Reducing resource waste in HPC through co-allocation, custom checkpoints, and lower false failure prediction rates

Dissertation submitted for the award of the title "Doctor of Natural Sciences" of the
Faculty of Physics, Mathematics, and Computer Science of the

Johannes Gutenberg Universität Mainz

Alvaro Frank
Born in San Jose, Costa Rica on the 08.10.1986

Mainz, 12 September 2022

1. Reviewer [redacted]

Fachbereich 08: Physik, Mathematik und Informatik
Institut für Informatik
Johannes Gutenberg Universität Mainz

2. Reviewer [redacted]

Fachbereich 08: Physik, Mathematik und Informatik
Institut für Informatik
Johannes Gutenberg Universität Mainz

Date of exam: 12 September 2022

Abstract

Bigger systems are being deployed by High Performance Computing (HPC) centers in order to fulfill the needs of modern scientific and big data applications and to match the increased amount of users in said systems. As modern scientific problems become more complex, HPC applications tend to have longer runtimes and higher resource requirements, which further drives the requirement for bigger systems. This thesis explores three methods to reduce wasted computational resources on modern HPC systems with thousands of components.

Unfortunately, not all HPC applications scale with the increased node counts or utilize all types of resources available in big HPC systems. This deficiency can leave partial resources underutilized through sub-optimal workloads of highly specialized scientific applications. Mixed workloads can help alleviate this drawback by potentially stressing different resources. Co-allocation is a method that can help increase resource utilization and job throughput by enabling mixed workloads to share compute nodes.

Increasing the amount of components in HPC systems also contributes to a higher risk of hardware failures, which in turn can crash user applications. HPC systems must account for this susceptibility to crashes by implementing resilience measures that reduce computation loss when user jobs fail. The current state-of-the-practice is to perform checkpoints at set intervals, which causes significant overhead every time computations are paused to save the state of applications. This cost of checkpointing raises further when the checkpoint intervals are chosen sub-optimally. Consequently, using optimal checkpoint intervals and predicting failures to only perform necessary checkpoints are two methods that can reduce overhead and increase throughput in HPC systems.

We therefore propose in this thesis three waste reduction methods related to sub-optimal resource utilization in scientific workloads, overhead from sub-optimal checkpoints and unnecessary checkpoints in failure prediction. The approaches explored here increase job throughput in HPC systems using co-allocation, reduce unnecessary checkpoints that are triggered after failure predictions and improve checkpoint intervals for common jobs with medium probability of failure. To accomplish these goals we first present a new node sharing strategy for batch systems and show how it can increase scheduling throughput when compared to standard node allocation methods. Secondly we propose a new optimal checkpoint interval for jobs with short to medium runtimes that can reduce the expected overhead from checkpointing. Finally we introduce a node failure prediction method tailored to big HPC systems that reduces false positive rates.

This thesis offers new insights into the inefficiencies that follow from job failures and resource under-utilization in big HPC systems, while also proposing three techniques that help alleviate said deficiencies.

Acknowledgement

I would first like to thank my supervisor [redacted] for the research opportunities he provided, as well as the necessary counsel that allowed me to accomplish this work. I also would like to thank [redacted], [redacted] and [redacted] for their input and guidance throughout. Finally I would like to express my very profound gratitude to my dad, my sister, her husband, and to my wife [redacted] for providing me with their unrelenting support and continuous encouragement throughout the years. This work would not have been possible without them. Thank you.

Contributors and Funding Sources

This section provides a description of my own contributions to the contents of this thesis and should be complete to my knowledge. I have personally performed the majority of the technical or scientific work and all of the writing in this thesis, but I cannot claim sole authorship for every idea present in the development of its contents. Most work shown here was achieved after various iterations and from normal doctoral advisory discussions with my supervisors. This thesis encompasses a variety of concepts in the topics of application co-allocation, checkpoint determination and failure prediction using machine learning methods. The experiments, simulations and analysis for failure prediction, checkpoint optimization and co-allocation were performed exclusively by me. I am also the primary author of all the publications, which make the basis for this thesis. The text of these publications has been re-written from the source of published papers [1, 2, 3] to fit the thesis, explain ideas in more detail or add more content. The idea of exploring node sharing to determine its possible benefits stems from my supervisors as part of the FAST project proposals. Similarly the idea of exploring the benefits of predicting failures to migrate jobs originates from the ENVELOPE project proposal that had as a goal increase resilience in HPC. The idea of improving checkpoint intervals for smaller jobs was mine and was also developed under the same ENVELOPE project. I am not the author of either of these project proposals, but the derived implementations to solve the proposed problems presented here were performed by me. I am not the author of applications used to test the co-allocation methods. The implementation of co-allocation algorithms within SLURM was performed by me. My work in failure prediction was performed in cooperation with partners at the Technical University of Munich who were responsible for half of the Table 4.1. My contributions to this overview consisted of four related works and calculating unnecessary checkpoint probabilities from my own classification metric also described in this work. The neural network chain method of failure prediction discussed in Section 4.2.3 is my own idea and implementation. The recurrent neural network voting system of Figure 4.5 was suggested by [redacted] as an alternative to the unanimous voting procedure. The implementation and all experiments were also part of my own work. The mathematical derivations and formulas described in Section 3.3 were done by me with the exception of optimized sub-equation 3.6d, which was suggested by [redacted] for simplicity but remains mathematically equivalent to my own work. All the figures and diagrams used are my own work and based on original designs or existing concepts but redone by me.

The work presented in this thesis was possible by the funding provided by the Federal Ministry of Education and Research of Germany under the FAST and ENVELOPE projects. Funding was also provided by the Center for Data Processing of the Johannes Gutenberg Universität Mainzunder the direction of [redacted].

Contents

1 Introduction 1

1.1 Motivation 1

1.2 High Performance Computing systems 3

1.3 Co-allocation of parallel applications 9

1.4 Failure probability aware checkpoint intervals 11

1.5 Failure prediction for parallel applications 13

2 Co-allocation of parallel applications 23

2.1 Related Work 24

2.1.1 Backfilling and *First-fit* scheduling 27

2.1.2 Co-allocation techniques 30

2.2 Methodology 34

2.2.1 Hardware and Software Setup 34

2.2.2 The scientific applications 35

2.2.3 Baseline allocation and alternative co-allocation setups 36

2.2.4 Batch system setup 38

2.2.5 Accounting 39

2.3 Evaluation and results 40

2.3.1 Isolated allocation experiments 41

2.3.2 Batch scheduling experiments 45

2.3.3 Profiling 47

2.3.4 Discussion of the results 51

3 Failure probability aware checkpoint intervals 53

3.1 Related work 56

3.2 Problem Formulation 59

3.3 Checkpoint Algorithm 61

3.3.1 Cost function 62

3.3.2 Iterative Algorithm 65

3.3.3 Poisson and Weibull examples 65

3.3.4 Usability 66

3.4 Simulation and Results 67
  3.4.1 Interval and cost function simulation 68
  3.4.2 Checkpoint cost overhead 71
  3.4.3 HPC system traces 72
  3.4.4 Discussion of the results 76

4 Failure prediction for parallel applications 79
4.1 Related Work 80
  4.1.1 Supervised binary classifier for failure prediction 83
  4.1.2 Classical Metrics on Classifier Performance 84
  4.1.3 Modeling Hardware Failures in HPC Systems 88
  4.1.4 Unnecessary Checkpoints 90
4.2 Data processing and prediction system 91
  4.2.1 Available data and statistical anomalies of hardware problems 91
  4.2.2 Data processing 95
  4.2.3 Failure Prediction with Neural Networks Ensemble 97
4.3 Evaluation and results 101
  4.3.1 Failure prediction metrics 102
  4.3.2 Discussion of the results 109

5 Conclusion and future work 111

Bibliography 121

Declaration 135

1

Introduction

1.1 Motivation

Scientists use high performance computing (HPC) systems to solve complex numerical problems that simulate characteristics of classical experiments. Similarly the fields of big data [4, 5] and machine learning [6, 7] make use of HPC systems to process large amounts of data using complex computations. These HPC computers are expensive to deploy and operate, which makes the utilization of their resources (e.g. use of cores for hours) also expensive [8]. For this reason HPC centers strive to increase efficiency by optimizing resource utilization and reducing waste of computational resources.

HPC applications also continuously improve to scale better in distributed memory systems and to use more nodes [9]. The large counts of processing units in individual HPC machines also incentivizes software developers to further increase the parallelism of their applications in shared memory systems. Despite these trends not all scientific applications efficiently scale or use all the types of resources present in an HPC node [10, 11, 12]. One reason for under-utilization is that shared resources, like memory channels or IO bandwidths, can be easily saturated by processes running on few CPU cores even at high core counts. Further increasing the number of cores per node can lead to diminishing returns when running applications competing for the same resources. This can leave other components or resources unused.

Within HPC systems, the scientific applications traditionally do not share hardware with each other. The HPC batch scheduling systems with priority queues limit one job per compute node and do not help with saturation. The exclusive allocation of one job can leave resources within the nodes unused based on the workload characteristics (*e.g.* accelerators idle on I/O heavy workloads). It is difficult to solve this problem if only one application can be used to saturate high core counts. The under-utilization of resources can also occur when the scheduler keeps too many nodes reserved (and therefore unused) for pending jobs. This can cause reserved nodes to remain unused for long periods of time while high priority jobs wait for more resources. The issue is also aggravated by schedulers not being allowed to choose long running, low priority jobs to complement resource utilization of unused components within already allocated nodes.

Another sub-optimal use of resources comes from inefficiencies in failure mitigation strategies. With the increase of components in HPC systems comes also a higher risk of failures. The resources are wasted every time computations get discarded after an application aborts due to a fatal error like hardware failures. This waste increases for HPC systems with high component counts or with systems that are very old and that have increased failure probabilities [13, 14, 15]. Saving the state of applications using checkpoints can diminish the wasted computation time after failures, but requires optimal checkpoint intervals or failure prediction methods to mitigate the inherent overheads of checkpoints.

This thesis provides three methods that reduce resource waste in HPC caused by, the under-utilization of resources in exclusive allocations of homogeneous workloads, the checkpoint overhead of sub-optimal checkpoints, and the cost of unnecessarily checkpointing applications with failure predictors. The three approaches used to mitigate these issues can be summarized as follows:

- Co-allocate pairs of good and distinct parallel jobs to the same set of nodes to increase effective hardware resource utilization.
- Predict HPC node failures with low false positive rates to reduce waste from unnecessary checkpoints.
- Use checkpoint intervals tailored for HPC jobs with lower probability of failing to reduce mitigation overhead.

These three techniques can reduce the significant overhead present in current systems that use existing alternative methods. For instance, we propose that specific co-allocated pairs of distinct applications can experience slowdowns less than twice their normal runtimes when sharing cores through simultaneous multithreading (SMT). The cost of increased individual application runtime is offset with a net reduction in core-hours needed to complete equivalent workloads. This is done by saturating resources that would otherwise stay unused by a single application. This method can use resources that would otherwise be unused in exclusive allocation schemes.

Furthermore, the prediction of eventual application failures due to faulty hardware can enable last moment checkpointing of applications that would otherwise have their partial computations fail and be wasted. Reduction of false alarms in failure prediction also reduces wasted overhead of unnecessary checkpoints.

Finally applications with small failure probabilities are very common in HPC systems. These applications waste resources doing checkpoints at intervals meant for long running and large applications. Optimization of intervals for these common jobs reduces the core-hours used to perform them. The following sections introduce each resource saving approach and the details of each method are explored in depth in their respective chapters.

1.2 High Performance Computing systems

HPC Clusters

High Performance Computing (HPC) is done in practice by combining the compute power of distributed computers to attain more performance than any individual workstation can. This is done to solve large computational problems in science or engineering that would otherwise take too long to solve or not fit within the memory of an individual computer. Common simulations performed with HPC are from topics like climate research, biology, energy, military, mechanical simulations, manufacturing, etc.

Figure 1.1 shows an architecture diagram of a mock HPC system using a cluster architecture with general purpose computers. This is the most popular form of architecture, with 93% of the TOP-500 [16] fastest HPC systems using a cluster setup. This mock HPC machine consists of file system nodes, compute nodes, administration nodes, and login nodes. Applications run on compute nodes reading and writing data on the global file system nodes and communicate through high speed compute networks like InfiniBand or OmniPath. The HPC system also has administration nodes running supporting services like the batch system that receive user job submissions. The Ethernet administrative network allows for traditional SSH management of nodes and the IPMI network enables the administrators to access the back-end BIOS settings and do tasks like cycling machines or installing operating systems. It also allows for an interference free collection of monitoring data that does not affect the computing interconnect.

The computations done in HPC systems are usually parallel and scale to large number of nodes, or data intensive requiring long running times to process all the data, or have a large number of independent tasks. An ideal application would have good shared memory and distributed memory scaling. Ideal strong scaling [17] would see the runtimes reduced by half when doubling the amount of compute units for

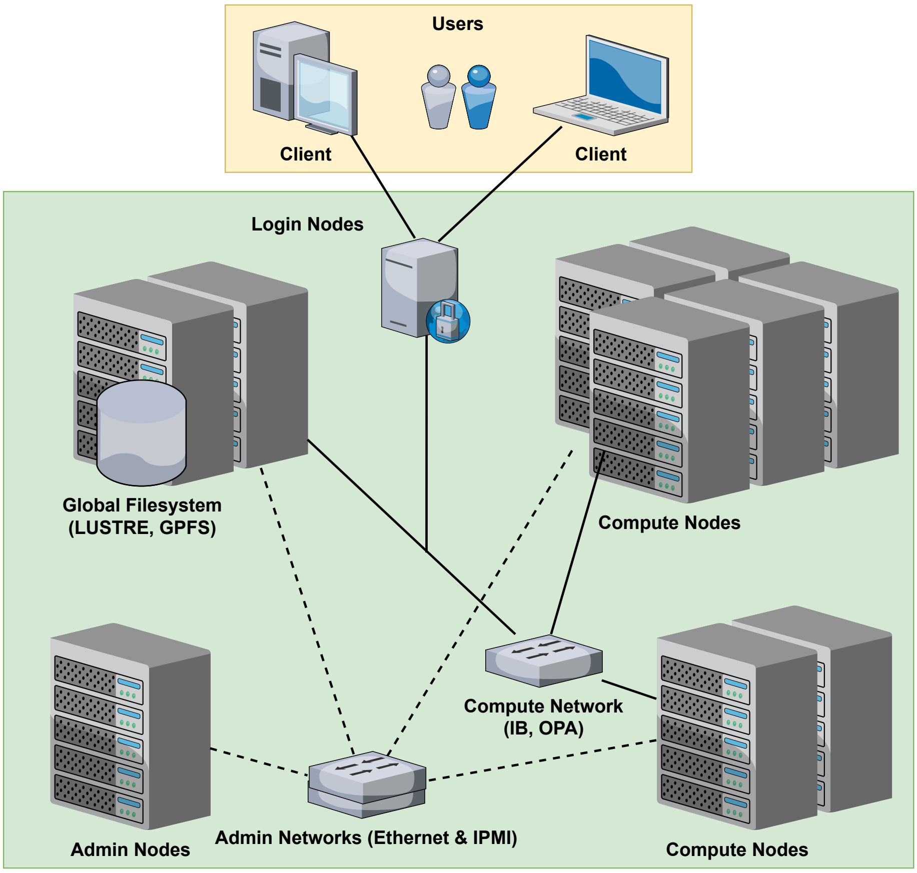

Fig. 1.1: HPC system architecture diagram with users accessing the HPC machine. The machine has multiple nodes that are connected with several networks.

a fixed problem size. This is in practice not possible and is bounded by Amdahl's law [18]. Ideal weak scaling would maintain the original runtime as the problem size and compute unit count are increased as described by Gustafson's law [19]. This need to solve larger problems is a driving force that pushes systems to become bigger and more complex. HPC systems provide large number of processors (CPU), memory and a high speed interconnect connecting the nodes. Each CPU has its own cache, each node has its own CPUs and memory and each node exchanges data with other nodes through a high–speed network.

