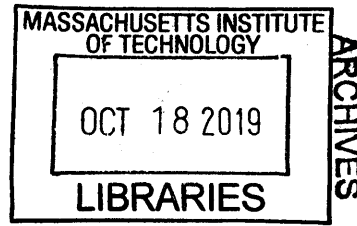


A Container-Based Lightweight Fault Tolerance Framework for High Performance Computing Workloads

Workloads
by
Mohamad Sindi



B.S. Computer Science, University of Kansas, 2003
M.S. Computer Science, George Washington University, 2009

Submitted to the Center for Computational Engineering and the Department of Civil and Environmental Engineering in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy in Computational Science & Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

SEPTEMBER 2019

© 2019 Massachusetts Institute of Technology. All rights reserved

Signature redacted

Author

Center for Computational Engineering
August 16, 2019


Signature redacted

Certified by


John R. Williams
Professor of Civil and Environmental Engineering
Thesis Supervisor

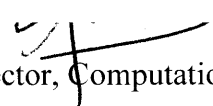
Signature redacted

Accepted by


Colette L. Heald
Professor of Civil and Environmental Engineering
Chair, Graduate Program Committee

Signature redacted

Accepted by


Nicolas Hadjiconstantinou
Co-Director, Computational Science and Engineering

A Container-Based Lightweight Fault Tolerance Framework for High Performance Computing Workloads

by

Mohamad Sindi

Submitted to the Center for Computational Engineering and the Department of Civil and Environmental Engineering on August 16, 2019 in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy in Computational Science & Engineering

Abstract

According to the latest world's top 500 supercomputers list, ~90% of the top High Performance Computing (HPC) systems are based on commodity hardware clusters, which are typically designed for performance rather than reliability. The Mean Time Between Failures (MTBF) for some current petascale systems has been reported to be several days, while studies estimate it may be less than 60 minutes for future exascale systems. One of the largest studies on HPC system failures showed that more than 50% of failures were due to hardware, and that failure rates grew with system size. Hence, running extended workloads on such systems is becoming more challenging as system sizes grow. In this work, we design and implement a lightweight fault tolerance framework to improve the sustainability of running workloads on HPC clusters. The framework mainly includes a fault prediction component and a remedy component.

The fault prediction component is implemented using a parallel algorithm that proactively predicts hardware issues with no overhead. This allows remedial actions to be taken before failures impact workloads. The algorithm uses machine learning applied to supercomputer system logs. We test it on actual logs from systems from Sandia National Laboratories (SNL). The massive logs come from three supercomputers and consist of ~750 million logs (~86 GB data). The algorithm is also tested online on our test cluster. We demonstrate the algorithm's high accuracy and performance in predicting cluster nodes with potential issues.

The remedy component is implemented using the Linux container technology. Container technology has proven its success in the microservices domain. We adapt it towards HPC workloads to make use of its resilience potential. By running workloads

inside containers, we are able to migrate workloads from nodes predicted to have hardware issues, to healthy nodes while workloads are running. This does not introduce any major interruption or performance overhead to the workload, nor require application modification. We test with multiple real HPC applications that use the Message Passing Interface (MPI) standard. Tests are performed on various cluster platforms using different MPI types. Results demonstrate successful migration of HPC workloads, while maintaining integrity of results produced.

Thesis Supervisor: John R. Williams

Title: Professor of Civil and Environmental Engineering

بِسْمِ اللَّهِ الرَّحْمَنِ الرَّحِيمِ

(The Arabic texts included are prayers thanking God, my parents, and my wife)

Acknowledgments

It is an understatement to say that pursuing a PhD at MIT is challenging. I am grateful to the many who helped throughout this journey. First, I am grateful to God, the most gracious, the most merciful, for his countless blessings and for giving me the opportunity to pursue my PhD at the best engineering school in the world. I am grateful for the patience and guidance he has bestowed upon me to succeed in this journey. It is a dream to be part of this magnificent institute, and with his grace, I am beyond humbled to be a part of it.

"رَبِّ أَوْزَعْنِي أَنْ أَشْكُرَ نِعْمَتَكَ الَّتِي أَنْعَمْتَ عَلَيَّ وَعَلَى وَالِدَيَّ وَأَنْ أَعْمَلَ صَالِحًا تَرْضَاهُ"

I would like to thank my advisor Professor John R. Williams. Without his continuous support and encouragement, this journey would have been impossible to complete. I appreciate the valuable guidance he provided me whenever I needed it, and the flexibility he gave me to experiment and pursue new research ideas. I am truly grateful for his mentorship and the opportunities he has given me. I am also appreciative and thankful to all of my committee members. I thank Professor Saurabh Amin for kindly serving as my committee's chair. His insightful suggestions and recommendations always helped me improve the quality of my writings and work, be it from our very first proposal meeting discussions, and until our final committee meeting. I would like to thank Christopher Hill, whom I now consider a friend and a colleague in our domain of work in HPC. I

appreciate his invitations to attend research-related HPC conferences, which I enjoyed attending along with his company. I also appreciate his guidance and bringing my attention to related research that is taking place in other local universities. His review of the thesis's first draft was beyond thorough and helped improve the quality of the work. I am thankful to Dr. Abel Sanchez, whom I think is one of the kindest academics I have met on campus. I have learned a lot from the classes he taught at MIT along with Professor Williams. I was academically exposed to practical examples in machine learning and classification problems, which inspired part of my research. The classes were also helpful in learning valuable soft skills, such as documenting and presenting research work through videos, which I ended up utilizing as part of this work.

I would like to thank my dear parents, Othman Sindi and Najwa Rafie, and my siblings. Even though we were thousands of miles apart, their perpetual encouragement, prayers, loving, and caring is what has gotten me through challenging times during my studies. I pray God to bless them and reward them for all what they have provided me.

"وَقُلْ رَبِّ ارْحَمْهُمَا كَمَا رَبَّيْتَنِي صَغِيرًا"

This journey would have not been tolerable without my loving wife, Nada Alamro, being by my side. Thank you for being there when I needed it, and for your sincere love and caring throughout the past few years. I pray God to bless you and bring comfort to you as you have brought it upon me.

"رَبَّنَا هَبْ لَنَا مِنْ أَزْوَاجِنَا وَذُرِّيَّاتِنَا قُرَّةَ أَعْيُنٍ"

I would like to thank Saudi Aramco for all of the support they have been providing me throughout my work with them for the past 20 years, and for their scholarship to pursue my PhD. I specifically would like to thank Saudi Aramco Fellow Dr. Ali Dogru.

Dr. Dogru always made sure to ask about me whenever he was visiting Cambridge. The delightful meetings we had together during his visits were always encouraging and supportive. He has been a mentor and a father figure to me, and I am forever grateful to him.

My technical research would have not been easily done without the generous support I have received from various research entities. I would like to thank the Sandia National Laboratories (SNL) and Dr. Adam Oliner for providing the HPC systems data, which was crucial to this research. It is not easy to come across such data, and I am very grateful for their support and efforts in providing it. I would like to thank Imperial College London, especially Dr. Tim Greaves, for their support with the Fluidity CFD code. Dr. Greaves went beyond expectations in providing me a specific Fluidity build for MPICH and creating an installation repository just for me, which was not publically available at that time. I would like to thank the team supporting the OPM Flow simulator, especially Dr. Arne Morten Kvarving, Dr. Alf Birger Rustad, and Dr. Markus Blatt. At the time of testing, the Red Hat version of the OPM Flow application was not MPI enabled, and the gentlemen helped providing me a new build repository that supported MPI. I would also like to thank Amazon and Schlumberger for providing me research grants and allowing me to use their computational platforms and software as part of my research.

I truly appreciate our CEE and CCE administrative staff, especially Kiley Clapper, Max Martelli, Kathleen Briana, Sarah Smith, and Kate Nelson for all of their support. Lastly and mostly, I would like to thank all of my friends and colleagues in MIT's Geonumerics group, CCES, Saudi Aramco Dhahran, and ASC Cambridge, there are too many of them to list.

Contents

Chapter 1 Introduction	16
1.1 Problem Background and Literature Review.....	16
1.2 Scope of Work	24
1.3 Contributions.....	31
1.4 Dissertation Outline	32
Chapter 2 Fault Prediction Component.....	34
2.1 The Data.....	34
2.2 The Algorithm.....	37
2.2.1 Algorithm Overview	37
2.2.2 Machine Learning Model of the Algorithm.....	44
2.2.3 Training Data for Machine Learning	45
2.2.4 Algorithm Summary	47
2.3 Results and Discussion	48
2.4 Summary	58

Chapter 3 Remedy Component	61
3.1 Overview of Proposed Remedy Environment	62
3.2 System Design	63
3.2.1 Hardware Cluster Setup	63
3.2.2 Software Setup	64
3.2.3 Container Setup.....	67
3.2.4 Network Setup	68
3.2.5 Summary of System Design.....	69
3.3 Selected HPC Applications for Testing	70
3.4 Testing and Results	72
3.4.1 Container Performance Benchmarks	73
3.4.2 Container Migration Testing.....	86
3.4.3 Results Integrity Check.....	91
3.4.4 Container Demo Videos.....	96
3.5 Challenges and Solutions.....	99
3.6 Summary	105
Chapter 4 Framework Integration and Testing	106
4.1 Overall View of Integrated Framework.....	106
4.2 Integration with the HPC Resource Manager	108
4.3 Full Cycle Testing of Framework	113
4.4 Summary	117
Chapter 5 Conclusions and Future Work	118
5.1 Summary and Conclusions	118
5.2 Future Work	120
Bibliography	122

List of Figures

Figure 1.1: Summary comparison of fault prediction method to previous related work .. 28

Figure 1.2: Summary comparison of remedy method to previous related work 30

Figure 2.1: Example Linux syslog message showing date, time, node name, and log message..... 36

Figure 2.2: Parallel processing architecture based on Spark framework..... 40

Figure 2.3: Using Bayes theorem to calculate probabilities of nodes being good or bad. 46

Figure 2.4: Example log line manually tagged by SNL with EXT_FS to indicate potential disk issue..... 46

Figure 2.5: Pseudo-code summarizing parallel algorithm to predict nodes with hardware issues 48

Figure 2.6: Confusion matrix for classification results..... 53

Figure 2.7: Confusion matrix for classification results..... 53

Figure 2.8: Example log line containing insignificant text token of a disk sector number 54

Figure 2.9: Confusion matrix for classification results.....	55
Figure 2.10: Parallel algorithm performance on multiple cores when processing supercomputer logs	57
Figure 2.11: Parallel algorithm scalability on multiple cores when processing supercomputer logs	58
Figure 3.1: Summary of system design architecture.....	70
Figure 3.2: Point-to-point MPI benchmarks for 1G network	75
Figure 3.3: Point-to-point MPI benchmarks for 10G network	76
Figure 3.4: Point-to-point MPI benchmarks for 25G network	77
Figure 3.5: Collective MPI benchmarks for 1G network	78
Figure 3.6: Collective MPI benchmarks for 10G network	79
Figure 3.7: Collective MPI benchmarks for 25G network	80
Figure 3.8: Benchmarks on m4.2xlarge instances	83
Figure 3.9: Benchmarks on c5.9xlarge instances.....	84
Figure 3.10: Benchmarks on m4.16xlarge and i3.metal instances	85
Figure 3.11: Overview of container migration test.....	87
Figure 3.12: Container migration steps.....	89
Figure 3.13: Results integrity check for the Palabos simulation	93
Figure 3.14: Results integrity check for the ECLIPSE simulator.....	93
Figure 3.15: Results integrity check for the Fluidity simulation	94
Figure 3.16: Md5sum integrity check for gif files produced by the Palabos simulation..	95
Figure 3.17: Md5sum integrity check for ParaView pvtu files produced by the Fluidity simulation.....	95

Figure 4.1: Full cycle of overall framework environment..... 108

Figure 4.2: Overall TORQUE resource manager setup with containers 111

Figure 4.3: Prologue script for TORQUE resource manager to generate containers host
file 112

Figure 4.4: Example TORQUE script to launch MPI job on containers 112

Figure 4.5: Summary of monitoring daemon performed tasks 114

List of Tables

Table 2.1: Breakdown of system log sizes.....	37
Table 2.2: Number of parallel RDD partitions created and sizes	41
Table 2.3: Reduced error logs after removing redundancy.....	42
Table 2.4: Effect of RDD repartitioning on algorithm's performance	43
Table 2.5: Prediction results for nodes with memory issues	50
Table 2.6: Prediction results for nodes with disk issues	50
Table 2.7: Newly discovered categories of hardware issues	51
Table 2.8: Tuned Linux and Spark system parameters.....	56
Table 3.1: Summary of tested HPC applications	71
Table 3.2: Input models used for application testing.....	82
Table 3.3: Average container migration times (using standard gpt2 disks)	90

Table 3.4: Average container migration times (using enhanced io1 disks).....	90
Table 3.5: Video links for container demos.....	96
Table 4.1: Examples of injected logs to simulate hardware error precursor	115
Table 4.2: Video demos for framework full cycle tests.....	116

Chapter 1

Introduction

1.1 Problem Background and Literature Review

High Performance Computing (HPC) supercomputers are used in many domains and industries. For example, such systems are heavily used in the oil and gas industry for performing reservoir simulations and seismic studies. The healthy operation of these systems in fields like the oil and gas industry is crucial to having a smooth workflow during the process of oil exploration and production. Such systems, however, are typically built of commodity hardware that is designed for performance rather than reliability. According to the latest June 2019 world's top 500 supercomputers list, 90.6% of the world's top supercomputers are based on commodity hardware HPC clusters [1]. Today's most powerful HPC supercomputers are running in petascale computing power with thousands of compute nodes consisting of several millions of compute processing

cores [2]. The Mean Time Between Failures (MTBF) for some of these petascale systems has been reported to be several days [3], while studies estimate the MTBF for future exascale systems to be less than 60 minutes [4], [5]. One of the largest studies conducted on HPC system failures was done on data collected for 9 years from more than 22 production HPC systems installed at Los Alamos National Lab (LANL) [6]. The study showed that more than 50% of the failures were due to hardware (e.g. memory, disk, and CPU failures) and that failure rates grew with system size. Hence, running sustainable workloads on such systems is becoming more challenging as the size of the HPC system grows. As exascale systems will become reality in the next few years, the need for better fault tolerance methods in HPC systems is becoming more crucial than ever.

The current de facto standard method to tolerate HPC workload failures is the application checkpoint-restart (CR) mechanism. In this scenario, the application periodically saves its state (i.e. checkpoint) in case a failure occurs, and once a failure occurs, the application can be restarted from its last checkpoint. The Berkeley Lab Checkpoint/Restart (BLCR) library [7] is popularly used to provide such mechanism for MPI-based applications [8], which requires installing additional Linux kernel modules on the system. In general, CR can have several limitations when it comes to large-scale HPC workloads. CR can have an impact on the performance and scalability of the application. It can introduce an undesirable overhead due to the needed writing and reading of data to storage systems to save and retrieve the checkpoint data. High CR overheads ultimately may lead to lowering the parallel efficiency of the workload as compute processors can spend more time doing the checkpoint process, rather than actually doing the workload's computation. Several studies showed that for a petascale system, the CR overhead could

be more than 50% of the total application execution time [9]–[11]. Other studies also estimate that future exascale systems could have a MTBF that is even smaller than the time required to complete a CR process [12]. The CR files can also consume significant storage space, which is undesirable for computer centers with limited storage or imposed disk usage quota per user. The CR mechanism is also more of a reactive fault tolerance method rather than a proactive one, meaning it will remedy the fault after the fact that your workload had failed due to a system failure, which leads to the interruption of your workload.

Other studies [13] have attempted to do proactive fault detection locally on HPC cluster nodes using third-party tools like the Intelligent Platform Management Interface (IPMI) tool [14], which also has its own limitations. In order for IPMI to work, the HPC node requires having an additional piece of hardware installed, called the baseboard management controller (BMC) chip, which provides the monitoring intelligence to the IPMI tool. The IPMI software will also need to be set up on each node individually as it is not a native standard tool in the operating system. Such mechanism used to continuously monitor the health of nodes may also introduce an undesirable overhead on the system, as the fault probing procedure is performed locally on each node.

Studies such as [15], [16] have also attempted to do fault detection by analyzing system health logs serially with less emphasis on having a time-efficient approach to do the analyzation. As mentioned previously, the MTBF for future exascale systems is estimated to be less than one hour. Hence, the use of serial algorithms to analyze massive system logs will not be efficient timewise as we are restricted with a relatively small MTBF value. Also to reduce the latency of obtaining results when doing online detection

on live supercomputer systems, study [16] used a sliding bounded history approach to limit the amount of log history information being processed, hence this could lead to an undesirable omitting or overlooking of some valuable data when analyzing the system health logs as a whole. Study [17], which is also related to [16], reported that their log analysis detection algorithm had a precision of 75% only. In addition, they only reported the precision of the algorithm when run on a limited amount of data that was collected for only a short period of three weeks from one system (8.3 million logs out of hundreds of millions). The frequency of running the detection algorithm was also limited to every 15 minutes, which could be insufficient if applied to a large system with a small MTBF.

Other studies [13], [18] used third-party tools such as Ganglia [19] and Nagios [20] to monitor system resources to detect system problems. However, such approaches also have their own challenges. The periodic polling done to monitor the system's health by local monitoring agents such as the Ganglia daemons can have an impact on the performance and scalability of the application in a large-scale HPC environment. Having these monitoring agents continuously running on the compute nodes can consume some of the system's compute and network resources instead of having them fully dedicated towards the HPC workload running on the system [21], [22]. Also monitoring tools such as Nagios can have limited analysis capabilities (e.g. simple pattern matching) and are infamous for being difficult to set up and maintain.

Another study [23] tried predicting failures based on data mainly from systems running the Windows operating system, which is not a popular operating system in the HPC domain. As illustrated in the latest June 2019 world's top 500 supercomputers list, none of the world's top systems are running on Windows [1]. In addition, the average

accuracy of the prediction methods reported was less than 80%. Methods used in the study such as the Hidden Markov Model (HMM) are also known for their undesired computational complexity, especially as the size of the data being processed grows larger. High computational costs during predictions are undesirable because we are restricted with small MTBF values for large systems, hence we cannot afford to use a mechanism that is time demanding during the prediction process.

Other studies like [24], [25] were limited in targeting only one specific type of failures, hard disks, and their prediction was based on analyzing data collected from a third-party tool called SMART [26]. In [24], they used a binning process to limit the amount of data being processed which can lead to some loss of information in the data being analyzed. In [25], they also used a sliding window approach, similar to [16], to limit the amount of data being processed when doing predictions, which may also lead to overlooking valuable data as a whole when predicting the likelihood of a failure. The study used the Support Vector Machine (SVM) approach to do the predictions, and in their best case, only 73% accuracy was achieved in predictions.

Study [27] tried predicting disk failures in a disk storage system. The predictions were based on analyzing kernel-level event logs that were collected from the storage system. The study used a Naive Bayes approach to do the predictions and they were able to predict all bad disks with only one false positive. However, the storage system being analyzed was small in size (only seventeen storage nodes) and the dataset used for testing was oversimplified and very small (only four failures in entire data). This study also targeted storage systems only and not complete HPC systems. The logs analyzed also seemed to be storage-vendor specific and not standard.

Another approach that has been studied to tolerate failures in HPC environments is the use of virtual machine (VM) technologies and their migration capabilities. Studies have previously investigated the use of VMs to migrate workloads from an unhealthy machine to another as a proactive fault tolerance measure [13], [28]. However, VMs have their own challenges in terms of the performance overhead they introduce. Traditional out of the box VM technologies can introduce a resource and performance overhead due to the virtual emulation of devices (e.g. network adapter) through a hypervisor. Some earlier studies showed that the network latency with VMs can be more than twice in some cases compared to a native system [29], [30]. Nevertheless, VM technologies have been evolving over time and efforts to improve their performance continue. Huang et al. [31] proposed a prototype I/O bypass mechanism to reduce the I/O overhead involved with VMs. Another more recent I/O bypass method is the single root input/output virtualization (SR-IOV) specification [32]. Liu [33] evaluated the performance of SR-IOV on 10 Gigabit networks and still reported a network latency overhead of around 41% comparing to native performance. The study also reported that SR-IOV introduced a 46% CPU utilization overhead on the system it ran on. Jose et al. [34] evaluated the performance of the Ohio State University (OSU) Micro-Benchmarks [35] with SR-IOV over InfiniBand [36], where collective benchmarks did not perform well, while point-to-point benchmarks were acceptable. Musleh et al. [37] later attempted to improve the performance of SR-IOV over InfiniBand by performing none trivial experimental tuning of network interrupts. The SR-IOV technology also currently still has its own challenges. SR-IOV is limited to having a hardware/software environment that supports this feature [38]. Another limitation is that SR-IOV can affect the capability of VM migration, which

is one of the main desirable features of VMs. For example, the current Red Hat Enterprise Linux (RHEL) 7 documentation clearly states that VM migration is not supported when SR-IOV is enabled [39]. Nevertheless, research attempts are currently ongoing to enable VM migration with SR-IOV, however such functionality it is still not mainstream and could require custom modifications to hypervisors or the use of custom Linux kernel modules and additional nonstandard third-party libraries [40], [41].

Another type of migration that has been studied in the domain of HPC is the process-level migration. In this case, the actual MPI processes are targeted for migration, rather than migrating a VM that contains them. Reber [42] attempted to use the Checkpoint-Restore in Userspace (CRIU) tool [43] to perform parallel process migration, however, they were not successful in migrating parallel MPI processes and were only able to migrate the serial none parallel version of their application. Another tool targeted to checkpoint and migrate MPI processes is the Distributed MultiThreaded CheckPointing (DMTCP) package [44]. The tool requires launching an additional coordinator process on one of the hosts involved in the computational environment. The application binaries will also have to be started through the tool's proxy launching script, which may add an overhead for the running application, especially if the application involves system calls. In addition, the tool does not work with graphical applications using X-Windows extensions like OpenGL, and currently cannot checkpoint graphical xterm terminals.

The current emerging alternative to using VMs is the container technology. Throughout the past recent years, several studies have looked into adopting containers into HPC environments. Jacobsen et al. [45] did some early work in deploying Docker-based containers in HPC environments. As a result, they implemented a system called

Shifter, which enables launching user-defined images of containers. Kurtzer et al. [46] developed a type of containers called Singularity with HPC in mind. Its implementation mitigates security implications that are present in other container types such as Docker. A few studies have also compared the performance of containers to those of VMs and native systems. Soltesz et al. [47] and Felter et al. [48] reported that the performance of containers was close to native and always outperformed the VMs, especially with I/O intensive and latency-sensitive workloads. Xavier et al. [49] compared the performance of three container types (LXC, Docker, and Singularity) to a native system and reported that Singularity performed the best. Ruiz et al. [50] evaluated the performance of the NPB benchmarks [51] in an LXC container environment over a 10 Gigabit network. They reported a performance overhead of up to 30% on some of the network intense runs. Their container network setup, however, was using the bridged networking scheme. This is different from the host-routed networking scheme used in this work, which relatively has a lower overhead. Zacharov et al. [52] even looked into evaluating the performance of Nvidia Docker containers when running scientific workloads on Nvidia general-purpose graphics processing units (GPGPUs). The performance overhead comparing container to native was less than 4% in the worst-case scenario. More recent studies from Ohio State University's Network-Based Computing (NBC) Laboratory have also looked into using containers in HPC [53]–[55]. This included performing various benchmarks and investigating methods to improve the performance of HPC applications on container-based environments. These studies were mainly performed in an InfiniBand environment using the MVAPICH MPI library.

Compared to the well-explored domain of VM migration, container migration is a young research topic that has not been extensively explored yet, especially in the scope of HPC. A few studies, however, have attempted to perform container migration in Linux environments. Pickartz et al. [56] implemented a prototype libvirt-based custom Linux driver to enable container migration with the help of the CRIU tool. However, the study did not seem to test migration with a real application, but rather with an artificially induced memory load on the system. Another study proposed a mechanism to migrate Docker containers based on a logging and replay method [57]. They used a nonstandard tool called ReVirt [58] to accomplish this. The migrated application, however, was a standalone web application and not HPC related. A more recent study looked into migrating containers using the CRIU tool [59]. However, the migrated container didn't have a real application running in it either, but rather only had a simple run of a serial Linux tool called memhog [60], which just allocates memory on a system.

1.2 Scope of Work

Based on our literature analysis in the previous section, we believe that proactive fault tolerance methods for HPC systems need to be improved in terms of fault prediction accuracy, performance (i.e. time needed to do fault prediction), and overhead reduction on the HPC workload running on the system. Hence, our proposed fault tolerance framework differs from previous work as it aims to overcome the limitations discussed in the literature review section.

In this work, we design and implement a lightweight fault tolerance framework for HPC systems with the aim to improve the sustainability of running workloads on HPC

clusters. The framework consists mainly of two components. The first is a fault prediction component, which proactively predicts hardware issues in the HPC system. The second is a remedy component, which takes remedial actions towards the HPC workloads running on the affected system.

The fault prediction component is implemented using a fast and accurate parallel algorithm that proactively predicts hardware failures in HPC systems with no overhead. Such predictions can alert system users and allow them to take remedial actions before failures impact workloads. The algorithm uses a machine learning approach for predictions when applied to supercomputer system logs. We test the algorithm on actual logs obtained from HPC systems from Sandia National Laboratories (SNL). The massive logs are generated by three supercomputers and consist of ~750 million logs, which equal ~86 GB of text data. The algorithm is also tested online on our test cluster. We demonstrate our algorithm's high accuracy and performance in predicting cluster compute nodes with potential issues.

The work done to implement the fault prediction component of our fault tolerance framework differs in several aspects from the related work discussed in the literature review section. We designed the fault prediction algorithm to run externally from the nodes and it uses Linux's native syslog service as its data source for log probing, hence no performance overhead is imposed locally on any of the compute nodes of the HPC system.

The algorithm is also designed to run in parallel in a time-efficient manner, which is significantly less than the estimated MTBF of future exascale systems. It also uses the Naive Bayes method, which is known to have a low linear computational complexity. In

addition to the algorithm being computationally efficient, this efficiency does not compromise the accuracy of the predictions as we target to achieve an accuracy greater than 95% when it comes to detecting true positives along with having low false positives and false negatives. The algorithm is also able to process massive supercomputer system logs entirely without having to sample or omit any valuable data.

Our work also involves analyzing a significant amount of real failures from Linux HPC supercomputers used in national labs and does not involve any artificial data or data from uncommon operating systems. The capabilities of our algorithm are also demonstrated on the complete data collected over a substantial period averaging around a year's long of data collection. Figure 1.1 shows a pictorial summary comparing how our fault prediction work differs from previous work discussed in the literature review section.

The second component of our fault tolerance framework is the remedy component. It is implemented using the Linux container technology. The container technology has proven its success in the microservices domain (e.g. webservers) as a scalable and lightweight technology used in data centers such as Google's. In our work, we adapt the container technology towards HPC workloads to make use of its resilience potential. By running HPC workloads inside containers, we are capable of migrating these workloads from compute nodes anticipating hardware problems, to healthy spare compute nodes while the workload is running. The migration process is orchestrated with CRIU. The container environment does not introduce any major interruption or performance overhead to the running workload. It does not require any application modification either. We test the remedy component with various real HPC applications that are MPI-based. Tests are performed on clusters with different hardware node specs, network

interconnects, and MPI implementations. Results demonstrate successful migration of HPC workloads inside containers with minimal interruption, while maintaining the integrity of the results produced.

We have prepared and shared several annotated YouTube videos to demonstrate the container migration of the various HPC workloads tested. This includes standard computational workloads, as well as workloads that produce in-situ visualization during the migration. To the best of our knowledge, we believe such video demonstration involving CRIU container migration of real HPC workloads is a first.

In addition to the container migration testing, this work performs comprehensive benchmarks comparing the performance of the container environment to the native system. The benchmarks are performed using various real HPC applications. Benchmark results show that running such applications inside containers provides a performance almost identical to the native system with negligible overhead.

