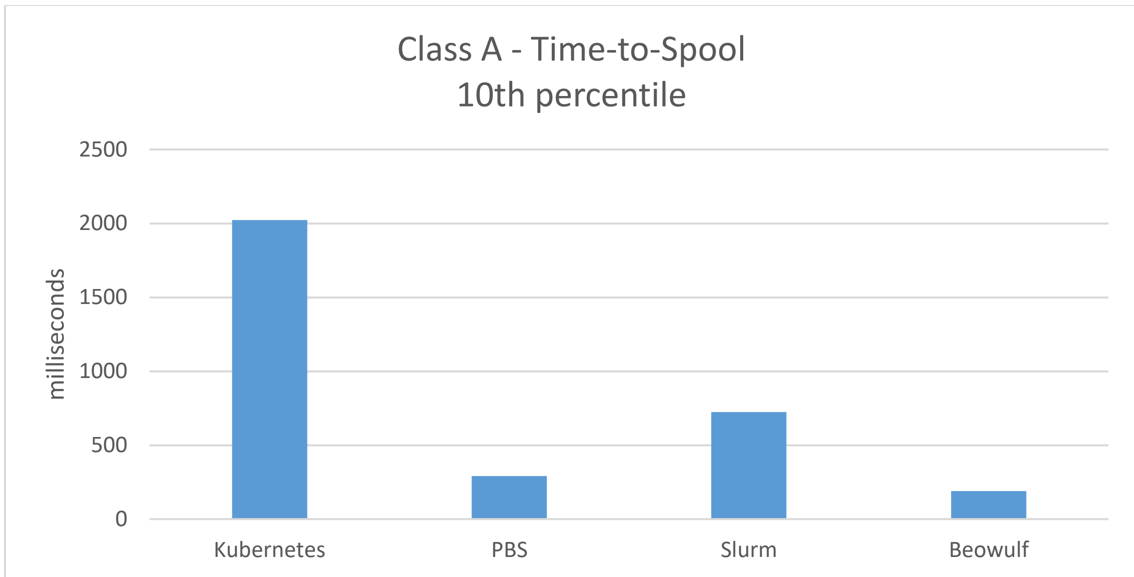

Figure 5: 10th percentile Time-to-Spool for Class A Jobs

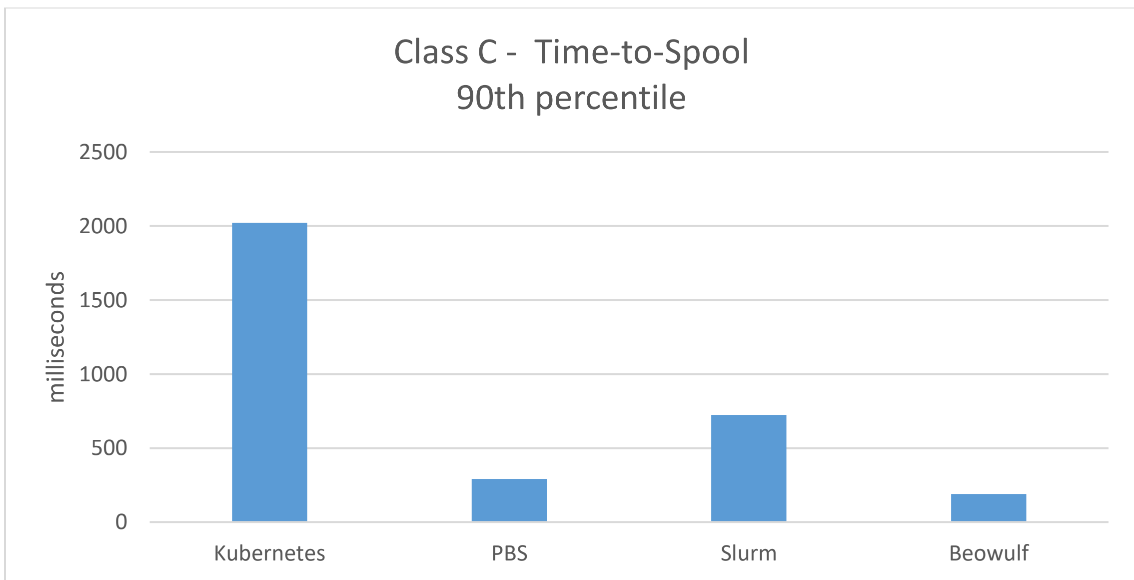

Figure 6: 90th Time-to-Spool for Class C Jobs

Of note, Kubernetes, Portable Batch Scheduler Professional, and Beowulf all had consistent and predictable spool times whereas Slurm spool times varied wildly, from as little as 80 milliseconds to 1000 milliseconds or more. This behavior can be seen especially in Class C of table 1 and figure 6. If Kubernetes time-to-spool is not considered in the full dataset then one will find that some of the Slurm Workload Manager spool times are statistically significant. The reason is that Kubernetes, Portable Batch Scheduler Professional, and Beowulf job handlers are all RAM based whereas the Slurm Workload Manager job handler is disk based. The Slurm Workload Manager jobs are first spooled to disk before execution. Since disk access times are slower and will occasionally be cached, the access times can vary from execution to execution.

RAM USAGE (KB)	MASTER	WORKER
BEOWULF	0	0
KUBERNETES	606404	290348
PBS	32164	1848
SLURM	2488	1296

Table 2: RAM Usage