Users need to use distributed memory programming models to run on HPC hardware. They can also do a hybrid approach by using shared memory models within the compute nodes. This is implemented as message–passing programs that divide data among nodes and where each node has all the data it needs to solve an independent sub-problem. The sub-problem can be parallelized using shared memory techniques. Computations then do data exchange cycles until the entire program completes.

These programs are created or configured by the users and then queued into a batch system like SLURM. The SLURM software is a cluster management and job scheduling system that allocates compute resources to applications, starts them and keeps accounting of used compute time [20]. While queued, the HPC jobs wait for resources where already running applications compute until they finish. If resources are available, SLURM starts the application with highest priority first. Applications usually need to wait for resources due to the high demand for computing time on the HPC systems, making the size of the cluster important.

The size of HPC systems has been recorded to steadily increase since the Top-500 started recording the computing power of HPC systems [16] in 1993. Figure 1.2 plots the performance of the most powerful HPC systems over the years between 1993 and 2020 using the LINPACK benchmark. The performance in GFlop/s (Giga floating point operations per second) has been steadily increasing for the top, bottom and sum of all systems. After 2003 the bottom 500 has been outperforming the top 1 from 1993, needing only 10 years to achieve such an increase. The performance improvement has shifted over the decades, from single threaded improvements initially, to high core and node parallelism, and eventually into accelerators. The raise in performance is expected to continue in the near future, leading to more nodes and more components within the HPC systems. Other rankings like the Green-500 and Graph-500 exist. The first focuses on energy efficiency in terms of GFLOPs per Watt of energy used and the former focuses on data intensive benchmarks and the performance of HPC systems for such problems.

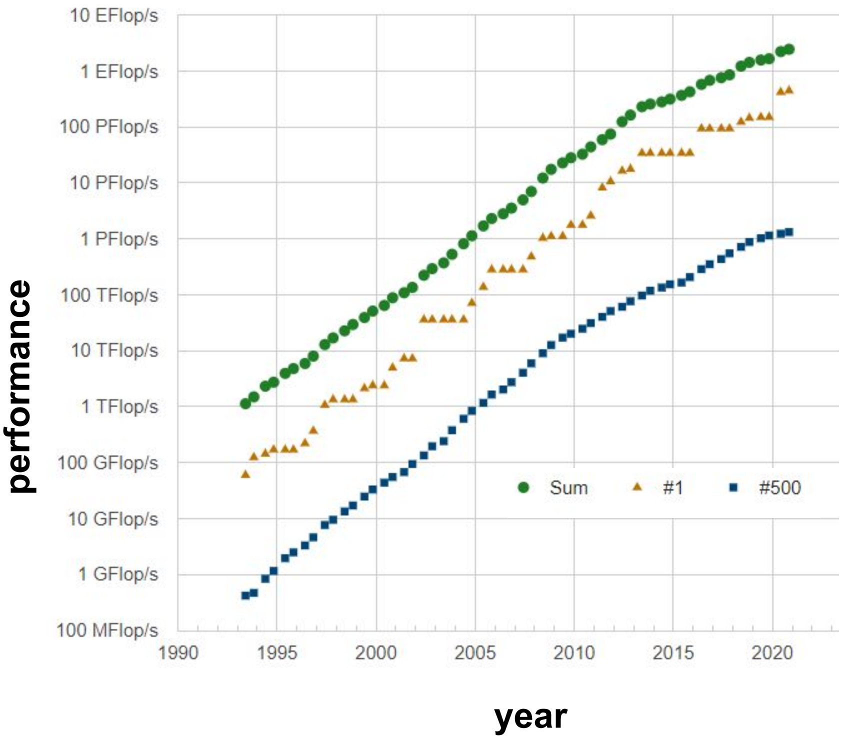

Fig. 1.2: Performance of the Top 500 HPC systems over the years (source top500.org [16]). The performance of the HPC systems at position 1 and position 500 has steadily increased with time.

Computing hardware

HPC applications can use distributed memory techniques like MPI on an HPC system and shared memory methods like OpenMP, pthreads and CUDA within a compute node itself. Figure 1.3 showcases a mock compute node with **von Neumann architecture** from an HPC system [21, 17]. The compute node has four cores with two sockets, uses non-uniform memory access (**NUMA**), two accelerators (one GPU shown) and a series of interconnects that allow the node to communicate to other compute and management nodes. The communication between internal components is mainly done through buses and to outside nodes through network infrastructure like OmniPath. The individual CPUs are placed within a sockets that communicate with each other using some sort of inter-socket-bus. The CPU depicted has its own interconnect to allow cores to interact. Each core has its own local cache, while the CPU itself offers a shared cache above the local cache. These are used in a cache hierarchy that might be several levels. If data that is needed is not found within the cache, the next level is asked, all the way to main memory. Not finding the data within a cache level is called a **cache miss** and finding it is referred to as a **cache hit** [21]. Similarly when data is changed within the CPU and it has to be updated, the

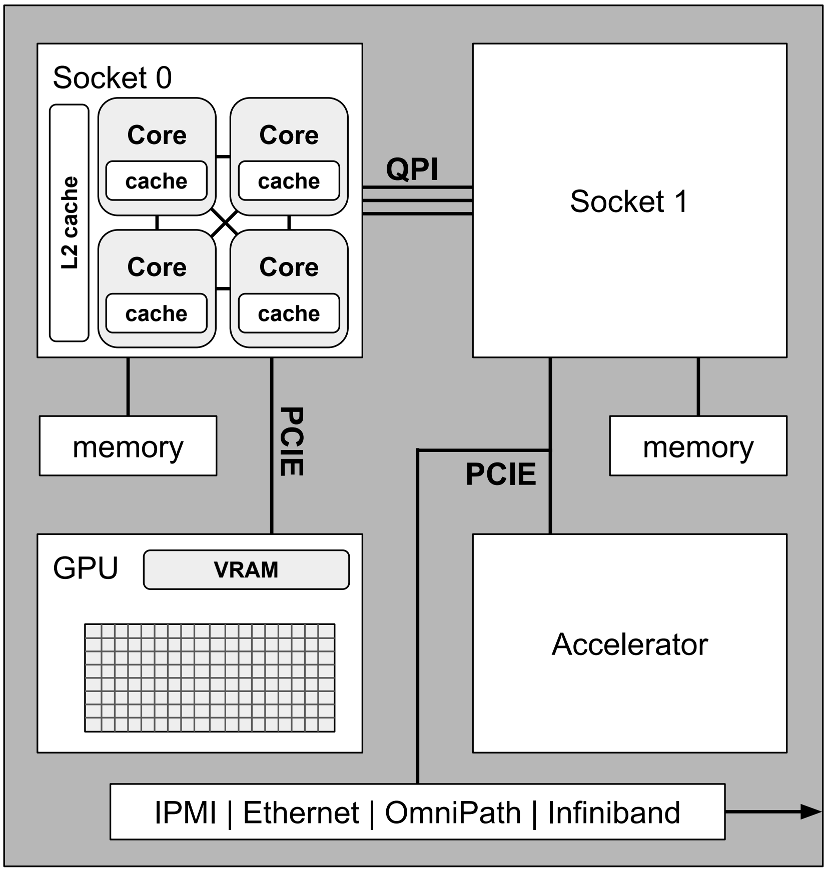

Fig. 1.3: Mock architecture of a two socket compute node with NUMA memory architecture, accelerators and network interconnects through PCIE, two level core cache and quad core CPUs.

system can choose to do **write through** of the data updating it all way from the first cache to the main memory, or it can do **write back** where updating of other caches is delayed and only updated using techniques like **Least Recently Used** or **FIFO** among others [21]. Other cores within the CPU can use **bus snooping** to monitor cache changes and invalidate its own copies. Each core in this example can be used as a "single instruction, multiple data" (**SIMD**) machine that processes arrays of data using single instructions from its individual cache. The accelerator in the Figure could also be a Vector Processor like a XeonPhi that also does **SIMD** operations. The whole CPU itself can be considered a "multiple instructions, multiple data" (**MIMD**) machine that has access to all shared memory to process different operations on each core as needed. Overall, this mock compute node represents a general HPC system that can have thousands of nodes forming a homogeneous cluster. It could also have varying hardware types and be a heterogeneous cluster.

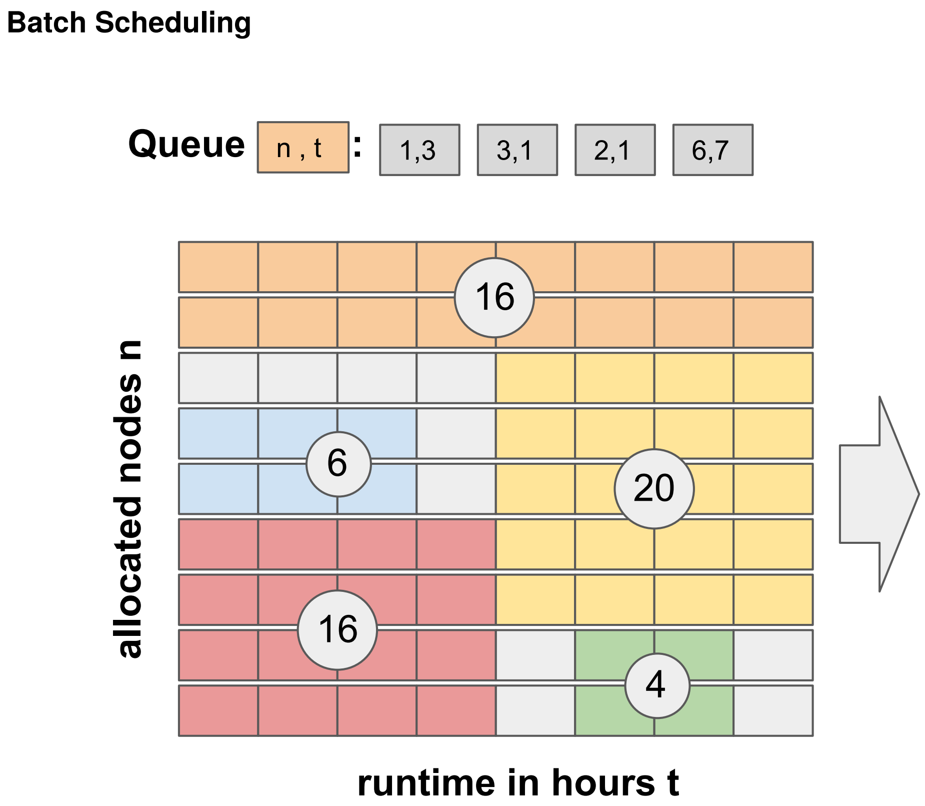

Fig. 1.4: Resource utilization in a mock HPC system with one hour node allocation for applications of different node counts and runtimes. The incoming queue is shown at the top with the highest priority job on the right [6,7] and least on the left [1,3]. The first number is the amount of nodes requested and the second number the runtime requested.

Resource allocation within HPC systems is traditionally performed by assigning compute nodes to a single application exclusively. User applications arrive into a queue of jobs, where it waits for the resources requested. The requests usually need an amount of compute nodes and hardware configurations that are not immediately available.

Figure 1.4 depicts a mock diagram of the node allocation for one hour chunks within an HPC system with 9 nodes. A queue of incoming jobs is shown at the top with the least priority job on the left [1,3] and the highest on the right [6,7]. The x-axis shows the runtime of applications of different colors in hours, while the y-axis shows the 9 nodes used by the applications. At the top we show an application in orange that used 2 nodes for 8 hours from start to finish, for a total of 16 node-hours. At the bottom a red application is shown that also used 16 node-hours, but with 4 nodes for 4 hours. A third application on the right in yellow used 20 core-hours, but only after enough resources were available. The blue application on the left used 6 node-hours, but the cluster was left with 3 nodes unused during its execution.

Only after red and blue end, could the yellow application start, leaving a total of 6 node-hours unused (shown in unnumbered light gray). These gaps are a result of scheduling and offer a potential optimization by using the nodes.

In practice, batch systems do not do a theoretically optimal scheduling decision [20]. Instead, they assign resources to waiting jobs based on their priorities that originate from their arrival time, waiting time and job importance. The classical batch system algorithm is backfill and does scheduling of higher priority jobs first and tries to fill unused gaps with lower priority jobs as long as those do not delay the start of the former. In the previous example of Figure 1.4 the low priority jobs [1,3] and [3,1] would be backfilled between jobs (6) and (16).

The example shows how scheduling is done in practice and how resource utilization is accounted as nodes-hours used by an application [22]. Accounting of resource utilization using core-hours is also a used metric in HPC.

1.3 Co-allocation of parallel applications

Co-allocation in HPC is the process of assigning two or more computing tasks to the same resources to be used simultaneously. Co-allocation allows processes from different HPC jobs to be run on the same nodes and therefore *to share nodes*. Sharing nodes can be done by exclusively assigning sockets or individual cores to different applications running on the same node or by sharing cores in the same sockets with Simultaneous Multi-Threading (SMT).

The underlying motivation for node sharing is that different applications might not stress the same resources simultaneously, leading to increased resource utilization. Previous works have shown that node sharing can be successful in isolated environments [23], but at the cost of application performance [24][25][26]. These studies were unfortunately limited to single node environments. Parallel applications that use distributed memory and message passing will have different utilization patterns and behave differently than single node applications under co-allocation.

HPC centers typically only support node sharing for single node jobs and avoid the co-allocation of parallel jobs. System administrators fear potential security risks, the complexity of a batch system supporting node sharing, and the eventual interference between co-allocated jobs that will lead to worse performance of individual applications. These concerns are justified, but require study to understand the potential benefits and drawbacks of co-allocation. This thesis studies the performance drawbacks for individual MPI applications and the possible resource savings for HPC systems when application pairs complete equivalent workloads using less node-hours.

This thesis explores how co-allocation can increase scheduling efficiency in Chapter 2. The discussion of security aspects is left out, but could be addressed by isolating applications in containers or virtual machines [27][28]. This thesis proposes instead that co-allocation of large parallel jobs can be integrated into batch environments to increase overall system efficiency.

First, we explore how different allocation approaches affect the performance of parallel MPI applications and the resources required to compute equal amounts of work. Second, we show that co-allocation can yield efficiency benefits for workloads that share nodes and over-subscribe physical cores through SMT. The findings are then used in a scheduling heuristic implemented in the SLURM batch system [20] that improves HPC system efficiency.

The experimental analysis uses five applications from the NERSC Trinity mini MPI parallel application suite [29]. These applications perform distributed memory and parallel workloads that reflect similar characteristics present in HPC batch systems. The main tasks performed to evaluate the benefits of co-allocation are:

- A detailed description on how to perform co-allocation experiments with a set of parallel scientific applications.
- The introduction of extensions for backfilling and first fit scheduling strategies to enable node sharing using SMT, application rank matching and application pair preference.
- An evaluation of various node sharing approaches to improve the efficiency of HPC systems. The computational efficiency will improve by up to 19% and the scheduling efficiency up to 25.2% compared to SLURM's backfill implementation.