This Work

Fault Prediction Method:

- Fault prediction algorithm runs externally from compute nodes with no overhead.
- Parallel algorithm with linear computational complexity (Naive Bayes method) doing predictions in a time significantly less than estimated MTBF (i.e. < 60 seconds).
- Algorithm tested on ~750 million real system logs.
- Prediction accuracy on massive size data logs > 95%.
- Algorithm analyzes entire data without having to sample or omit any data.
- Data analyzed covers substantial collection period averaging ~1 year.

Previous Related Work

Fault Prediction Method:

- Methods use local probing of compute nodes and third party agents causing overhead.
- Serial methods or methods with higher computational complexity, with less emphasis on a time-efficient approach to comply with challenging MTBF.
- Tests done on significantly smaller amounts of log data (a few million in best case).
- Prediction accuracy on medium size data logs ~70-80%, and > 95% on small oversimplified data logs.
- Uses data sampling methods that may lead to omitting or overlooking valuable data.
- Data analyzed covers shorter collection periods.

Figure 1.1: Summary comparison of fault prediction method to previous related work

The work done to implement the remedy component of our fault tolerance framework differs in several aspects from the related work discussed in the literature review section. All of the container migration related work we came across was done using serial or artificial workloads. To the best of our knowledge, we believe this work is the first in the HPC domain to demonstrate successful migration of distributed MPI-based real HPC workloads using CRIU and containers. We use real HPC applications from different domains to test the container migration and performance. Most of the related work, with either VMs or containers, used generic benchmarks such as the High Performance LINPACK (HPL) [61] or the NAS Parallel Benchmarks (NPB) [51], which are still valuable tests, but not truly reflective of real HPC applications. In addition, most of the related work focus their study and solution on a single implementation of MPI (e.g. MVAPICH), a single type of interconnect (e.g. InfiniBand), or a single type of machine specs (e.g. fixed memory size). However in this work, we cover a broader scope of study. We test our container environment with three of the popularly used MPI implementations, MPICH [62], Open MPI [63], and Intel MPI [64]. We test three different interconnects which include, 1, 10, and 25 Gigabit Ethernet networks. We also test on four different machine types that vary in specs (e.g. different memory architectures, memory sizes, and CPU cores). Figure 1.2 shows a pictorial summary comparing how our remedy component work differs from previous work discussed in the literature review section.

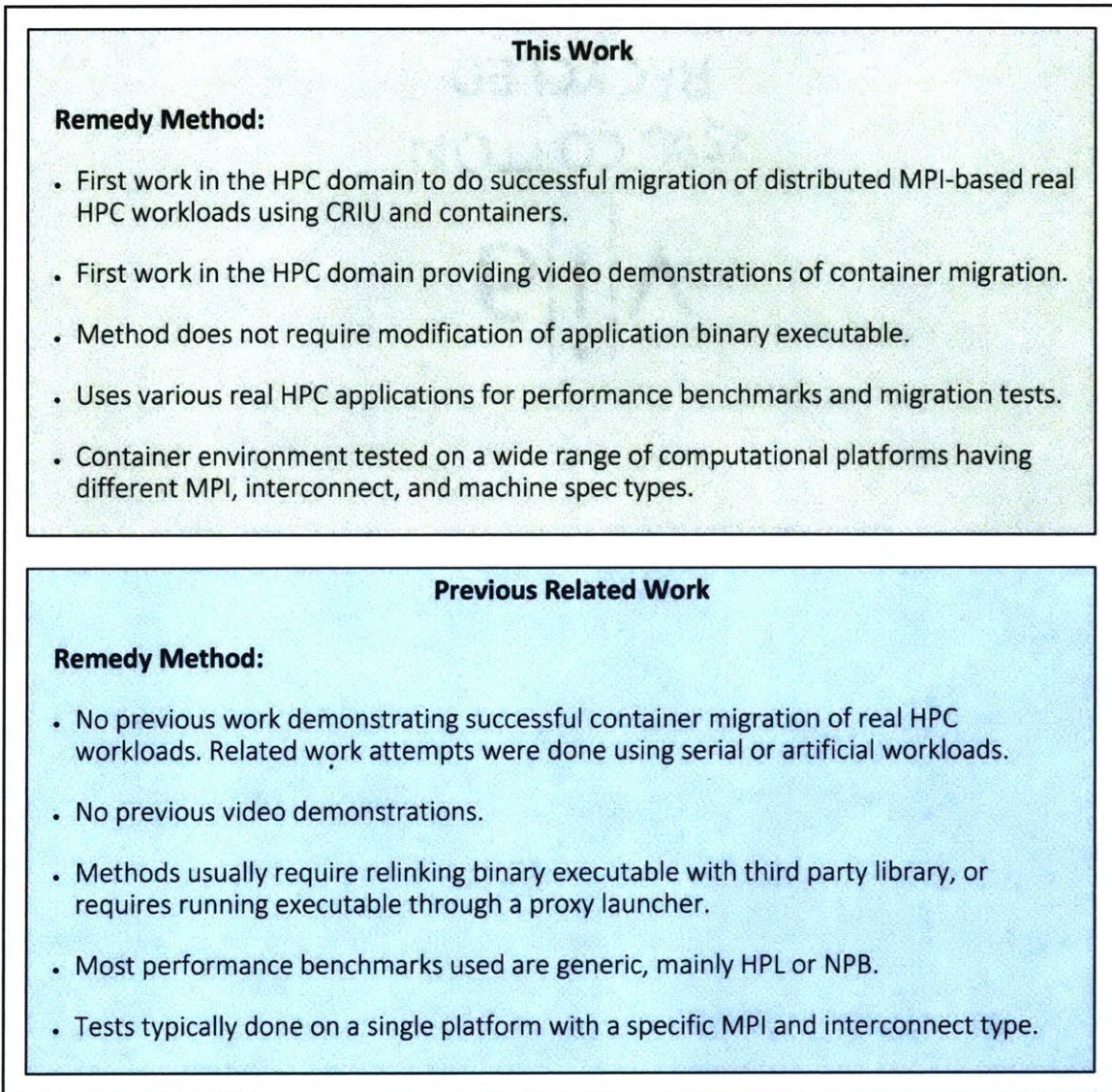


Figure 1.2: Summary comparison of remedy method to previous related work

1.3 Contributions

The main contributions of this work can be summarized as follows:

- 1) We provide a proactive fault tolerance method for HPC systems that is improved in terms of fault prediction accuracy, performance (i.e. time needed to do fault prediction), and overhead reduction on workload compared to previous related work.
- 2) The use of the container technology in the HPC domain as a fault tolerance mechanism is a field that has not been thoroughly explored yet. To the best of our knowledge, we believe this work is the first in this domain to demonstrate successful migration of MPI-based real HPC workloads using CRIU and containers.
- 3) We implement a code fix to the open-source CRIU library to enable successful migration of containers having NFS mounts inside of them.
- 4) We prepare and share several annotated YouTube videos to demonstrate the container migration of the various HPC workloads tested. This includes standard computational workloads, as well as workloads that produce in-situ visualization during the migration. To the best of our knowledge, we believe such video demonstration is also a first in the HPC domain.
- 5) Related work benchmarking containers mainly use generic HPC benchmarks (e.g. HPL and NPB) on a single system. In this work, we do comprehensive benchmarks using various real scientific HPC applications on a wide range of computational platforms. The benchmarks provide results comparing the performance on containers compared to the native system and quantify the overhead involved.
- 6) As the use of containers in HPC is a relatively young topic, we believe that the challenges we faced with the wide range of tests performed, and the solutions adopted, are all valuable experiences to share with the HPC community.

1.4 Dissertation Outline

In Chapter 2, we go over the implementation of the fault prediction component of the fault tolerance framework. We start by discussing the specifics of the data being analyzed, followed by the details of the prediction algorithm. We discuss the machine learning algorithm, as well as the training data used. We also provide pseudo code to summarize the algorithm's functionality. We share the results obtained with the algorithm and discuss them in terms of accuracy, performance, and scalability. Finally, a summary and conclusion is presented.

In Chapter 3, we go over the proposed remedy environment that will be used once faults are predicted on a system. We start with an overview of the environment, followed by the system design details of the implementation. This includes the setup details of the hardware, software, containers, and network. We also present the various HPC applications that are being tested with the environment. After that, we discuss the tests performed and the results obtained. We also discuss the challenges faced during the testing and the solutions that were adopted to overcome them. A summary and conclusion is presented at the end.

In Chapter 4, we go over the integration of the fault prediction component, the remedy component, the HPC resource manager, and our custom monitoring daemon, which all ultimately compose the full HPC fault tolerance framework. We first present an overview of the fully integrated framework. Next, we discuss our method to integrate the HPC resource manager into the container environment. We follow by discussing our tests and demonstrating the framework's successful full functionality. We conclude with a summary afterwards.

Finally, Chapter 5 concludes the overall work and summarizes our main contributions. We also discuss future research directions for testing our framework on other computational platforms.

Chapter 2

Fault Prediction Component

In this chapter, we discuss the implementation details of the fault prediction component of the framework. First, we present the details of the analyzed data. Next, we discuss the details of the prediction algorithm. The machine learning algorithm is presented, and the preparation of the training data is discussed as well. We summarize the algorithm with pseudo code afterwards. Results obtained with the algorithm are then discussed. Finally, a summary and conclusion is presented.

2.1 The Data

Our work targets HPC systems running the Linux operating system, as it is currently the most widely used operating system for HPC systems. The latest June 2019 list of the world's top 500 supercomputers shows that 100% of the systems are running on Linux [1]. The main data source that HPC system administrators consider looking at to

investigate health issues with an HPC system is the Linux system logs. The Linux operating system by default collects these system logs locally on each node of an HPC cluster in a text format. Such operation is considered native and standard for the Linux operating system to perform. This makes it a desirable source of data as there is no added overhead for collecting the data locally on the nodes, unlike non-native third-party methods discussed previously in the literature review section. These system logs are insightful to monitor issues related to the system, which could help in predicting hardware failures. The system health data logs collected locally from nodes are typically forwarded and aggregated in a repository for archiving purposes in data centers. Linux's standard 'syslog' logging daemon provides a lightweight out of the box functionality to forward such logs to a central archival repository [65]. The log forwarding is usually done through the node's general Ethernet network. This does not interfere with the workload taking place on the cluster since the workload is usually running on a separate high bandwidth network such as an InfiniBand network. These logs are typically collected in HPC centers for security purposes as they contain details of activities taking place on the system.

The collected system logs follow the open-source syslog Berkeley Software Distribution (BSD) protocol standard used by Linux [66]. This means that logs within a certain Linux distribution would have a standard and similar structure in terms of content. Figure 2.1 shows an example line of such logs where it contains information such as the date and time of when the log was generated, name of node that generated the log, and the actual log message.

```
2015.11.12.07:52:29 node123 kernel: EXT3-fs error (device sda3) in ext3_dirty_inode: IO failure
```

Figure 2.1: Example Linux syslog message showing date, time, node name, and log message

Obtaining such system logs for research purposes can be very challenging. HPC systems are mainly used by national labs or industrial entities. These logs usually contain sensitive data such as user names, IP addresses, and other security-related data. This makes it difficult for HPC centers to publically share the data. Centers would typically have to anonymize the data before sharing it. This can be a tedious and time-consuming process, which centers may be unwilling to go through. In addition, HPC centers usually would like to publically show the positive side of the systems they own, for example, how powerful or stable the systems are, and where they rank among the world's top supercomputers. Hence, centers can be reluctant to sharing system failure data as it could affect the global image of these prestigious systems. HPC hardware vendors may also be hesitant to share failure data, as it is a bad advertisement for their hardware products. We tried to contact personnel from some US national labs to acquire such data. Personnel from two national labs were contacted, Los Alamos National Labs (LANL) and Sandia National Labs (SNL). We chose to contact these two entities as they had a history of providing HPC related data for researches. LANL confirmed that they collect and have such data logs from their systems, however unfortunately they no longer release such collections of data for a number of reasons they did not want to go into. As for SNL, we were able to get anonymized system logs from three of their former supercomputers.

The data provided by SNL was collected from three supercomputers, Thunderbird, Spirit, and Liberty. Data obtained from the supercomputers consisted of ~750 million

logs and equal to ~86 GB of text data. The collection period for these logs varied among the three supercomputers and consisted of several hundred days of log collection for each, with an average of around 372 days of log collection. Just to have a relative perspective of how large this data is, the size of the entire recent English language Wikipedia data is around 127 GB [67]. We counted the number of raw system logs from each of the three SNL supercomputers and Table 2.1 shows the approximate size breakdown for each.

Supercomputer Name	Total Log Size (GB)	Number of Raw Log Lines
Thunderbird	27	211,212,192
Spirit	34	272,298,969
Liberty	25	265,569,231

Table 2.1: Breakdown of system log sizes

2.2 The Algorithm

In this section, the implementation of the fault prediction algorithm is presented. First, an overview of the algorithm is described, followed by a section presenting the machine learning model of the algorithm. After that, we describe the training data used with the machine learning approach. Finally, we provide a summary of the algorithm’s steps.

2.2.1 Algorithm Overview

The next step after obtaining the system logs was to develop an algorithm to automate predicting nodes with faulty hardware as a proactive measure. With an estimated MTBF of less than one hour for future exascale systems, we wanted to design an algorithm that can process massive system logs consisting of hundreds of millions of lines of logs in only a matter of a few minutes or less. With that in mind, we planned to develop a

parallel algorithm that runs on a multi-processor platform in order to process this large amount of data in a timely manner. Some exascale plans refer to designing systems in two different scenarios, a slim node, and a fat node scenario [11], [68]. In both cases, the system will provide a total of a billion computing cores. With slim nodes, the system would have 1,000,000 nodes, each having 1000 cores. In the fat node scenario, the system would have 100,000 nodes, each having 10,000 cores. We can get a rough estimate of the size of system logs produced by such a massive system based on the average amount of logs produced per day per node by the supercomputers at SNL. If we went with the worst-case scenario in terms of the number of nodes (i.e. slim nodes setup), this gave us an estimate of around 60 GB of system logs data produced per day for such an exascale system. With this in mind, we targeted that our algorithm should be able to process similar amounts of data in a matter of a few minutes or less.

We decided to use the open-source Apache Spark framework as our development framework for the algorithm [69]. Spark provides utilities to write programs that can run in parallel on a local multi-core system, and even on a distributed clustered system. It is also currently being used by large enterprise entities such as Amazon and Yahoo. The framework gives you the flexibility to write your code in several options of programming languages including Scala, Java, and Python and provides APIs for all of these languages to access the parallel functions available in Spark. It also provides the capability of loading big data and having it processed fast in-memory once it is loaded. This was a very desirable feature for us as it supported our need to process tens of gigabytes of supercomputer system logs data in a scalable and parallel fashion. As Spark provides APIs for several programming languages, we have chosen Scala as our language of

implementation [70]. This choice was made because the Spark parallel functions beneath the hood were also implemented in Scala, hence using Scala gave the best performance out of the three programming language options. We also discussed this choice with the inventor of Spark, MIT's Professor Matei Zaharia, who also confirmed that Scala is the recommended language to be used with Spark for performance.

In order to have a better understanding of how our algorithm works, we briefly introduce some of the parallel processing concepts that we utilize from the Spark framework. The main data element which enables parallelism in Spark is called a resilient distributed dataset (RDD) [71]. We can think of an RDD as a collection of partitions of our original dataset in which each partition can be operated on in parallel by a computer's processing core. This is a similar concept to the MapReduce model used in big data processing [72], in which we have a map step that splits a dataset into smaller parts and works on them, then we have a reduce step that collects and combines the solutions from the smaller parts to produce a result. However in the case of RDDs, most of the processing of these partitions can be done in-memory. This can improve the performance of the algorithm significantly compared to models like MapReduce, where processing the data can involve slower frequent accessing of data from disk. Another concept that Spark uses for parallel processing is the concept of a driver and worker. A driver can be thought of as your main program that is responsible for scheduling tasks on the workers to process the partitions of your dataset, and collecting the results of those tasks. The workers, on the other hand, are the executors of those scheduled tasks, which we can think of as the multiple computing cores working in parallel. Figure 2.2 illustrates the concept of RDDs and the driver/worker model with the Spark framework.

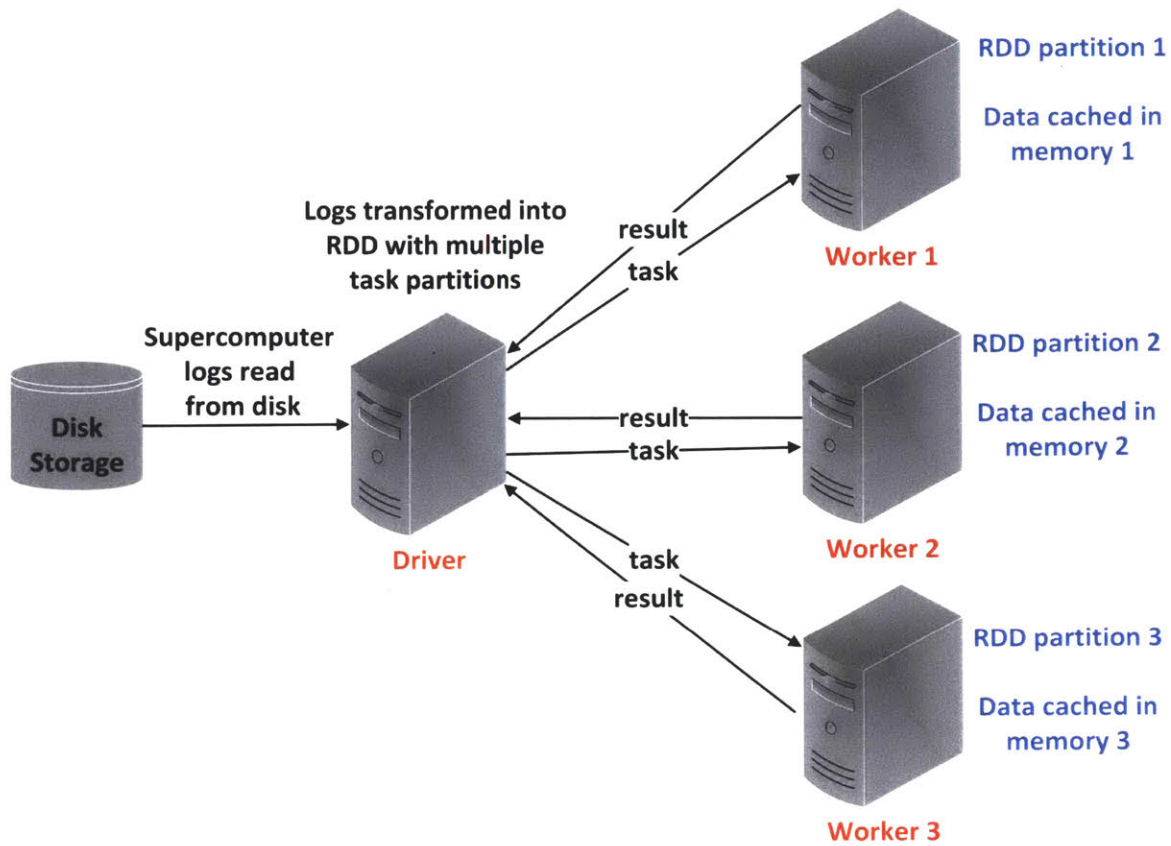


Figure 2.2: Parallel processing architecture based on Spark framework

The algorithm starts by loading the syslog data for one of the three supercomputers from disk, and then decomposes it into smaller partitions using the Spark library. This is the first step needed in order to be able to process the large log file in parallel by multiple computer cores. The number of partitions calculated by Spark mainly depends on the size of the original data file. The number of partitions is not necessarily a one-to-one mapping with the number of parallel processing cores available. Instead, the number of partitions is an integer multiple of the number of processing cores available. This results in an oversubscription of tasks on the processing cores, which usually leads to a better performance and cores' utilization since a single task typically does not fully utilize an entire processing core on its own. Table 2.2 shows the number of partitions initially

calculated for the three supercomputer log files, where each partition is on average roughly 32 MB in size.

Supercomputer Name	Total Log Size (GB)	# RDD Partitions	Size of Each Partition (MB)
Thunderbird	27	828	32.61
Spirit	34	1,018	33.40
Liberty	25	769	32.51

Table 2.2: Number of parallel RDD partitions created and sizes

The structure of each log line follows a tabular format consisting of columns. Hence, once the log dataset has been transformed into parallel RDD partitions, the algorithm then transforms each line of logs into a comma-separated values (CSV) structure, which is also done in parallel. The CSV structure gives us more flexibility later on to manipulate the data.

We notice that the content of such supercomputer logs can have a significant amount of redundancy in it by nature. For example, if a compute node was having a hard disk issue and was producing a log message similar to that in Figure 2.1, each time the node tries to access that troubled disk, the same message can be reported in the logs repeatedly. As we have shown in Table 2.1, the number of logs for each supercomputer was quite significant and averaging to around a quarter of a billion logs per supercomputer. With this in mind, the algorithm next tries to reduce the dimensionality of the data by removing the redundancy in it. We use a concept borrowed from the databases world and the Structured Query Language (SQL) language to accomplish this. The SQL language provides a “distinct” statement, which is typically used to return distinct values from a database table. Our algorithm applies a similar distinct concept but to an RDD structure that we preprocess in a tuple format containing the node name and the logs. This enables

us to remove the redundant error entries from the log files in a parallel fashion. By removing the redundancy, this improves the overall processing time of the data, and reduces the memory footprint on the processing cores. We were amazed that this step of the algorithm reduced the size of the datasets dramatically with an average of around 86% reduction in log size. Table 2.3 shows the results of applying the reduction step of the algorithm to the datasets from the three supercomputers.

Supercomputer Name	# Raw Log Lines	# Reduced Error Log Lines	Reduction %
Thunderbird	211,212,192	52,452,811	75
Spirit	272,298,969	34,453,762	87
Liberty	265,569,231	11,589,710	96

Table 2.3: Reduced error logs after removing redundancy

To improve the performance even further, the algorithm also tunes the number of RDD partitions that Spark originally created and reduces it depending on the percentage of log reduction. For example, in the case of Thunderbird, only 25% of the data remained after the reduction, hence the algorithm reduces the number of RDD partitions generated by Spark to 25% as well, resulting in 207 partitions instead of the original 828. The repartitioning improved the average overall runtime of the algorithm by around 34% when running on 16 computing cores as shown in Table 2.4. Having too many partitions for a reduced amount of data could cause a communication overhead and not enough amount of work for the processes working on the partitions, hence our repartitioning technique helps improve the performance as the data is reduced.

Supercomputer Name	Elapsed Time with Original Partitioning (seconds)	Elapsed Time with Tuned Partitioning (seconds)	Performance Improvement %
Thunderbird	66	47	40
Spirit	71	54	31
Liberty	41	31	32

Table 2.4: Effect of RDD repartitioning on algorithm's performance

Looking at the content of the raw logs, we can also see that the order of logs coming from the different nodes of a cluster is scattered, meaning that logs from different nodes will be interleaved between each other. The next step that the algorithm performs is to scan the hundreds of millions of interleaved log lines and group the logs coming from each specific node together. This process is done in parallel and produces tuple objects containing a node name and a list of all its logs. At this stage, we have a reduced preprocessed data structure containing each node's name and the logs associated with it.

Compute nodes that compose a cluster typically have identical or similar specs in terms of hardware and software installed on them, hence they tend to produce very similar syslog system logs. In addition, as we have previously shown in Figure 2.1, system logs provided by the Linux syslog standard are descriptive of the events taking place and can have meaningful word semantics as they are meant to be read by system administrators. The vocabulary used in the logs in terms of errors is also somehow limited. With this in mind, the algorithm tries to predict the faultiness of nodes using the statistical characteristics of the logs. We use a machine learning approach to do the predictions, which is covered in more details in the following section.

2.2.2 Machine Learning Model of the Algorithm

The algorithm uses a Naive Bayes (NB) classifier model with a Bernoulli setup to predict node failures [73]. The model is considered a supervised machine learning approach to do predictions. This means we train the model by exposing it to some previous precursor failure/non-failure data from one of the HPC systems provided by SNL, then the model tries to predict if nodes in other HPC system logs would have failures or not. In the domain of machine learning, this is considered a classification problem that has a discrete number of outcomes. In this case, it is a binary outcome of either “0” meaning that the node was predicted as not faulty, or “1” meaning the node was predicted as a faulty node. This is similar to the concept of using NB classifiers for spam detection in emails [74]. As an analogy, we can think of the content of an email to be classified as spam or not to be equivalent to the content of logs coming from a specific node in the HPC system, which needs to be classified as faulty or not. The classification of the text “document” is then done based on its words content. The training data containing previous precursor failures/non-failures from HPC systems will ultimately build two sets of vocabulary for us, one for faulty nodes and one for normal nodes. The Bernoulli model of the NB classifier will use a binary mechanism (0 or 1) to flag the absence or presence of words in a document in one of the two sets of vocabulary we have. The Bernoulli model does not care about how frequently a word occurs in a document, it just cares if the word occurred or not. This also explicitly models the absence of words in a document. This is different from the typical NB classifier used in classification problems, usually referred to as the Multinomial NB model, where the frequency of word occurrences is taken into account [73]. The decision to use the Bernoulli model instead of

the Multinomial model for the NB classifier was influenced after testing both models, where the Bernoulli model produced more accurate results with less false positives. Also since we reduced the data by removing the redundancy in it, using the Multinomial NB model, which depends on word frequencies to do predictions, might not be suitable to use on data that has been intentionally reduced by removing redundant words from it.

The NB classifier relies on the simple concept of conditional probability. It tells us the probability that a hypothesis is true if some event has happened [75]. In our case, given logs of a specific node to be classified, it assigns to this node probabilities for the two possible classes, good or bad node. Using Bayes' theorem, the probability can be calculated as in Figure 2.3. The NB classifier has linear complexity requiring a single pass over the data for training and testing. This is a desired characteristic that makes it a commonly used classification method due to its performance efficiency compared to other classifier methods, which may be iterative in nature and more expensive when it comes to computational complexity [73].

2.2.3 Training Data for Machine Learning

As SNL has provided us with logs of three of their supercomputers, we chose a portion of the logs from the Thunderbird supercomputer to be the basis of our training data for the machine learning algorithm. We chose this supercomputer because it was the largest from the three in terms of the number of nodes, hence it had relatively more hardware failures compared to the other two and provided richer training data. A previous study was done by SNL to categorize the types of errors found in these logs [15]. The process involved manually adding tag labels to errors found in the logs using visual inspection and simple Linux shell scripting. We use these tags to extract the training data

we need for our machine learning algorithm. The training data needs to include logs from nodes that had failures (i.e. bad nodes) and logs from nodes that did not have failures (i.e. good nodes). For the bad nodes, we picked training data covering two types of common hardware failures, disk and memory. SNL labeled logs containing potential disk issues with the label “EXT_FS”, while logs containing potential memory issues were labeled with the label “ECC”. Figure 2.4 shows an example of such log lines tagged with the label “EXT_FS” indicating a potential disk issue. The training data was around 10% of Thunderbird’s node data.

- What is the probability that a node’s given logs D belong to a class C?
- i.e. What is $P(C|D)$? (a posterior probability computed by Bayes Theorem)
- C can be one of two classes, “b” for bad node, or “g” for good node.

$$P(C|D) = \frac{P(D|C) P(C)}{P(D)} \quad (\text{denominator is fixed, } P(D) \neq 0)$$

- With some manipulations of probabilities, we can find the fixed denominator to be:

$$P(D) = P(D | C = "b") P(C = "b") + P(D | C = "g") P(C = "g")$$

- The terms $P(C = "b")$ and $P(C = "g")$ are called “class prior” probabilities which are easily found from the training data. They are the fractions of the bad and good nodes from the total number of nodes in the training data.
- The remaining conditional probability terms can be calculated also based on the training data as follows:

$$P(D | C = "b") = \prod_i P(w_i | C = "b"), \quad P(D | C = "g") = \prod_i P(w_i | C = "g")$$

- The term $P(w_i | C)$ is the probability that the i^{th} word of a new given document occurs in a document from class C’s training data.