RAM usage (in kilobytes) was observed during the experiments and recorded in table 2. Since Beowulf does not include a batch scheduler, it was recorded as 0 kb usage. Also observed is the very small footprint of Slurm Workload Manager. This is a consequence

of all jobs being spooled to disk and not managed in memory. Kubernetes is very memory intensive and consumes the most RAM.

The initial hypothesis was that the addition of a batch scheduler would degrade performance of the jobs. The results from these experiments were very surprising. Slurm Workload Manager and Portable Batch Scheduler Professional both performed remarkably better than the Beowulf cluster. They performed better in time-to-process, as can be seen in Table 3 and Figure 7. They also performed better in terms of total-time as can be seen in Table 4 and Figure 8. This is unexpected given that the Beowulf cluster had the shortest time-to-spool. The only situation where a batch scheduler performed worse than a plain Beowulf cluster was the Kubernetes cluster.

TIME-TO-PROCESS (SEC)	**SHORT**	**LONG**
KUBE	96.2	14508
PBS	91.2	14147
SLURM	91.6	14151
BEOWULF	94.5	14166

Table 3: Time-to-Process (sec)

TOTAL-TIME (SEC)	SHORT	LONG
KUBE	98.1	14510
PBS	91.6	14147
SLURM	92.2	14152
BEOWULF	94.7	14166

Table 4: Total-Time (sec)