Figure 1.5 gives an initial example of how to co-allocate four applications that would normally use four individual exclusive nodes. The applications are shown in color, the nodes in gray squares and the CPUs in white squares within the nodes. On the left, all four applications use their nodes exclusively, with only the top right application using a single CPU. Co-allocation is shown on the right, where the left pair of applications share one CPU using SMT and the right pair share no CPUs. The nodes that remain free can be used for new applications, and as long as the original

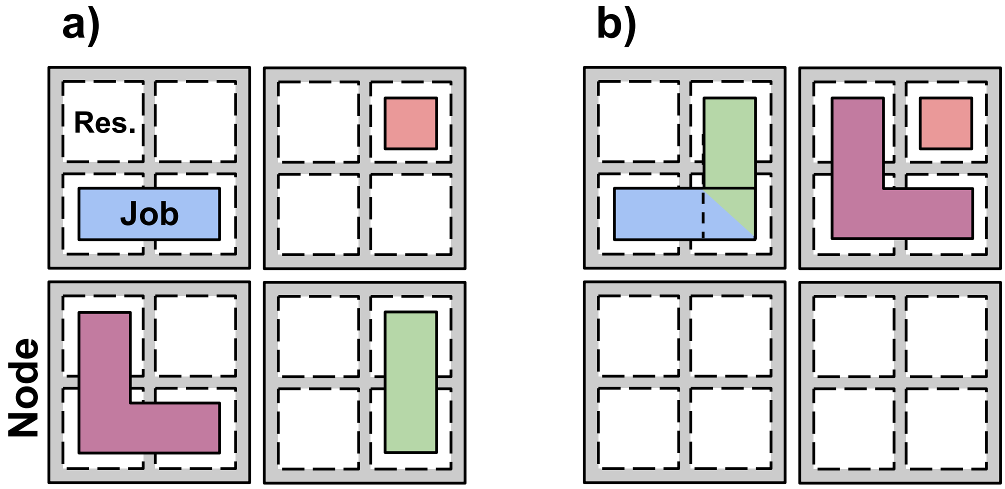

Fig. 1.5: Example of four applications using four nodes that consist of four CPUs. System a) shows the applications using four nodes without sharing and system b) sets two applications in two shared nodes.

four application do not slow down more than twice their original expected runtimes, gains can be had. These gains are indeed possible by fully utilizing internal CPU pipelines using SMT, and by using otherwise unused resources during other phases like network communication.

The co-allocation chapter will show that selecting certain pairs of application to co-allocate will have better savings than using indiscriminate pairs. On average, the computational efficiency of workloads using co-allocated backfill improved by 13% and the scheduling efficiency by 22.5%. Overall we provide strong evidence to suggest that co-allocation can benefit HPC systems by providing resource utilization savings for certain workloads.

1.4 Failure probability aware checkpoint intervals

With the increase of nodes being deployed in HPC systems and with as many more nodes being used by applications, the probability of at least one component failing during the application's execution time also increases. This creates a risk of losing computation time to failures and being forced to restart computations. This risk can be reduced by using mitigation techniques like checkpointing at set intervals.

Checkpointing techniques store the state of computations from running applications, enabling them to resume work in case a failure occurs and the program stops. These checkpoints are normally performed at intervals, in order to cover as much of the runtime as possible. While more frequent checkpoints reduce the amount of lost work for each failure, they also limit the efficiency of applications, especially for large jobs [30]. This is because creating a checkpoint requires pausing the application, using time and resources to create it. It is therefore important to produce checkpoints at intervals that minimize the cost of checkpoints and also account for the dependence between the interval length and the potential lost computations.

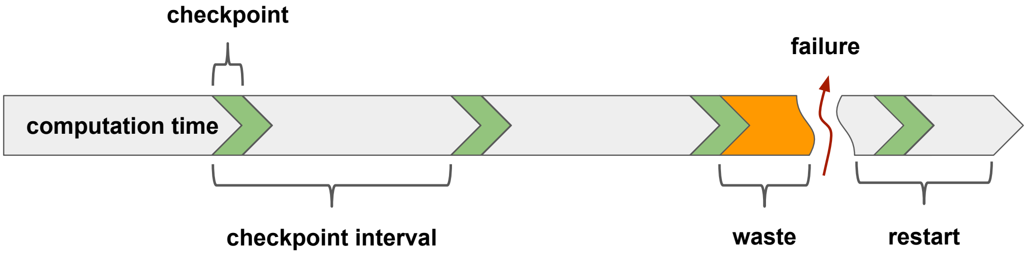

Fig. 1.6: Periodic checkpointing example where a failure occurs and computations not saved are lost.

Figure 1.6 depicts an example of periodic checkpointing for an application that fails upon encountering a failure. The computation time is divided in intervals by the time it takes to produce a checkpoint and keeps running until it finishes or it fails. After a failure the program restarts from the previous checkpoint, having lost any computations between the last checkpoint and the failure. All the computations performed after the last checkpoint are lost and considered wasted. After restarting from the last checkpoint the application continues execution.

The first relevant theoretical method for finding optimal checkpointing intervals was proposed by Young [31] and was later improved by Daly [32]. Their methods were derived for Poisson failure distributions and were intended to be used for single running processes and with the assumption that failures nearly always occur. The Young interval accounted for all the failures that would occur on long running processes. Unfortunately, this is not a good model for HPC parallel jobs that might have low probability of failing. Although not intended for HPC jobs, the Young intervals would only apply to very long running and big jobs with extremely high probability of failing.

In this thesis, the assumption is challenged and we investigate how previous checkpointing works waste resources by providing inefficient checkpoint intervals for common HPC jobs that have medium failure probabilities. In Chapter 3 we show with data mining that many checkpointable HPC jobs only have a medium probability of failure (MPF) and that they can require different checkpoint intervals. MPF jobs do not fulfill the modeling assumptions that failures almost always happen, as either their runtimes are not long enough, or the time between node failures is too long for the node count used. Not accounting for these aspects in previous studies resulted in over- or underestimations of the optimal checkpointing intervals for MPF jobs, which in turn leads to inefficiencies in resource utilization.

Chapter 3 will propose a solution with a new cost function for checkpointing MPF jobs and an iterative algorithm that finds the statistically best checkpoint interval. The proposed algorithm will also be close to the optimal interval under the previous methods of Young and Daly that use their assumptions. The checkpoint algorithm accounts for the number of nodes used, the runtime of the application, and the failure properties of the underlying HPC system. The algorithm has the added property of converging to the optimal results of Daly for jobs with high failure probabilities.

The experimental results for existing real HPC traces will show that the proposed method can achieve cumulative checkpoint cost savings (considering all jobs and requeuing) from 7.1% to 25.7% for failures from a Weibull distribution, and from 7.3% up to 27.5% for failures from a Poisson distribution. These savings are in relation to the best 5 alternative methods. Finally we will also show that the method achieves savings between 6.0% and 26.6% compared to alternative methods when only inaccurate values for the mean time between failures (MTBF) are available.

1.5 Failure prediction for parallel applications

Typical HPC applications have tightly interconnected parallel processes that can cause the failure of the entire program when a single process fails. These failures can occur when an individual hardware part fails and the probability of these failures increases with the amount of components present in the HPC system. This susceptibility to failures combined with the cost of checkpoints makes failure prediction an important mitigation approach. By using system health data, a binary classifier can be trained to classify the time series data into either possible incoming failures (positive classification) or non-failure (negative classification) events. These classifiers work as a failure predictor that can use the classification ahead of time (lead-up time) to produce a checkpoint before the failure occurs. The failure predictor system **(FPS)** can therefore predict failures and then trigger a checkpoint on demand (if it predicts a failure will occur), reducing the overhead of constant checkpoints with intervals. We will propose an FPS that uses multiple neural networks to classify data into events through two voting procedures. The first voting procedure will require all networks to vote for a failure unanimously, and the second will use a final network to learn the voting procedure. The failure warnings predicted by an FPS are the positive classifications from the binary classifier, where **true positives** are correct failure predictions and **false positives** predictions where no failure occurs afterwards. We will go into more detail about these classification metrics, but for now it is important to note that the role of the FPS is to trigger a checkpoint whenever a prediction is made due to a positive classification.

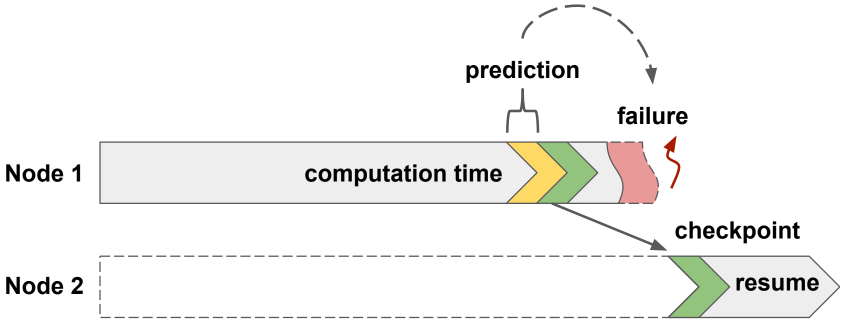

Fig. 1.7: Failure prediction example over time as a failure is avoided by a prediction that is follow by a restart on a different HPC node.

Figure 1.7 provides an example of how an FPS can help react against an incoming failure by triggering a checkpoint before an application crash. The trigger was caused by a positive classification of sample data gathered prior to the prediction. The computations of the application are shown in light gray, the prediction using health time data in yellow and the checkpoint/restart procedure in green. In this example the failure predictor is capable of classifying the health data with enough lead-up time to perform a checkpoint (and migration) before the failure. A usable FPS should avoid unnecessary checkpoints by minimizing the amount of false alarms produced. This can be achieved by tuning the binary classifier to produce better classification metrics.

Classification Metrics

This work will deal specifically with supervised learning of binary classifiers that are trained to classify input sample data into one of two output classes [33, 34]. These will be failures and non-failures (ok) events. The supervised learning aspect is done by training the binary classifier with input sample data already labeled [33, 34, 35] into one of either failure or non-failure class. These failure labels are referred to as failure events, and are mined using a combination of HPC batch system logs and node sensor data.

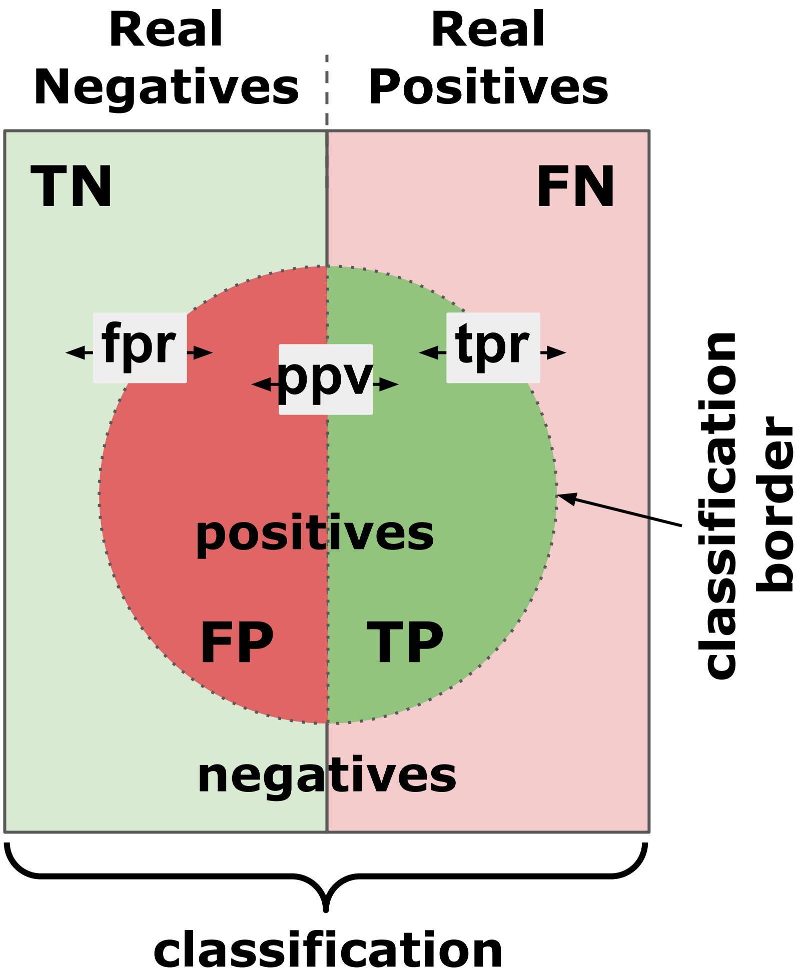

Fig. 1.8: Diagram of how binary classification sorts real classes into classified classes. The different ratios give the classical ML metrics.

The binary classifier in this context should associate input sample data with events that are positive (failure) or negative (non-failure, ok). The following classical machine learning metrics describe the performance of a binary classifiers that sorts data between failures and ok events:

- *Positives (P)* refer to all samples classified as failures.
- *Negatives (N)* refer to all samples classified as non-failure / ok events.
- *True Positives (TP)* refer to amount of correctly classified failures.
- *True Negatives (TN)* refer to correctly classified non-failure / ok events.
- *False Negatives (FN)* refers to misclassified failures as ok.
- *False Positives (FP)* to ok events that are misclassified as failures.
- *Precision:* the probability of a positive prediction to be correct, also expressed as a the positive predictive value ($\boldsymbol{ppv}$) calculated with $\frac{TP}{TP+FP}$.

- *Recall:* the probability of correctly classifying a positive event, or true positive rate ($\boldsymbol{tpr}$) calculated with $\frac{TP}{TP+FN}$.
- *Fall-Out:* the probability of misclassifying a positive event as a negative event, also expressed as false positive rate ($\boldsymbol{fpr}$) calculated with $\frac{FP}{FP+TN}$.

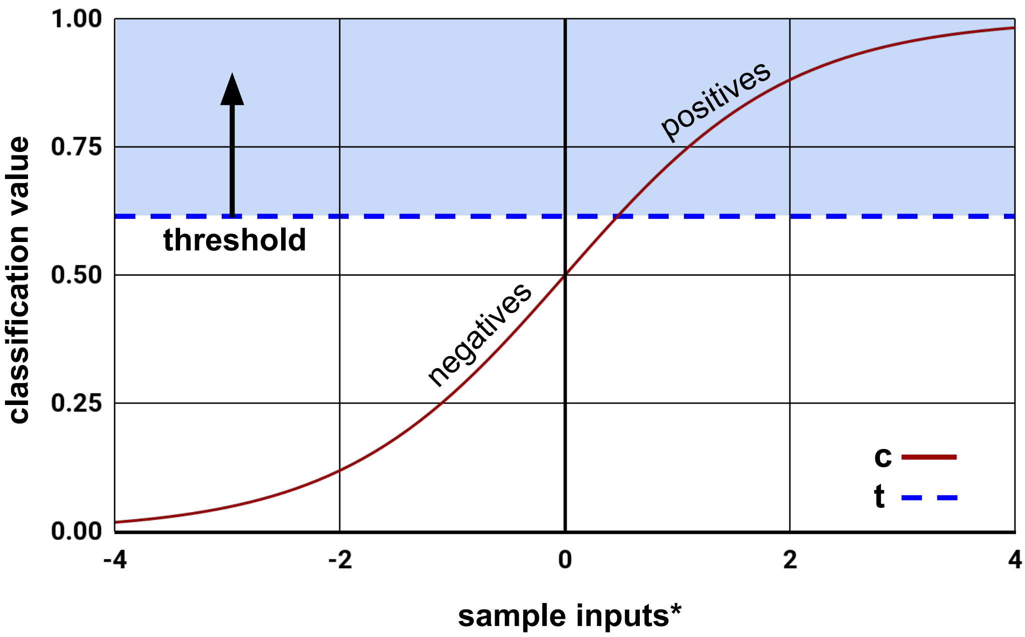

Fig. 1.9: Plot depicting a Sigmoid $S(x) = \frac{x}{1+e^{-x}}$ activation function that classifies between two classes if the activation value c from $S(x)$ is above t.

Binary classifiers give a preliminary classification value $c \in \mathbb{R}$ traditionally $\in [0, 1]$ that is used to decide (classify) the class of a sample. If the value c is above a threshold t such that $c > t$, the classifier chooses a specific class for the sample, i.e: positive [33]. Figure 1.9 shows the relationship between then the classification value c and the decision threshold t. The $\boldsymbol{tpr}$ and $\boldsymbol{fpr}$ are calculated for a range of thresholds and their change is plotted. As the classifier processes a series of sample inputs (*usually multi-dimensional), it decides the class of the sample based on the threshold. The metrics of Figure 1.8 can be refined by adjusting the threshold and sorting sample data between positive and negative classifications. A trivial and permissive threshold to maximize positives ($t = 0$) would allow all samples to be classified as positive. This would effectively maximize true positives, which is desired, but would also maximize false positives, which is not desired. A negative sample will be miss-classified as positive if the value $c > t$. On the other hand, the classifier threshold could trivially be chosen such that all samples are classified as negative, maximizing both true and false negatives ($t = 1$). A non-trivial approach requires evaluating the $\boldsymbol{fpr}$ and $\boldsymbol{tpr}$ while changing the threshold and finding an acceptable value for t. A different approach could apply a different function or decision method to the final classification (i.e: $argmax$ with multiple c_i). Plotting the $\boldsymbol{tpr}$ against $\boldsymbol{fpr}$ for all thresholds as seen on Figure 1.9 helps choose a suitable threshold for a usable true positive rate and acceptable low false positive rate.