Figure 2.3: Using Bayes theorem to calculate probabilities of nodes being good or bad

```
2015.11.12.07:52:29 EXT_FS node123 kernel: EXT3-fs error (device sda3) in ext3_dirty_inode: IO failure
```

Figure 2.4: Example log line manually tagged by SNL with EXT_FS to indicate potential disk issue

Preparing the training data was a complex task on its own due to the massiveness of the data and the fact that the raw logs are interleaved and not grouped by nodes. This

required us to write another aiding program in Spark to produce the training data. As a result, we were able to produce a text file containing training data for bad nodes and another for good nodes. Each line entry in the training data corresponds to data coming from one specific node with redundancy removed.

Since the training data are logs containing mainly English words, we also had to normalize this data. This required stemming the data by removing punctuations (e.g. full stops and commas), brackets, standalone digits, and stop words (e.g. “is” and “are”). We used the Apache Lucene English Analyzer library to perform the normalization [76]. This process to normalize the data is also applied to the test data for consistency.

Furthermore, machine learning algorithms typically expect to take their input training data in a numerical format, however, the training data are logs containing mainly English words. Hence, we had to convert these English words into digits using a hashing function. The produced hashed results can then finally be used to train the model.

We next test the algorithm to predict node failures in the remaining never seen system logs of the three supercomputer systems. Ultimately in a live system, once the potentially faulty nodes are identified, the algorithm is to report these nodes to the resource manager software running on the cluster. The resource manager then flags them as potentially faulty nodes and takes them out of the resources pool to be further investigated by the system administrators.

2.2.4 Algorithm Summary

Figure 2.5 provides pseudo-code that summarizes the overall parallel algorithm to predict nodes with hardware issues in supercomputer system logs, as it was described in details in the previous sections of Chapter 2.

Algorithm: Parallel Faulty-Node Prediction Algorithm Based on Spark	
Training data input:	Preprocessed training data from the Thunderbird supercomputer
Testing data input:	Supercomputer Linux syslog system data file (aggregated from all cluster nodes)
Output:	List of highly likely nodes having potential for hardware failures
-----Training Part-----	
load training data files and convert into RDD <i>training_data</i> stored in-memory	
for all RDD partitions in <i>training_data</i> do in parallel (on multi-cores)	
normalize the training data by stemming it	
hash the normalized training data to convert it from words to numerical digits	
train the Naive Bayes Bernoulli model using the training data and produce the <i>NB_Bernoulli_model</i>	
end for	
-----Prediction Part-----	
load syslog data file and convert it into RDD <i>raw_logs</i> stored in-memory	
calculate # of produced RDD partitions <i>rdd_partitions</i>	
set tabular schema headers identifying columns in <i>raw_logs</i>	
for all RDD partitions in <i>raw_logs</i> do in parallel (on multi-cores)	
apply CSV formatting for <i>raw_logs</i>	
save in-memory CSV formatted node name and logs in RDD <i>csv_logs</i>	
for all RDD partitions in <i>csv_logs</i> do in parallel	
apply dimensionality reduction to remove redundancy and save in <i>reduced_logs</i>	
end for	
tune # of <i>rdd_partitions</i> based on the percentage of log reduction	
for all RDD tuned partitions in <i>reduced_logs</i> do in parallel	
group scattered node logs by node name and save in <i>grouped_logs</i>	
end for	
for all RDD node partitions in <i>grouped_logs</i> do in parallel	
normalize the logs by stemming them	
hash the normalized logs to convert them from words to numerical digits and save it in <i>test_data</i>	
end for	
for all RDD node partitions in <i>test_data</i> do in parallel	
predict node failures in test data using the <i>NB_Bernoulli_model</i> and save results in <i>predicted_faulty_nodes</i>	
end for	
end for	
report predicted faulty nodes to cluster workload manager through APIs for faulty nodes bookkeeping	

Figure 2.5: Pseudo-code summarizing parallel algorithm to predict nodes with hardware issues

2.3 Results and Discussion

In the section, we will discuss the results obtained from applying our fault prediction algorithm to the logs from SNL. We focus on three aspects, accuracy, performance, and scalability of the algorithm.

SNL’s previous study [15] focused on correctly characterizing and quantifying errors found in these logs by manually labeling the errors through visual inspection and simple scripts. As a result of this great effort, SNL provided tag labeled logs as a baseline for

comparison. We use these tags to help us determine the accuracy of our algorithm when it comes to predicting faulty nodes. The algorithm does not read these tags when it is predicting faulty nodes. We compare the algorithm's results to those that were produced by the manual tagging done by SNL.

We identified the unique nodes that were manually tagged by SNL for two common hardware issues (i.e. disk and memory), then compared it to those reported by our algorithm. As explained previously, the training data we used to build our machine learning NB prediction model was based on logs from the Thunderbird supercomputer. With this in mind, we first tested the NB model with data from the two other supercomputers, Spirit and Liberty. In addition, as we only used a small portion of the overall logs from Thunderbird to build our training data (~10%), we tested the NB model using the remaining logs from Thunderbird as well. By looking at results in Table 2.5 and Table 2.6, we see that the algorithm was able to predict all the troubled nodes that were manually tagged by SNL. In addition, it flagged 6 new nodes with potential faulty memory, and 13 new nodes with potential faulty disks, which were missed by SNL's tagging. Looking further at the logs of the 19 newly discovered nodes, 16 of them were true positives with legitimate errors, highlighted in italics in Table 2.5 and Table 2.6, while only 3 nodes, highlighted in bold in Table 2.6, were false positives.

It is also important to note that in the case of the 3 false positives, the logs of those nodes contained logs that could be mistaken as hardware disk errors, but they were actually not. For example, in the case of node liberty2p, the logs were showing an input/output (I/O) error due to a software file locking issue on a user's home directory on an external storage system. Such an error looks very similar to a local hardware disk issue

as both could generate I/O errors. This observation gave us even more confidence in our machine learning algorithm, as those false positives could justifiably be confused as local hardware failures even by an unexperienced human system administrator analyzing the logs visually.

Supercomputer Name	# of Nodes with Faulty Memory (Manually Tagged)	# of Nodes with Faulty Memory (Algorithm Predicted)	Newly Predicted Nodes
Thunderbird	81	84	<i>cn624, dn261, badmin1</i>
Spirit	0	2	<i>sn105, sn138</i>
Liberty	0	1	<i>liberty1p</i>

Table 2.5: Prediction results for nodes with memory issues

Supercomputer Name	# of Nodes with Faulty Disks (Manually Tagged)	# of Nodes with Faulty Disks (Algorithm Predicted)	Newly Predicted Nodes
Thunderbird	82	85	<i>en122, an1002, bn410</i>
Spirit	9	14	<i>sn105, sn111, sn216, shpnfs, sadmin2</i>
Liberty	0	5	<i>ln156, ln30, ln72, liberty2p, ladmin2</i>

Table 2.6: Prediction results for nodes with disk issues

In addition, even though our training data only contained logs of issues from two common categories (memory and disk), the machine learning algorithm successfully learned from such patterns and discovered new never before seen categories. This included 6 new faulty nodes with 3 new categories of hardware issues (power supply, processor, and management controller battery issues) as illustrated in Table 2.7, which were all true positives.

# of Nodes with Faulty Power Supply (Algorithm Predicted)	# of Nodes with Faulty Processor (Algorithm Predicted)	# of Nodes with Faulty MC Battery (Algorithm Predicted)	Newly Predicted Nodes
3	1	2	PS: <i>badmin2, eadmin1, eadmin2</i> Processor: <i>bn697</i> MC Battery: <i>cn600, dn539</i>

Table 2.7: Newly discovered categories of hardware issues

Overall, the algorithm was run on ~750 million logs collected from more than 5000 nodes. The algorithm predicted 197 nodes as potentially having hardware issues. Out of the 197 nodes, 172 nodes (true positives) matched all those manually tagged by SNL. In addition, the algorithm was successful in identifying 22 newly discovered nodes not tagged by SNL, which were all true positives. Only 3 nodes out of the 197 predicted were false positives. In machine learning classification problems, a confusion matrix is commonly used to describe the accuracy of the classification model [77]. From the confusion matrix, we can calculate the overall error rate of the prediction algorithm and its accuracy. Figure 2.6 shows the confusion matrix and the calculated overall error rate and accuracy for our results.

Furthermore, we wanted to see how the process of normalizing the data affected the accuracy of the algorithm. With this in mind, we ran the same experiment again but without normalizing the data, Figure 2.7 shows the results that were obtained. The number of false positives didn't change significantly compared to the run with the normalized data, 3 nodes vs. 6 nodes, however we noticed that a significant amount of bad nodes, 119 nodes, were missed and were incorrectly classified as good nodes. Having such a large number of false negative nodes can indeed affect the stability of the workload running in an HPC environment, as these troublesome nodes would go without

being noticed. Hence, our data normalization process technique was crucial in improving the accuracy of the prediction and reducing the data complexity. We believe that normalizing the data removes many text tokens from the logs that are insignificant for the prediction process. The presence of such text tokens also negatively affects the accuracy of the prediction as we have seen in our results. For example, as part of the normalizing process, we stem text tokens containing standalone digits found in the logs. Due to the nature of these system logs, we have seen that there is a great amount of such insignificant standalone digits in the logs. For instance, if a node had a potential disk issue, you could get a log similar to that in Figure 2.8. In this case, the log contains a standalone digit token “27156729” which represents a sector number in the potentially troubled disk. These sector numbers can be unique from one log to another, and having millions of these logs coming from different nodes may generate millions of unique disk sector numbers, which are insignificant to the predictions. This complicates the prediction process as these tokens cause the training and test data to grow with no need, and evidently affect the prediction’s accuracy.

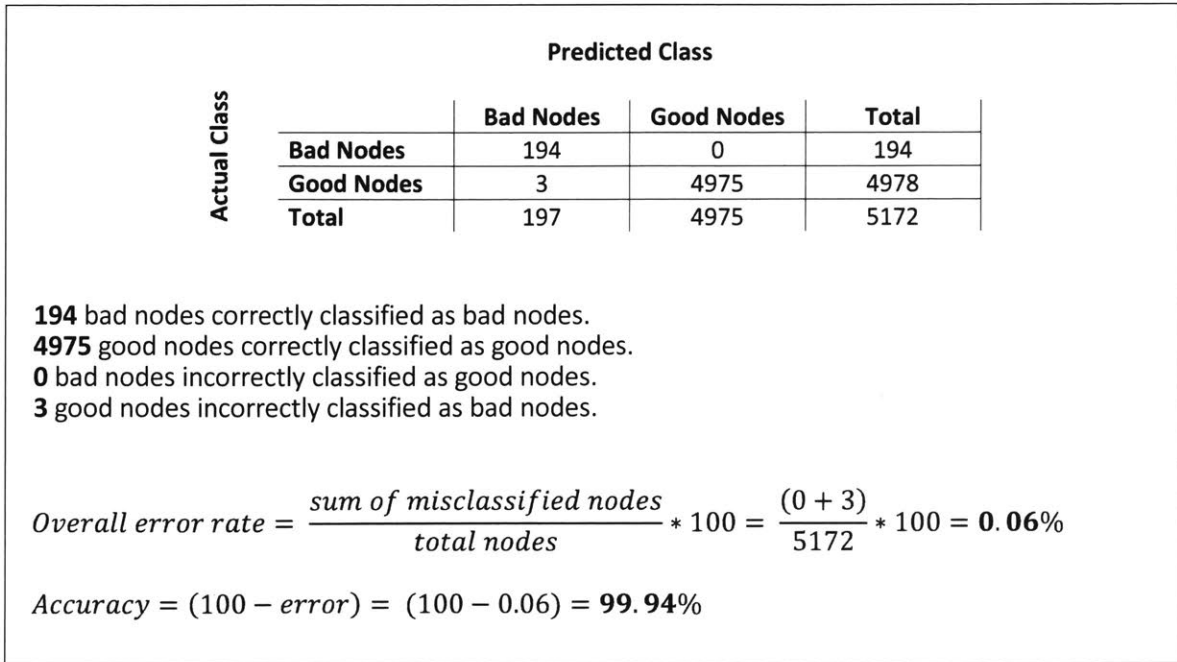


Figure 2.6: Confusion matrix for classification results
(NB Bernoulli model with data normalization)

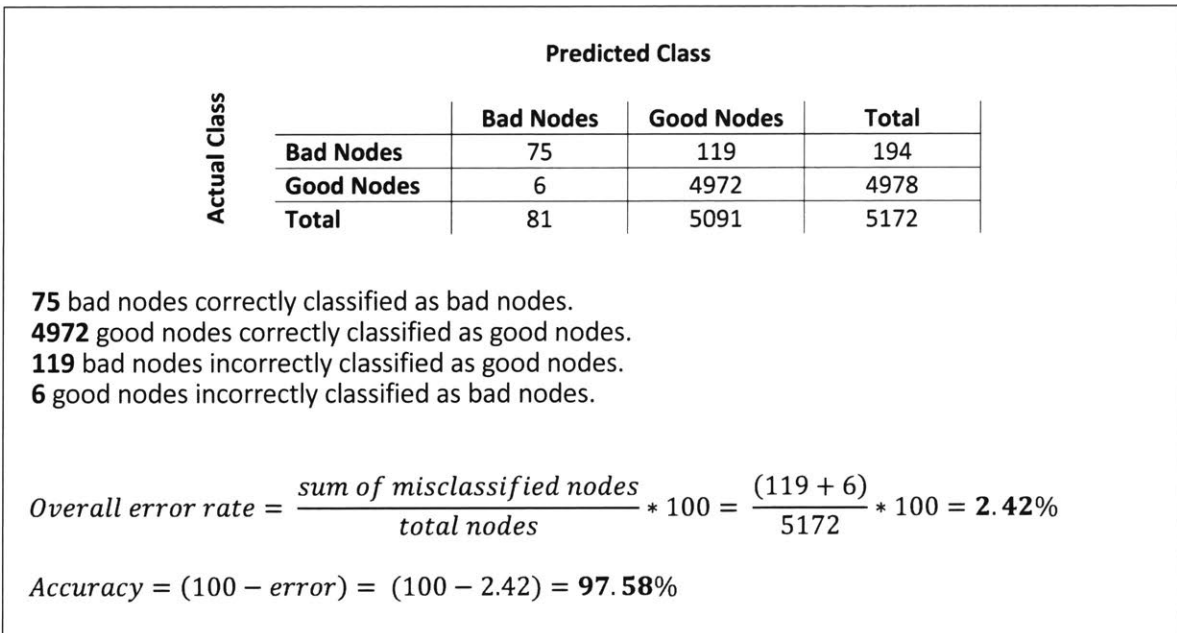


Figure 2.7: Confusion matrix for classification results
(NB Bernoulli model without data normalization)

```
2015.12.07.04:00:03 node123 kernel: end_request: I/O error, dev sda, sector 27156729
```

Figure 2.8: Example log line containing insignificant text token of a disk sector number

In addition, the decision to use the Bernoulli model instead of the Multinomial model for the NB classifier was influenced after testing both models with our algorithm. We observed that the Bernoulli model produced more accurate results with less false positives. The Bernoulli model takes into account if a text token is present or not in the document being classified, without taking into account how frequent that text token occurs in the document. The Multinomial model, on the other hand, takes into account the frequency of text token occurrences when classifying the documents. Figure 2.6 and Figure 2.9 compare the results obtained and their accuracy for both Bernoulli and Multinomial models when used with the NB classifier. The results show that the Multinomial model produced far more false positives compared to the Bernoulli model, 773 vs. 3, while the number of true positives was the same for both models. This large number of false positives reduced the prediction accuracy with a noticeable 15% drop in accuracy. Having such a large number of good nodes flagged incorrectly as bad nodes can affect the workflow of the HPC environment. This can be very disruptive to the HPC system administrators that are monitoring the health of these systems and a waste of time and effort to investigate such a large number of false red flags. We believe that our approach of combining the Bernoulli model of the NB classifier along with our methods of preprocessing the data in terms of reduction, normalization, and grouping, has led to having a high prediction accuracy, even when processing massive amounts of data.

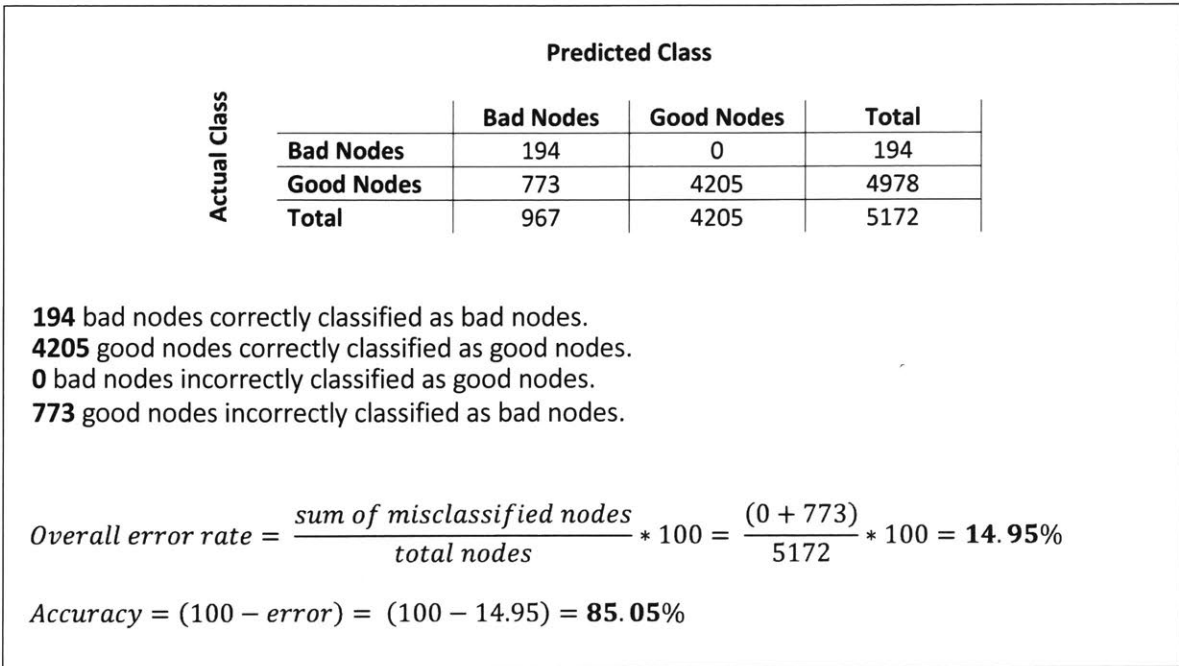


Figure 2.9: Confusion matrix for classification results
 (NB Multinomial model with data normalization)

Next, we share the performance and scalability results of the parallel algorithm when running on 1 core and up to 16 cores. We ran the algorithm on an AWS Memory Optimized node of instance type “r3.8xlarge” [78]. The node had 16 computing cores, 244 GB of memory, and a solid-state disk (SSD) of size 320 GB that stored the supercomputer logs. The Linux operating system used was the 64-bit Red Hat Enterprise Linux (RHEL) 7 distribution.

Initially when we ran the algorithm using the default standard settings of the Linux operating system and Spark, the runs failed with errors coming from both Linux and Spark. The runs were performed using a normal Linux system user (none root). The Linux errors were related to exceeding the allowable memory usage and the maximum number of open files for a normal user. The Spark errors were related to running out of space when writing temporary scratch files on disk. Investigating this further, we had to

tune some of the default Linux and Spark system settings in order for the system to be capable of handling the large system log files we were processing. From the Linux operating system side, we had to tune the memlock and nofile system settings. The memlock parameter specifies how much memory the system user can lock into their address space, while the nofile parameter can be used to increase the maximum number of open file descriptors for the user [79]. The default values for those Linux system parameters were too low and had to be tuned to resolve the limitation issue. From the Spark system side, we had to adjust the “spark.local.dir” parameter that is used to point to the temporary directory used for scratch space in Spark. Instead of using the default /tmp directory, which was limited in space, we pointed that parameter to a larger directory on an SSD disk in order to accommodate more data without running out of space. Also to tune the performance of Spark even further, we adjusted the “spark.driver.maxResultSize” parameter from its default value of 1 gigabyte to 20 gigabytes. This increased the total size of results collected from all partitions being processed, which helped improve the performance when running the algorithm [80]. Table 2.8 shows a summary of the system parameters we tuned, in order for the algorithm to run with the large amount of data being processed.

Parameter	Tuned Value
memlock	unlimited
nofile	63536
spark.local.dir	/home/tmp (20GB SSD partition)
spark.driver.maxResultSize	20g

Table 2.8: Tuned Linux and Spark system parameters

Figure 2.10 and Figure 2.11 show the performance and scalability results of the parallel algorithm after tuning the parameters. Figure 2.10 shows that the elapsed time for

processing the logs is directly proportional to the size of the supercomputer log files mentioned previously in Table 2.1. When running on 16 cores, the average elapsed time to do the full prediction cycle was only 44 seconds. Figure 2.11 shows the scalability results of the algorithm when processing the three supercomputer logs, which illustrates a similar behavior for all three. The scalability was close to linear with the smaller number of cores, and then it deviates as we go up to 16 cores. We believe the scalability is affected by several factors. This includes the increased communication and scheduling overhead that the Spark library will have as we increase the number of cores, the size of the log file being analyzed, and the redundancy of the data in the logs. Comparing these scalability results to similar scalability results published by Databricks, a leading company providing commercial support for Spark that was founded by the inventor of Spark, our scalability results look very similar to those published by Databricks [81].

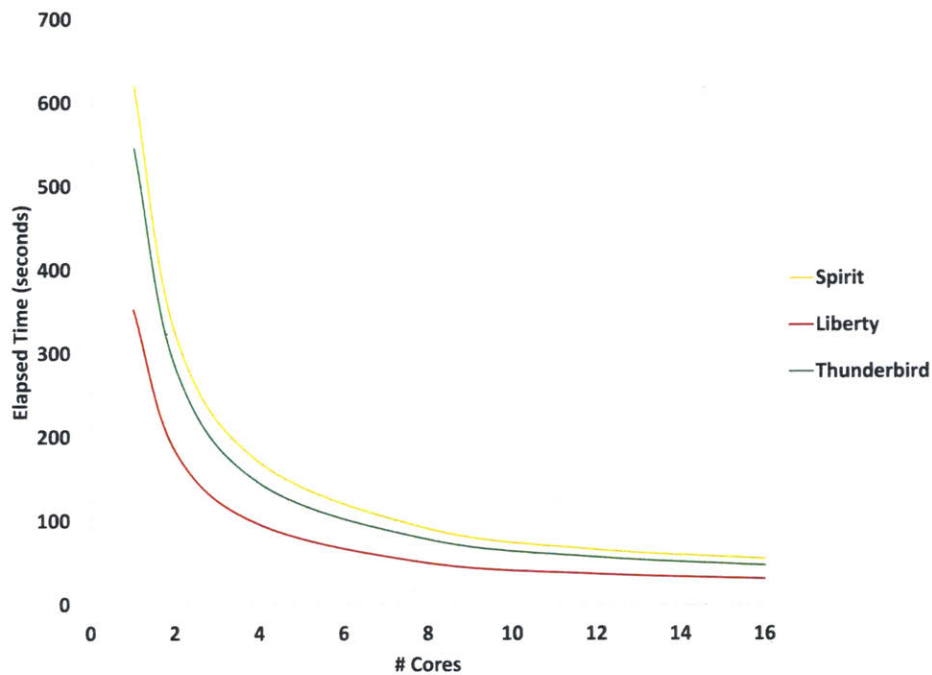


Figure 2.10: Parallel algorithm performance on multiple cores when processing supercomputer logs

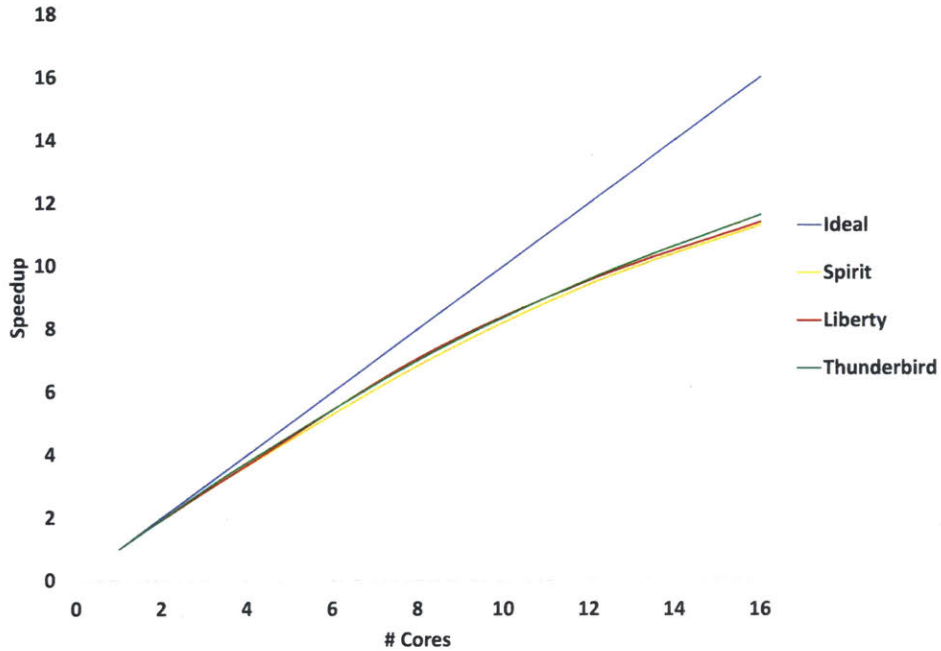


Figure 2.11: Parallel algorithm scalability on multiple cores when processing supercomputer logs

2.4 Summary

In this chapter, we have proposed and demonstrated a fast and accurate parallel algorithm that proactively predicts hardware issues in HPC systems. With such proactive predictions, we can alert users of the systems and allow them to take remedial actions before the failures impact workloads running on the system. The algorithm uses a supervised machine learning approach based on the Naive Bayes Bernoulli classifier model. It predicts hardware failures based on the statistical characteristics of the Linux system logs produced by the HPC systems.

Our objectives when developing the prediction algorithm were to be able to predict hardware failures with a high level of accuracy, to be able to perform the prediction in a time significantly less than the estimated MTBF for future exascale systems, and to perform the prediction without affecting the system's performance. We have shown how

these objectives were met when testing the algorithm on ~750 million real system logs from three SNL supercomputer systems.

The algorithm demonstrated a high level of accuracy with only 3 nodes incorrectly classified out of more than 5000. The algorithm was successful in predicting all troubled nodes that were previously manually identified by SNL. In addition, it was capable of predicting new troubled nodes that were not previously flagged by SNL. Furthermore, as the algorithm uses a supervised machine learning approach for the predictions, it was capable of predicting new categories of hardware failures that it was not exposed to in the training data. As for performance, the algorithm was capable of processing the data entirely and producing the prediction results in a desirable time. It performed the full prediction cycle in less than one minute for each of the three different supercomputer logs, with an average run time of only 44 seconds when running on a machine with 16 computing cores. Since we designed the algorithm to run externally from the compute nodes, no performance overhead will be imposed locally on the nodes running HPC workloads. In addition, since the algorithm uses Linux's natively generated logs as its data source, there is no additional overhead introduced on the system to generate and collect the data. This is unlike related studies discussed, where the data analyzed is generated by third-party tools and local agents, which consume system resources and impose a local overhead on the compute nodes.