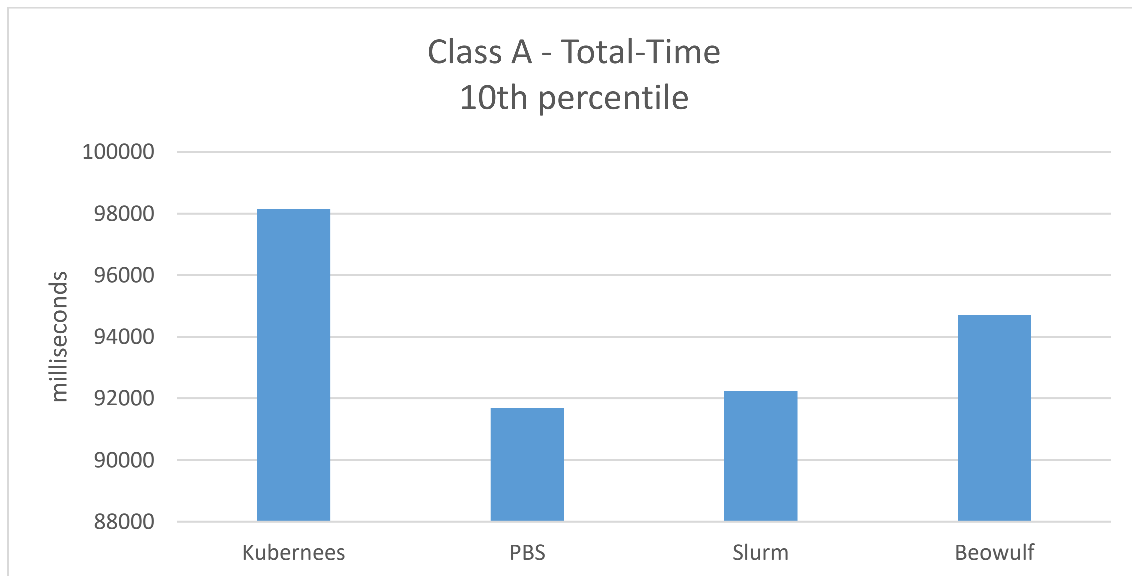

Figure 7: 10th percentile for Total-Time for Class A Jobs

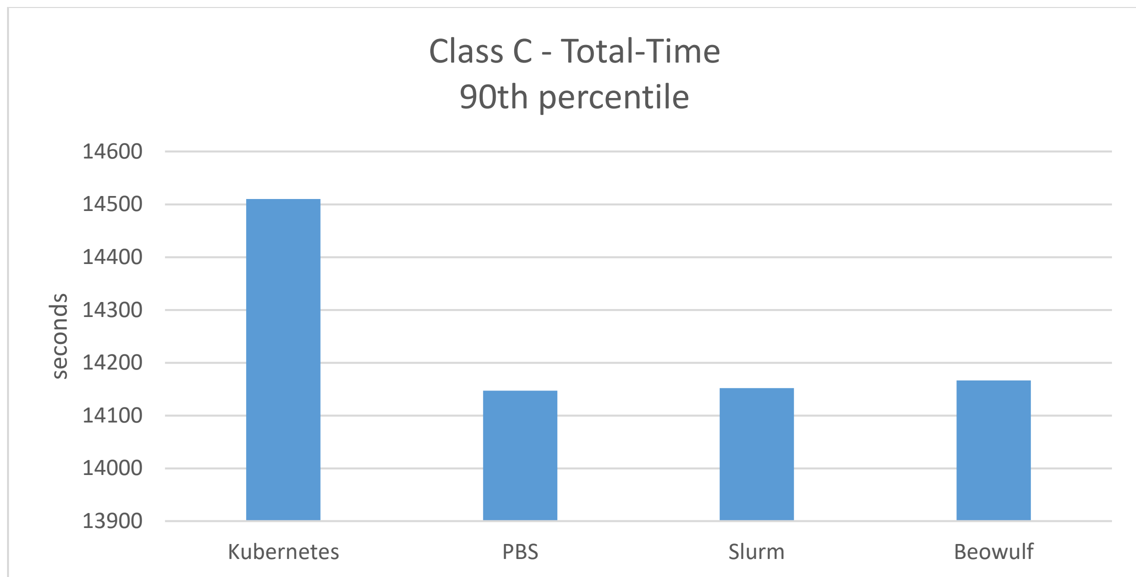

Figure 8: 90th percentile for Total-Time for Class C Jobs

It was discovered that one reason Slurm Workload Manager and Portable Batch Scheduler Professional outperform the Beowulf cluster is the fact that Beowulf relies on SSH for inter-node communication. Since SSH is encrypted, communication is slower for Beowulf. The communications in Slurm Workload Manager and Portable Batch Scheduler Professional clusters are not encrypted. The reason that this security situation would be tolerated is that the nodes that are employed in a high-performance computing cluster are fenced within an environment. The cluster is not accessible except through the front-end node.

The Kubernetes cluster also suffers from utilizing SSH for its communication protocol. In addition to SSH, our Kubernetes setup also relies on a virtual network and custom

dynamic DNS solutions to determine worker node availability. The added layer of the virtual network and the DNS lookups significantly affects its performance.

Chapter 5

FUTURE WORK AND CONCLUSION

In terms of future work, the research indicates that there are some implementation changes that could significantly improve performance. For Kubernetes, for example it needs to be determined if Weave-Net is the appropriate network plugin for the cluster. A comparison of network plugins for Kubernetes in conjunction with OpenMPI would be a great point of future research. Another way that Kubernetes cluster could be optimized is by moving from SSH to RSH for fenced networks. This same optimization could be applied to Beowulf clusters as well.

One additional optimization for Kubernetes would be to create a static, custom pod as the front-end node. Once the custom pod is provisioned then the batch job would select the front-end node instead of creating new pods each time. Provisioning all pods including the front-end pod ahead-of-time would eliminate most of the startup time.