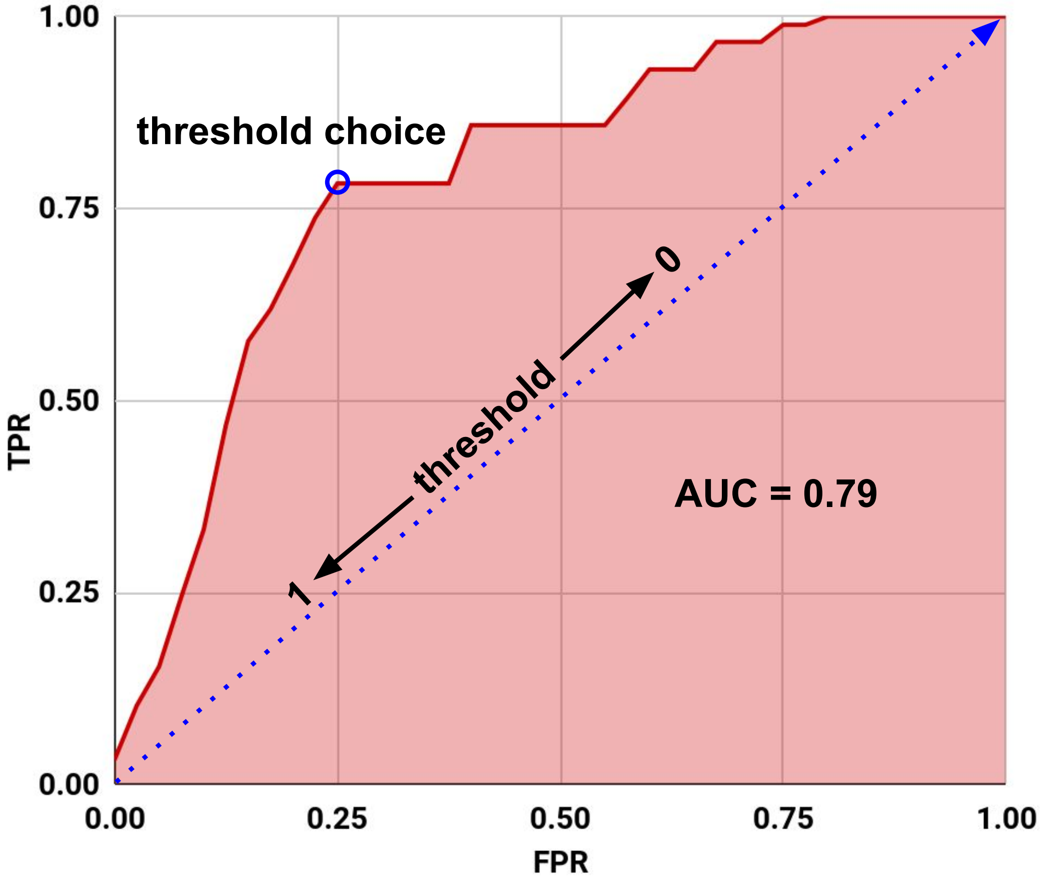

Fig. 1.10: Plot of an ROC curve and AUC value (0.79) from a fictional binary classifier. The horizontal axis shows the false positive rate and the vertical axis the true positive rate. The rates are plotted for all classification thresholds and an arbitrarily good threshold (high $\boldsymbol{tpr}$ and low $\boldsymbol{fpr}$) value is chosen. As the rate is decreased, more data samples are classified as positive.

Figure 1.10 shows the receiver operating characteristic **(ROC)** curve for a binary classifier [33, 34] with a threshold $t \in [0, 1]$ for an output. If the classification value is above the threshold, the sample is considered positive. As the threshold is decreased ($\longrightarrow 0$), more failures and non-failure events are classified as positives, increasing both the $\boldsymbol{fpr}$ and $\boldsymbol{tpr}$. An arbitrary chosen threshold marked in blue can be selected to achieve a low enough $\boldsymbol{fpr}$ (0.25) that still maintains a desirable high $\boldsymbol{tpr}$ (>0.75) for prediction purposes. The threshold choice could slightly be moved to the right on the FPR horizontal axis, but this would increase the undesired $\boldsymbol{fpr}$ and not increase the $\boldsymbol{tpr}$ significantly.

A related measure to ROC is the area under the curve (AUC) that quantifies the quality of the classifier across all possible classification thresholds [33, 34]. The AUC is a single value $A \in [0, 1]$, where values of A close to 0 show a classifier that fails to classify any true positive samples as positive, and values of A close to 1 show a classifier that can correctly classify all true positives and true negatives (at some threshold). The more $A \longrightarrow 1$, the better the classifier performs across its thresholds. The AUC is not required to minimize false positives (the ROC suffices) and is not explored in much detail beyond its visualization in ROC plots.

Although the ROC is widely used in the machine learning literature, it is rarely reported in works that deal purely with failure prediction of HPC systems. This makes comparison to previous works difficult. The ROC analysis of this work is nevertheless still helpful to minimize the fpr as much as possible.

Scalability problem

The common metrics are not sufficient to evaluate the false positive quality of an FPS in highly parallel HPC environments. Typical FPS are trained to predict failure events of individual components and can achieve good enough false positive rates for single nodes. However, HPC systems need to predict failures for thousands of nodes and make mitigation decisions based on all the nodes used by the parallel application, since any failing node will make the entire distributed application crash.

Even a low false positive rate for continuous predictions of individual components will accumulate when predicting failures for hundreds of nodes. When considering thousands of predictions, the individual low probability of classifying a false positive per component translates into a high probability of **at least one** FPS classifying a false positive. At large scales, false alarms become costly because any positive prediction (including false positives) will trigger a checkpoint. This results in at least one false positive from thousands of nodes constantly triggering checkpoints, effectively rendering FPS unusable for large scale systems.

Despite good *true positive* failure prediction rates that can identify plenty of failure events in previous works, FPS have not been successfully used in practice for HPC due to false predictions triggering too many resilience measures like unnecessary checkpoints. We call this situation of least one FPS predicting a false positive, the **false positive prediction problem** or FPP for short. Reducing the false positive prediction rates is therefore important to avoid unnecessary preemptive checkpoints when using thousands of nodes. This FPP occurs when the standard performance metrics for machine learning binary classifiers are combined to thousands of nodes.

In Section 4.1.4 we will introduce a new global metric, namely *the probability of triggering unnecessary checkpoints* or UC for short. This UC will calculate the chance of at least one false positive event happening in n node predictors.

Neural networks as binary classifiers

In Chapter 4 we will show an FPS that can attain low enough false positive rates and make them usable on large scale systems where predictions are performed for every component. A usable FPS with low false positives is achieved by aggregating batch system traces, hardware sensors, and operating system counters into one FPS that predicts batch job failures on a per-node basis with sufficient lead-up time for checkpoints. We implement our FPS using artificial neural networks trained with features (min, max, avg, among others) of hardware sensors and labeled with failure and non-failure tags based on expert knowledge.

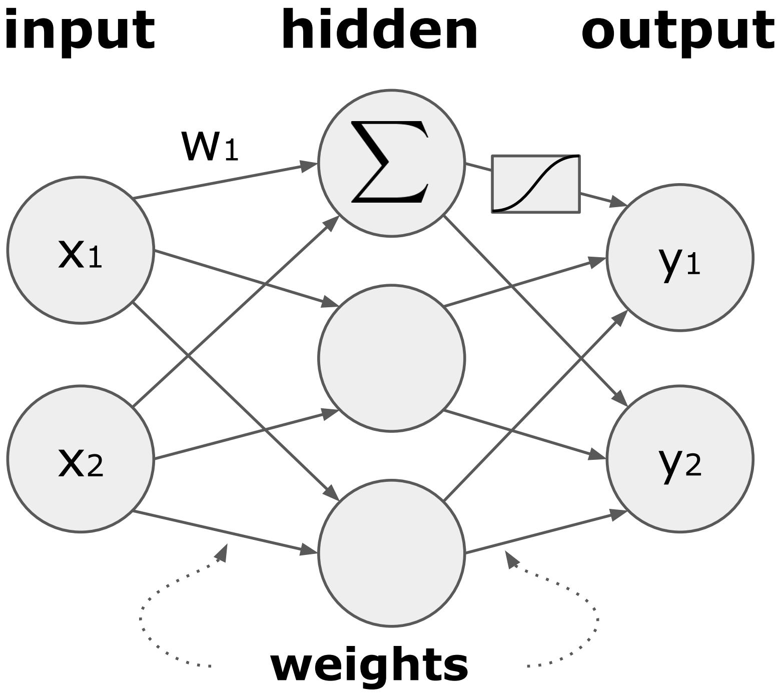

Fig. 1.11: Diagram of a three layer feed forward neural network with a single hidden layer and two inputs and outputs.

Figure 1.11 shows a mock three layer feed forward artificial neural network [33, 35]. Each circle represents an artificial neuron. The input layer simply feeds the inputs x_i to each neuron in the hidden layer. The hidden layer then linearly sums the product of each input with its weight $\sum x_i \cdot w_i$. The sum in the hidden layer neuron is later passed through the activation function before being sent to the next neuron with a new weight [33, 35]. The outputs y_i are then used to determine the classification result. In the case of binary classification, two neurons can be used with a $\mathrm{softmax}(\mathrm{y_i})$ function to reduce the classification to a single value decision.

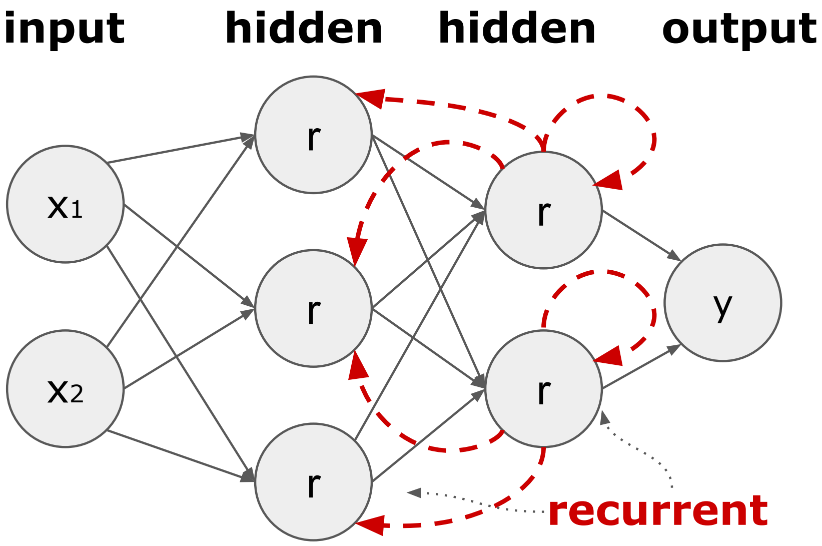

Fig. 1.12: Diagram of a mock arbitrary recurrent neural network with two inputs and one output. The recurrent links to neurons in the same or previous layers are shown in red.

Figure 1.12 shows a mock recurrent neural network with internal recurrent layers that can receive inputs from themselves or layers further in the network. Recurrent networks try to model temporal properties in the classification by use of the recurrent links to previous layers and with themselves [33].

An example of a specialized recurrent neural networks are Long Short-Term Memory (LSTM) networks. These networks can learn relations between time sequences too, but are also able to overcome the vanishing gradient problem (slow training) present in recurrent network training [36, 37]. An LSTM recurrent network will be used to supplement and aggregate the results of multiple basic feed forward networks that predict failures at distinct time intervals.

Proposed Failure Prediction System

The FPS is built using an ensemble of multiple neural networks [34, 33] that vote with their classifications to produce a final prediction. An ensemble is just multiple classifiers that make a joint decision for a final classification.

The viability of failure prediction to reduce waste will be evaluated by:

- Showing that per-component predictions trigger too many unnecessary checkpoints within production HPC systems.
- Defining a job-wide quality metric for failure prediction systems based on the probability of triggering unnecessary checkpoints (UC).
- Showing that previous works with reasonable classification rates still provide too high UC probabilities.
- Introducing ensemble of neural networks (NN) that vote to predict failures.
- Demonstrating that the ensemble voting methods can reduce false alarms for hundreds of nodes due to the low enough UC probability.
- Showing that the proposed FPS with machine learned voting has a false positive prediction rate of 0.00004, a recall rate of 0.7448 and a UC probability of 28.93% for 8,192 nodes.

The proposed FPS could supplement existing interval-based checkpointing strategies [38].

2

Co-allocation of parallel applications

HPC systems are typically used by diverse users with various complex scientific applications that have different workloads and resource utilization patterns. A university or state HPC system might be open to users from research groups of the social sciences, biology, mathematics and physics among others [39, 40]. The user count is also reflected in the diversity of scientific problems. Mathematicians might be interested in using HPC platforms for purely numerical problems [41], biologists could process Big Data Genetic studies [42, 43] and chemists might solve particle simulations or protein folding problems [44]. These are just some examples from the plenitude of scientific problems that can be computationally processed in an HPC system.

Unfortunately not every scientific workload can fully utilize the various compute resources (e.g: CPU, Memory, Network) present in a compute node. A compute intensive application doing matrix-matrix multiplications could fully saturate the memory bandwidth and not use the network at all [45, 46]. A parallel plasma physics computation could on the other hand use the network interconnect more effectively [47]. Individual scientific application workloads usually saturate a limited amount of resources, using them in steps at different times. This happens for MPI applications that are usually highly interconnected with various phases of parallel communication and computation. Any individual process waiting for bandwidth, I/O operations or access to accelerators could stall the progress of all other processes waiting for synchronization at a communication MPI call.

Despite the variety of workloads available to the HPC system, most compute nodes do not have all of their resources saturated. This happens because batch systems only allocate a single application per node, leaving behind the opportunity to use heterogeneous workloads for resource saturation. This limitation is not technical, but rather traditionally imposed by policy in order to reduce security risks that could arise from sharing local data or memory.

There are also concerns that the performance of the applications will degrade significantly when nodes are shared. We will tackle this aspect by offering evidence that sharing nodes can increase macro performance for the overall HPC system. The variety of applications can be exploited by combining certain workloads in order to increase resource utilization within individual nodes. It will be shown that certain stages of the runtime from applications will degrade differently depending on the workload combination.

The types of experiments performed further in this chapter will also use parallel programs to test how the inter-application interference behaves when sharing high node counts of up to 128 nodes. This will then be scaled from isolated experiments of application pairs to an entire batch system of queued applications with various types and sizes. These batch experiments explore the limitations of scheduling co-allocation pairs when using priority queues. They will also give a novel overview of the resource utilization savings that HPC systems can expect from computing their batch loads using co-allocation.

The scheduling limitations of co-allocated applications will be addressed by combining batch scheduling heuristics with co-allocation strategies that pair suitable applications together. This will make it is possible to better saturate the resources of individual compute nodes while also reducing idle nodes. Overall we will show a higher job throughput for co-allocation batch systems and less core-hours needed to complete the same workloads.

2.1 Related Work

Node sharing can be used to pair different applications that have orthogonal resource needs with the goal of utilizing more resources on a node. Time sharing enables shared access to the node by multiple applications. Stillwell *et al.* showed with simulations that controlled time-sharing scheduling algorithms can offer improvements over traditional HPC batch scheduling [48]. Their simulations assumed that all tasks are CPU-bound and did not address inter-job interference. In contrast to simulating artificial workloads, we performed experiments with real MPI workloads, where applications do MPI communication and numerical computations.