We believe that our algorithm was able to achieve this high level of accuracy and performance, even when applied to massive amounts of data, due to the successful combination of techniques we used. Our approach of combining the Bernoulli model of the NB classifier along with our methods of preprocessing the data in terms of reduction,

normalization, and grouping has led to the desired high prediction accuracy. Performance wise, the decision to base the algorithm on the linear complexity NB model contributed to the algorithm's desired performance. Furthermore, our choice of using the Apache Spark framework for implementation facilitated achieving the desired performance due to the parallel in-memory processing capabilities. In addition, the tuning we performed for the Linux and Spark system parameters helped improve the performance of the algorithm further. Finally, our technique of repartitioning the data after reduction and adjusting its default number of partitions boosted the performance of the algorithm even further.

Chapter 3 [†]

Remedy Component

In Chapter 2 of this work, we have discussed our implementation of a fault prediction algorithm for HPC systems. In this chapter, we will go over the proposed remedy environment to use once such faults are identified on a system. We start with an overview of the environment, followed by the system design details of the implementation. This includes the setup details of the hardware, software, containers, and network. We also present the various HPC applications that were chosen for testing with the environment. After that, we discuss the testing details and results obtained. We also discuss the challenges faced during the testing and the solutions that were adopted to overcome them. Finally, a summary and conclusion is presented.

[†] Work from this chapter has been accepted in the IEEE High Performance Extreme Computing conference (HPEC '19) for publication. The publication was also competitively selected as Best Paper Finalist.

3.1 Overview of Proposed Remedy Environment

The method used to design the remedy environment is based on running the HPC workloads inside Linux containers. Container technology has proven its success in the microservices domain as a scalable and lightweight technology used in data centers. In our work, we adapt the container technology towards HPC workloads to make use of its migration capabilities, which can be thought of as a resilience mechanism. By running inside containers, we are capable of migrating the HPC workloads from compute nodes anticipating hardware problems, to healthy spare compute nodes while the workload is running. Migration is performed using the CRIU tool with no application modification. The container environment does not introduce any major interruption or performance overhead to the running workload. We also provide application benchmark results comparing the performance of the container environment to the native system.

The container environment is tested with various real HPC applications that are based on the MPI standard. The applications tested come from both academia (open-source) and the industry (closed-source). When choosing the test applications, we selected ones written in various programming languages that are commonly used in HPC applications (e.g. C++, C, and Fortran). The applications also vary in terms of their runtime results. Some produce in-situ live visualizations of the results as an HPC simulation is being migrated, while some just produce the typical time-stepping text output to the terminal. Verifying the integrity of the results produced by the migrated workloads is also addressed in this study. The applications chosen produce various types of output data files (text and binary), which are then used for post-processing to verify the integrity of results. We test the applications with different hardware node types as well as different

network interconnect types (e.g. 1, 10, and 25 Gigabit Ethernet networks). The MPI applications are tested with a various number of processing cores ranging from 4 and up to 144 cores running on multiple nodes. In addition, we test the migration with three of the commonly used MPI implementations, MPICH, Open MPI, and Intel MPI.

3.2 System Design

In this section, we will go over the details of the system design of the HPC remedy environment. This will include the setup of the clusters' hardware, software, containers, and network.

3.2.1 Hardware Cluster Setup

The cluster test environment was set up using the Amazon Web Services (AWS) infrastructure [82]. We believe that our container environment setup should also work with traditional Linux clusters that are not AWS based. By using AWS however, it gave us the option to set up test clusters with various node specs in terms of the number of processing cores, system memory, and different network interconnect speeds. For our testing, we chose four types of node instances that varied in terms of specs [83]. The lower spec nodes were of type m4.2xlarge (4 physical cores each, 32 GB RAM, 1 Gig network). The medium spec nodes were of type c5.9xlarge (18 physical cores each, 72 GB RAM, 10 Gig network). As for the higher spec nodes, we tested using two types, m4.16xlarge (32 physical cores each, 256 GB RAM, 25 Gig network) and i3.metal (36 physical cores, 512 GB RAM, 25 Gig network). All instance types, except for i3.metal, use AWS's hardware virtual machine (HVM) technology to launch the instances. The i3.metal instances however are AWS's newer bare metal instances. They provide

applications running on them with direct access to the resources of the underlying server (e.g. processor and memory resources) in a non-virtualized environment. We specifically chose instances that are both HVM based and bare metal based to make sure that the container migration testing would work with both types of environments (virtualized and non-virtualized).

For all instances, we disable the CPU's hyper-threading mode, CPU's c-states power saving mode, and CPU's turbo boost mode [84]. These CPU features are commonly disabled in HPC environments as they can affect the performance/consistency of the HPC applications running on the system.

All instances access a shared network storage using Amazon's Elastic File System (EFS), which provides a volume of persistent storage that is mounted on the cluster instances [85]. The shared storage volume is mounted on the instances through the Network File System versions 4.1 (NFSv4.1) protocol [86]. This shared storage holds all the input data for the HPC applications as well as the produced results from running the HPC applications.

3.2.2 Software Setup

In this section, we will discuss the software setup of the environment in terms of the operating system and container technology used, as well as the system tools used for launching and maintaining the containers on the system.

Container technology has been gaining a lot of attention in the past few years due to its low overhead on the system it runs on, and its capability to provide a performance that is almost identical to a native system. Containers are more lightweight on the system compared to traditional VM technologies. A container running on a node will share the

same operating system kernel and common files without replicating the operating system. This makes it less memory and space demanding compared to traditional VMs. Containers also do not rely on a hypervisor to access the underlying system resources. On the other hand, traditional VMs use hypervisors that can introduce a resource and performance overhead due to the virtual emulation of devices through a hypervisor. Such device emulation can affect the performance of the application running on the system. For example, the emulation of a network card in a VM can affect the communication latency between different instances, which could be a deal breaker for latency-sensitive HPC applications. Container technology also provides a convenient way to package an application and its dependencies for ease of deployment.

Looking at the different types of container technologies currently available, there are mainly two common types of containers, application containers and system containers [87], [88]. Application containers typically host individual applications inside a container and are popularly used to host microservices such as webservers. The resources of the underlying server where these containers are hosted (e.g. memory, processors, etc.) can then be partitioned among the multiple application containers running on the system. The most popular example of this type of containers is Docker, which was initially released in 2013 as an open-source project [89]. System containers, on the other hand, tend to encapsulate the entire underlying server environment inside a container. This is similar to the concept of having a VM, however there is no hypervisor overhead involved. A good example of this type of containers is the OpenVZ containers, which is a mature implementation initially released in 2005 as an open-source project [90].

We decided to use system containers to set up our environment and chose OpenVZ containers. We believe that system containers are more suitable towards an HPC environment as a single container can behave as a full HPC compute node, encapsulating all resources of the underlying system. OpenVZ also provided out of the box container features such as assigning IP addresses to containers, secure shell (ssh) connections, and NFS shared file system mounting, which are all crucial features to any HPC environment.

In addition, we use the CRIU tool to facilitate the container migrations. CRIU is an open-source project that provides a checkpoint/restart functionality for Linux based applications and processes. The project was initially released in 2012. The CRIU tool can be used to capture the states of CPU, memory, disk, and network of a process or a hierarchy of processes. Such functionality of CRIU can also be applied to containers as containers are considered Linux processes themselves. We also use Parallels' "prctl" command line tool to provision and manage the containers [91]. In addition, the AWS Command Line Interface (CLI) package [92] is used for the network management of the instances. Finally, the "vzpkg" tool [93] is used to install any extra system packages needed inside the containers.

The OpenVZ version that we targeted for testing was 7.0.7. The kernel provided by OpenVZ is a variation based on Red Hat Enterprise Linux (RHEL) 7.4 kernel 3.10.0-693. The OpenVZ variation however has patches that support the container functionality and its management. To our understanding, the patches applied to the kernel are currently not mainstream in the Linux kernel. The CRIU version used was version 3.4.0.41-1, while the "prctl" container management tool used was version 7.0.148-1. The AWS CLI package used was version 1.15.61. The "vzpkg" tool used was version 7.0.57-2.

It is also important to note that the only officially supported method to install OpenVZ is through their ISO installation image. This created a challenge for us as our testing environment was set up on the AWS platform, and there was no official way to install an AWS instance with an ISO image. After discussing this with the OpenVZ community, it was brought to our attention that there is a commercially supported version of OpenVZ, called Virtuozzo, which AWS currently provides instances for. Due to the limitation of not being able to install via ISO image on AWS, we ended up setting our environment using the equivalent Virtuozzo 7.0.7 image provided by AWS. The concept of OpenVZ to Virtuozzo is similar to the concept of the CentOS Linux to RHEL, where OpenVZ and CentOS are the open-source projects that are the basis for their commercial distributions, while Virtuozzo and RHEL are the commercial distributions that are bundled with enterprise support to the users. Technically, the kernel and software versions for both the open-source and commercial variations appeared identical to us.

3.2.3 Container Setup

HPC applications typically would fully utilize the underlying compute node they are running on. With this in mind, our setup consisted of launching a single container on each node instance, and that single container would then have full access to the resources of the underlying server it runs on (e.g. memory, processors, etc.). The launching of the containers was orchestrated using the “prctl” container management tool. By default, the launched containers do not have access restrictions in terms of how many processing cores to use from the underlying machine. However when it comes to memory, we had to specify the amount of memory we wanted to allocate for the container when it was being launched. As we were testing with different types of instances that had different specs,

we set the containers to use the majority of the memory available depending on the instance type, while leaving a sufficient amount of memory for the underlying operating system just as a precaution measure. For example, on the i3.metal instances, the containers were launched with 500 GB of RAM out of the 512 GB available.

The “prctl” container management tool was also used to write scripts to automate launching the containers with specific settings. This included setting the container’s operating system template, name, IP address, users, DNS, NFS mounts, ssh keys, default bash shell environment variables, default bash shell limits, and the netfilter firewall settings. The container template used was the CentOS 7 template. In addition, we used the “vzpkg” tool to automate installing any extra packages we needed inside the container (e.g. MPI libraries, third-party packages needed by the HPC applications being tested, etc.).

For each of the HPC applications we were testing, a separate container installation image was created. Each container image only included the software packages and application binaries needed for that specific HPC application. This kept the containers minimal in size.

3.2.4 Network Setup

For every node instance in the cluster, four IP addresses were allocated. The native instance was assigned 2 IPs, a public and a private one. The container running on the native instance was also assigned 2 IPs, a public and a private one. The public IPs are used to access the native instance and the container by the ssh protocol from any remote terminal. The private IPs are used for local communication between the instances or between the containers. The IPs of the native system are associated with the instance’s

primary network interface, while the IPs of the container are associated with the instance's secondary network interface. By having this setting, the IPs of the containers can be treated as floating IP addresses, meaning that if a container is migrated from one physical machine to another, then the IP addresses associated with the container can also be migrated along with the container. This concept of having a floating IP address was crucial in order for the HPC applications to be successfully migrated through containers.

By default, the assigned IP addresses in the AWS platform are not static. This means if you reboot or stop an instance, then there is no guarantee that the instance will come back having the same IP address once it is up again. In typical HPC environments, the cluster nodes usually would have static IPs associated with them that do not change upon reboots. Luckily, the AWS platform provides what is called Elastic IP addresses, which are static IPs that do not change if an instance was rebooted or stopped [94]. For our setup, all IPs were set to be static. Having static IPs made it more convenient for us when doing our benchmarking and container migration testing. The details of how the containers' IPs were setup is similar to what is found in this reference [95].

Finally, the container network is set up in a host-routed mode [96]. This means that the native node acts as a local router to any packets going through the container. This type of networking mode for containers usually provides better latency performance compared to other modes, such as the bridged networking mode [97].

3.2.5 Summary of System Design

Figure 3.1 gives a pictorial summary of the system design architecture for the container-based test environment.

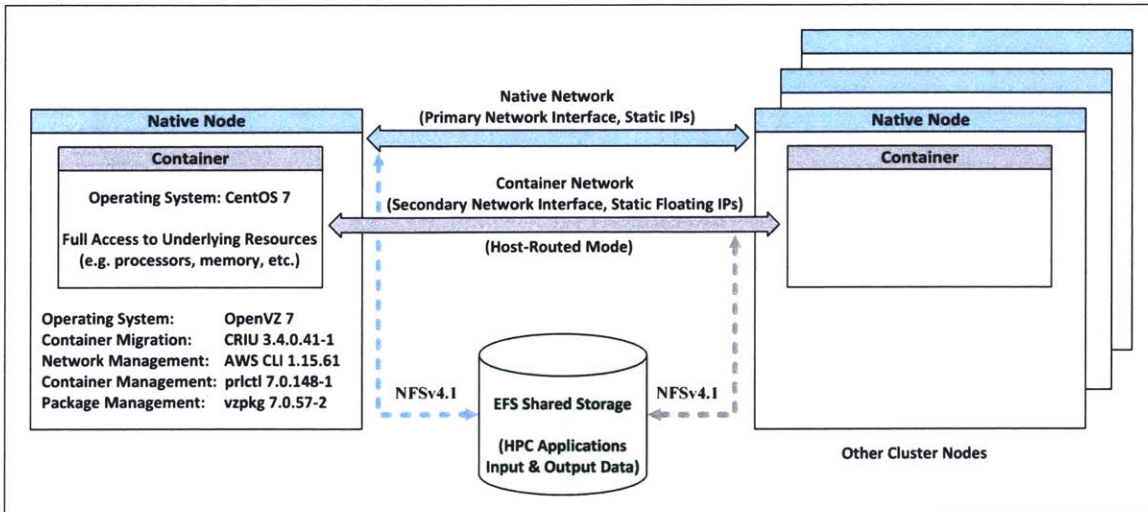


Figure 3.1: Summary of system design architecture

3.3 Selected HPC Applications for Testing

In this section, we briefly go over the HPC applications chosen for the container migration testing and performance benchmarks. We chose six HPC applications that are all MPI based. They cover various domains and are implemented in different programming languages. The applications tested were the Ohio State University (OSU) Micro-Benchmarks [35], Palabos [98], Flow [99], Fluidity [100], GalaxSee [101], and the ECLIPSE* industry-reference reservoir simulator by Schlumberger [102]. Table 3.1 summarizes these applications.

Application	Main Developers	Language	Domain
OSU Micro-Benchmarks	Ohio State University	C	MPI benchmark for network bandwidth and latency
Palabos	- Academic: University of Geneva - Industry: FlowKit CFD	C++	CFD/Complex Physics using lattice Boltzmann method
Flow	Open Porous Media Initiative (Oil companies + Academia)	C++	Reservoir simulation
Fluidity	Imperial College London	Fortran, C++	- CFD solving Navier-Stokes - Geophysical fluid dynamics - Ocean Modelling - Adaptive unstructured mesh
GalaxSee	- Shodor Education Foundation - National Center for Supercomputing Applications - George Mason University	C++	N-body galaxies simulation
ECLIPSE	Schlumberger (commercial)	Fortran	Industry-reference reservoir simulator

Table 3.1: Summary of tested HPC applications

The OSU Micro-Benchmarks suite is a set of MPI benchmarks used to test the bandwidth and latency performance of MPI functions on HPC networks. It is open-source and the suite includes various tests for the common MPI operations. For our benchmarks, we focused on testing a few of the point-to-point MPI benchmarks as well as a few of the collective MPI benchmarks. This application is the only one out of the six we tested that is not actually a scientific HPC application, but rather a network benchmark. It allowed us to compare that container’s network performance to that of the native system. OSU micro-benchmarks version 5.3.2 was used for testing.

Palabos is an open-source computational fluid dynamics (CFD) solver that is based on the lattice Boltzmann method. It provides a variety of CFD simulation test models that can be used for benchmarking. Palabos version v2.0r0 was used for testing.

Flow is an open-source fully-implicit reservoir simulator. It can be used to run industry-standard oil reservoir simulation models. It provides a limited number of

simulation test models. The project is currently maintained by the Open Porous Media (OPM) initiative. Flow version 2018.04-0 was used for testing.

Fluidity is an open-source multiphase CFD solver. It supports adaptive unstructured meshes and is capable of numerically solving the Navier-Stokes equation. It provides a wide range of simulation test models, which include ocean modeling and geophysical fluid dynamics. It is mainly maintained by the University of Imperial College London. Fluidity version 4.1.15 was used for testing.

GalaxSee is a simple open-source application that performs the N-body galaxies simulation and visualization. It allows you to provide parameters for your simulation such as the number of stars in the galaxy to simulate, star mass, and the simulation period. GalaxSee version MPI 0.9 was used for testing.

The ECLIPSE simulator is a commercial closed-source reservoir simulator developed by Schlumberger. It is considered the industry-reference reservoir simulator in the oil and gas industry. It provides various synthetic reservoir models for testing. ECLIPSE simulator version 2017.2 was used for testing.

3.4 Testing and Results

In this section, we will go over the details of the two types of tests performed in the HPC container environment. This includes container benchmark tests, and container migration tests. We will also go over the integrity check of the results produced from the migrated workloads. In addition, we will provide references for videos demonstrating the container migration tests using real HPC workloads.

3.4.1 Container Performance Benchmarks

The first type of tests conducted was to benchmark the HPC applications on the container environment and compare the performance to the native system. A native system is a cluster of instances with no containers running on them. A container environment would be the same cluster of instances, however on each instance we would launch a single container that encapsulates the entire underlying native system. The native instances and the containers have their own distinct IP addresses. The `mpirun` launcher is then used to start the MPI jobs. When launching an MPI job on the native instances, the MPI hostfile will have the IP addresses of the native instances. Similarly, when launching on the containers, the MPI hostfile will have the IP addresses of the containers. Once the MPI jobs are completed, we record the elapsed time of execution for both the native and container environments. Each benchmark is run four times and timings are averaged. We also calculate the overhead introduced from running on containers. All application benchmarks were performed using the MPICH MPI library version 3.0-3.0.4-10, except for the ECLIPSE simulator benchmark where we used the Intel MPI library version 5.0.2.044. Intel MPI was the supported library for the binary executable of the ECLIPSE simulator.

The first application tested was the OSU Micro-Benchmarks. For the point-to-point MPI benchmarks, we tested the `osu_latency` and `osu_bw` benchmarks, which report the network's latency and bandwidth respectively. For the collective MPI benchmarks, we tested `osu_allgather` and `osu_allreduce`. These are two of the commonly used collective MPI operations in HPC applications, and they report the latency of the collective operations. Each benchmark was performed on a pair of instances of the same type. In the

case of the point-to-point benchmarks, one processor from each instance participated in the benchmark. For the collective benchmarks, all available processing cores on each instance were used. As our instances varied in network specs, the tests were performed on all three network types available, which included 1, 10, and 25 Gigabit networks. Figure 3.2, Figure 3.3, and Figure 3.4 show the point-to-point MPI benchmarks for the three network types. Figure 3.5, Figure 3.6, and Figure 3.7 show the collective MPI benchmarks for the three network types.

In the case of the 1 Gigabit network, the bandwidth of the container and native system were almost identical. With the 10 Gigabit network, there was a slight bandwidth reduction on the containers, while on the 25 Gigabit network there was a more noticeable reduction in bandwidth. Nevertheless, the overall average bandwidth overhead was only around 3.9 % when using the container environment. As for latency, there was a slight overhead when using the containers for most of the tests, with both point-to-point and collective benchmarks. Overall, the average latency overhead was around 6.8 % when using the container environment. We do not fully understand why the 25 Gigabit network behaved a bit differently than the others, but we believe this might be related to the need for more appropriate network tuning for 25 Gigabit networks from the Linux side (e.g. kernel parameters, etc.).