Slurm Workload Manager out of the box does not appear to require any optimizations. Any optimizations would be in terms of additional configuration of the supporting OpenMPI libraries themselves. In order to better assess Slurm Workload Manager versus

Portable Batch Scheduler, it might be beneficial to unroll the for-loop within the batch scripts. Portable Batch Scheduler Professional also provides an MPI wrapper script (pbs_mpirun) that was not leveraged during the experiments which could potentially boost performance, since the benchmarks are OpenMPI based. Also, the job array functions within Portable Batch Scheduler and Slurm Workload manager should be leveraged to see how they compare against one another. Future research might entail evaluating batch scheduler backfill algorithms and job arrays and developing methods to evaluate those scheduler features.

5.1 Conclusion

The purpose of this research is to develop a method to evaluate the strength and weaknesses of a variety of high-performance computing schedulers. Beowulf clusters are wonderful for dedicated jobs with single users but do not provide any native batch scheduling to take advantage of idle resources. While Kubernetes does provide some batch job facilities, ease of development, and process isolation; it did not perform as well as expected overall. In conclusion, the data that was collected suggests that most batch schedulers are uniquely tuned to improve performance of high-performance compute jobs. This advanced tuning is especially pronounced in Slurm Workload Manager and Portable Batch Scheduler.

REFERENCES

Print Publications:

[Eijkhout11]
Eijkhout, Victor, et al (2011), Introduction to High-Performance Scientific Computing

[Sterling01]
Sterling, Thomas (2001). Beowulf Cluster Computing with Linux.

Electronic Sources:

[Alfonso17]
de Alfonso, Carlos, et al, "Container-Based Virtual Elastic Clusters," The Journal of Systems & Software, vol. 127, May 2017, pp. 1–11. EBSCOhost, doi:10.1016/j.jss.2017.01.007.

[Altair18]
Altair, "Altair PBS Professional 18.2 Big Book (IG, AG, HG, RG, UG, & PG)", https://www.pbsworks.com/pdfs/PBS18.2_BigBook.pdf, 2018, last accessed October 20, 2018.

[AWS15]
Amazon Web Services, "An Introduction to High Performance Computing on AWS:Scalable, Cost-Effective Solutions for Engineering, Business, and Science," https://d0.awsstatic.com/whitepapers/Intro_to_HPC_on_AWS.pdf, 2015, last accessed September 1, 2017.

[Becker97]
Becker, D., et al, "Frequently Asked Questions," http://www.beowulf.org/overview/faq.html#3, last revision 1997, last accessed September 20, 2017.

[Brown04]
Brown, Robert, "Engineering a Beowulf-style Compute Cluster", http://rgbrown.org/Beowulf/beowulf_book/beowulf_book.pdf, 2004, last accessed April, 2019.

[Flynn66]
Flynn, Michael, “Very high-speed computing systems”, Proceedings of the IEEE, volume 54, pages 1901-1909, December 1966. https://www.researchgate.net/publication/2989747_Very_High-Speed_Computing_Systems/download, last accessed April, 2019.

[Gupta15]
Gupta, Arun, “Key Concepts of Kubernetes”, http://blog.arungupta.me/key-concepts-kubernetes/, last revision January 14, 2015, last accessed September 23, 2017.

[Haase99]
Haase, Gundolf, “Parallelization of numerical algorithms”, http://www.numa.uni-linz.ac.at/Staff/haase/parvor_e/node9.html, last revision December 2013, last accessed April, 2019.

[HPC2N17]
HPC2N, Umeå University, “Beginner’s Guide to Clusters,” https://www.hpc2n.umu.se/documentation/guides/beginner-guide, last modified September 21, 2017, last accessed September 23, 2017.

[Kaur14]
Kaur, Kiranjot et al, “A Comparative Analysis: Grid, Cluster and Cloud Computing”, International Journal of Advanced Research in Computer and Communication Engineering, 2014, Vol. 3, Issue 3, https://pdfs.semanticscholar.org/5e6e/9b4b7f4d986bc9f1246198c50c8d43d2d695.pdf, last accessed April, 2019.

[Kubernetes17]
Kubernetes, “Production-Grade Container Orchestration”, https://kubernetes.io/, last revision 2017, last accessed September 23, 2017.