Xiong *et al.* [49] proposed a framework for co-allocating applications in HPC clusters by profiling applications offline based on their resource utilization intensity. Their work focused on predicting application interference using machine learning and support vector machines (SVM). Their scheduling decision uses the prediction of the SVM to decide whether to co-allocate the highest priority job in the queue. The co-allocation scheduler proposed later in this chapter uses a lookup table of good and bad pairings to decide whether two applications should be paired. It does not require prior profiling of individual application metrics and training of machine learning methods. This work uses similar MPI applications and expands the problem sizes from 8 to 128 nodes. In addition to individual experiments of application pairs, this work shows savings for batch systems containing thousands of jobs. This work also shows that the combined makespan of the co-allocated pairs is reduced, despite the increased runtime of each individual application.

Yang *et al.* [50] developed an oversubscription framework for long running cloud applications that uses performance metrics to select target co-allocation machines. Their work also shows that oversubscription increases resource utilization and that workload makespan can be reduced.

Kuchumov *et al.* [51] recognized that batch schedulers can leave compute nodes underutilized and proposed the use of control groups to limit and assign fair shares of CPU time, memory and network to each application sharing cloud nodes. Their work left unclear the policy by which the resources should be assigned to each application, it performed no parallel experiments showcasing the benefits of such a technique, and focused on cloud systems. The work done in this chapter takes the opposite approach by allowing selected application pairs to compete for resources and saturate each hardware element as much as possible.

Leng *et al.* [52] have shown that the use of SMT in computationally intense scenarios does not degrade performance due to already highly utilized resources. We corroborate their findings and use HPC parallel applications that involve inter process communication to make interleaving use of CPU possible.

De Blanche and Lundqvist investigated slowdowns due to resource sharing when using similar processes. They showed that making the same application share the same node (*Terrible Twins*) can lead to overloaded resources [26]. Those were single-node experiments while this thesis concentrates on the same effects for multiple nodes.

Wende *et al.* [53] showed how oversubscription of two processes per hardware CPU thread of the same scientific application, cause more than a factor two performance degradation compared to no oversubscription. In the isolated experiments we were able to corroborate these results and included the gained knowledge in a co-allocation scheduling heuristic.

Iancu *et al.* [54] also explored the throughput benefits of oversubscribing two single node applications. They showed that co-allocating pairs of shared memory workloads does improve throughput. In the evaluation of this chapter we also show that these throughput benefits remain when co-allocating parallel multi-node applications and with batch system workloads. They also show that partitioning cores exclusively among applications leads to no benefits and that the Linux scheduler can affect performance significantly.

The Portable Hardware Locality tool hwloc [55] can impact performance by scheduling OpenMP threads according to their affinities and by placing MPI processes according to their communication patterns. The importance of application-specific local scheduling and placement is evident and we use static placement with core sharing mentioned previously for multiple nodes. An automatic system will be shown to be unnecessary in our co-allocation scenarios, since only one of the core pinning methods ends offering good performance.

The node sharing approach we present is implemented as an extension of existing batch system algorithms using *Backfill* [56][57]. Optimal scheduling is complex and requires prior knowledge of applications, making it hard to apply them in practice. Józefowska and Weglarz introduced the notion of discrete-continuous scheduling [58] to focus on the run- and release-time of jobs [59]. Their runtimes consider how the quantity of allocated resources affects a jobs processing time. Their analysis is a mathematical programming formulation that can minimize the scheduler's make-span. Unlike their work that requires knowledge of resources, runtimes and changes in applications sizes, this work uses basic heuristics to improve scheduling without attaining mathematical optimality.

Brinkmann *et al.* proposed a deterministic approximation algorithm that determines a resource aware schedule [60] to improve system performance. Their algorithm distributes resource requests among jobs to prevent them from exceeding the resource limits and to minimize the makespan by focusing heavily on resource assignment. Despite their analytical makespan reduction for single node multi-core systems, the authors only showed results for a single node, while an HPC system consists of a large number of nodes with heavily interconnected tasks. In this thesis we try to expand on this work testing scheduling heuristics on multi-node systems using MPI applications. We also take an empirical approach by testing MPI applications and do not perform a mathematical (analytical) optimization.

Dealing with static resource requirements is a challenge for co-allocation. Süß *et al.* showcased the difficulty of dealing with static resource requirements due to applications changing their requirements at runtime [61][62]. They proposed a

VarySched scheduler that tries to solve resource contention per node by controlling access to requested resources by applications. Although we do not solve resource contention at a node level, the batch scheduling approach developed could be compatible with it.

Theoretical scheduling approaches that search for optimal scheduling approaches in homogeneous distributed systems exists, but are not compatible with live HPC systems and do not necessarily apply to co-allocation. An example limitation is present when minimizing the scheduling makespan of n jobs on m identical parallel machines. Graham [63] showed that scheduling jobs in a sorted non-ascending order of their runtimes bounds an optimal solution by a factor of $\left(\frac{4}{3} - \frac{1}{3m}\right)$. This type of list sorted scheduling that aims to minimize a Longest Processing Time (LPT) is unacceptable in production HPC systems where parallel jobs have priorities based on arrival time, quotas and institutional rules. This problem is already NP-Hard [64, 65] and considering more complex scenarios like vectors of resource requirements [66] does not improve the complexity. This work will therefore focus on taking runtime measurements with running jobs on batch systems. It will also implement basic co-allocation heuristics on a production batch system for ease of deployment.

Previous works that improve scheduling of resources have achieved better makespans for multiple resources on multi-core systems while limiting their investigations to analytical or theoretical representations of system workloads. In this thesis we will explore the benefits of co-allocation when using simple scheduling heuristics for a series of actual MPI parallel workloads that improve resource utilization to complete equivalent workloads.

2.1.1 Backfilling and *First-fit* scheduling

Allocation is the mapping of available computing nodes to pending parallel applications and is the responsibility of the HPC systems batch system scheduler. Users submit jobs to the batch system where they queue for resources and wait for them to be available. Batch scheduling is the process of scheduling user submitted heterogeneous jobs for execution on an HPC system. Traditionally the batch system chooses high priority jobs from the queue and assigns them computing nodes exclusively. Depending on the size of a job, available resources and priority policies assigned to the users, a job might wait for resources to be available. Allowing for co-allocation in scheduling algorithms can allow for multiple jobs to be placed on shared computing nodes. This allows the system to make use of the orthogonal resource needs of jobs and improve overall system utilization.

HPC schedulers have plenty of configuration options that make them behave in complex and unique manners. The following properties are common options of relevance to co-allocation:

- Independent and long running HPC jobs can be configured to work without real-time user intervention. These HPC jobs define a set of required resources to run the job.
- A scheduler can have job queues where the jobs wait according to their arrival time, size or priority. Each job queue is sorted by priority, usually obeying first come first served or system defined hierarchies.
- Jobs with a higher priority receive resources and run before jobs of a lower priority.
- A job that mistakenly exceeds its allocated resources or allotted time requirements will be terminated.

Optimally scheduling multiple requests for r resources on n nodes is known as vector-scheduling. Solving vector-scheduling in an optimal way is too complex, with non-polynomial runtime to an optimal schedule. Due to the complexity of scheduling algorithms, batch systems do not perform a real-time optimal allocation with a plan-ahead schedule. Instead, schedulers in HPC systems make the scheduling decision in a very short time window with limited available information. Scheduling algorithms limit the choice of which program to start next, to the high priority jobs that fit in the available resources.

The *Backfill* algorithm allows for the highest priority job (HPJ) to run if enough nodes are available [67]. If no resources are available straight away, *Backfill* will reserve currently free resources and compute when the HPJ can start based on the current jobs running and their expected end times. A scheduler using *Backfill* will not start lower priority jobs (LPJ) that would delay the start of all the pending HPJ. *Backfill* can however run any LPJ on currently available resources as long as their end time is before the scheduled start of all pending HPJ and they can actually make use of the available resources.

The previously described backfilling behavior is depicted in Figure 2.1, where a queue of four jobs wait for resources. The left side shows the state of jobs and resources at timestamp T_0. Jobs J_i require n_i compute nodes (first numeral) and run for t_i time units (second numeral) and are represented as rectangles. The jobs are ordered in a FIFO priority with the job at the bottom in blue having the highest priority.

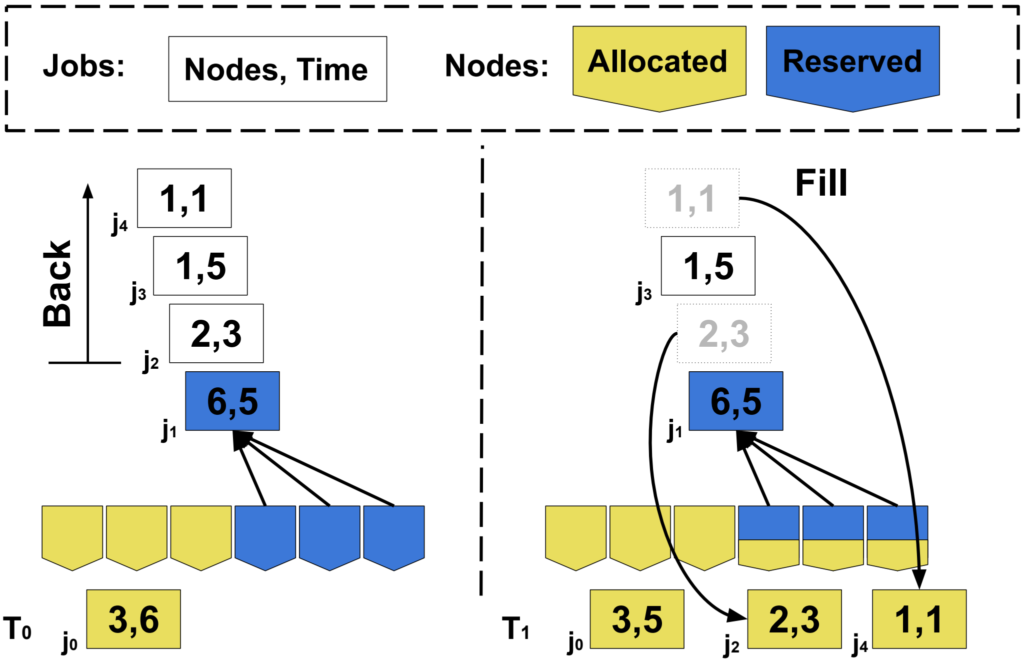

Fig. 2.1: Scheduling steps performed by a *Backfill* algorithm where backfilling is taking place to use idle resources.

At T_0 job j_0 $(3, 6)$ is running and has six more time units left to run while using three nodes. The j_1 job $(6, 5)$ is waiting for j_0 to finish and has reserved three nodes that are currently idle (see arrows) while it waits for the other three nodes to become available. The rest of jobs are waiting for their turn in the queue.

At timestamp T_1 the *Backfill* algorithm backfills the jobs j_2 $(2, 3)$ and j_4 $(1, 1)$. The job j_0 still has five time units left to run and the *Backfill* algorithm chooses the two jobs to use the reserved resources of the pending $(6, 5)$ job. The job j_3 $(1, 5)$ would also use one single node like j_4 $(1, 1)$ and has the highest priority of the two, but starting it would delay the start of the pending j_1 job. For this reason, *Backfill* choose to skip j_3 $(1, 5)$ and backfills j_4 $(1, 1)$ instead.

The backfilling algorithm starts to leave a lot of available pending nodes unused when the queued jobs request long running hours and hundreds of nodes. In this scenario the *Backfill* step is unable to find suitable jobs that can use pending nodes and that will finish before the start of a higher priority job. This drawback will also apply to a backfilling scheduler that is trying to find suitable co-allocation pairs of applications.

The *First-fit* scheduling algorithm can mitigate this issue by choosing the first job that can fit in a given set of idle nodes. This allows it to fill in the gaps and use the pending nodes at the cost of the FIFO rule. Job priorities are only used for choosing which job is considered first and does not limit the start of jobs with lower priorities. *First-fit* has the advantage of making use of otherwise idle resources in *Backfill* and this can allow it to achieve a better throughput. The disadvantage of *First-fit* is a limited fairness, where big jobs requesting large amounts of resources can starve by never starting since there is always a smaller job that could be run.

Both scheduling algorithms can be configured to allow sharing of nodes among multiple applications. In both cases, the general properties of the scheduling algorithms remain. The main difference when enabling co-allocation is that *First-fit* can take advantage of more jobs when choosing a partner for an already running job. *First-fit* is also less restrictive because all pending jobs can be considered while *Backfill* can only start jobs that do not delay HPJ that are pending.

2.1.2 Co-allocation techniques

We explore multiple techniques of co-allocation to measure the impact that various mappings of computing tasks to MPI ranks can have on runtimes and system throughput. This is done by comparing co-allocation against the conventional technique of mapping application tasks to exclusive cores. The focus is kept on MPI parallel applications due to its predominance in HPC applications.

Parallel applications map their MPI ranks to physical cores or to logical cores. The mapping two ranks to logical cores of the same physical core is used to oversubscribe the physical core using simultaneous multithreading (SMT) techniques like hyperthreading (HT). At the MOGON II system from the Johannes Gutenberg University Mainz we were able to determine that within twelve months (11/2017 – 10/2018) 22% of the jobs utilized one compute process per physical core in the compute nodes. All other jobs (78%) oversubscribed cores by running two or more compute processes per physical CPU core. Oversubscribing within the same application is traditionally considered sub-optimal for compute heavy workloads [54].

Figure 2.2 shows three methods of pinning/mapping tasks to cores in a two socket compute node. The a) sub-figure shows traditional one task per core exclusive mapping, b) oversubscription by the same application using HT and c) co-allocation with oversubscription by two distinct applications.

Figure 2.3 depicts five job (process) mappings for two applications P and Q. Each processor (solid black dot) consists of two physical cores, each core supports hyperthreading and each node consists of four cores. MPI ranks running on the cores are

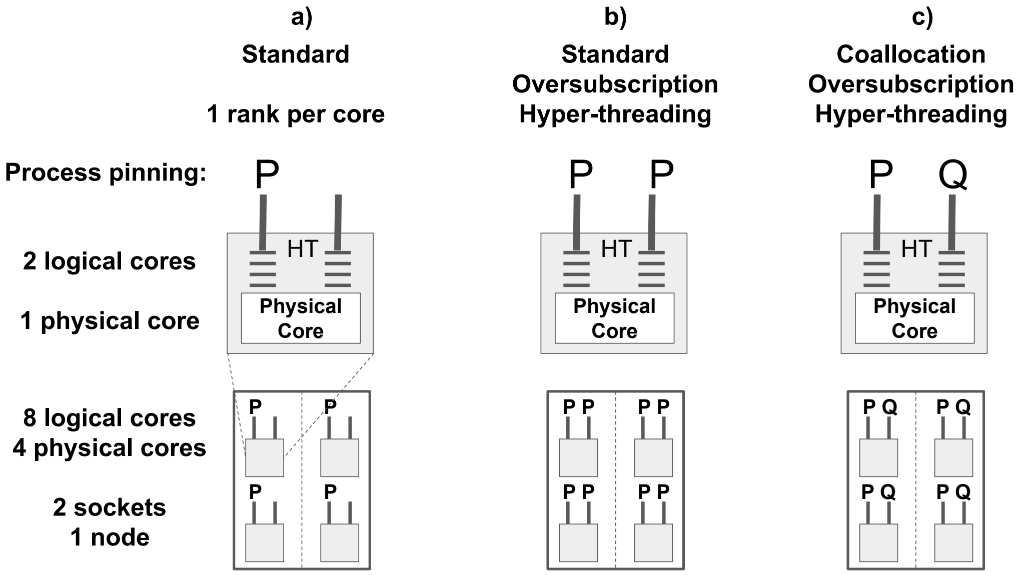

Fig. 2.2: Methods to map processes to cores in compute nodes.