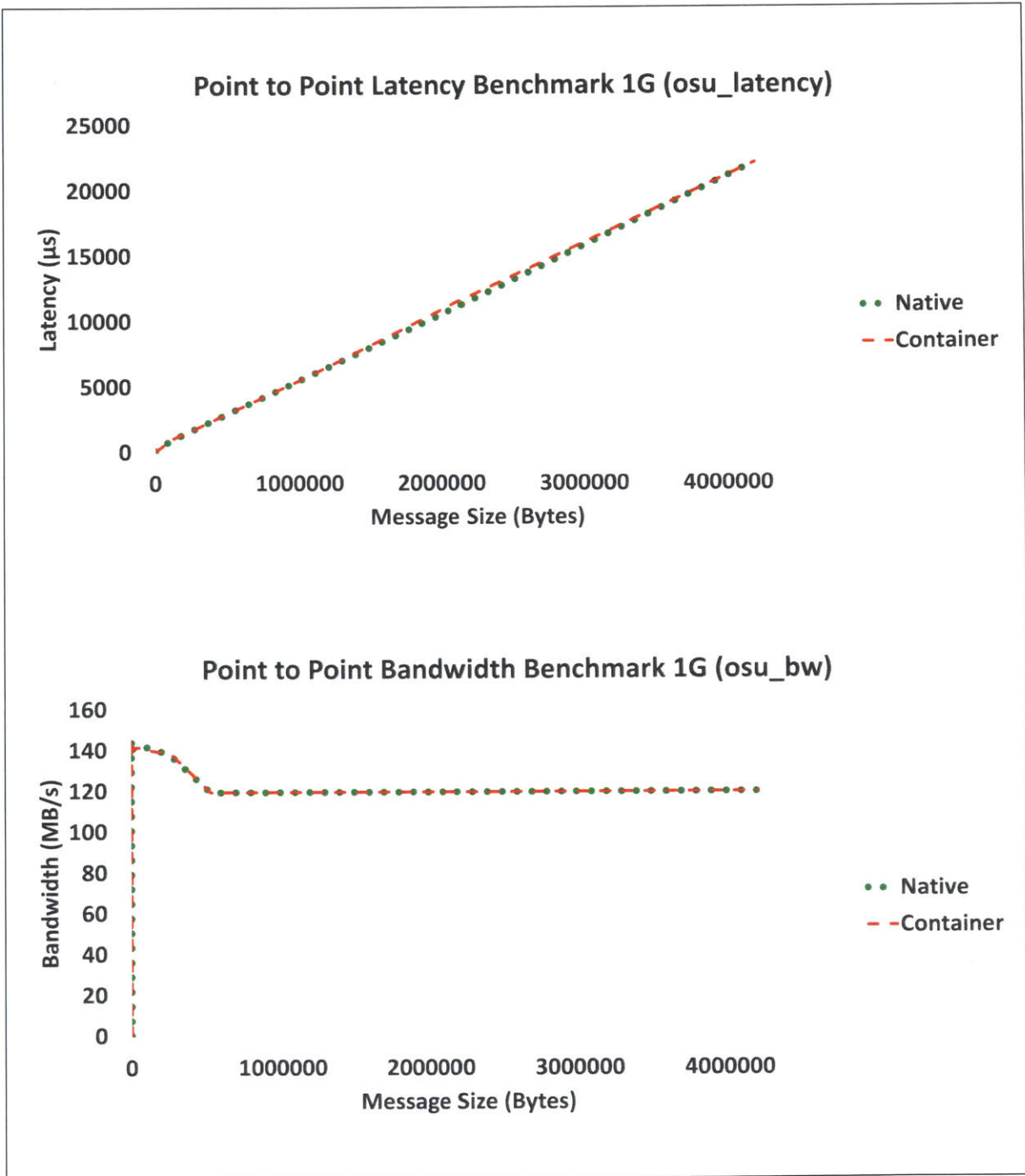


Figure 3.2: Point-to-point MPI benchmarks for 1G network

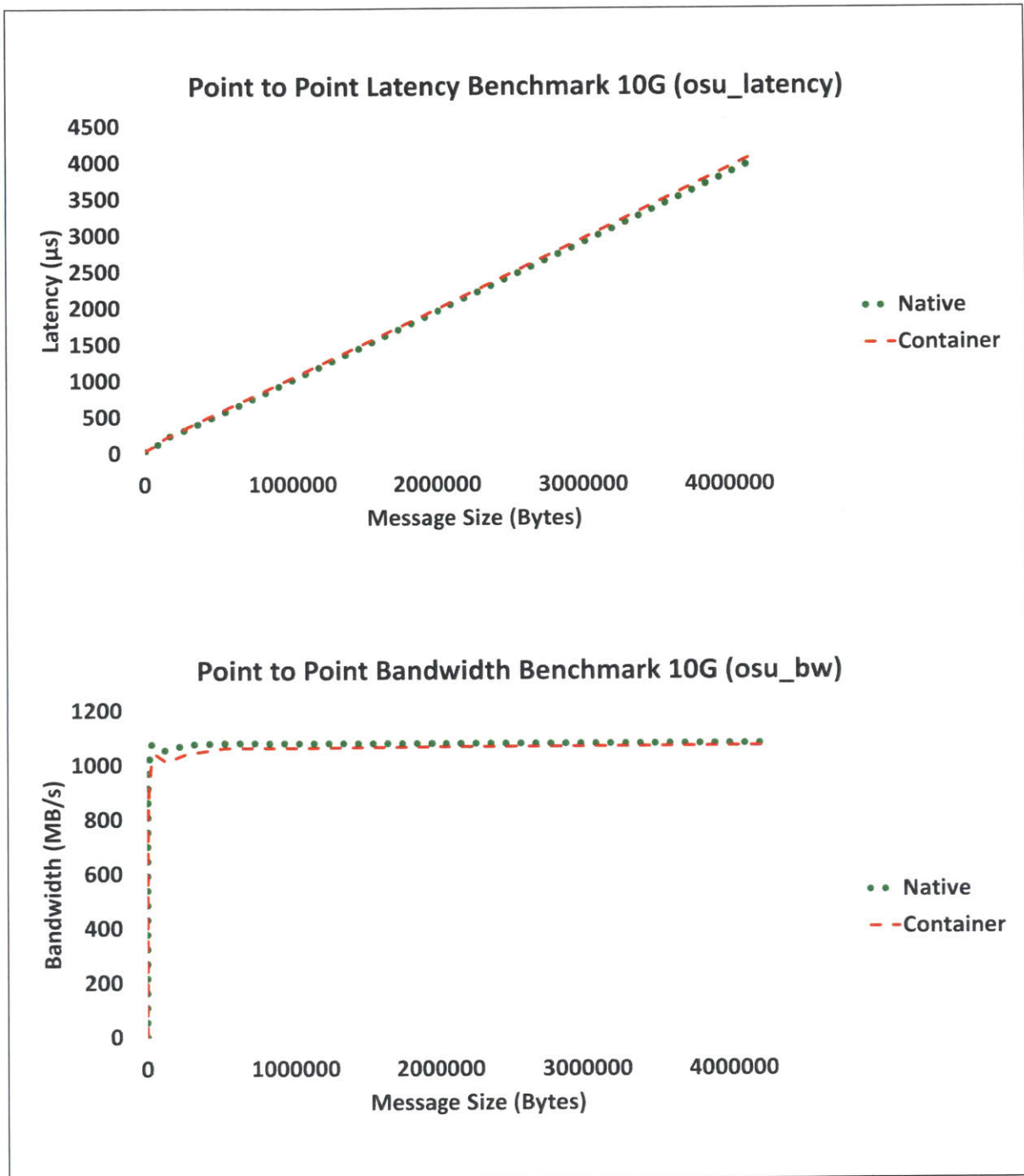


Figure 3.3: Point-to-point MPI benchmarks for 10G network

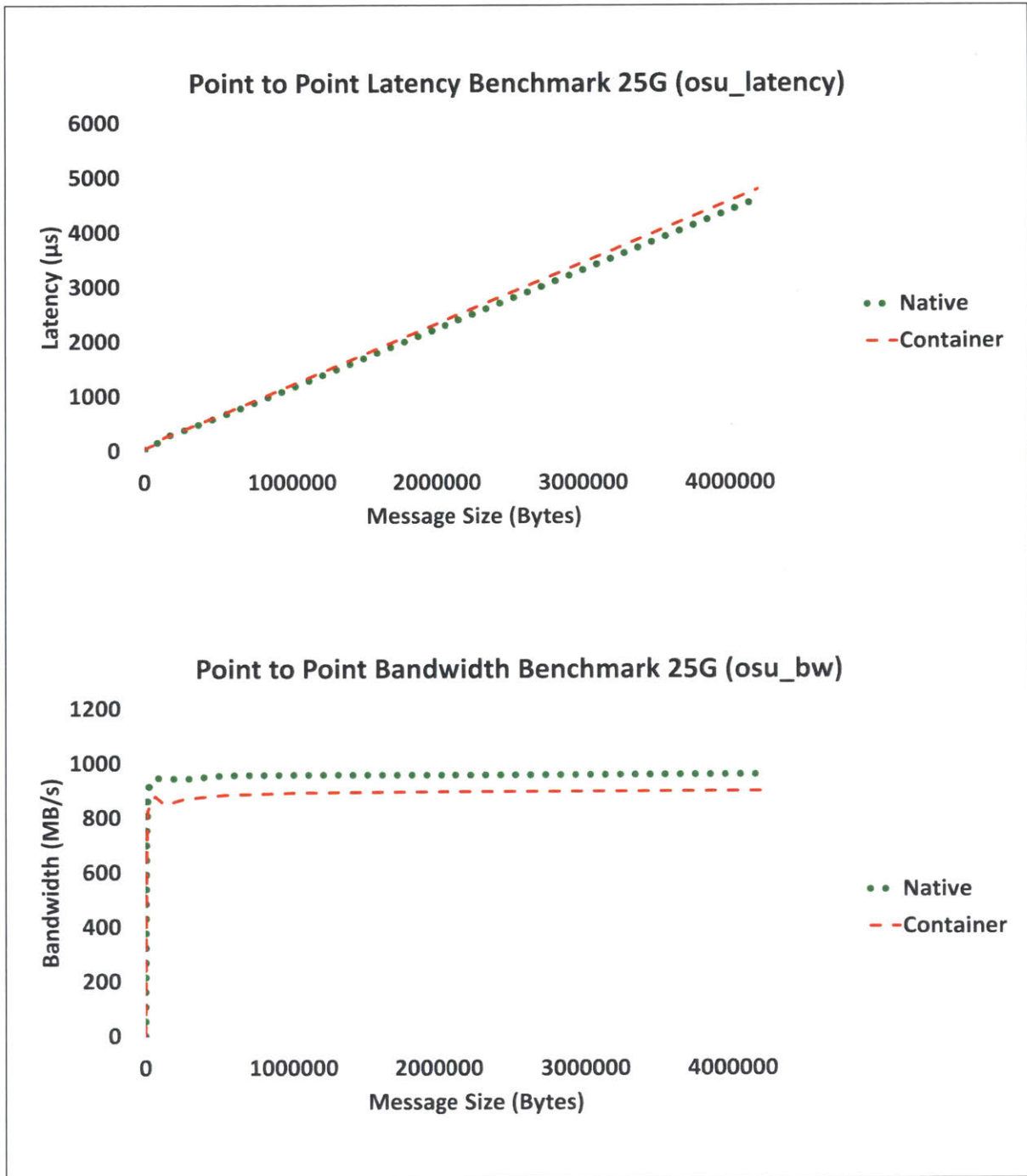


Figure 3.4: Point-to-point MPI benchmarks for 25G network

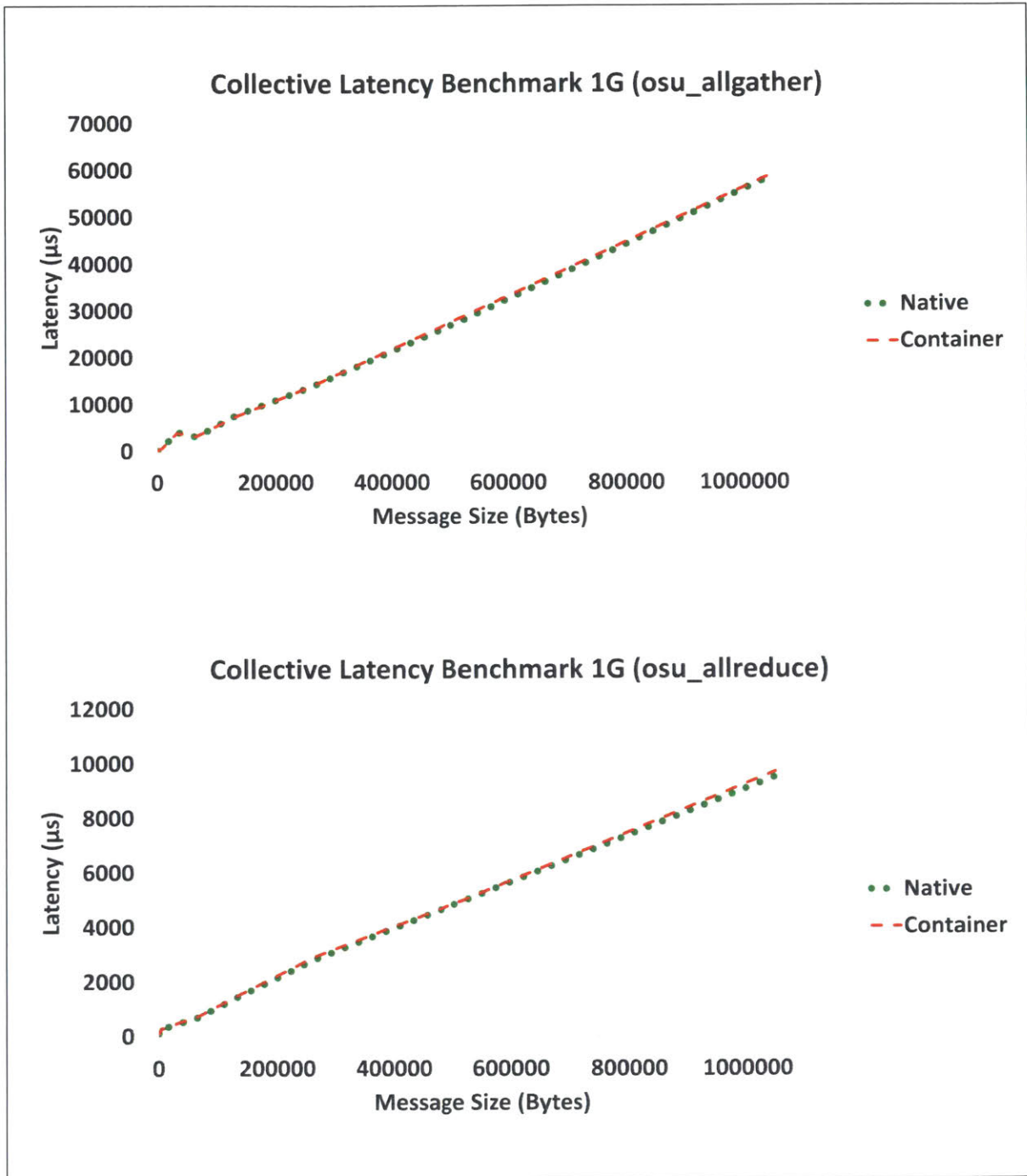


Figure 3.5: Collective MPI benchmarks for 1G network

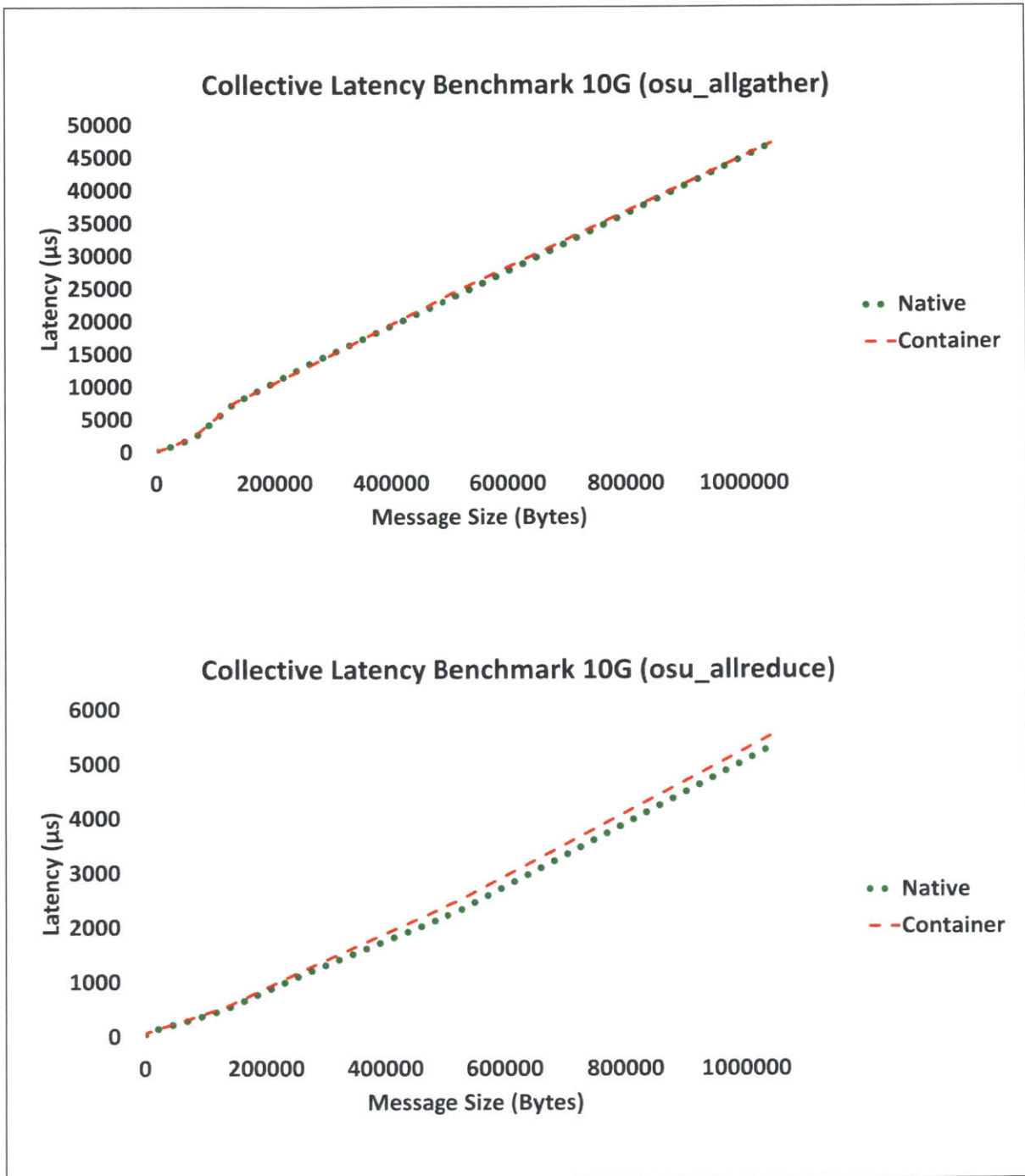


Figure 3.6: Collective MPI benchmarks for 10G network

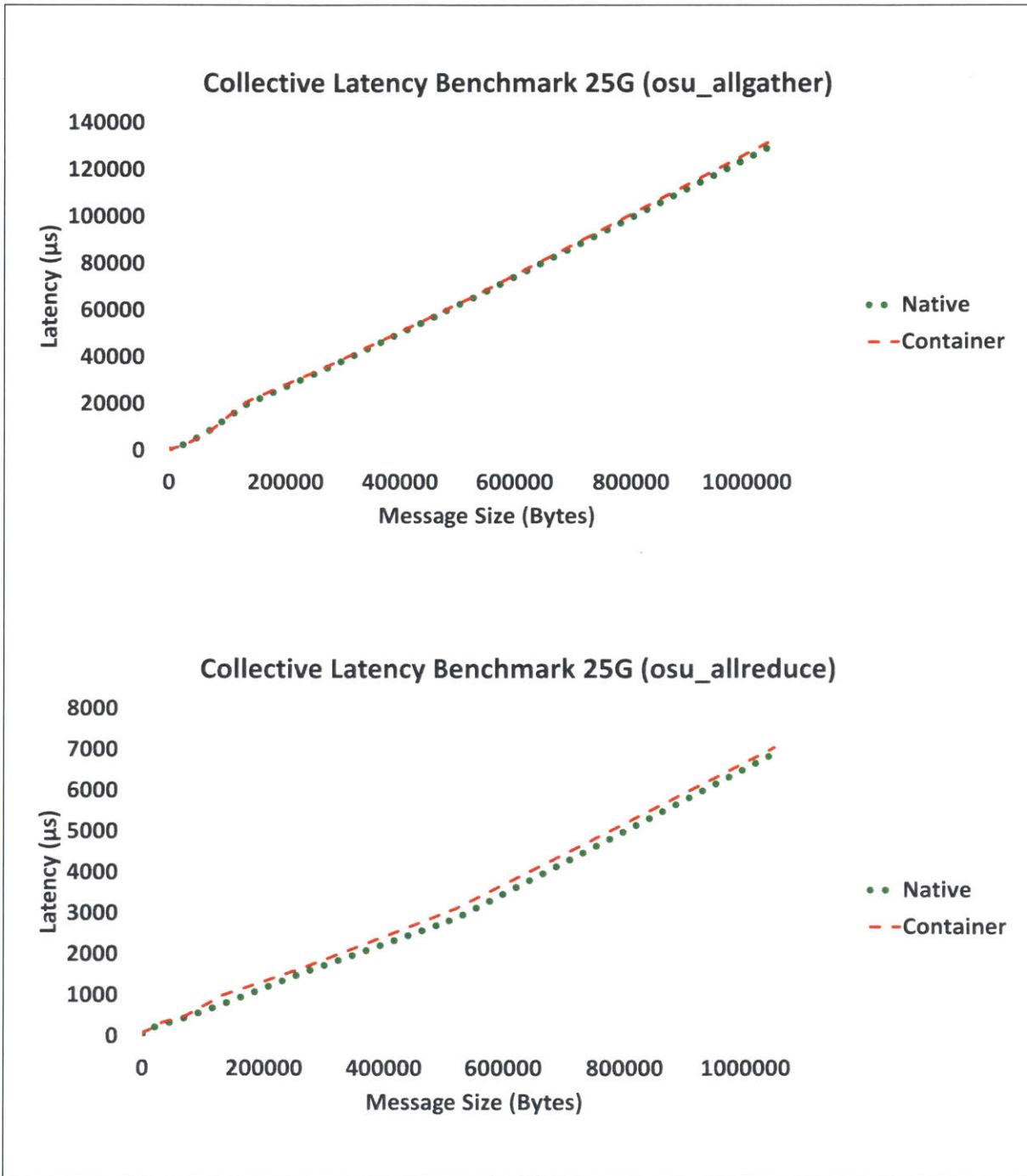


Figure 3.7: Collective MPI benchmarks for 25G network

Such slight performance overhead with containers was to be expected. We believe that the difference in performance was due to the way the network routing is done. The container's network is run in a host-routed mode. In this case, the native node acts as a router to the packets passing through the container. Hence, the container's traffic will have to go through that one extra step of routing, which contributes to the slight overhead we are seeing with the containers.

Next, we proceeded with testing the remaining HPC applications. The benchmarks were performed on the various instance types with MPI job sizes ranging from running on 4 processing cores, and up to 144 cores. The input models we used for each application varied depending on the number of processing cores we were running on. For example, when testing the Fluidity application, we used a moderately sized model "tephra_settling" when benchmarking on 4 cores, however when benchmarking on 144 cores, we used a significantly larger model "tides_in_the_Mediterranean_Sea" so that the problem size will not be too small when running on a larger number of cores. Also, note that for the ECLIPSE simulator and Flow applications, we were only able to test on up to 8 processing cores. For the ECLIPSE simulator, the commercial license we had was limited to only 8 parallel processes. For Flow, the test models provided with the application were limited in size (e.g. Norne model was the largest) and problem sizes were too small to be run on more than 8 cores. Table 3.2 summarizes the input models used for all the runs.

Figure 3.8, Figure 3.9, and Figure 3.10 show the average elapsed times for the HPC application runs and compare the performance of the container and native environments. Even though we previously saw a slight overhead in latency and bandwidth when using

containers with the OSU Micro-Benchmarks, the performance overhead when testing with real scientific HPC application was very negligible and close to native performance. The average performance overhead when running on containers was only 0.034%. In the worst case, the maximum overhead observed was around 0.9%. Overall, the performance on the container environment was acceptable for all the HPC applications tested and was almost identical to the performance of the native environment.

AWS Instance Type	Application	Model	# Nodes	# Cores
m4.2xlarge	Fluidity	tephra_settling	1	4
m4.2xlarge	Fluidity	water_collapse	2	8
c5.9xlarge	Fluidity	water_collapse	1	18
c5.9xlarge	Fluidity	backward_facing_step_3d	2	36
m4.16xlarge	Fluidity	tides_in_the_Mediterranean_Sea	4	128
i3.metal	Fluidity	tides_in_the_Mediterranean_Sea	4	144
m4.2xlarge	Palabos	rayleighTaylor2D_2000x600	1	4
m4.2xlarge	Palabos	rayleighTaylor2D_2000x600	2	8
c5.9xlarge	Palabos	rayleighTaylor2D_2000x600	1	18
c5.9xlarge	Palabos	rayleighTaylor2D_4000x1200	2	36
m4.16xlarge	Palabos	rayleighTaylor2D_8000x2400	4	128
i3.metal	Palabos	rayleighTaylor2D_8000x2400	4	144
m4.2xlarge	GalaxSee	#stars: 8000, mass: 10, time: 10000	1	4
m4.2xlarge	GalaxSee	#stars: 8000, mass: 10, time: 10000	2	8
c5.9xlarge	GalaxSee	#stars: 18000, mass: 10, time: 10000	1	18
c5.9xlarge	GalaxSee	#stars: 36000, mass: 10, time: 10000	2	36
m4.16xlarge	GalaxSee	#stars: 38400, mass: 10, time: 1000	4	128
i3.metal	GalaxSee	#stars: 43200, mass: 10, time: 1000	4	144
m4.2xlarge	Flow	SPE9	1	4
m4.2xlarge	Flow	Norne	2	8
m4.2xlarge	ECLIPSE	ONEM1_4	1	4
m4.2xlarge	ECLIPSE	ONEM1_8	2	8

Table 3.2: Input models used for application testing



Figure 3.8: Benchmarks on m4.2xlarge instances

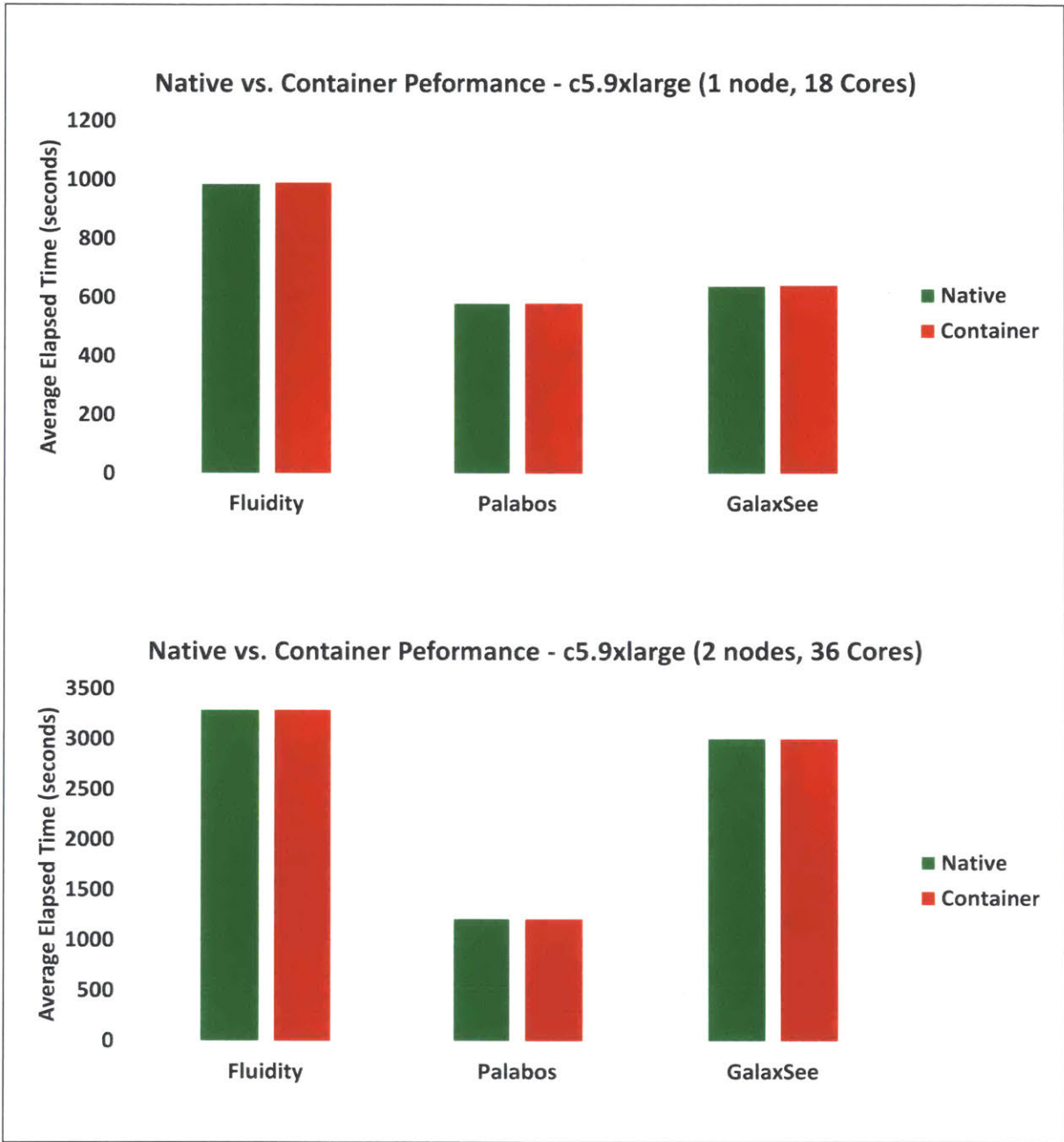


Figure 3.9: Benchmarks on c5.9xlarge instances

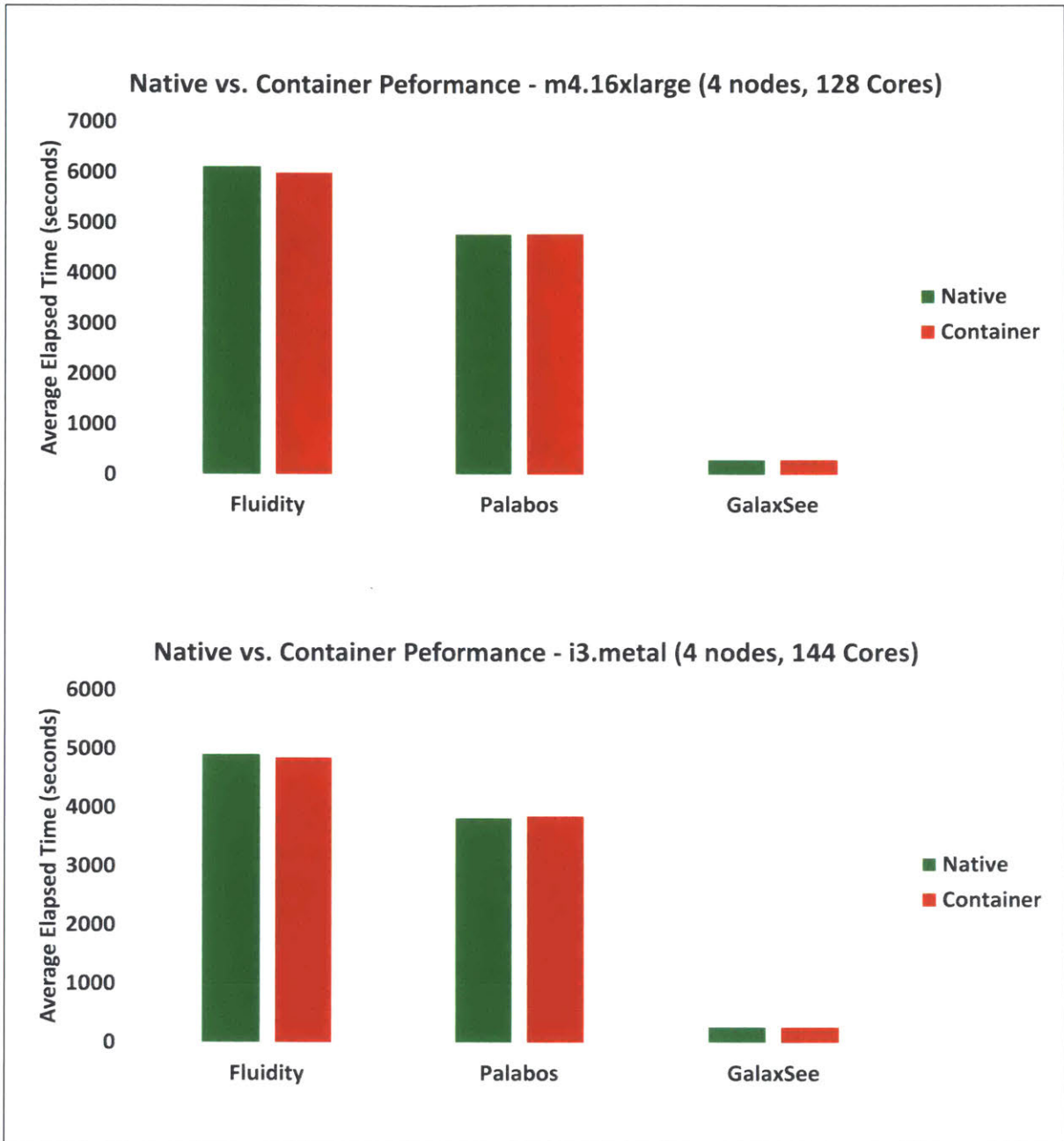


Figure 3.10: Benchmarks on m4.16xlarge and i3.metal instances

3.4.2 Container Migration Testing

The second type of tests conducted was to test migrating containers while an MPI-based HPC application was running inside of them. As we strived to accomplish this goal, we were faced with various technical challenges that had to be addressed before succeeding. In this section, we only highlight the testing and results obtained after resolving all the challenges, while we dedicate section 3.5 of this chapter to go over the details of the challenges faced and solutions adopted during testing. We next discuss the migration mechanism, followed by a discussion about the migration timings for the different applications tested.

Our goal was to launch a distributed MPI application inside containers that are hosted on several native machines, then try to successfully migrate one of the containers from one native machine to another (i.e. spare machine) while the MPI job was running. Figure 3.11 gives an overview of the task we were trying to accomplish. We used the “prctl” tool as our interface to control the states of the container. The “prctl” tool uses the CRIU library to manipulate the container’s state. The AWS CLI tool was also used to migrate the container’s floating IP address between machines.