[Madani11]
Madani, Basem et al, “Performance Benchmark and MPI Evaluation Using Westmere-based Infiniband HPC Cluster,” International Journal of Simulation -- Systems, Science & Technology; 2011, Vol. 12 Issue 1, p20-26, 7p, EBSCOhost, login.dax.lib.unf.edu/login?url=http://search.ebscohost.com/login.aspx?direct=true&db=aci&AN=96568498&site=eds-live&scope=site, last accessed October 20, 2018.

[Namur17]
Université de Namur, “Slurm Quick Start Tutorial”, https://support.ceci-hpc.be/doc/_contents/QuickStart/SubmittingJobs/SlurmTutorial.html, last revision September 18, 2017, last accessed September 23, 2017.

[Neiswanger15]
Neiswanger, Willie et al, "Embarrassingly Parallel Variational Inference in Nonconjugate Models", https://arxiv.org/pdf/1510.04163, last modified October 14, 2015, last accessed April,2019.

[NPB18]
"NAS Parallel Benchmarks", NASA Advanced Supercomputing Division, https://www.nas.nasa.gov/publications/npb.html, last accessed September 20, 2017

[OpenMPI18]
"A High-Performance Message Passing Library", https://www.open-mpi.org/, last accessed September 20, 2017

[Reuther18]
Reuther, Albert, et al. "Scalable System Scheduling for HPC and Big Data." Journal of Parallel and Distributed Computing, vol. 111, Jan. 2018, pp. 76–92. EBSCOhost, doi:10.1016/j.jpdc.2017.06.009.

[Rouse07]
Rouse, M, "high-performance computing (HPC)," https://searchdatacenter.techtarget.com/definition/high-performance-computing-HPC, last accessed September 18, 2017.

[Sarkar12]
Sandip, Sarkar et al, "Benchmarking of High Performance Cluster Reynolds", International Journal of Advanced Research in Computer Science, 3 (3), May – June, 2012,21-32. EBSCOhost, login.dax.lib.unf.edu/login?url=http://search.ebscohost.com/login.aspx?direct=true&db=aci&AN=91876563&site=eds-live&scope=site, last accessed 20 Oct. 2018.

[Slurm13A]
"Slurm Workload Manager", https://slurm.schedmd.com/slurm.html, last revision November 24, 2013, last accessed September 23, 2017

[Slurm13B]
"Quick Start Guide", https://slurm.schedmd.com/quickstart.html, last revision March 13, 2016, last accessed September 23, 2017

[Soner12]
Soner, Seren, and Can Özturan, "Integer Programming Based Heterogeneous CPU–GPU Cluster Schedulers for SLURM Resource Manager," Journal of Computer and System Sciences, vol. 81, Feb. 2015, pp. 38–56. EBSCOhost, doi:10.1016/j.jcss.2014.06.011.

[Vaughan-Nichols17]
Vaughan-Nichols, S. J, “Build your own supercomputer out of Raspberry Pi boards,” http://www.zdnet.com/article/build-your-own-supercomputer-out-of-raspberry-pi-boards/, last revision June 21, 2017, last accessed September 20, 2017.

[Yang13]
Yang, Xiaoyu et al, “Principles, Methodologies, and Service-Oriented Approaches for Cloud Computing”, https://books.google.co.in/books?id=gMieBQAAQBAJ&printsec=frontcover#v=onepage&q&f=false, last accessed April, 2019.

APPENDIX A

Slurm Workload Manager Code Listing

slurmkick.sh

```
#!/bin/bash
TIME0=$(date +%s%3N)
sbatch slurmbatch.sh $1 $2 $3
echo "${TIME0}" > slurm-$1-$2-time0.txt
```

slurmbatch.sh

```
#!/bin/bash
# set max wallclock time
#SBATCH --time=5-00:00:00
# num nodes
#SBATCH --nodes=1
# set name of job
#SBATCH --job-name=ep4
# mail alert at start, end and abortion of execution
#SBATCH --mail-type=ALL
# send mail to this address
#SBATCH --mail-user=futralj@gmail.com
### Run the executable
# run the application
export PATH=/bin/:${PATH}
TIME1=$(date +%s%3N)
```

```
echo "${TIME1}" > /home/student/jobs/slurm/slurm-$1-$2-
time1.txt

sar -rub 1 > /home/student/jobs/slurm/stats-$1-$2-
${HOSTNAME}.txt &

for i in $(seq -s' ' $1); do

  mpirun --mca btl ^openib
/home/student/Downloads/NPB3.3.1/NPB3.3-MPI/bin/$3

Done

pkill -f sar

TIME2=$(date +%s%3N)

echo "${TIME2}" > /home/student/jobs/slurm/slurm-$1-$2-
time2.txt
```