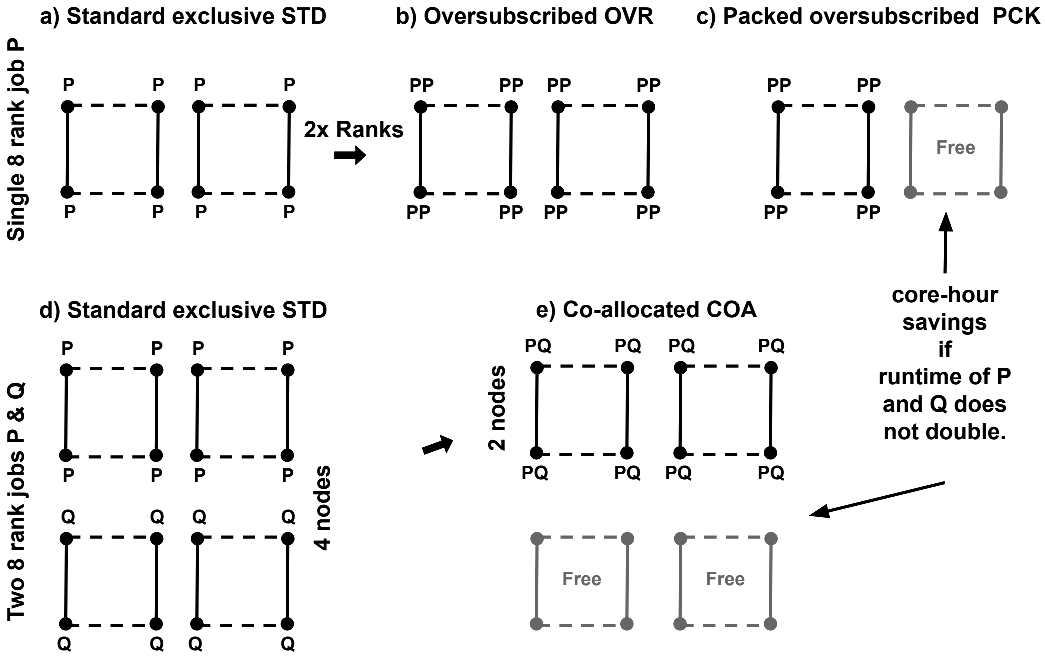
a) Standard exclusive STD
b) Oversubscribed OVR
c) Packed oversubscribed PCK
Single 8 rank job P
2x Ranks
Free
core-hour savings if runtime of P and Q does not double.
d) Standard exclusive STD
e) Co-allocated COA
Two 8 rank jobs P & Q
4 nodes
2 nodes
P
Q
PP
PQ

Fig. 2.3: Standard mapping, oversubscribed and co-allocation techniques for applications running on a four core node.

depicted by characters. Cores in the same socket are connected by solid lines while cores on different sockets are connected by dashed lines.

Figure 2.3-a shows *standard exclusive allocation* (STD) where one rank of job P is pinned to one physical core exclusively. Under exclusive allocation every compute process (rank) from an application has unrestricted access to their assigned CPU core. The standard exclusive allocation is used as a baseline for experiments. The baseline is expected to produce minimal runtimes since CPU cores are not shared. The runtime is also shorter because node resources are not shared with other applications, and because the performance is not degraded due to intra-application interference or oversubscription.

Oversubscription of one application requires either the doubling of ranks as shown in the *standard oversubscribed allocation* (OVR) technique or to use the existing rank count and pack the ranks into half the nodes using *packed oversubscribed allocation* (PCK) (see Fig. 2.3-b and Figure 2.3-c respectively). Both scenarios solve the same computational problem, with PCK utilizing half the computational hardware.

The more relevant allocation technique for co-allocation (COA) shares nodes between applications, which transforms the STD mapping of two applications using four nodes shown in Figure 2.3-d into sharing of half the nodes in (Figure 2.3-e). COA uses hyper-threading and pins one process of **each** application to the same physical core through neighbouring logical cores.

An alternative co-allocation could have each application exclusively use one socket each, with memory mapping to the socket. This would isolate the cores and the memory bandwidth for each application while still sharing the nodes. In this setup network and I/O could be shared, but not the CPU pipeline or memory bandwidth. This would still keep the same utilization of exclusive resources as the non-co-allocated version while limiting the potential gains from saturation of resources. Exploratory experiments showed no benefits from this setup and a previous work showed similar insights [54].

Computation intensive applications do not gain performance from distributing their workload to more threads when the compute power stays the same. Oversubscription of different applications could take advantage of interleaved (out-of-order) execution of two processes as provided by HT. This can be done due to the orthogonal resource requirements that exist in large sets of varied scientific programs. A simplified depiction of Hyper-threading enabling out-of-order execution based on Intel documentation is shown in Figure 2.4. HT can saturate some internal parts of the physical core with two processes. This offers speedups when compared to simply running each process one after another.

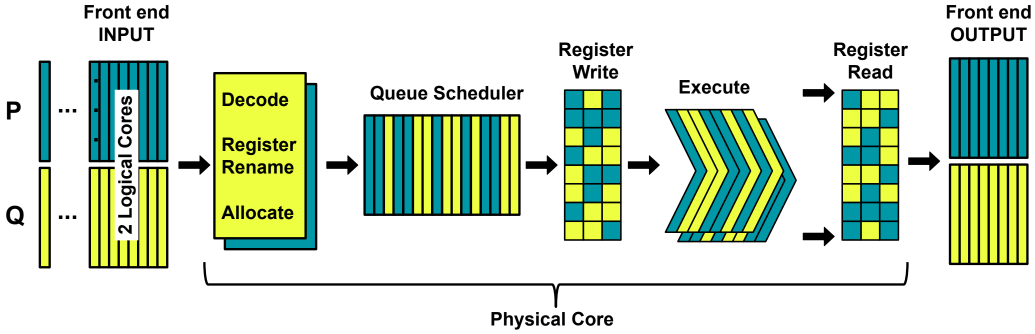

Fig. 2.4: Simplified diagram of a hyper-threading pipeline used by two processes P & Q to exploit out-of-order execution.

Backfilling and *First-fit* can be configured to use both STD and COA methods when scheduling applications. Exclusive node allocation is considered the default approach in most batch systems while sharing of nodes through co-allocation is traditionally not used due to performance concerns.

The co-allocation methods here go beyond standard node sharing that let the software threads do time sharing as regulated by the operating system. Co-allocation pins two different jobs to the same cores but different hyper-threading (HT) threads. This lets the hardware schedule the processes as hardware threads instead of letting the operating system decide time shares. With twice the number of threads, HT can lead to cache contention and pollution while communication and I/O-bound applications can benefit by interleaving their operations.

It is nevertheless expected that placing two instances of the same application on the same node can lead to significant slowdowns, as the same application instances can compete for the same resources at the same time. Equal instances of a program solve the same computational problem and produce similar loads. Sharing the hardware will therefore lead to contention of the same hardware elements (i.e: network adapter, ALU), which will indeed slow down the two instances. This setup will also be explored experimentally.

Measurements of cache-hits and cache-misses across different co-allocation configurations using Intel's *Processor Counter Monitor (PCM)* [1] tool where indented and tried but ultimately unsuccessful in the HPC hardware used. The PCM tool required root configurations that were not accepted within the HPC production system used for experiments.

---

[1] https://github.com/opcm/pcm

2.2 Methodology

We will explore how specific co-allocation pairings can increase a batch system's throughput with two types of co-allocation experiments. First, five applications were paired and run in permutations within the same compute nodes. These isolation experiments allow to determine which pairs can finish the same computations in less node-hours. Using half of the nodes to run both applications will share resources and lead to runtime slowdowns for each application. The goal is to achieve slowdowns that are less than twice the original runtime, leading to savings in the total core-hours needed to solve equivalent workloads. The slowdowns are less than twice the original runtimes when using half the computing nodes, so fewer node-hours can be attained. Second, multiple application instances were submitted to a batch system with co-allocation capabilities, where applications could share nodes in a dynamic environment. These batch experiments can corroborate that the isolated findings do translate to savings in a complex and realistic environment. Both experiments compare co-allocation to standard exclusive strategies. In this section we continue by describing the software and hardware setups and how the experiments were performed.

2.2.1 Hardware and Software Setup

All experiments were performed in the MOGON II HPC system at the Johannes Gutenberg University Mainz using up to 512 Skylake nodes, 85 GB of memory and a 100 Gbit/s Intel Omnipath interconnect. Each Skylake node has two 16-core Xeon Gold 6130 CPUs with hyper-threading, leading to 64 usable logical threads per node. Both CPUs were configured to behave as four independent NUMA nodes with independent cache hierarchies by using the Skylake Mesh Architecture [68].

Application were compiled using GCC 6.3 and OpenMPI 3.1.0 [69] without OpenMP multithreading on a CENTOS 7 distribution. Disabling native multithreading limits the multi-core implementation of algorithms to MPI processes and makes the mapping of ranks to logical cores simpler. The base batch system SLURM used in the batch experiments was version 17.11.7. It was also expanded with custom co-allocation capabilities to run within the existing SLURM batch system already present in MOGON II.

2.2.2 The scientific applications

For the experiments we used a subset of the NERSC Trinity mini MPI parallel application suite [29, 70] so that we could have a realistic batch job queue of scientific MPI applications that can be scheduled using the techniques described in Section 2.1.1. The applications were selected for their MPI workloads that solve different scientific problems and for their configurable test inputs that make experimental runs easily reproducible. The five applications selected [29, 70] are as follows:

- AMG is a multigrid equation solver for unstructured grids.
- GTC simulates the turbulent transport of particles in plasmas using the Vlasov equation.
- miniGHOST simulates heat diffusions across homogeneous domains using a stencil technique with Dirichlet boundary conditions.
- MILC is a lattice QCD (quantum chromo dynamics) application, which has been derived from the MIMD Lattice Computation (MILC) Collaboration.
- SNAP mimics the computational workload, memory requirements, and communication patterns of PARTISN, which builds a neutral particle transport simulator.

The applications were configured to run with inputs suitable for 512, 1,024, 2,048, and 4,096 MPI ranks equivalent to using 16 to 128 MOGON II compute nodes per job. It used a standard allocation technique of one rank per physical core. The rank counts and problem sizes were chosen for suitable scaling of the applications where adding or removing computational power decreases or increases runtimes respectively (see [29]). However, we did not investigate the scaling properties of the applications. We did not perform weak or strong scaling tests in the strict sense, since they are not relevant to compare the changes between exclusive allocation and co-allocation.

From the documentation distributed with each application and the LBNL technical report [71], the applications used have the following resource utilization properties:

- AMG is highly synchronous and memory bound with parallel efficiency being largely determined by the latency of network communication and memory-access speeds.
- GTC is reliant on random access to memory and is mostly bound by the interconnect due to many small messages used in collective communications.

- GHOST is memory bandwidth bound with some network sensitivity due to global collective communications required at each time step in simulations.
- MILC is computationally intense and memory bandwidth bound with a minimal sensitivity to interconnect since it uses a mix of point to point and collective operations.
- SNAP utilizes large quantities of memory and is memory bandwidth bound. It is also parallelized through spatial decomposition.

2.2.3 Baseline allocation and alternative co-allocation setups

To explore the benefits of co-allocation in batch systems an isolated baseline from the scientific applications is needed. This is done using the allocations previously depicted in Figure 2.3, where each application solves the same computational problem using the following four allocations:

1. Standard exclusive allocation (STD) uses N nodes and places R ranks on R physical cores (one to one). This provides a baseline without oversubscription or co-allocation.
2. Standard oversubscribed allocation (OVR) uses N nodes and maps $2 \cdot R$ ranks to two ranks per physical core. This doubles the number of ranks while keeping the compute power and the problem size the same and will show how adding more threads benefits less from multi-threading.
3. Packed oversubscribed allocation (PCK) takes R ranks and $N/2$ nodes by mapping two ranks to one physical core. It effectively reduces the computational power of STD by a factor of two and will show how two threads benefit less from multi-threading.
4. Co-allocation (COA) takes two applications with R ranks and N nodes each to explore possible benefits in core-hours needed to solve equivalent workloads. The applications share the nodes by mapping one rank of each application to the same physical core through SMT, effectively running two distinct STD configurations in the same node count. The goal of COA is to showcase that specific distinct workloads can be processed in less core-hours when using co-allocation.

The first experiments run applications in isolation using STD to show the applications' runtime in traditional configurations to work as a baseline. Co-allocation is then explored to show jobs running in pairs and sharing physical cores.

The co-allocation pairings were limited to configurations with the same number of ranks for each application. This was done to ensure that all nodes of an MPI program

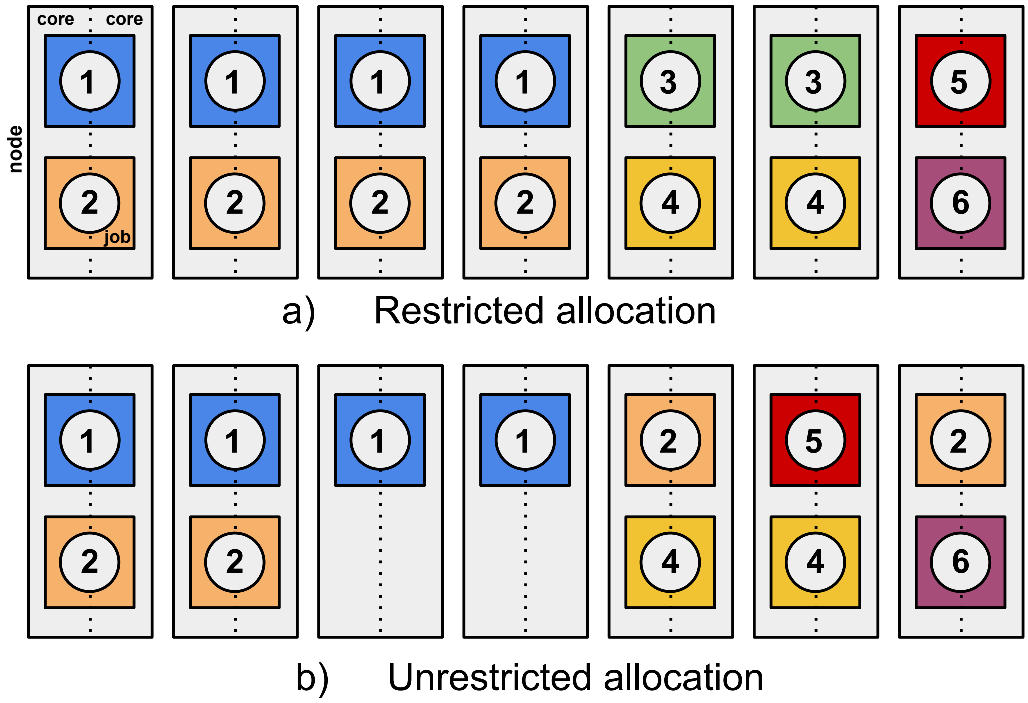

Fig. 2.5: Figure a) shows applications that share nodes using the same rank count. Unrestricted rank counts are avoided as shown in b) due to possible multi-job interference or non-interference of some ranks.

are influenced by the same co-allocated application. Any pairing of applications with different ranks would have a subsection of processes running without interference and another with interference by a different application. Such asymmetric configuration would be difficult to analyze in isolated experiments of pairs and in batch experiments.

The rank size restriction is shown in Figure 2.5-a, where only applications with equal rank counts share nodes in an example system. Figure 2.5-b shows how an unrestricted technique would lead to possible multi-job interference. It is important in the co-allocation batch experiments that jobs share all of their nodes or none at all, avoiding the drawbacks of co-allocation scenarios presented in Figure 2.5-b. Having a subset of nodes shared and another subset used exclusively would slow down Job 1 across all its exclusive nodes. This could be beneficial to the alternative of not running if not enough nodes are available, but is detrimental to performing experiments that consider all possible combinations of partial node-sharing. Furthermore, SLURM's current API enforces that the all the ranks of a job must be pinned to the same core id's across all used nodes in a cluster. This makes configurations similar to Job 2 in Figure 2.5-b impossible.