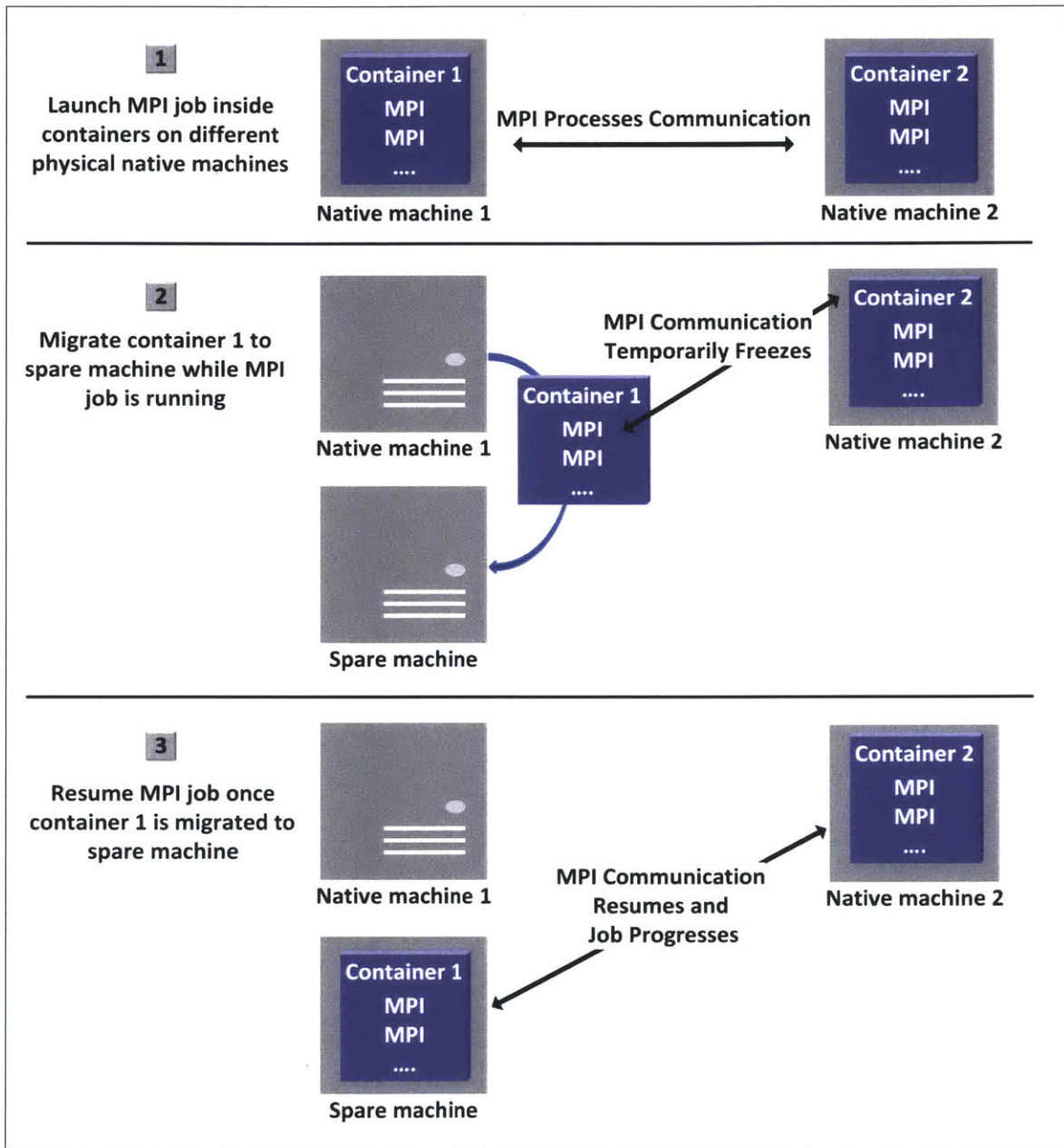


Figure 3.11: Overview of container migration test

Overall, the migration steps can be described as follows. First, the container is put in a “suspend” state as MPI processes are running inside of it. This will cause the MPI job to freeze temporarily. The MPI processes on the other participating HPC nodes will still be alive, but waiting for the suspended container to come back. As part of the suspension process, CRIU takes a dump of the container’s state and stores it locally on the native machine. Next, the container’s floating IP address is migrated to the spare node. We automate the IP migration by writing a custom script influenced by Sabat’s [103], which utilizes the AWS CLI tool for automation. After that, the container is put in a “migrate” state, which will copy the dumped container’s state to the spare machine we are migrating to. Finally, the container is put in a “resume” state on the spare machine. This restores the container’s state and unfreezes it. At this point, the MPI job resumes from the same state before it temporarily froze and progresses. The migration and resuming are done as user root. The HPC application’s input data, as well as the output produced by the run, all reside on a shared NFS storage system, so there is no need to copy any of that data between the machines involved with the migration. We automate the entire migration process with a custom script. Figure 3.12 summarizes the migration steps.

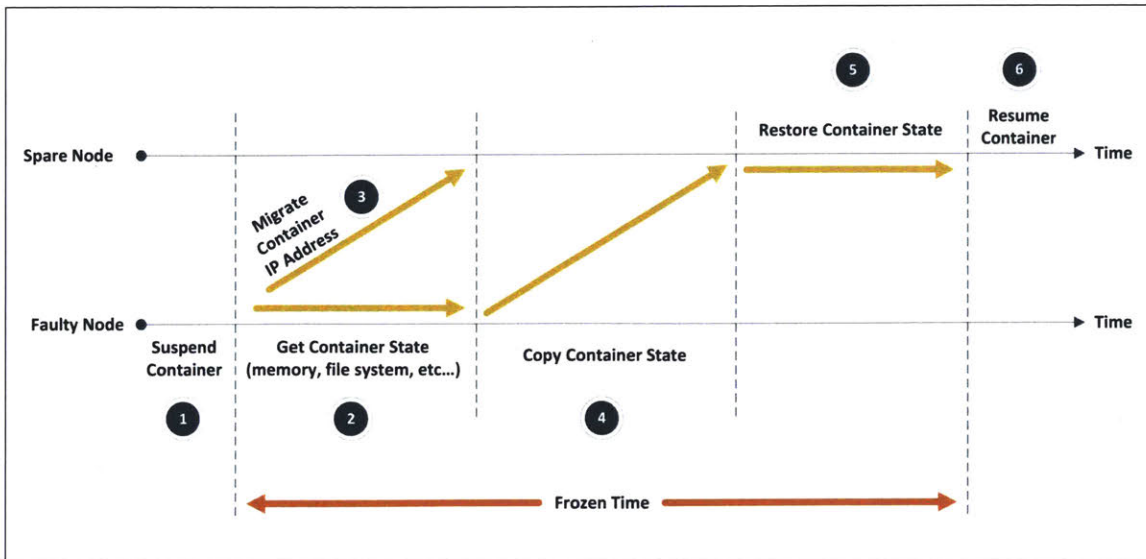


Figure 3.12: Container migration steps

In addition to performing the migration tests, we also observed the time it takes to complete the container migration for the various HPC applications being tested. Table 3.3 shows the average migration times that we obtained initially. The overall average migration time was 34 seconds. Looking at the timings, the migration time was influenced by the size of application binaries and dependent libraries stored inside the container. In the case of Palabos, GalaxSee, and the ECLIPSE simulator, the application binaries and dependent libraries were stored on a shared NFS storage and not inside the container, thus making the container's size relatively smaller. However in the case of Fluidity and Flow, the installation of the applications was performed using Linux's RPM packages. This method ends up installing the application binaries and their dependencies inside the container, thus increasing the size of the container.

We also noticed that the migration time did not change much whether we were using the 1, 10, or 25 Gigabit network. This was initially puzzling to us as we were expecting that faster network speeds would speed up the copying of the migration data from one

machine to another. It seemed to us later that the bottleneck was not the network speed, but rather the read/write speeds of the local disks storing the container migration dumped data. The AWS instances by default use a General Purpose SSD (gpt2) local disk that has a limited throughput. The results obtained in Table 3.3 were done using the standard gpt2 disks. AWS also provides higher throughput disks called Provisioned IOPS SSD (io1) disks. We tested the migration with the io1 disks as well and the migration times improved as illustrated in Table 3.4. Using the enhanced disks reduced the migration times by around 35%. With that, the improved overall average container migration time was now 22 seconds instead of 34 seconds.

Application	Migration Time (seconds)
Fluidity	50
Flow	35
Palabos	30
GalaxSee	29
ECLIPSE	26

Table 3.3: Average container migration times (using standard gpt2 disks)

Application	Migration Time (seconds)
Fluidity	33
Flow	23
Palabos	19
GalaxSee	19
ECLIPSE	17

Table 3.4: Average container migration times (using enhanced io1 disks)

Another attempt to improve the migration time even further was to try to store the container data in a shared NFS storage instead of it being on the machine’s local disk. We hoped that this approach would eliminate the time spent on copying the container data over the network during migration. With OpenVZ, the data of the container is stored as a single-image file on a local disk partition called “/vz”. We attempted to have a single

shared “/vz” partition for all containers which was mounted from a shared NFS storage. However, this caused the containers to fail to even startup. At the time of our testing, it turned out that OpenVZ did not support hosting single-image container files through an NFS shared storage. Their user guide clearly states that the “/vz” partition needs to be on a local EXT4 file system [104]. With that in mind, we did not pursue the NFS approach any further.

3.4.3 Results Integrity Check

As part of the container migration testing, it was crucial for us to check the integrity of the results produced by the HPC applications once the migration was completed and the MPI job has progressed and finalized. We wanted to make sure that the migration was not causing any data corruption to the results produced, especially that the types of data produced varied by the application.

The different HPC applications being tested produced several types of output files. For example, Fluidity produced binary files (e.g. .vtu and .pvtu files) for post-processing with ParaView [105]. Palabos also produced binary files for post-processing with ParaView (e.g. .vtk files) as well as GIF image files. The ECLIPSE simulator and Flow produced text files (e.g. .log and .prt files) as well as binary output files for post-processing (e.g. .unsmry and .egrid files). GalaxSee did not generate any output files, however it produced an in-situ animated visualization of the simulation. Our approach to verifying the integrity of the results was as follows. For each application, we do a full run on the container environment without any container migration and we save the application results. Next, we do the same run but trigger a container migration at a random time step during the run. We take note at which time step in the run we triggered

the container migration. Once the migration is done, we let the run complete and save the results produced. Finally, we do a side-by-side comparison of the results produced with and without container migration.

To compare the application results, we used several comparison methods. For applications that produced data files that can be post-processed, we visually compared results using the appropriate post-processing tools (e.g. ParaView, etc.). For applications that produced text files, we compared the end results reported in the files. We used tools such as WinMerge and vimdiff to do a side-by-side comparison of the files [106]. We also used the md5sum tool [107] to do a checksum comparison on binary files produced (e.g. .gif, .pvtu, etc.). For the GalaxSee application, we did a visual comparison of the animated results since the application did not generate any output files for comparison.

Next, we share some examples demonstrating the data integrity checks. Figure 3.13 is from a Palabos run for a simulation of the Rayleigh Taylor instability. The container migration was triggered randomly at time step 2400 during the simulation. We compare the results produced with and without container migration at time step 2400, at time step 3000 after the migration completed and the simulation has progressed a bit, and lastly at the final time step. From the visual results, we can see that both results were the same with no corruption in the data produced. Figure 3.14 shows a side-by-side comparison for the log text files produced by a run for the ECLIPSE simulator, with and without container migration. Both runs finished at the same final time step 231 and produced identical results. Figure 3.15 is from a Fluidity run for a simulation of tephra, volcanic ash particles, settling through a water tank. The container migration was triggered randomly at time step 100. We compare the results produced with and without container

migration at time step 100, at time step 125 after the migration completed and the simulation has progressed, and lastly at the final time step. Again, both post-processed results looked the same. Figure 3.16 and Figure 3.17 show an md5sum comparison check of the binary files produced by Palabos (.gif files) and Fluidity (.pvtu ParaView files) respectively. The checksums were identical for the files produced with and without migration. For all HPC applications tested, we did not have any discrepancy issues with the results. All the runs with and without container migration produced consistent results.

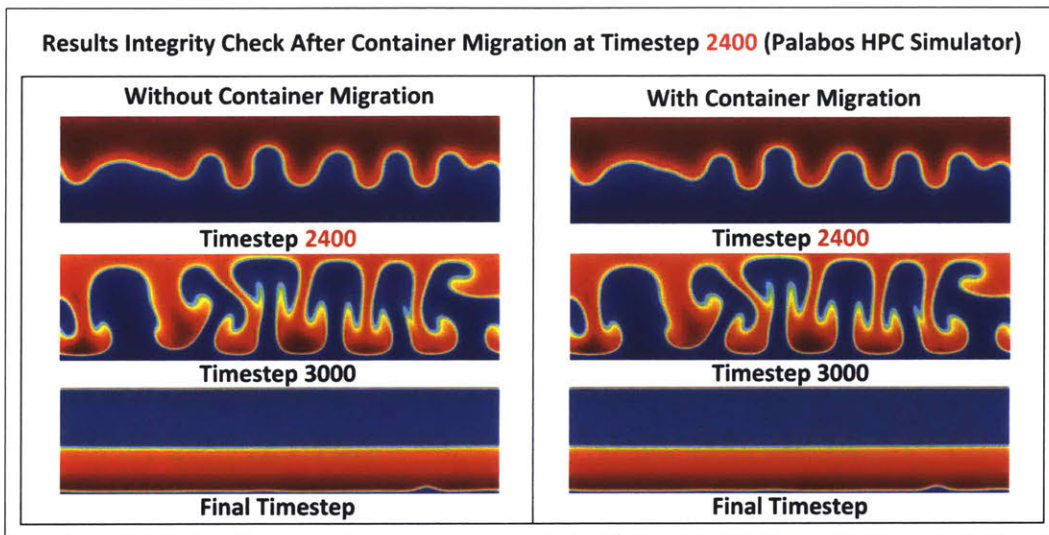


Figure 3.13: Results integrity check for the Palabos simulation

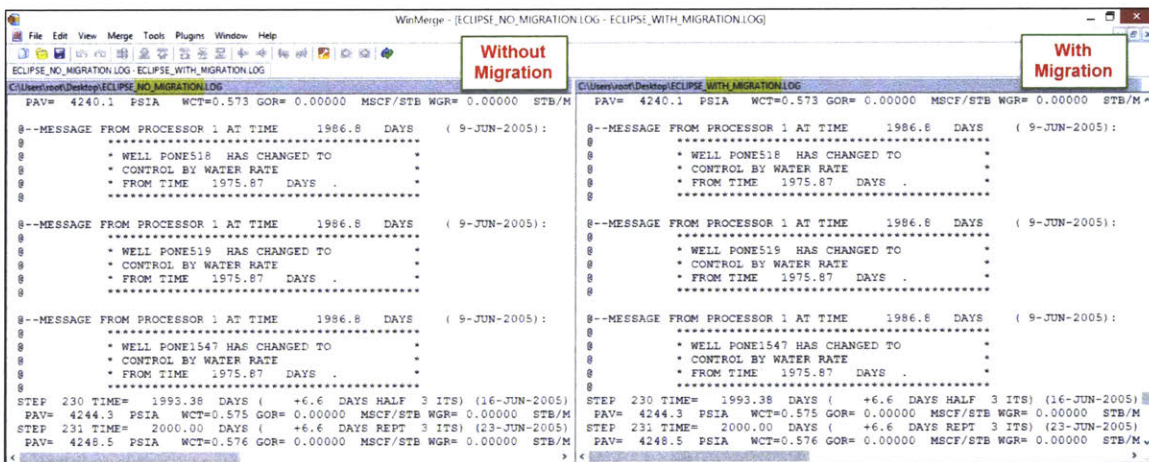


Figure 3.14: Results integrity check for the ECLIPSE simulator

**Results Integrity Check After Container Migration
at Timestep 100 (Fluidity HPC Simulator)**

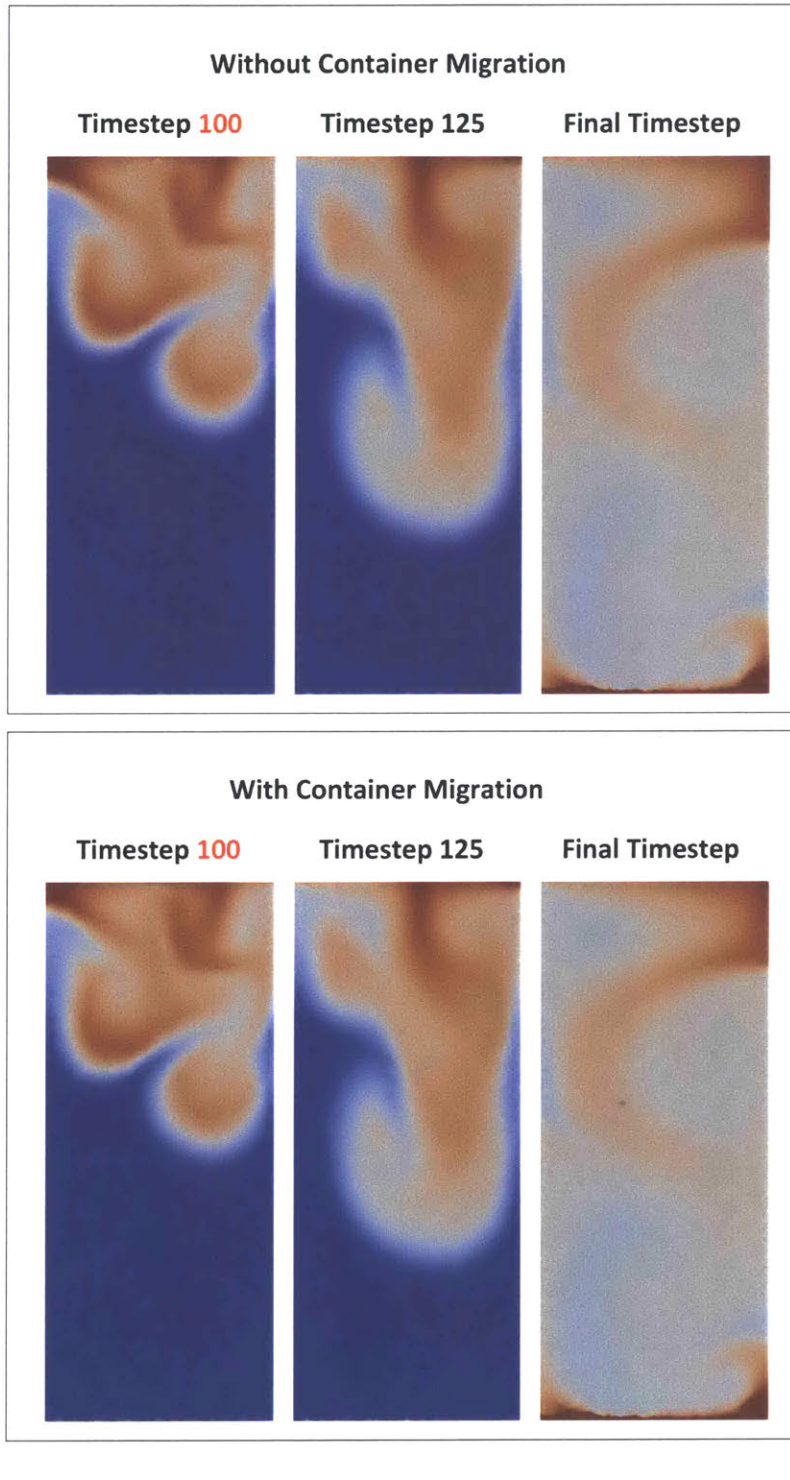


Figure 3.15: Results integrity check for the Fluidity simulation

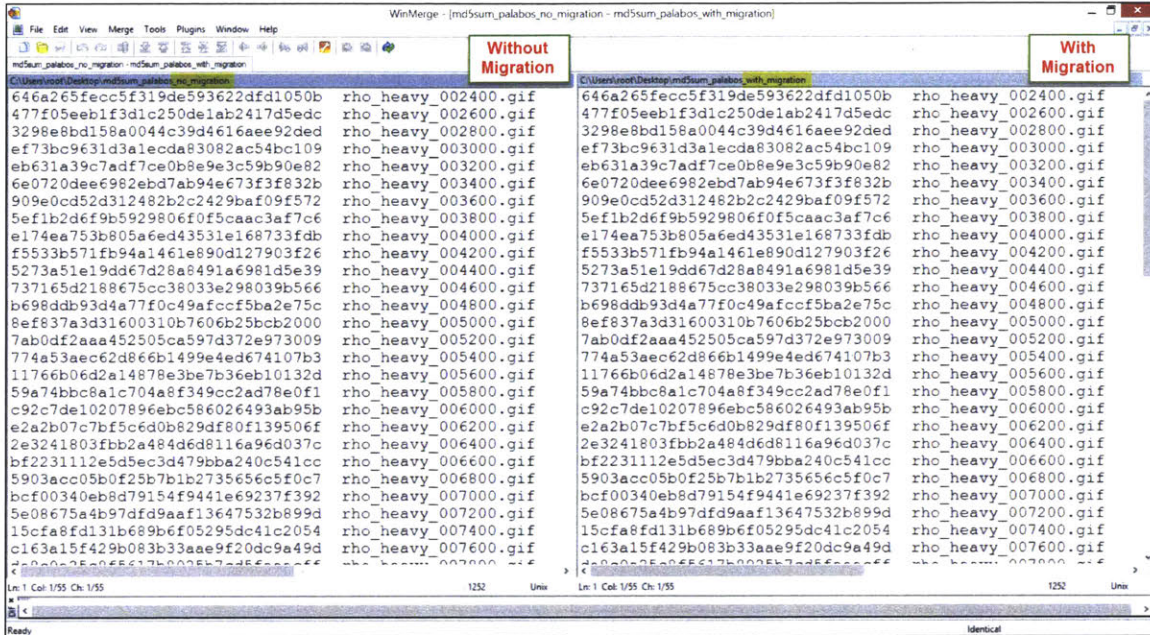


Figure 3.16: Md5sum integrity check for gif files produced by the Palabos simulation

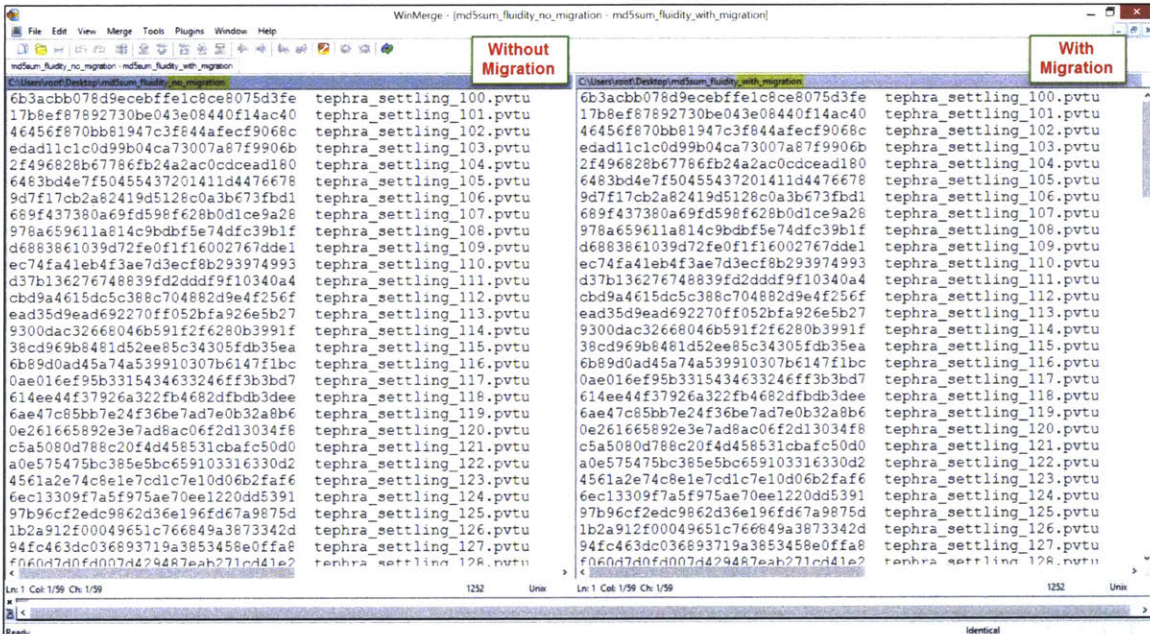


Figure 3.17: Md5sum integrity check for ParaView pvtu files produced by the Fluidity simulation

3.4.4 Container Demo Videos

In this section, we share several YouTube videos demonstrating the successful container migration for some of the HPC applications we tested. We describe the details of each individual test and reference the links for the videos. The videos are visually annotated so that the viewer can have a clear understanding of the test scenarios.

The test environment is the same as illustrated in Figure 3.11. The general scenario of the demo tests is as follows. The MPI job is initially launched on two containers that are hosted on two different native machines. While the job is running, one of the containers is migrated to a spare machine. During migration, the MPI job temporarily freezes with minimal interruption. Once the migration is completed, the MPI job resumes. During the test, the Linux “top” command is run inside each container to monitor the state of the MPI processes. On the native machines, we use the Linux “wall” command to run the “prctl” tool every one second to list the containers and their states on each native machine. This helps visually catching the transition states of the containers as they are being migrated. There are some slight variations of the test scenarios for the different applications, which we will address next as we describe the videos. Table 3.5 provides the YouTube link references for the video demos.

Application	Demo Video Link
Palabos	https://youtu.be/1v73E2Ao3Mk
ECLIPSE	https://youtu.be/5tz6JP2UgTk
GalaxSee	https://youtu.be/NIT7nJ-yENc
Flow	https://youtu.be/KNTVHQnMVHU

Table 3.5: Video links for container demos

The first video demonstrates the container migration for the Palabos application [108]. The demo shows the application being launched on two containers with two MPI processes on each. The run is for a 2D simulation of the Rayleigh Taylor instability. As the application is running, we can see the results and time stepping on the launch terminal after the application is started with the mpirun launcher. The run also shows a 2D in-situ visualization of the simulation results as it is running. The container migration for container “sindi_ct1” is then triggered at a random time step, in this case it was after time step 2400. During the migration, we can see the state of the container changing from “running” to “suspended” on the native machine it was initially launched on. We can see that the MPI job freezes temporarily and the terminal displaying the time stepping results also freezes. Similarly, the 2D visualization of the simulation temporarily freezes. Once the container is migrated successfully to the spare machine, the state of the container changes back to “running” and the time stepping and 2D visualization of the MPI job resumes and progresses from the same point before it froze. The overall migration time was around 35 seconds.

The second video demonstrates the container migration for the ECLIPSE simulator application [109]. We were curious to see how the MPI job behaves if the container was to be migrated more than once. In this scenario, we migrate the container back and forth in a ping-pong fashion between two native machines. We migrated the containers back and forth six times with no issues. The video however only shows an example by migrating back and forth once. The job is launched on two containers, with four MPI processes each. The simulation is run using the “PARALLEL_15000CELLS” test model. The first container migration is triggered during a random time step, in this case it was

after time step 36. During migration, the time stepping output to the terminal freezes. Once the migration to the spare machine is completed, we can see the simulation continuing at time step 37 and progressing. Next, the container is migrated again, this time from the spare machine back to the original machine where it was initially running on. The second migration is triggered after time step 70, and once it is completed, the application resumes at time step 71 and progresses. The average time of both migrations was 28.7 seconds.

The third video demonstrates the container migration for the GalaxSee application [110]. In this scenario, we try to swap two containers simultaneously, “sindi_ct1” and “sindi_ct2”, from two different native machines while an MPI job is running inside of them. We did this test because we were curious to see how the native machine would behave as it was getting ready to pack and migrate the data of one of the containers, and at the same time, it was also getting ready to receive the data of another container being migrated into it. The demo shows the application being launched on two containers with four MPI processes on each. The simulation is done for eight thousand stars and an in-situ visualization is displayed during the simulation. During the swap, the visualization temporarily freezes, then resumes from the same visual image stage before it froze and progresses. Looking at Table 3.3, the migration time we reported previously for GalaxSee was 29 seconds. However, in this case it was a bit higher and around 38.5 seconds. We believe this is due to the simultaneous migrations taking place as the data of the migrated containers are being copied at the same time between the two native machines involved.

The last video demo is for the Flow application [111]. This test is different from the others as it does not involve container migration, but rather further tests the container’s

suspend and resume capabilities. In this scenario, we launch an MPI job on two containers with four MPI processes each. Next, we suspend all the containers involved in the job, in this case the job freezes after time step 6. Once all containers are suspended, we power off the two native machines hosting the two containers and then power them on again. This can be seen in the video as we issue the “reboot” command on both native machines. Once both machines are up again, we resume both containers. The MPI job then resumes successfully at time step 7 and progresses. The demo shows that the container’s resilience capabilities are not only limited to migrations. The ability to suspend an entire MPI job then resuming it can be useful for HPC centers. For example, this might be useful in events such as scheduled system downtime maintenance, or perhaps when doing system patching and upgrades.

3.5 Challenges and Solutions

We were faced with various technical challenges throughout our testing when using containers in an HPC environment. In this section, we will go over the challenges encountered and the solutions adopted. Some of the challenges affected the container’s migration, while some affected the container’s functionality and performance.