APPENDIX B

Portable Batch Scheduler Professional Code Listing

pbskick.sh

```
#!/bin/bash
export PATH=/bin:${PATH}
TIME0=$(date +%s%3N)
qsub -v RUNS=$1,ITER=$2,TEST=$3 pbsbatch.sh
echo "${TIME0}" > pbs-$1-$2-time0.txt
```

pbsbatch.sh

```
#!/bin/bash
#PBS -N pbs
### Merge output and error files
#PBS -j oe
### Select 1 nodes
#PBS -l select=1:ncpus=1
### Run the executable
# run the application
export PATH=/bin/:${PATH}
TIME1=$(date +%s%3N)
echo "${TIME1}" > /home/student/jobs/pbspro/pbs-$RUNS-
$ITER-time1.txt
sar -rub 1 > /home/student/jobs/pbspro/stats-${RUNS}-
${ITER}-${HOSTNAME}.txt &
for i in $(seq -s' ' $RUNS); do
```

```
  mpirun --mca btl ^openib
/home/student/Downloads/NPB3.3.1/NPB3.3-MPI/bin/$TESTdone

pkill -f sar

TIME2=$(date +%s%3N)

echo "${TIME2}" > /home/student/jobs/pbspro/pbs-$RUNS-
$ITER-time2.txt
```

APPENDIX C

Kubernetes Code Listing

daemon.yaml

```
apiVersion: extensions/v1beta1
kind: DaemonSet
metadata:
  generation: 1
  name: ssh-openmpi-worker
spec:
  revisionHistoryLimit: 10
  selector:
    matchLabels:
      app: ssh-openmpi
  template:
    metadata:
      creationTimestamp: null
      labels:
        app: ssh-openmpi
    spec:
      containers:
      - args:
        - -c
        - cp /data/id_* ~/.ssh/; chmod 644
~/.ssh/id_rsa.pub; chmod 600 ~/.ssh/id_rsa;
```

```
          cp /data/id_rsa.pub /root/.ssh/authorized_keys;
/usr/sbin/sshd; sleep 5;
          SERVERS=$(dig +short ssh-
openmpi.default.svc.cluster.local | paste -sd ','
          -); echo ${SERVERS} | ssh-keyscan -f - >
/root/.ssh/known_hosts; sleep infinity
        command:
        - /bin/sh
        image: ironmerchant/openmpi
        imagePullPolicy: Always
        name: ssh-openmpi-worker
        ports:
        - containerPort: 22
          protocol: TCP
        resources: {}
        terminationMessagePath: /dev/termination-log
        terminationMessagePolicy: File
        volumeMounts:
        - mountPath: /data
          name: ssh-openmpi-worker-volume
      dnsPolicy: ClusterFirst
      restartPolicy: Always
      schedulerName: default-scheduler
      securityContext: {}
      terminationGracePeriodSeconds: 30
      volumes:
      - hostPath:
          path: /home/student
          type: ""
        name: ssh-openmpi-worker-volume
```

```
  templateGeneration: 1
  updateStrategy:
    rollingUpdate:
      maxUnavailable: 1
    type: RollingUpdate
status:
  currentNumberScheduled: 0
  desiredNumberScheduled: 0
  numberMisscheduled: 0
  numberReady: 0
```

job.yaml.tmpl

```
apiVersion: batch/v1
kind: Job
metadata:
  name: openmpi-controller-job
spec:
  template:
    spec:
      containers:
      - name: openmpi-controller
        image: ironmerchant/openmpi
        command: ["/bin/sh"]
        args: [
          "-c",
          "/data/jobs/kube/kubebatch.sh $(NUM_ITER)
$(NUM_RUNS) $(JOB_NAME)"
        ]
        env:
```

```
        - name: "NUM_ITER"
          value: "{{NUM_ITER}}"
        - name: "NUM_RUNS"
          value: "{{NUM_RUNS}}"
        - name: "JOB_NAME"
          value: "{{JOB_NAME}}"
        ports:
        - containerPort: 22
        volumeMounts:
        - name: openmpi-controller-volume
          mountPath: /data
      nodeSelector:
        dedicated: master
      tolerations:
      - key: node-role.kubernetes.io/master
        effect: NoSchedule
      restartPolicy: Never
      volumes:
      - name: openmpi-controller-volume
        hostPath:
         path: /home/student
```

service.yaml

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: ssh-openmpi
```

```
  name: ssh-openmpi
spec:
  clusterIP: None
  ports:
  - name: ssh
    port: 22
    protocol: TCP
    targetPort: 22
  selector:
    app: ssh-openmpi
  sessionAffinity: None
  type: ClusterIP
status:
  loadBalancer: {}
```

<u>kubekick.sh</u>