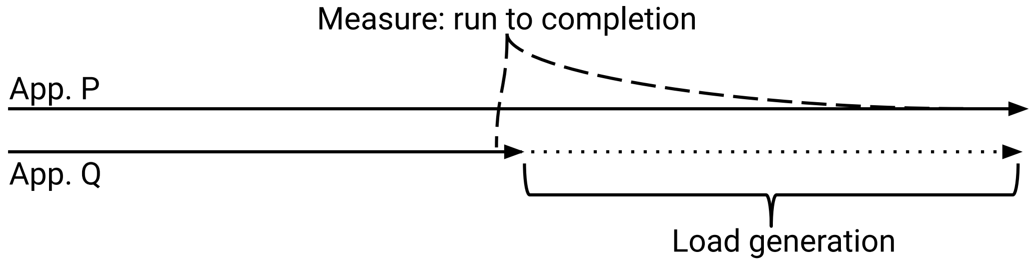

Fig. 2.6: Applications P and Q run to completion. App. P finishes last while Q generates load after finishing.

The isolated co-allocation experiments are designed to run both applications simultaneously and to completion. To counteract the drawback that one application will always finish first, the faster was restarted until the slower one finished (load generation; see Figure 2.6). The extra individual runtimes from this load generation are discarded and not shown in the results. The effects of the load generation are nonetheless present in the runtime of the slower application. This is done to guarantee that the results of isolated tests reflect interference for both applications from start to finish.

2.2.4 Batch system setup

For the batch experiments we created a queue of 4,096 jobs by randomly selecting jobs from the set of the five NERSC Trinity mini MPI applications to create a uniform distribution. Each job randomly got a runtime repetition factors $c \in \{1, 2\}$ that works as an input for the application to solve a static computational problem c times. The unit c also gives *Backfill* the required runtime information to consider whether a job with a shorter runtime can be backfilled in idle nodes. The effective runtime of one single undisturbed execution is 1-4 minutes. The rank sizes n for the jobs have also been randomly and uniformly selected from $n \in \{512, 1024, 2048, 4096\}$ MPI processes.

Simulations started with all jobs already queued. The queue of 4,096 jobs gave the scheduling algorithms a big enough source of pending jobs to fetch suitable jobs to schedule. SLURM was also configured to attempt scheduling the entire queue of jobs every 15 seconds. Baseline batch simulations were performed using STD exclusive allocation and co-allocation.

For co-allocation, two different pairing strategies were used. The first COA strategy naively pairs any possible applications together as long as they have the same amount of ranks. The second COA strategy follows the *Terrible Twins* [26] heuristic and disallows multiple instances of the same application type to share nodes. All co-allocation methods allow jobs to share nodes as long as ranks are equal and regardless of runtime. In all COA strategies, if the scheduler is unable to find a partner application, the already running job will keep using its cores exclusively (full performance) until a suitable job is found for co-allocation. The co-allocation scheduling algorithms are implemented in the SLURM resource manager that runs within the existing systems SLURM and ran on the MOGON II HPC cluster with production system settings.

The goal of the batch experiments is to show that a basic heuristic based on co-allocation can improve batch system throughput by processing more computational work using the same amount of hardware. The isolated allocation tests on the other hand establish a baseline to compare against within a controlled environment.

2.2.5 Accounting

Isolated experiments

For isolated experiments all application pairs start simultaneously. The resources used to complete the computational work were measured for STD and COA experiments. STD experiments use resources in terms of core hours $T_{cores} = t \cdot c$ that account for c **physical** cores used over the (wall clock) runtime t of the application. For co-allocation experiments the individual core hours of a pair of applications T_P and T_Q or the cumulative core hours for the whole pair $T_{P+Q} = T_P + T_Q$ are considered. As mentioned previously, it is expected that running certain pairs together will require less resources than solving equivalent workloads individually.

Batch scheduling

For batch system experiments we used custom metrics based on the available core hours T_{cores} in the HPC system and the core hours $T_{comp} = \sum_i t_i \cdot c_i$ used for computing all finished applications. The T_{cores} lets us know how many core-hours the HPC system had available for the applications to use and T_{comp} determines how much the applications actually used. The more cores remain unused due to scheduling gaps, the bigger the $T_{cores} > T_{comp}$ difference is. In T_{comp} the t_i is the time application i has been running using c_i cores. The exclusive allocation methods (standard, packed and oversubscribed) use 32 cores per node and application in the MOGON II system. Co-allocation methods share those 32 physical cores effectively using half the node per application. The T_{comp}^{STD} is the metric that determines how many core hours a queue of jobs is expected to use using standard exclusive allocation based on the average runtime of each application under STD isolated experiments. We then calculate ratios based on these core-hours metrics and call them **efficiencies**. Note that they are **not** equivalent to scaling efficiency and is only used to described how well a core-hour metric compares to another.

Our custom efficiency metrics have been derived from core hours. The **utilization efficiency** $E_{util} = T_{comp}/T_{cores}$ is the fraction of the available core hours used by the batch scheduler to perform computations. Any unused nodes will hurt this metric since they are not being used to schedule jobs. **Utilization efficiency** helps to quantify the ability of a given scheduling algorithm to pack applications into the available core hours. The **computational efficiency** $E_{comp} = T_{comp}^{STD}/T_{comp}$ compares the core hours used by the scheduler to run applications (no waiting time) to the expected runtime of an exclusive standard allocation (isolated experiments) technique for the same set of applications.

The schedules of different algorithms can be compared using the **scheduling efficiency** $E_{sched} = E_{util} \cdot E_{comp} = T_{comp}^{STD}/T_{cores}$, which considers how many standard core hours T_{comp}^{STD} were scheduled using the available core hours T_{cores}. The scheduling efficiency is T_{comp}^{STD} normalized against the available core hours, which effectively evaluates how much work was processed by the system. The scheduling efficiency considers how many core hours it would have taken to finish the algorithms specific schedule under exclusive allocation and compares it with the actual core hours taken to compute the schedule.

It is important to note that each algorithm can fundamentally compute a different queue of jobs due to backfilling and job selection. The **computational** and **scheduling** efficiencies can be over 100% when batch systems use less resources than an equivalent STD exclusive allocation in isolated environments.

2.3 Evaluation and results

This section showcases the isolated and batch experiments performed using co-allocation in batch scheduling environments while taking advantage of SMT for interleaved resource usage by heterogeneous scientific applications. The first set of tests are isolated allocation experiments and the second batch scheduling experiments. The isolation experiments will show a clearer picture of how co-allocation can affect core-hours and will give baseline for T^{STD}_{comp}. The batch experiments will show a realistic behaviour of co-allocation where applications do not start simultaneously but are paired dynamically as resources become free.

2.3.1 Isolated allocation experiments

First we show the runtime and core hours that each application consumes under STD, PCK, COA and OVR allocation techniques in Table 2.1. Runtimes are shown in green first for each application and core hours are second in blue. In this table darker shades highlighting shorter runtimes and fewer core hours respectively. The values show the runtimes and core hours of the application listed on the left. These values are 50% of the core hours used in case of co-allocation, the other 50% are used by the co-allocated application. The second application at the top is the co-allocated application generating interference load as discussed in Section2.2. The cumulative core hours are discussed later in Table 2.2.

The runtimes show that STD allocation offers lower runtimes for all applications and configurations. PCK has half the available computational hardware and slows down applications significantly. The core-hours needed to complete the workload of the main application are also never better than the alternative of co-allocation. With COA the runtime slowdowns depend on the co-allocated application listed at the top of the table. Each application has at least three good alternative pairs where COA offers less core-hours to complete the main applications workload.

The OVR technique keeps the same physical cores as STD, doubles the number of ranks and uses SMT hyper-threading. Partitioning the same workload on more threads makes the runtimes slower than STD for nearly all scenarios. The reason for this is that OVR pairs *terrible twins*, where more ranks do the same workload and compete for exactly the same hardware [26] resources.

Normal STD allocation achieves better runtimes in most cases, but is typically less resource efficient by requiring more core hours when compared to co-allocation. Figure 2.3-d depicts how COA uses half of the resources when running a pair of applications in shared hardware. Whenever applications slow down by less than two times the base runtime, the core hours required will be less and offer therefore better use of resources. Fewer core hours are depicted in Table 2.1 with darker shades of blue.

The results for individual applications show that at least one co-allocation setup is more resource efficient than a STD counterpart (see blue shadings). Any application

		STD	PCK	COA					OVR
	ranks	exclusive		gtc	amg	milc	ghost	snap	2x ranks
gtc runtime	512	222.3	399.6	374.1	341.8	408.7	482.2	371.2	220.4
	1024	225.8	435.1	384.4	357.4	422.3	481.8	372.2	319
	2048	248.3	603.8	417.6	385.9	470.1	526.7	405.4	418.2
	4096	286.7	798.2	466.9	431.9	518.3	533.3	466.5	494
gtc corehours	512	31.6	28.4	26.6	24.3	29.1	34.3	26.4	31.3
	1024	64.2	61.9	54.7	50.8	60.1	68.5	52.9	90.7
	2048	141.3	171.7	118.8	109.8	133.7	149.8	115.3	237.9
	4096	326.2	454.1	265.6	245.7	294.9	303.4	265.4	562.1
amg runtime	512	76.9	133	120.6	132.6	134.6	149	128.5	80.3
	1024	81.2	144.4	127.5	135.7	139.6	153	129.7	80.5
	2048	85.6	153.1	129.9	139.2	148.2	161	136.6	97.6
	4096	93.2	165.3	140.7	144	163	165	147.1	126.8
amg corehours	512	10.9	9.5	8.6	9.4	9.6	10.6	9.1	11.4
	1024	23.1	20.5	18.1	19.3	19.9	21.8	18.4	22.9
	2048	48.7	43.5	36.9	39.6	42.2	45.8	38.9	55.5
	4096	106.0	94.0	80.0	81.9	92.7	93.9	83.7	144.3
milc runtime	512	220.1	433.5	309.5	293.4	324.8	430	320.1	244.5
	1024	229.8	465.5	321.3	303.3	325.4	437.2	317.8	279.1
	2048	232.8	493.6	325.6	307.5	322.2	416.7	322.3	314.5
	4096	241.2	561.2	341.1	322.7	324.4	406.3	334.8	371.3
milc corehours	512	31.3	30.8	22.0	20.9	23.1	30.6	22.8	34.8
	1024	65.4	66.2	45.7	43.1	46.3	62.2	45.2	79.4
	2048	132.4	140.4	92.6	87.5	91.6	118.5	91.7	178.9
	4096	274.4	319.3	194.0	183.6	184.5	231.1	190.5	422.5
ghost runtime	512	186.7	370.4	243.4	231.9	320.8	362.3	258.9	289.1
	1024	195.2	386.6	259	242.6	349	376.1	262.3	305.6
	2048	218.8	414.8	268.6	263.6	384.5	384.4	273.6	709.4
	4096	184.2	474.2	273.3	255.3	397.8	384.3	273.1	823.9
ghost corehours	512	26.6	26.3	17.3	16.5	22.8	25.8	18.4	41.1
	1024	55.5	55.0	36.8	34.5	49.6	53.5	37.3	86.9
	2048	124.5	118.0	76.4	75.0	109.4	109.3	77.8	403.6
	4096	209.6	269.8	155.5	145.2	226.3	218.6	155.4	937.4
snap runtime	512	254.8	502.1	460.9	441.7	594.6	690.8	482.3	274.6
	1024	208.6	384.1	365.3	347.8	477.3	542.9	373.1	230.9
	2048	219.7	408.5	376.6	363.4	476.3	536.2	386.4	282.5
	4096	293.5	583.3	482.6	491	595.6	611.8	504.2	581.7
snap corehours	512	36.2	35.7	32.8	31.4	42.3	49.1	34.3	39.1
	1024	59.3	54.6	52.0	49.5	67.9	77.2	53.1	65.7
	2048	125.0	116.2	107.1	103.4	135.5	152.5	109.9	160.7
	4096	333.9	331.8	274.5	279.3	338.8	348.0	286.8	661.8

Tab. 2.1: Runtime and core hours for experiments up to 4,096 ranks using standard, packed, oversubscribed and co-allocation techniques. Darker shades highlight better values.

paired with GTC, AMG, and SNAP always uses fewer core hours to complete, which makes these three great applications to co-allocate with. Pairing with MILC also requires fewer or similar core hours than STD in three scenarios. The worst co-allocation partner is GHOST, because all applications require comparable or more core hours when paired with GHOST. The highlighted shadings in Table 2.1 show that shorter runtimes do not imply fewer core-hours.

Table 2.1 shows the individual performance and resource utilization for the application listed on the first column on the left. Table 2.2 shows the cumulative resource utilization of the application pairs combined. The OVR technique is not shown in this new table since it offers no individual benefits in terms of resource utilization compared to COA. It is more relevant to investigate the requirements of solving application pairs sequentially one after another using the STD and PCK techniques ($T_{P+Q} = T_P + T_Q$) using co-allocation.

Table 2.2 shows the combined core hours required to solve fifteen application pairings using STD, PCK and COA. At a first glance most co-allocations require less core-hours than STD, with the exception of GHOST plus GHOST and SNAP. These two pairings offer close resource utilization to STD or worse in the case of GHOST+GHOST at 4,096 ranks and GHOST+SNAP at 512 and 1,024 ranks.

Figure 2.7 further shows how COA can offer relative savings where all but five COA pairings (MILC+GHOST, MILC+SNAP, GHOST+GHOST, GHOST+SNAP, SNAP+SNAP) produce savings of at least 10% and up to 25% for AMG+MILC, AMG+GHOST and MILC+MILC. All application runtimes have standard deviations below 5% when run under isolated conditions. Figure 2.7 also shows that avoiding GHOST+GHOST and SNAP+SNAP and allowing MILC+MILC can reduce resource utilization for the same workload. Pairing aware scheduling strategies could further increase gains when co-allocation applications, but are not explored beyond the no-twins heuristic.

The co-allocation experiments shown here are done in isolation outside of a batch system and offer initial evidence that the benefit of sharing nodes is application dependent. The existence of beneficial and detrimental pairings becomes evident when testing the pairings in isolation. If the pairs are chosen carefully, COA can yield benefits on a batch scheduling system. The potential benefits of COA in batch systems are explored next.

ranks	STD	PCK	COA	STD	PCK	COA	STD	PCK	COA
512	63.2	56.8	53.2	42.6	37.9	32.9	62.9	59.2	51.1
1024	128.5	123.8	109.3	87.3	82.4	69.0	129.6	128.1	105.8
2048	282.5	343.5	237.6	190.0	215.3	146.7	273.7	312.1	226.3
4096	652.4	908.2	531.2	432.2	548.1	325.7	600.6	773.3	488.9
	gtc+gtc			gtc+amg			gtc+milc		
512	58.2	54.8	51.6	67.9	64.1	59.2	21.9	18.9	18.9
1024	119.8	116.9	105.4	123.6	116.5	104.9	46.2	41.1	38.6
2048	265.7	289.7	226.2	266.2	287.9	222.4	97.4	87.1	79.2
4096	535.8	723.9	458.9	660.1	785.9	539.9	212.1	188.1	163.8
	gtc+ghost			gtc+snap			amg+amg		
512	42.2	40.3	30.4	37.5	35.8	27.1	47.2	45.2	40.5
1024	88.5	86.7	63.0	78.6	75.5	56.3	82.4	75.2	67.9
2048	181.1	184.0	129.6	173.2	161.5	120.8	173.7	159.7	142.2
4096	380.5	413.3	276.3	315.6	363.8	239.1	440.0	425.9	363.0
	amg+milc			amg+ghost			amg+snap		
512	62.6	61.7	46.2	57.9	57.2	53.4	67.5	66.5	65.0
1024	130.7	132.4	92.6	120.9	121.2	111.8	124.7	120.8	113.1
2048	264.9	280.8	183.3	256.9	258.4	227.9	257.4	256.6	227.2
4096	548.9	638.5	369.1	484.0	589.0	457.4	608.4	651.1	529.3
	milc+milc			milc+ghost			milc+snap		
512	53.1	52.7	51.5	62.8	62.0	67.5	72.5	71.4	68.6
1024	111.0	110.0	107.0	114.9	109.6	114.5	118.7	109.3	106.1
2048	248.9	236.0	218.7	249.5	234.2	230.3	250.0	232.4	219.8
4096	419.2	539.5	437.2	543.5	601.6	503.4	667.9	663.7	573.7
	ghost+ghost			ghost+snap			snap+snap		

Tab. 2.2: Cumulative core hours for running a pair of applications under STD, PCK and COA techniques. Darker shades show fewer core hours required.