The first issue faced was during the testing of the multiple MPI libraries we were evaluating with the container environment. Three types of commonly used MPI libraries were targeted, MPICH, Intel MPI, and Open MPI. We were very keen to test MPICH with our environment since MPICH, and its derivatives, is considered the most widely used implementation of MPI in the world according to its website [62]. We believed that getting MPICH to work was crucial as it is the baseline for several other MPI

implementations such as MVAPICH, IBM MPI, Cray MPI, Microsoft MPI, and Intel MPI. Testing Open MPI was also important to us since it is a popular implementation in the HPC domain and is not a derivate of MPICH.

The testing with MPICH and Intel MPI was trouble-free. We did not have any technical issues with them in terms of launching MPI jobs on the containers, or when it came to migrating the containers. On the other hand, we faced a few challenges with Open MPI. We used Open MPI version 1.10.7, which was the current during our time of testing. Initially, the basic launching of an MPI job to run inside the containers was failing. The MPI jobs would fail to start and gave the error “tcp_peer_send_blocking: send() to > socket 9 failed: Broken pipe”. Investigating this further, Open MPI appeared not to support virtual IP interfaces. In this case, OpenVZ uses such interfaces inside the containers for the network communication (i.e. `venet0` and `venet0:0` interfaces). This limitation was documented on the Open MPI frequently asked questions (FAQ) page at the time of our study [112]. The FAQ pages mention that this might get fixed in future releases. It also references a trouble ticket that addresses this issue [113]. At the time of our study, that ticket was still in open state and not resolved.

Even though we faced this limitation with Open MPI, we applied a hack to the Open MPI source code that would enable us to run MPI jobs on the OpenVZ containers, which can be found in this reference [114]. It is a workaround and is not in the Open MPI production code, hence there is no guarantee for its proper functionality or stability. However after applying the hack, we were able to run MPI jobs successfully to completion on the containers.

Another challenge we had with Open MPI was related to the connection timeout behavior of its MPI daemons. During the container migration testing, the MPI processes in the migrated container are suspended, and as a result, the entire MPI job freezes temporarily until the migration completes and the job resumes. During this suspension phase, we wanted to see what kind of tolerance the remaining active MPI processes would have before they timeout and die. To test this tolerance, we did some tests where we intentionally suspend a container for more than thirty minutes while the MPI processes running on it freeze, and then we migrate and resume the container. The MPI jobs had no issues when doing this test with MPICH or Intel MPI, even with the prolonged suspension time. However, with Open MPI it was a different case. With Open MPI, we were successful in migrating the containers as long as the migration time was around one minute. In the test case where the migration time exceeded one minute (e.g. around one minute and thirty seconds), the MPI daemons/processes TCP network connection seemed to timeout and die before the migration completed, which caused the MPI job to fail. The error produced was the following “ORTE has lost communication with its daemon located on node. This is usually due to either a failure of the TCP network connection to the node, or possibly an internal failure of the daemon itself. We cannot recover from this failure, and therefore will terminate the job”. Investigating this further, the Open MPI FAQ page suggested adjusting several Linux kernel parameters to change the default tolerance behavior of the TCP connection timeouts [115]. We tried adjusting these Linux kernel parameters to increase the timeout behavior (e.g. `tcp_syn_retries` and `tcp_keepalive_time` parameters), however this did not seem to have an effect on the Open MPI daemons and they still timed out. We also came across a

similar issue that was reported previously in the Open MPI online forum. At the time of our study and according to one of the lead run-time environment developers for Open MPI, the model to adjust the TCP connection timeout behavior for the MPI daemons was currently not supported, but might be added in future releases [116]. Having this in mind, we decided not to investigate this issue further as it seemed to be only specific to Open MPI. It is also worth noting that adjusting the Linux kernel parameters to increase the timeout behavior was effective with MPICH and Intel MPI, unlike Open MPI.

A different and more challenging issue encountered was related to having a shared NFS storage mounted inside the containers. This shared storage is crucial to the functionality of the HPC applications as it is the central location storing the input/output data for the applications. Initially, when we attempted to migrate a container with an active NFS mount, the migration process immediately hanged and the migration eventually failed. The hanging was taking place in the first step of the migration process, which is suspending the container. The suspend operation would hang until it timed out and then gave an error message stating that the container suspension had failed. Investigating this further, we were able to pinpoint that the issue was related to the CRIU library. To be more specific, CRIU was hanging during the execution of the code in one of its files "nfs-ports-allow.sh". Debugging that code further, we found that the code was hanging in the "nfs_server_ports" function. The function tries to run the Linux "rpcinfo" tool to remotely query NFS ports on the shared storage system mounted inside the containers, which was causing the hanging and failure of migration. We implemented a simple fix to CRIU's original code to overcome this issue. This involved disabling the "nfs_server_ports" function causing the hanging, and instead of relying on the "rpcinfo"

tool, we replaced that code with a variable that references the appropriate standard NFS port numbers. For example, in our case the storage mounts were using the NFSv4.1 protocol, so the standard corresponding port number was port 2049. After applying this fix to the CRIU code, we were able to successfully migrate the containers with no issues.

Another problem we had in regards to container migration was specific only to the ECLIPSE simulator application. Initially when attempting to migrate a container with the ECLIPSE simulator running in it, the migration would fail in its first stage while suspending the container. Investigating this further, it was clear to us that the issue was with the CRIU library. Logs were showing the message “remote posix locks on NFS are not supported yet”. Apparently, CRIU currently has a limitation of not being able to suspend processes that use file locks on a shared storage system [117]. Some parallel applications may place such file locks on some of their generated output files while the application is running. Investigating the output files produced by the ECLIPSE simulator, it was indeed the case. The simulation run was producing a database file with the file extension “.dbprtx” which had a file lock applied to it. Looking at the release notes of the ECLIPSE simulator, it mentions that producing this file is optional and that it might cause issues on shared file systems that do not support file locking (e.g. Lustre file system) [118]. The solution to this was to prevent the generation of this optional file during the simulation run. This was achieved by setting the keyword “MESSSRVC” to off in the input model file for the ECLIPSE simulator. After applying this fix, we were able to successfully migrate the containers with no issues.

The final issue encountered was concerning the performance of the applications on the containers. The performance issue only surfaced when we were benchmarking on the

instances that had 10 and 25 Gigabit networks. For some of the runs, the benchmarks on the containers were more than 20% slower compared to the native system. Investigating this further, it turned out to be a network tuning issue. One of the network parameters that can affect the network's performance is the Maximum Transmission Unit (MTU). The MTU is the size of the largest allowed packet that can be transmitted over the network. Having larger MTU means you can transfer more data in an individual network packet being transmitted. As a result, this can improve network performance and latency. When launching instances with 10 and 25 Gigabit networks, the default MTU setting for their network interfaces was set to a value of 1500 bytes. This value is usually more suitable for 1 Gigabit networks. Hence, we had to manually change the default MTU value from 1500 to 9000 (usually referred to as jumbo frames) in order to take advantage of the enhanced networking provided by the underlying system. This was also the recommendation we later found in AWS's documentation for local node communication in a cluster [119]. We used the Linux "ip link" tool to change the MTU value of the network interfaces. It is very important to note that it was not sufficient to only change the MTU on the interfaces of the native system, but it was also crucial to change it on the network interfaces inside the containers as well (e.g. `venet0` interface). Even though the container's network is run in a host-routed mode, we noticed that changing the MTU only on the interfaces of the native system enhanced the performance for the native runs only, while the performance on the container runs was noticeably slower. However, once we applied the change to the network interfaces on both the native and container environments, the performance improved and results became consistent.

3.6 Summary

In this chapter, we went over the remedy component of our fault tolerance framework. This included the design and testing details of a remedy environment that utilizes the Linux container technology. We presented our successful experiences in using CRIU container migration as a means to improve the resilience of running HPC workloads on commodity clusters, which are commonly prone to hardware failures. We have tested the container's migration capabilities using various real scientific HPC applications that are based on the MPI standard. Results show that we can successfully migrate containers with HPC workloads between different physical machines with minimal interruption, without application modification, and without any data corruption to the produced results. To the best of our knowledge, we believe such successful demonstration of CRIU container migration using real HPC application is a first in the HPC domain. We have also conducted a broad range of performance benchmarks on containers using real HPC applications. This covers benchmarking using three network interconnect types and four different machine types varying in specs. Results show that the performance on containers was close to native. With this remedy environment design and obtained results, we have shown that the container technology can be a feasible resilience mechanism for workloads running on commodity HPC clusters.

Chapter 4

Framework Integration and Testing

In Chapters 2 and 3 of this work, we have discussed the implementation of the fault prediction component and remedy component, respectively, of our fault tolerance framework. In this chapter, we will go over the integration of these two components along with an HPC resource manager and a custom monitoring daemon, which all ultimately compose our full HPC fault tolerance framework. We present an overview of the fully integrated framework, and then discuss how the HPC resource manager was integrated. We follow that by sharing tests to demonstrate the framework's successful full functionality. Finally, a summary is presented.

4.1 Overall View of Integrated Framework

The overall fault tolerance framework will be the result of integrating our fault prediction component and remedy component along with an HPC resource manager and a

custom monitoring daemon. The following scenario summarizes a full operational cycle of how the framework will work. First, a user submits an MPI workload to run on an HPC cluster requiring several hours or days to complete. The workload is submitted to the cluster using a typical HPC resource manager. The workload is then prone to failure if one of the compute nodes was to have a hardware issue. While the workload is running, the framework's fault prediction component is continuously on the look for compute nodes that could potentially fail soon. The prediction component is capable of processing massive amounts of online node logs as they are being produced live from the HPC system. The fault predictions are done in an appropriate time that is significantly less than the system's MTBF. Once the prediction component identifies a node with potential problems, it will report the node to the cluster's resource manager to offline it from the resources pool and maintain the bookkeeping of the node's status. As the offlined node still has an active workload running on it, the remedy component next migrates the workload from the faulty node to a healthy spare node. This involves minor interruption to the workload during migration, but the workload continues normally on the healthy node once migrated. Finally, the faulty node is now free from any active workload and ready for maintenance. A full cycle of the overall framework environment is summarized in Figure 4.1 with steps.

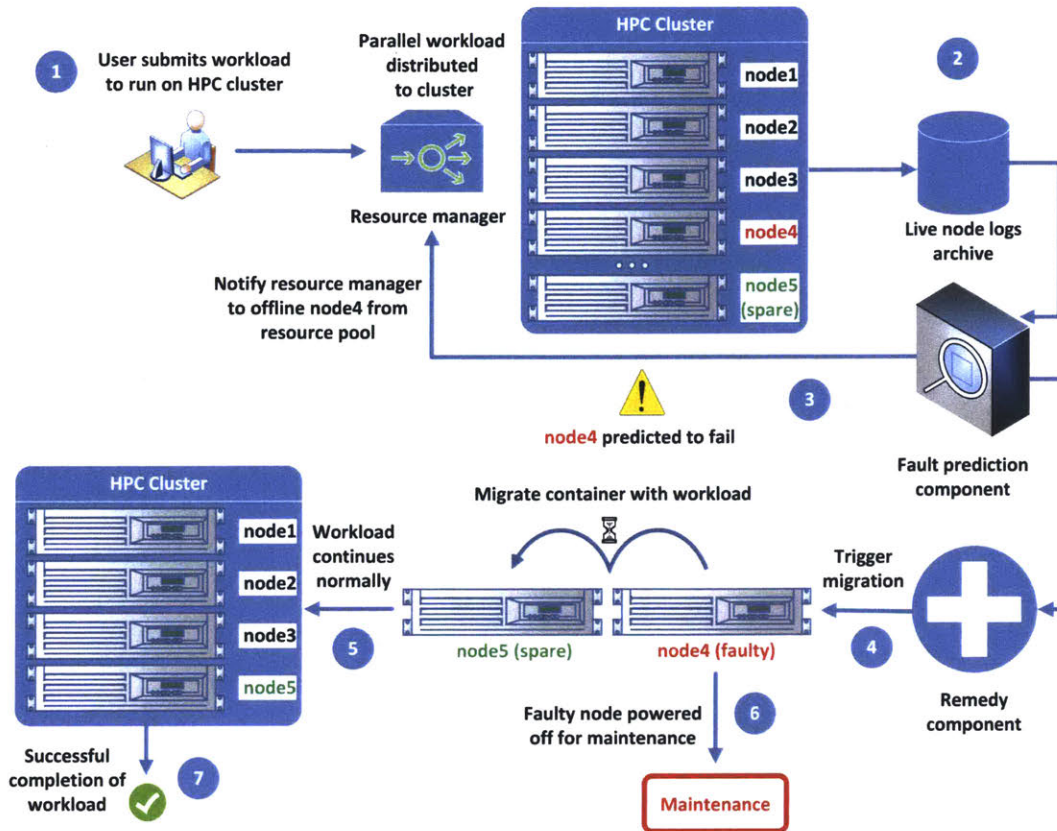


Figure 4.1: Full cycle of overall framework environment

4.2 Integration with the HPC Resource Manager

One of the critical components of an HPC environment is the workload resource manager. A resource manager is a software component usually used to schedule the execution of HPC workloads on an HPC cluster through a queueing system. It is also used to do the bookkeeping of the nodes in a cluster in terms of their status and health. There are several implementations of such resource managers, some are open-source, while others are commercial providing professional support. Common examples of HPC resource managers include TORQUE [120], PBS [121], SLURM [122], and UGE [123]. They all provide similar functionalities, but might have different command syntax to accomplish the tasks.

For our test environment, we used the open-source implementation of TORQUE (version 6.0.1) as the HPC resource manager. It is based on the original Open Portable Batch System (Open PBS) project developed by NASA [124], however with additional extended functionalities to provide better scalability. A typical configuration of this resource manager would involve a server, scheduler, and client daemons to be set up. On the master cluster node where jobs would be launched, the `pbs_server` and `pbs_sched` daemons are set up. The `pbs_server` is the master daemon monitoring and controlling all compute nodes of the cluster. The `pbs_sched` is the basic default scheduler daemon for the queuing system. There are more sophisticated scheduler implementations to use, such as the commercial Moab scheduler [125], however for our testing purposes, the default scheduler was sufficient. Finally, the `pbs_mom` is the client daemon running on each compute node, which reports its status to the `pbs_server`.

In integrating our container environment with the resource manager, we wanted to minimize any changes to the typical resource manager setup you would have on a standard cluster. As described previously, every physical node in our cluster has a pre-running container set up on it, which encapsulates the entire node resources. The physical node has an IP address distinct from the IP address of the container running on it. The IP of the container is a floating IP address that moves with the container as it is being migrated. The `pbs_server` maintains a node list of IP addresses or hostnames of the compute entities it is managing. Our initial thought was to populate the `pbs_server` node list with the IPs of the containers instead of the typical setup where it would contain the IPs of the physical compute nodes. This would easily allow submitting workloads directly to the containers through the resource manager. However, such setup would then

complicate the bookkeeping done by the resource manager to maintain the node's status and health. This is because the resource manager will no longer be managing physical machines, but rather virtual container entities, which do not have a physical health status. With that in mind, we decided to keep the TORQUE system setup identical to what you would find on a non-container environment, where the `pbs_server` node list will contain the IPs of the physical compute nodes. The container integration with the resource manager is instead done dynamically when the workload is being submitted.

Submitting a workload through an HPC resource managing system typically requires preparing a launching script, which is then submitted to the job queueing system. Different resource managers can have different syntax options to use inside these launching scripts. The resource manager also provides several built-in environment variables accessible inside the launching script, which can be used to aid in launching the MPI job on the cluster (e.g. variable containing total number of compute cores job will use, etc.). Our approach to integrating the container environment with the resource manager was done using some of those environment variables provided by the system.

In TORQUE, the environment variable `PBS_NODEFILE` is automatically generated inside your launch script once the jobs are submitted through the resource manager. This variable points to a temporary file containing the IPs or hostnames of the physical compute nodes assigned to your job, which is then used as the hostfile that the `mpirun` command uses to launch the MPI job on the assigned nodes. Instead of using the `PBS_NODEFILE` to launch the job, we wrote a custom script that uses the content of that file to probe the physical nodes and retrieve the IPs of the containers currently running on those physical nodes. The script then constructs a new hostfile having the IPs of the

containers. The new hostfile is named `container_hosts` and is automatically placed in the same directory from which the job was submitted. The new hostfile is then used to launch the MPI job with the `mpirun` launcher. With this approach, we are able to launch jobs on the containers directly, and at the same time, the `pbs_server` is still doing the health and status bookkeeping of the physical compute nodes running the containers.

This approach to integrate the containers with the resource manager also enables us to make such integration almost transparent to the end users of the cluster. TORQUE allows the resource manager administrators to automatically execute a preparation script before any job is launched on the system. This type of script is referred to as a prologue script in the TORQUE system. We utilize prologue to launch our custom script that generates the `container_hosts` file having the IPs of the containers. With this technique, the only minor change that the end users will have to do in their launching scripts is to reference the `container_hosts` file instead of the default `PBS_NODEFILE` when launching the job with the `mpirun` launcher. The users will use the standard TORQUE `qsub` command to submit the job to the resource manager. Figure 4.2 illustrates the overall setup of the TORQUE resource manager with containers. Figure 4.3 shows the prologue script used. Figure 4.4 shows a sample TORQUE submission script that will launch the jobs on the containers.

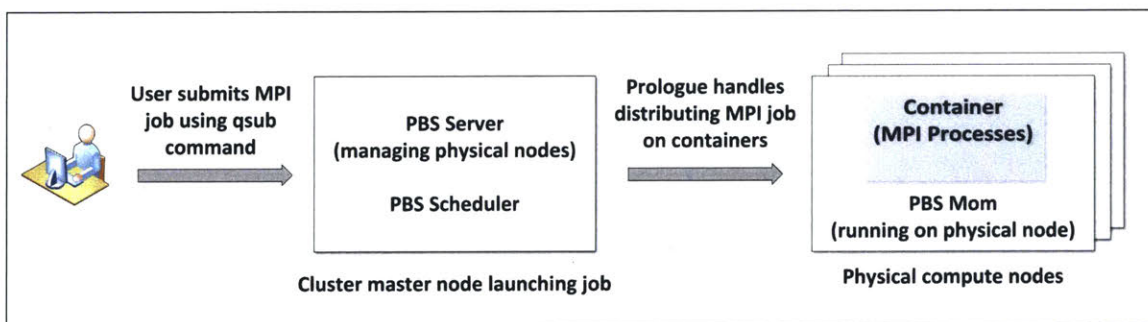


Figure 4.2: Overall TORQUE resource manager setup with containers

```

#!/bin/bash
#Change working directory to where the workload was launched from
cd $PBS_O_WORKDIR
#Remove any existing old host files
rm -f container_hosts
#Generate new host file with container IPs
for node in `cat $PBS_NODEFILE | uniq`;
do
    export ct_ip=`ssh $node "prctl list -a -o ip | grep -v IP_ADDR | xargs"`
    echo "$ct_ip:$PBS_NUM_PPN" >> container_hosts
done
#Change permission of new host file for user to access
chmod 666 container_hosts

```

Figure 4.3: Prologue script for TORQUE resource manager to generate containers host file

```

#!/bin/sh
#PBS -N palabos_job
#PBS -l nodes=2:ppn=4,walltime=00:15:00
#PBS -q batch
#PBS -o palabos_out
#PBS -e palabos_err
#PBS -V
#PBS -X
#Change working directory to where workload was launched from
cd $PBS_O_WORKDIR
#Launch MPI job on containers
mpirun -enable-x -np $PBS_NP -hostfile ./container_hosts ./rayleighTaylor2D.exe

```

Figure 4.4: Example TORQUE script to launch MPI job on containers

4.3 Full Cycle Testing of Framework

In this section, we describe the test environment used to demonstrate the overall functionality of the framework. The test scenario performed is similar to the steps illustrated previously in Figure 4.1. We also provide video examples demonstrating a full cycle test of the framework's fault prediction and remedy functionality using real applications and the TORQUE resource manager.

A full cycle test starts by a user submitting an MPI job to the container-based test cluster. The submission is done through the cluster's resource manager using the standard `qsub` command. The job is then observed while it is running and being managed by the resource manager. As the job runs, the Linux system logs from all nodes in our test environment are gathered live to the logs' central archival repository. At the time of testing, there were more than one million text logs already gathered in the central logs archival repository, as nodes have been forwarding their live logs for a while. Next, a hardware precursor error is simulated on one of the compute nodes that the job is currently running on. This is done by injecting a single real precursor log into the online live system logs using the Linux `echo` command. Table 4.1 shows several examples of such logs that we gathered and used for testing. Some of the test logs came from SNL, while others came from examples we gathered online from the web [126]–[134]. As we inject the log, the fault prediction component of the framework is continuously monitoring the live logs of the system while the MPI job is running. The fault prediction component next detects that a node has a potential hardware issue and alerts the resource manager. The resource manager then puts the affected node offline from the active resources pool. This also alerts the remedy component to take action. The remedy

component will then trigger the migration of the container running on the affected node and have it migrated to a healthy spare node. The workload continues progressing normally after being migrated. Finally, the affected node is free from any workload running on it and can be taken down for maintenance.

We implement a monitoring daemon that encapsulates the framework's functionality entirely. The monitoring daemon is run on a single standalone node. For convenience, we run it on the master node of the cluster where the resource manager server is running and where the live central logs of the cluster are accessible. It is set up to run as Linux cron scheduled job [135], where it is scheduled to run every 60 seconds. Once a failure precursor log is injected into the live system logs, the monitoring daemon was capable of predicting the troubled node within the 60 seconds timeframe it was scheduled to run on. Overall, the tasks performed by the monitoring daemon can be summarized in a chronological order as in Figure 4.5.

1. Run fault prediction algorithm on live stream of cluster's system logs every 60 seconds.
2. Notify resource manager about predicted faulty nodes.
3. For every identified node that is still active in the resource pool:
 - Offline node from current resource pool with timestamp.
 - Attach a note to the offlined node in the resource manager bookkeeping system, which references the potential troublesome system logs.
 - Notify remedy component and trigger migration of container from affected node to healthy spare node.
 - Email incident details to system administrator for record keeping and to have affected node down for maintenance.

Figure 4.5: Summary of monitoring daemon performed tasks

Hardware Category	Log
Memory	Memory device status is non-critical Memory device location: DIMM1_A Possible memory module event cause:Single bit warning error rate exceeded
Memory	Memory device status is critical Memory device location: DIMM1_B Possible memory module event cause:Single bit error logging disabled
Memory	MC2 Error: : SNP error during data copyback
Memory	MC1 Error: Parity error during data load
Memory	MC4 Error (node 1): DRAM ECC error detected on the NB
Memory	Data Cache Error: during L1 linefill from L2
Memory	Instruction Cache Error: Parity error during data load
Memory	Bus Unit Error: GEN parity/ECC error during data access from L2
Memory	Northbridge Error (node 1, core 0): L3 ECC data cache error
Memory	bus error 'generic participation, request timed out generic error mem transaction generic access, level generic'
Memory	Northbridge Error (node 1): DRAM ECC error detected on the NB
Disk	in start_transaction: Journal has aborted
Disk	in ext3_dirty_inode: IO failure
Disk	ext3_journal_start_sb: Detected aborted journal
Disk	in ext3_reserve_inode_write: IO failure
Disk	ext3_get_inode_loc: unable to read inode block
CPU	Processor sensor detected a failure value Sensor location: PROC_2 Chassis location: Main System Chassis Previous state was: Unknown Processor sensor status: Presence detected, IERR
CPU	NMI: IOCK error (debug interrupt?) for reason 60 on CPU 0.
Power Supply	Power supply detected a failure Sensor location: PS 1 Status Chassis location: Main System Chassis Previous state was: OK (Normal) Power Supply type: AC Power Supply state: Presence detected, Failure detected, AC lost
Fan	Fan sensor detected a failure value Sensor location: FAN 4A RPM Chassis location: Main System Chassis Previous state was: Unknown Fan sensor value (in RPM): 0
Battery	Voltage sensor detected a failure value Sensor location: ROMB Battery Chassis location: Main System Chassis Previous state was: OK (Normal) Discrete voltage state: Bad

Table 4.1: Examples of injected logs to simulate hardware error precursor

Finally, we provide two video examples demonstrating a full cycle test of the framework’s functionality. The YouTube links for the video demos can be found in Table 4.2. The first video uses the Palabos application [136], while the second one uses the Fluidity application [137]. In both videos, the application is launched on two containers with four MPI processes on each. Several commands of the TORQUE resource manager can be seen used to submit the application and check the status of the job and resources. The application is submitted using the qsub command, followed by checking its running status using the qstat command. After that, the “pbsnodes -l” is used to list any nodes in the system that are offlined or down, and at this point none exist. As the application is running, we can see a hardware precursor error artificially being injected, using the Linux echo command, into the live system logs. Shortly and within a 60 seconds timeframe, we see the MPI job temporarily freezing as the monitoring daemon has predicted that the affected node could fail soon, and then triggered the migration of the container from the affected node to a healthy spare node. Once migrated, the MPI job continues progressing successfully. Finally, the “pbsnodes -l” command is used again to list any nodes that are in an offline or down state, and at this point we see that the troubled node was automatically offlined by the monitoring daemon and has been taken off the active resources pool.

Application	Demo Video Link
Palabos – Full Cycle Demo	https://youtu.be/KW60Ju6wWac
Fluidity – Full Cycle Demo	https://youtu.be/DCtFhxve5Hc

Table 4.2: Video demos for framework full cycle tests

4.4 Summary

In this chapter, we presented how the HPC fault tolerance framework was fully integrated and tested. The integration of the fault prediction component, the remedy component, the TORQUE resource manager, and our custom monitoring daemon all compose the overall framework. An overview of how a full operational cycle of the framework was also presented. We then discussed how TORQUE's prologue was used to integrate the resource manager with the container-based environment. With this setup, the resource manager was successful in distributing the launched MPI jobs on the containers, and at the same time, it could still do the bookkeeping of the status and health of the physical nodes hosting the containers. After that, we discussed the steps involved to do a full cycle test of the framework. This included submitting a real HPC job through the resource manager, then artificially injecting various hardware precursor errors into the live logs of a participating node to simulate hardware issues. In all test cases, the monitoring daemon was capable of predicting the troubled node within a 60 seconds timeframe once the precursor errors were injected. The active workload running on the troubled node was also successfully migrated to a spare node and progressed normally. Finally, we provided two example YouTube videos successfully demonstrating the steps of a full cycle test of the framework, which used real HPC applications and the cluster's resource manager.

Chapter 5

Conclusions and Future Work

5.1 Summary and Conclusions

In this work, we designed and implemented a lightweight fault tolerance framework for HPC workloads. The framework serves to improve the resilience of running HPC workloads on commodity HPC clusters that are frequently prone to hardware issues. The main contributions of this work can be summarized as follows.

First, a fault prediction component was implemented to proactively predict hardware issues on compute nodes of a cluster. This is done by using our own developed parallel algorithm that uses a machine learning approach to analyze system logs of an HPC system. The algorithm was tested on massive amounts of real data, ~750 million logs, obtained from three supercomputers from SNL. In addition, the algorithm was also tested online on our live test cluster. Compared to previous related work, our fault prediction

algorithm provides improved results in terms of fault prediction accuracy, performance (i.e. time needed to do fault prediction), and overhead reduction on the underlying system.

Second, a remedy component was implemented to sustain the HPC workloads running on the affected system once faults are predicted. This was done by adapting the Linux container technology into an HPC environment in order to make use of its migration capabilities. A code modification to the open-source CRIU library was also contributed to enable the successful migration of containers having NFS mounts. By using our container-based HPC environment, we were able to migrate actively running HPC workloads from machines having potential hardware problems, to healthy spare machines without affecting the running workload. The environment was tested successfully using five real HPC applications with three different implementations of the MPI library (MPICH, Intel MPI, and Open MPI). The environment did not impose any significant interruption or overhead to the HPC workload running on the system, nor did it require any modifications to the application. To the best of our knowledge, we believe this work is the first in the HPC domain to demonstrate successful migration of MPI-based real HPC workloads using CRIU and containers.