```
#!/bin/bash -ex
for i in 0 1 2 3 4; do
  ssh compute-0-${i} "nohup sar -rub 1 >
/home/student/jobs/kube/n1/stats-$1-$2-compute-0-${i}.txt
&"
done
sed "s/{{NUM_ITER}}/${1}/g" job.yaml.tmpl > job.yaml
sed -i.orig "s/{{NUM_RUNS}}/${2}/g" job.yaml
sed -i.orig "s/{{JOB_NAME}}/${3}/g" job.yaml
kubectl label nodes cisvm-rocks71.ccec.unf.edu
dedicated=master || true
TIME0=$(date +%s%3N)
echo "${TIME0}" > /home/student/jobs/kube/n1/kube-$1-$2-
time0.txt
```

```
kubectl apply -f service.yaml
kubectl apply -f daemon.yaml
while [[ $(kubectl get pods | wc -l) < 6 ]]; do
  echo "Not online yet..."
  sleep 1
done
kubectl apply -f job.yaml
while [[ ! $(kubectl get pods | grep "Completed") ]]; do
  sleep 10
done
for i in 0 1 2 3 4; do
  ssh compute-0-${i} "pkill sar"
done
kubectl delete -f job.yaml
```

kubebatch.sh

```
#!/bin/bash
TIME1=$(date +%s%3N)
echo "${TIME1}" > /data/jobs/kube/n1/kube-$1-$2-time1.txt
cp /data/id_* ~/.ssh/
chmod 644 ~/.ssh/id_rsa.pub
chmod 600 ~/.ssh/id_rsa
export SERVERS=$(dig +short ssh-
openmpi.default.svc.cluster.local | paste -sd ',' -)
export SERVERS=$(echo ${SERVERS} | cut -d',' -f5-)
echo ${SERVERS} | ssh-keyscan -f - > ~/.ssh/known_hosts;
for i in $(seq -s' ' $1); do
  mpirun --mca btl ^openib\
```

```
        --host ${SERVERS}\
        --allow-run-as-root\
        /tmp/NPB3.3.1/NPB3.3-MPI/bin/$3
done
TIME2=$(date +%s%3N)
echo "${TIME2}" > /data/jobs/kube/n1/kube-$1-$2-time2.txt
```

APPENDIX D

Beowulf Code Listing

plainkick.sh

```
#!/bin/bash
export
LD_LIBRARY_PATH=/usr/lib64:/usr/lib64/openmpi:${LD_LIBRARY_
PATH}
TIME0=$(date +%s%3N)
echo "${TIME0}" > plain-$1-$2-time0.txt
./plainbatch.sh $1 $2 $3 &> log-$2.txt &
```

plainbatch.sh

```
#!/bin/bash
export PATH=/bin/:/usr/bin:${PATH}

ssh compute-0-0 "nohup sar -rub 1 >
/home/student/jobs/plain/stats-$1-$2-${HOSTNAME}.txt &"
TIME1=$(date +%s%3N)
echo "${TIME1}" > plain-$1-$2-time1.txt
for i in $(seq -s' ' $1); do
  mpirun --mca btl ^openib\
         --host compute-0-0\
         /home/student/Downloads/NPB3.3.1/NPB3.3-MPI/bin/$3
done
TIME2=$(date +%s%3N)
```

```
echo "${TIME2}" > plain-$1-$2-time2.txt
ssh compute-0-0 'pkill sar'
```

VITA

Jeremy Futral has a Bachelor of Applied Science in Information Technology from the University of North Florida, 2012 and expects to receive a Master of Science in Computer Science from University of North Florida in Spring 2019. Dr. Roger Eggen is Jeremy Futral's current thesis advisor.