Third, we prepared and shared several annotated YouTube videos demonstrating the successful container migration of the various HPC workloads tested, as well as videos demonstrating the full operational cycle of our framework's functionality. This included testing standard computational workloads, as well as workloads that produced in-situ visualizations during the migration. To the best of our knowledge, we believe such video demonstration is also a first in the HPC domain.

Fourth, we performed comprehensive performance benchmarks on containers using various real scientific HPC applications. The benchmarks were performed on a wide range of computational platforms, and they compared the performance of containers to the native system. Previous related work benchmarking containers mainly use generic HPC benchmarks (e.g. HPL and NPB) and use a single computational platform for benchmarking.

Finally, as the use of containers in HPC is a relatively young topic, we believe that the challenges we faced with the wide range of tests performed, and the solutions adopted, are all valuable experiences to share with the HPC community.

5.2 Future Work

For future work, we plan to look into performing container migration for HPC workloads that use other container types such as Singularity and Docker. We also plan to investigate container migration in HPC environments having InfiniBand networks. InfiniBand is a popularly used interconnect type in HPC environments, which we think is crucial to be tested with container migration as well. We believe that InfiniBand will have its own unique challenges when it comes to container migration. However, the use of technologies such as IP over InfiniBand (IPoIB) [138] might help facilitate accomplishing such task. Also in an InfiniBand environment, we will have to test migration using the MVAPICH MPI library [139], as it is the most popularly used MPI implementation for such environments. Studies such as [52] were also able to run workloads on GPGPUs using containers, so it would also be interesting to look into the possibility of container migration when GPGPUs are involved, or when other accelerator

devices such as Intel Xeon Phi [140] are used. We will also be interested to look into methods to reduce or eliminate the container migration time for HPC workloads. This includes methods such as using shared file systems to host the container's data, or using live migrations techniques that could help hide interruptions during migration.

Bibliography

- [1] The TOP500 Project, “TOP500 List Statistics,” Jun-2019. [Online]. Available: <http://www.top500.org/statistics/list>.
- [2] The TOP500 Project, “TOP500 Supercomputer Sites,” Jun-2019. [Online]. Available: <https://www.top500.org/lists/2019/06/>.
- [3] F. Cappello, A. Geist, B. Gropp, L. Kale, B. Kramer, and M. Snir, “Toward exascale resilience,” *The International Journal of High Performance Computing Applications*, vol. 23, no. 4, pp. 374–388, 2009.
- [4] R. Riesen *et al.*, “Redundant computing for exascale systems,” *Sandia National Laboratories*, 2010.
- [5] K. Bergman *et al.*, “Exascale computing study: Technology challenges in achieving exascale systems,” *Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep*, vol. 15, 2008.
- [6] B. Schroeder and G. Gibson, “A large-scale study of failures in high-performance computing systems,” *IEEE Transactions on Dependable and Secure Computing*, vol. 7, no. 4, pp. 337–350, 2010.
- [7] P. H. Hargrove and J. C. Duell, “Berkeley lab checkpoint/restart (blcr) for linux clusters,” in *Journal of Physics: Conference Series*, 2006, vol. 46, p. 494.
- [8] D. W. Walker and J. J. Dongarra, “MPI: a standard message passing interface,” *Supercomputer*, vol. 12, pp. 56–68, 1996.
- [9] R. A. Oldfield *et al.*, “Modeling the impact of checkpoints on next-generation systems,” in *24th IEEE Conference on Mass Storage Systems and Technologies (MSST 2007)*, 2007, pp. 30–46.
- [10] I. Philp, “Software failures and the road to a petaflop machine,” in *HPCRI: 1st Workshop on High Performance Computing Reliability Issues, in Proceedings of the 11th International Symposium on High Performance Computer Architecture (HPCA-11)*, 2005.

- [11] T. Herault and Y. Robert, *Fault-tolerance techniques for high-performance computing*. Springer, 2015.
- [12] M. Snir *et al.*, “Addressing failures in exascale computing,” *The International Journal of High Performance Computing Applications*, vol. 28, no. 2, pp. 129–173, 2014.
- [13] A. B. Nagarajan, F. Mueller, C. Engelmann, and S. L. Scott, “Proactive fault tolerance for HPC with Xen virtualization,” in *Proceedings of the 21st annual international conference on Supercomputing*, 2007, pp. 23–32.
- [14] D. S. Wung, “Intelligent platform management interface (IPMI),” SLAC National Accelerator Lab., Menlo Park, CA (United States), 2010.
- [15] A. Oliner and J. Stearley, “What supercomputers say: A study of five system logs,” in *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN’07)*, 2007, pp. 575–584.
- [16] A. J. Oliner, A. Aiken, and J. Stearley, “Alert detection in system logs,” in *2008 Eighth IEEE International Conference on Data Mining*, 2008, pp. 959–964.
- [17] J. Stearley and A. J. Oliner, “Bad words: Finding faults in spirit’s syslogs,” in *2008 Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*, 2008, pp. 765–770.
- [18] K. Charoenpornwattana, C. Leangsuksun, G. Vallee, A. Tikotekar, and S. Scott, “A Scalable Unified Fault Tolerance for HPC Environments,” *9th LCI International Conference on High-Performance Clustered Computing*, Apr. 2008.
- [19] M. L. Massie, B. N. Chun, and D. E. Culler, “The ganglia distributed monitoring system: design, implementation, and experience,” *Parallel Computing*, vol. 30, no. 7, pp. 817–840, 2004.
- [20] W. Barth, *Nagios: System and network monitoring*. No Starch Press, 2008.
- [21] F. D. Sacerdoti, M. J. Katz, M. L. Massie, and D. E. Culler, “Wide area cluster monitoring with ganglia,” in *null*, 2003, p. 289.
- [22] M. Massie *et al.*, *Monitoring with Ganglia: Tracking Dynamic Host and Application Metrics at Scale*. O’Reilly Media, Inc., 2012.
- [23] W. Peng, T. Li, and S. Ma, “Mining logs files for data-driven system management,” *Acm Sigkdd Explorations Newsletter*, vol. 7, no. 1, pp. 44–51, 2005.
- [24] G. Hamerly and C. Elkan, “Bayesian approaches to failure prediction for disk drives,” in *ICML*, 2001, vol. 1, pp. 202–209.
- [25] E. W. Fulp, G. A. Fink, and J. N. Haack, “Predicting Computer System Failures Using Support Vector Machines,” *WASL*, vol. 8, pp. 5–5, 2008.
- [26] B. Allen, “Monitoring hard disks with smart,” *Linux Journal*, no. 117, pp. 74–77, 2004.
- [27] P. Broadwell, “Component failure prediction using supervised Naïve Bayes classification,” <http://citeseerx.ist.psu.edu/viewdoc/summary>, 2002.
- [28] A. Polze, P. Troger, and F. Salfner, “Timely virtual machine migration for proactive fault tolerance,” in *2011 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops*, 2011, pp. 234–243.
- [29] J. P. Walters and V. Chaudhary, “A fault-tolerant strategy for virtualized HPC clusters,” *The Journal of Supercomputing*, vol. 50, no. 3, pp. 209–239, 2009.

- [30] B. Davda and J. Simons, “RDMA on vSphere: Update and future directions,” in *Open Fabrics Workshop*, 2012.
- [31] W. Huang, J. Liu, B. Abali, and D. K. Panda, “A case for high performance computing with virtual machines,” in *Proceedings of the 20th annual international conference on Supercomputing*, 2006, pp. 125–134.
- [32] P. Kutch, “Pci-sig sr-ioV primer: An introduction to sr-ioV technology,” *Intel application note*, pp. 321211–002, 2011.
- [33] J. Liu, “Evaluating standard-based self-virtualizing devices: A performance study on 10 GbE NICs with SR-IOV support,” in *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, 2010, pp. 1–12.
- [34] J. Jose, M. Li, X. Lu, K. C. Kandalla, M. D. Arnold, and D. K. Panda, “SR-IOV support for virtualization on infiniband clusters: Early experience,” in *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, 2013, pp. 385–392.
- [35] Ohio State University, “OSU Micro-Benchmarks,” 2018. [Online]. Available: <http://mvapich.cse.ohio-state.edu/benchmarks>. [Accessed: 01-Apr-2018].
- [36] T. Shanley, *Infiniband Network Architecture*. Addison-Wesley Professional, 2003.
- [37] M. Musleh, V. Pai, J. P. Walters, A. Younge, and S. Crago, “Bridging the virtualization performance gap for HPC using SR-IOV for InfiniBand,” in *2014 IEEE 7th International Conference on Cloud Computing*, 2014, pp. 627–635.
- [38] Mellanox, “HowTo Configure SR-IOV for ConnectX-3 with KVM (InfiniBand).” 05-Dec-2018.
- [39] RedHat, “SR-IOV Support For Virtual Networking.” 05-Jan-2018.
- [40] W. L. Guay *et al.*, “Early experiences with live migration of SR-IOV enabled InfiniBand,” *Journal of Parallel and Distributed Computing*, vol. 78, pp. 39–52, 2015.
- [41] J. Zhang, X. Lu, and D. K. Panda, “High-performance virtual machine migration framework for MPI applications on SR-IOV enabled InfiniBand clusters,” in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2017, pp. 143–152.
- [42] A. Reber, “Process migration in a parallel environment.” Universität Stuttgart, 23-Jun-2016.
- [43] Virtuozzo, “Checkpoint-Restore in Userspace (CRIU).” [Online]. Available: <https://www.criu.org>.
- [44] J. Ansel, K. Arya, and G. Cooperman, “DMTCP: Transparent checkpointing for cluster computations and the desktop,” in *2009 IEEE International Symposium on Parallel & Distributed Processing*, 2009, pp. 1–12.
- [45] D. M. Jacobsen and R. S. Canon, “Contain this, unleashing docker for hpc,” *Proceedings of the Cray User Group*, 2015.
- [46] G. M. Kurtzer, V. Sochat, and M. W. Bauer, “Singularity: Scientific containers for mobility of compute,” *PloS one*, vol. 12, no. 5, p. e0177459, 2017.
- [47] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, “Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors,” in *ACM SIGOPS Operating Systems Review*, 2007, vol. 41, pp. 275–287.

- [48] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, “An updated performance comparison of virtual machines and linux containers,” in *2015 IEEE international symposium on performance analysis of systems and software (ISPASS)*, 2015, pp. 171–172.
- [49] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. De Rose, “Performance evaluation of container-based virtualization for high performance computing environments,” in *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, 2013, pp. 233–240.
- [50] C. Ruiz, E. Jeanvoine, and L. Nussbaum, “Performance evaluation of containers for HPC,” in *European Conference on Parallel Processing*, 2015, pp. 813–824.
- [51] D. H. Bailey, “NAS parallel benchmarks,” *Encyclopedia of Parallel Computing*, pp. 1254–1259, 2011.
- [52] I. Zacharov, O. Panarin, E. Ryabinkin, K. Izotov, and A. Teslyuk, “Virtualization for Scientific Workload,” in *2018 International Scientific and Technical Conference Modern Computer Network Technologies (MoNeTeC)*, Moscow, 2018, pp. 1–5.
- [53] J. Zhang, X. Lu, and D. K. Panda, “High performance MPI library for container-based HPC cloud on InfiniBand clusters,” in *2016 45th International Conference on Parallel Processing (ICPP)*, 2016, pp. 268–277.
- [54] J. Zhang, X. Lu, and D. K. Panda, “Performance characterization of hypervisor- and container-based virtualization for HPC on SR-IOV enabled InfiniBand clusters,” in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2016, pp. 1777–1784.
- [55] J. Zhang, X. Lu, and D. K. Panda, “Is Singularity-based Container Technology Ready for Running MPI Applications on HPC Clouds?,” in *Proceedings of the 10th International Conference on Utility and Cloud Computing*, 2017, pp. 151–160.
- [56] S. Pickartz, N. Eiling, S. Lankes, L. Razik, and A. Monti, “Migrating Linux containers using CRIU,” in *International Conference on High Performance Computing*, 2016, pp. 674–684.
- [57] C. Yu and F. Huan, “Live migration of docker containers through logging and replay,” in *2015 3rd International Conference on Mechatronics and Industrial Informatics (ICMII 2015)*, 2015.
- [58] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen, “ReVirt: Enabling intrusion analysis through virtual-machine logging and replay,” *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 211–224, 2002.
- [59] R. Stoyanov and M. J. Kollingbaum, “Efficient Live Migration of Linux Containers,” in *High Performance Computing*, vol. 11203, R. Yokota, M. Weiland, J. Shalf, and S. Alam, Eds. Cham: Springer International Publishing, 2018, pp. 184–193.
- [60] A. Kleen, “memhog: Allocation of memory with policy for testing,” 2019. [Online]. Available: <http://man7.org/linux/man-pages/man8/memhog.8.html>.
- [61] J. J. Dongarra, P. Luszczyk, and A. Petitet, “The LINPACK benchmark: past, present and future,” *Concurrency and Computation: practice and experience*, vol. 15, no. 9, pp. 803–820, 2003.

- [62] W. Gropp, L. Ewing, N. Doss, and A. Skjellum, “MPICH High-Performance Portable MPI.” [Online]. Available: <https://www.mpich.org>. [Accessed: 21-Mar-2017].
- [63] E. Gabriel *et al.*, “Open MPI: Goals, concept, and design of a next generation MPI implementation,” in *European Parallel Virtual Machine/Message Passing Interface Users’ Group Meeting*, 2004, pp. 97–104.
- [64] Intel, “Intel MPI Library.” [Online]. Available: <https://software.intel.com/en-us/mpi-library>.
- [65] R. Gerhards, “The syslog protocol,” 2009.
- [66] C. Lonvick, “The BSD syslog protocol,” 2001.
- [67] The Wikimedia Foundation, “English Wikipedia Data,” Jun-2017. [Online]. Available: <https://dumps.wikimedia.org/wikidatawiki>. [Accessed: 01-Jun-2017].
- [68] J. Dongarra *et al.*, “The international exascale software project: a call to cooperative action by the global high-performance community,” *The International Journal of High Performance Computing Applications*, vol. 23, no. 4, pp. 309–322, 2009.
- [69] M. Zaharia *et al.*, “Apache spark: a unified engine for big data processing,” *Communications of the ACM*, vol. 59, no. 11, pp. 56–65, 2016.
- [70] M. Odersky *et al.*, *The Scala language specification*. Citeseer, 2004.
- [71] M. Zaharia *et al.*, “Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing,” in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, 2012, pp. 2–2.
- [72] J. Dean and S. Ghemawat, “MapReduce: simplified data processing on large clusters,” *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.
- [73] C. Manning, P. Raghavan, and H. Schütze, in *Introduction to information retrieval*, Cambridge University Press, 2009, pp. 253–286.
- [74] V. Metsis, I. Androutsopoulos, and G. Paliouras, “Spam filtering with naive bayes-which naive bayes?,” in *CEAS*, 2006, vol. 17, pp. 28–69.
- [75] N. Silver, in *The signal and the noise: why so many predictions fail—but some don’t*, New York, USA: Penguin, 2012, pp. 243–251.
- [76] The Apache Lucene Project, “Apache Lucene English Analyzer Library,” 2017. [Online]. Available: https://lucene.apache.org/core/5_3_1/analyzers-common/org/apache/lucene/analysis/en/EnglishAnalyzer.html. [Accessed: 27-Feb-2017].
- [77] G. Shmueli, N. R. Patel, and P. C. Bruce, in *Data mining for business intelligence: Concepts, techniques, and applications*, New Jersey, USA: John Wiley and Sons, 2010, pp. 182–184.
- [78] Amazon Web Services, “Amazon EC2 Previous Generation Instance Types,” 2017. [Online]. Available: <https://aws.amazon.com/ec2/previous-generation>.
- [79] RedHat, “Optimizing System Performance,” 2017. [Online]. Available: https://access.redhat.com/documentation/en-US/Red_Hat_Directory_Server/8.2/html/Performance_Tuning_Guide/system-tuning.html. [Accessed: 04-Nov-2017].
- [80] Apache Software Foundation, “Spark Configuration,” 2017. [Online]. Available: <https://spark.apache.org/docs/latest/configuration.html>. [Accessed: 03-Nov-2017].

- [81] X. Meng, “Mllib: Scalable machine learning on Spark,” in *Stanford ICME Spark Workshop*, 2014.
- [82] Amazon, “Amazon Web Services (AWS),” 2019. [Online]. Available: <https://aws.amazon.com>.
- [83] Amazon Web Services, “Amazon EC2 Instance Types,” 2019. [Online]. Available: <https://aws.amazon.com/ec2/instance-types>.
- [84] Amazon, “Processor State Control for Your EC2 Instance,” 2019. [Online]. Available: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/processor_state_control.html.
- [85] Amazon Web Services, “Amazon Elastic File System (EFS),” 2019. [Online]. Available: <https://aws.amazon.com/efs>.
- [86] S. Shepler, M. Eisler, and D. Noveck, “Network file system (NFS) version 4 minor version 1 protocol,” 2010.
- [87] T. Taylor, “Application Containers vs. System Containers: Understanding the Difference,” 04-Oct-2016. [Online]. Available: <https://www.sumologic.com/blog/application-containers-vs-system-containers-understanding-difference>. [Accessed: 01-Apr-2019].
- [88] A. Goodman, “Application vs System Containers,” 17-Oct-2017. [Online]. Available: <https://www.excella.com/insights/application-vs-system-containers>. [Accessed: 01-Apr-2019].
- [89] C. Anderson, “Docker [software engineering],” *IEEE Software*, vol. 32, no. 3, pp. 102–c3, 2015.
- [90] M. Furman, *OpenVZ essentials*. Packt Publishing Ltd, 2014.
- [91] OpenVZ, “OpenVZ Command Line Reference: prlctl.” [Online]. Available: https://docs.openvz.org/openvz_command_line_reference.webhelp/_prlctl.html. [Accessed: 01-Apr-2019].
- [92] Amazon, “AWS Command Line Interface (CLI).” [Online]. Available: <https://aws.amazon.com/cli>. [Accessed: 01-Apr-2019].
- [93] OpenVZ, “OpenVZ Command Line Reference: vzpkg.” [Online]. Available: https://docs.openvz.org/openvz_command_line_reference.webhelp/_vzpkg.html. [Accessed: 01-Apr-2019].
- [94] Amazon Web Services, “Elastic IP Addresses.” [Online]. Available: <https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/elastic-ip-addresses-eip.html>. [Accessed: 22-Apr-2019].
- [95] Virtuozzo, “Using Virtuozzo in the Amazon EC2.” [Online]. Available: https://docs.virtuozzo.com/wiki/Using_Virtuozzo_in_the_Amazon_EC2#Configure_the_external_IP_address_for_the_container. [Accessed: 10-Jan-2018].
- [96] OpenVZ, “Container Network Modes - Host-Routed Mode for Containers.” [Online]. Available: https://docs.openvz.org/openvz_users_guide.webhelp/_host_routed_mode_for_containers.html. [Accessed: 20-Jun-2019].
- [97] OpenVZ, “Container Network Modes - Bridged Mode for Containers.” [Online]. Available: https://docs.openvz.org/openvz_users_guide.webhelp/_bridged_mode_for_containers.html. [Accessed: 20-Jun-2019].

- [98] J. Latt, “Palabos, Parallel Lattice Boltzmann Solver,” *FlowKit, Lausanne, Switzerland*, 2009.
- [99] Open Porous Media (OPM) Initiative, “Flow: Fully-Implicit Black-Oil Simulator,” 2018. [Online]. Available: https://opm-project.org/?page_id=19. [Accessed: 27-Mar-2018].
- [100] Imperial College London AMCG, “Fluidity manual v4.1.12.” 22-Apr-2015.
- [101] D. Joiner, “GalaxSee-MPI: Gravitation N-Body Simulation.” [Online]. Available: <https://www.shodor.org/refdesk/Resources/Tutorials/MPIExamples/GalaxSee.php>. [Accessed: 04-Mar-2018].
- [102] Schlumberger, “ECLIPSE Industry-Reference Reservoir Simulator, * Mark of Schlumberger,” 2017. [Online]. Available: <https://www.software.slb.com/products/eclipse>.
- [103] T. Sabat, “AWS CLI Script to Assign a Secondary IP,” 09-Jun-2015. [Online]. Available: <https://codepen.io/tsabat/post/aws-cli-script-to-assign-a-secondary-ip>.
- [104] OpenVZ, “OpenVZ User’s Guide.” 20-Dec-2016.
- [105] J. Ahrens, B. Geveci, and C. Law, “Paraview: An end-user tool for large data visualization,” *The visualization handbook*, vol. 717, 2005.
- [106] K. Varis *et al.*, “WinMerge 2.14 Help,” *WinMerge website (manuall. winmerge.org)*, pp. 1–28, 2016.
- [107] Ubuntu Documentation, “Howto md5sum,” 06-Sep-2015. [Online]. Available: <https://help.ubuntu.com/community/HowToMD5SUM>. [Accessed: 21-Jun-2019].
- [108] M. Sindi, “Container Migration Demo for MPI-based HPC Workloads: Palabos Application,” *Massachusetts Institute of Technology*, 03-Apr-2019. [Online]. Available: <https://youtu.be/1v73E2Ao3Mk>.
- [109] M. Sindi, “Container Migration Demo for MPI-based HPC Workloads: The ECLIPSE Simulator by Schlumberger,” 03-Apr-2019. [Online]. Available: <https://youtu.be/5tz6JP2UgTk>.
- [110] M. Sindi, “Container Migration Demo for MPI-based HPC Workloads: GalaxSee Application,” *Massachusetts Institute of Technology*, 03-Apr-2019. [Online]. Available: <https://youtu.be/NIT7nJ-yENc>.
- [111] M. Sindi, “Container Demo for MPI-based HPC Workloads: SuspendResume Entire MPI Job - Flow Application,” *Massachusetts Institute of Technology*, 03-Apr-2019. [Online]. Available: <https://youtu.be/KNTVHQnMVHU>.
- [112] Open MPI, “Open MPI FAQ: Does Open MPI support virtual IP interfaces?,” 22-Jan-2019. [Online]. Available: <https://www.open-mpi.org/faq/?category=tcp#ip-virtual-ip-interfaces>. [Accessed: 03-Apr-2019].
- [113] Open MPI Team, “Open MPI Issue #160: TCP BTL does not support Linux virtual interfaces,” 01-Oct-2014. [Online]. Available: <https://github.com/open-mpi/ompi/issues/160>. [Accessed: 03-Apr-2019].
- [114] J. Squyres, “Open MPI Hack to Work with OpenVZ Containers,” 24-Jun-2016. [Online]. Available: <https://www.mail-archive.com/users@lists.open-mpi.org/msg29585.html>. [Accessed: 04-Dec-2017].
- [115] Open MPI, “FAQ: Linux Kernel TCP Parameters,” 22-Jan-2019. [Online]. Available: <https://www.open-mpi.org/faq/?category=tcp>.

- [116] R. Castain, “Open MPI ORTE Connection Timeout,” 16-May-2016. [Online]. Available: <https://www.mail-archive.com/users@lists.open-mpi.org/msg29262.html>.
- [117] CRIU, “What cannot be checkpointed.” 13-Jul-2017.
- [118] Schlumberger, “ECLIPSE Industry-Reference Reservoir Simulator (Version 2017.2) - Release Notes.” 2017.
- [119] Amazon, “Network Maximum Transmission Unit (MTU) for Your EC2 Instance,” 2019. [Online]. Available: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/network_mtu.html.
- [120] G. Staples, “Torque resource manager,” in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, 2006, p. 8.
- [121] R. L. Henderson, “Job scheduling under the portable batch system,” in *Workshop on Job Scheduling Strategies for Parallel Processing*, 1995, pp. 279–294.
- [122] A. B. Yoo, M. A. Jette, and M. Grondona, “Slurm: Simple linux utility for resource management,” in *Workshop on Job Scheduling Strategies for Parallel Processing*, 2003, pp. 44–60.
- [123] Univa, “Univa Grid Engine (UGE),” 2019. [Online]. Available: <http://www.univa.com/products/>. [Accessed: 22-Jun-2019].
- [124] NASA, “Portable Batch System (PBS),” 05-Dec-2018. [Online]. Available: [https://www.nas.nasa.gov/hecc/support/kb/portable-batch-system-\(pbs\)-overview_126.html](https://www.nas.nasa.gov/hecc/support/kb/portable-batch-system-(pbs)-overview_126.html).
- [125] Adaptive Computing, “Moab Workload Manager,” 2012. [Online]. Available: <http://docs.adaptivecomputing.com/mwm/7-1-3/help.htm#overview.htm>.
- [126] Experts-Exchange.com, “Hardware Issue Example,” 17-Jun-2012. [Online]. Available: <https://www.experts-exchange.com/questions/27759824/failed-to-prefill-DIMM-database-from-DMI-data.html>. [Accessed: 22-Jun-2019].
- [127] C. Kuenzler, “Hardware Issue Example,” 24-May-2018. [Online]. Available: <https://www.claudiokuenzler.com/blog/777/hardware-error-parity-error-during-data-load-clean-the-cpu-heatsink>. [Accessed: 22-Jun-2019].
- [128] LinuxQuestions.org, “Hardware Issue Example,” 17-Jun-2014. [Online]. Available: <https://www.linuxquestions.org/questions/slackware-14/hardware-error-in-syslog-4175508316/>. [Accessed: 22-Jun-2019].
- [129] LinuxQuestions.org, “Hardware Issue Example,” 03-Mar-2013. [Online]. Available: <https://www.linuxquestions.org/questions/slackware-14/kernel-message-bad-memory-bad-cpu-4175452580/>. [Accessed: 22-Jun-2019].
- [130] Oracle, “Hardware Issue Example,” 2010. [Online]. Available: <https://docs.oracle.com/cd/E19432-01/820-1120-22/chapter7.html>. [Accessed: 22-Jun-2019].
- [131] Red Hat, “Hardware Issue Example,” 25-Jun-2015. [Online]. Available: <https://access.redhat.com/solutions/70160>. [Accessed: 22-Jun-2019].
- [132] Red Hat, “Hardware Issue Example,” 05-Dec-2018. [Online]. Available: <https://access.redhat.com/solutions/42261>. [Accessed: 22-Jun-2019].
- [133] J. Reyes, “Hardware Issue Example,” 01-May-2016. [Online]. Available: <http://johnrey.es/index.php/2016/05/01/linux-kernelhardware-error-mc4-error-node-1-dram-ecc-error-detected-on-the-nb/>. [Accessed: 22-Jun-2019].

- [134] Superuser.com, “Hardware Issue Example,” 25-Mar-2014. [Online]. Available: <https://superuser.com/questions/733417/syslogd-hardware-error>. [Accessed: 22-Jun-2019].
- [135] M. S. Keller, “Take command: cron: Job scheduler,” *Linux Journal*, vol. 1999, no. 65es, p. 15, 1999.
- [136] M. Sindi, “Container Migration Full Cycle Demo for MPI-based HPC Workloads: Palabos Application,” *Massachusetts Institute of Technology*, 22-Jun-2019. [Online]. Available: <https://youtu.be/KW60Ju6wWac>.
- [137] M. Sindi, “Container Migration Full Cycle Demo for MPI-based HPC Workloads: Fluidity Application,” *Massachusetts Institute of Technology*, 22-Jun-2019. [Online]. Available: <https://youtu.be/DCtFhxve5Hc>.
- [138] J. Chu and V. Kashyap, “Transmission of ip over infiniband (ipoib),” 2006.
- [139] MVAPICH Team, “MVAPICH2 2.3. 1 User Guide,” 01-Mar-2019. [Online]. Available: <http://mvapich.cse.ohio-state.edu/static/media/mvapich/mvapich2-2.3.1-userguide.pdf>. [Accessed: 22-Jun-2019].
- [140] G. Chrysos, “Intel® xeon phi coprocessor (codename knights corner),” in *2012 IEEE Hot Chips 24 Symposium (HCS)*, 2012, pp. 1–31.