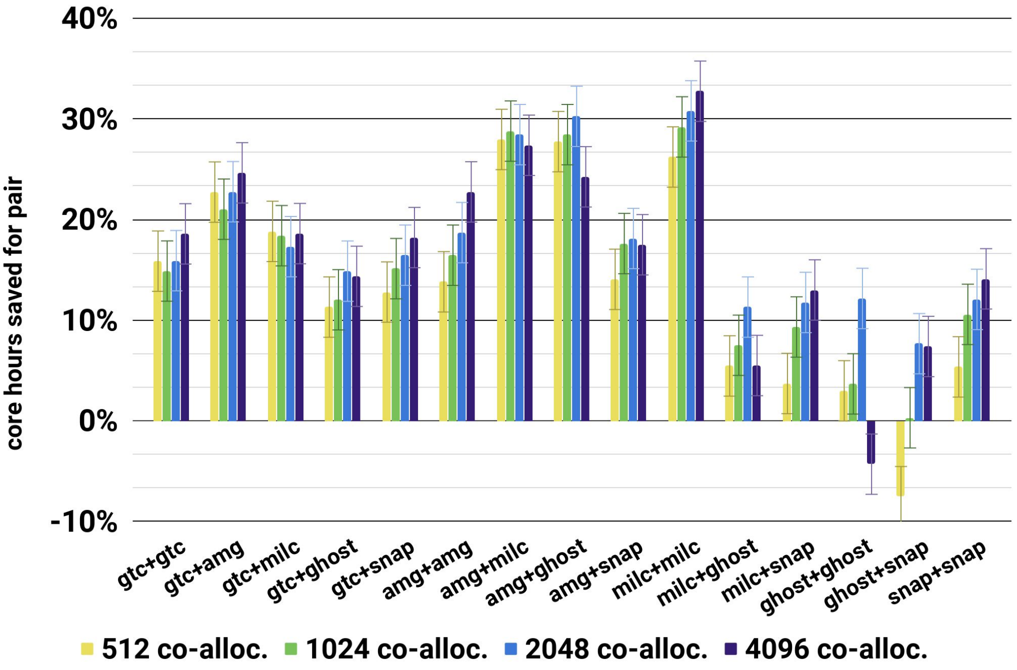

Fig. 2.7: Cumulative core hour savings of co-allocation vs. STD.

2.3.2 Batch scheduling experiments

The main hypothesis of this work is that specific co-allocation pairings of applications can benefit batch systems in terms of scheduling efficiency by reducing the required core-hours needed to solve equivalent workloads. All full batch experiments were performed four times (with queues of jobs) with the *Backfill* and *First-fit* scheduling algorithms using SLURM. Each batch experiment ran for one hour and used 512 nodes with 32 physical cores each and 16,384 cores in total. The SLURM scheduling system was allowed to schedule and run as many jobs as possible in one hour time. Furthermore, the applications were configured using the setup described in Section 2.2.4. Three batch experiments were tested, one with STD allocation, one with a naive co-allocation scheme unaware of bad pairing and a scheme not allowing twins in the co-allocation heuristic.

Figure 2.8 depicts the average utilization, computational, and scheduling efficiency for each tested scheduling algorithm and allocation method for a total of six setups. Utilization efficiency is the fraction of the available core hours used by the batch-scheduler to perform computations. The measured utilization efficiency shows no overhead required to utilize co-allocation with *Backfill* and instead shows a slight increase in utilization for co-allocation, although still within the margin of error.

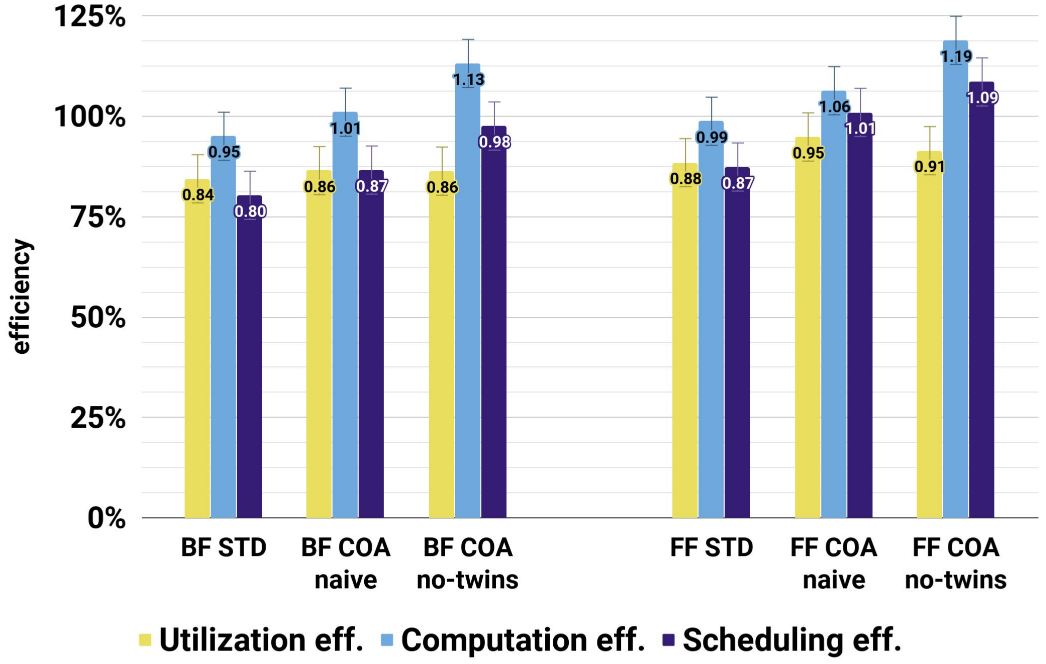

Fig. 2.8: Efficiency for *Backfill* and *First-fit* algorithms when using standard, naive and no twins.

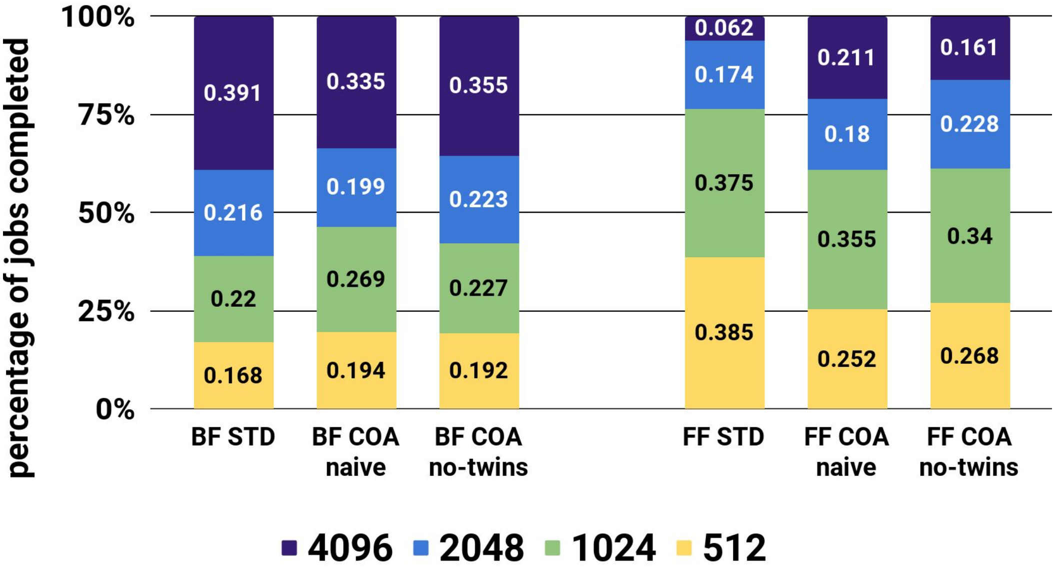

Fig. 2.9: Percentage of different ranks run by schedulers.

The utilization efficiency shows that no scheduling algorithm can achieve perfect utilization of the available core-hours within the batch experiments.

The computational efficiency compares the batch core hours used to finish all applications (not including waiting), to the expected runtime of an exclusive standard allocation in isolated experiments. The computational efficiency of co-allocation techniques is competitive with the exclusive standard methods for both *Backfill* and *First-fit*. The no-twins setups has 113% computational efficiency with *Backfill* and 119% with *First-fit*. Naive co-allocation achieves 101% and 106% for *Backfill* and *First-fit* respectively. These computational efficiencies show that COA methods can improve upon exclusive utilization of resources.

The scheduling efficiency compares the actual core hours taken to compute the queue of jobs (including waiting for resources) to the core hours it would have taken to finish the batch-algorithms specific schedule while using exclusive allocation. This scheduling efficiency also increases for the co-allocation versions. The *Backfill* schedule with no twins allowed has 98% scheduling efficiency and 109% for *First-fit* no twins techniques. The scheduling efficiency of no-twins improves slightly over the naive techniques since most twin application interfere by competing for the same resources. This shows that available resources to complete the queue of jobs are used more efficiently.

The amount of jobs each algorithm can schedule and run can change due to the algorithm and pairing restrictions. Figure 2.9 shows the amount of jobs scheduled from each size. The difference between *Backfill* and the *First-fit* can be attributed to *First-fit* algorithms having a bias to prefer smaller ranks sizes. *First-fit* often fills small amounts of free nodes with small jobs, which gives it a bias. This is a good area to explore improved algorithms combined with co-allocation to reduce biases and to further increase computational efficiency.

2.3.3 Profiling

From the previous experiments it is clear that co-allocation can improve efficiency. To understand the source of these benefits we can profile the applications. The profiling information will show which sections of the algorithms slow down the most.

In Figures 2.10 to 2.14 the average runtimes for the different sections of each application are shown. The main sections showcased are the computational workload,

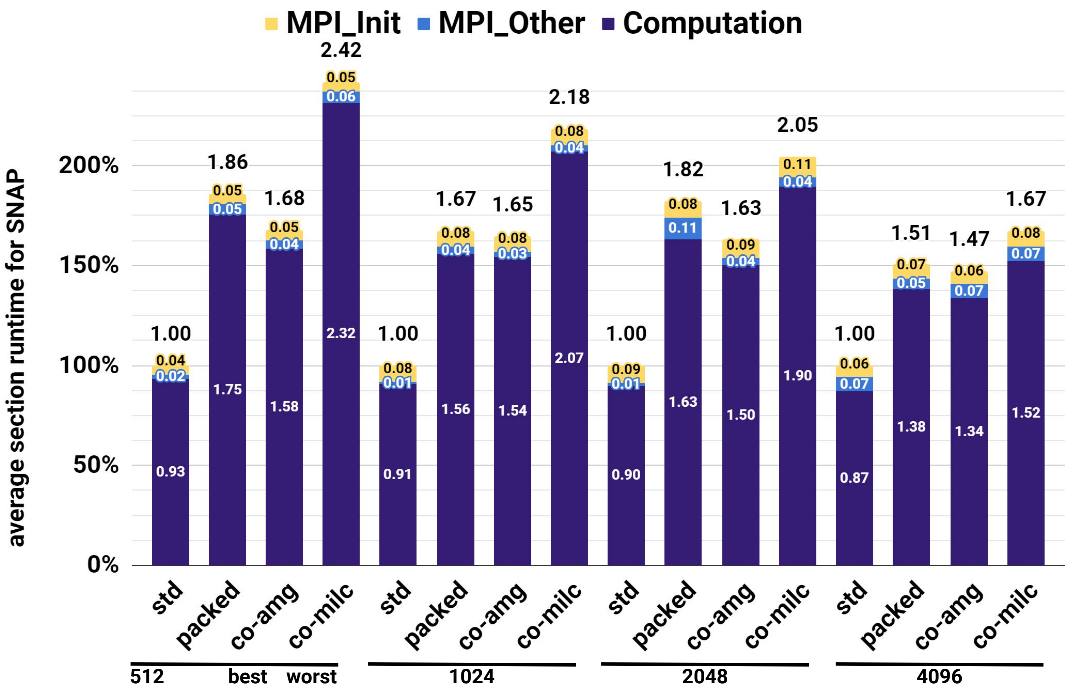

Fig. 2.10: Average runtime of SNAP functions relative to STD.

significant MPI API functions (selected based on their share of the required runtime) and other miscellaneous MPI functions.

Each application is evaluated based on the rank sizes by showing the standard allocation first, packed allocation second, best and worst runtimes of co-allocation last. The packed version is presented for comparison purposes. It oversubscribes hardware with the homogeneous workload of the application itself.

The first notable MPI function across all applications is the initialization section MPI_INIT, which is relatively short and only sees a big rise in runtime share for AMG (see Figure 2.13). This initialization should become negligible for application runtimes in the order of hours instead of minutes.

The second notable MPI function that increases its runtime when co-allocated is the global communication function MPI_Allreduce. GTC in Figure 2.11 sees a doubling of the MPI_Allreduce runtime for packed. It also increases for the best co-allocation scenario with AMG and increases even further with GHOST, which is the worst case. AMG and GHOST also see increased runtimes for their packed and co-allocated versions when compared to STD.

Computation is the third section profiled for the applications. With co-allocation the number of ranks within a node running both applications are doubled. From

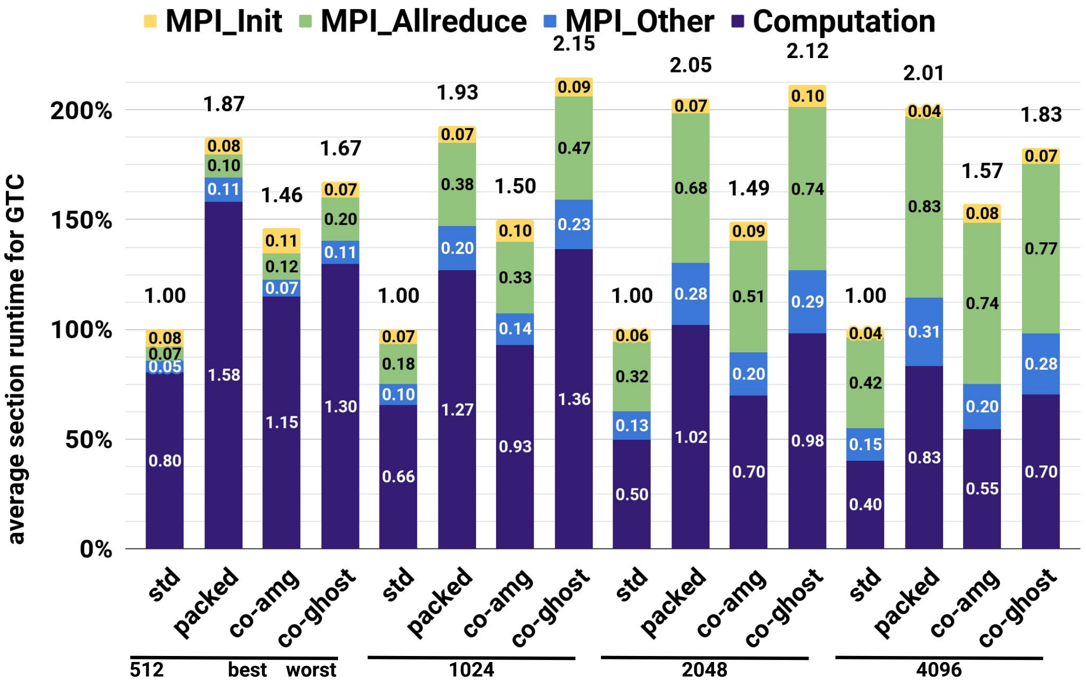

Fig. 2.11: Average runtime of GTC functions relative to STD.

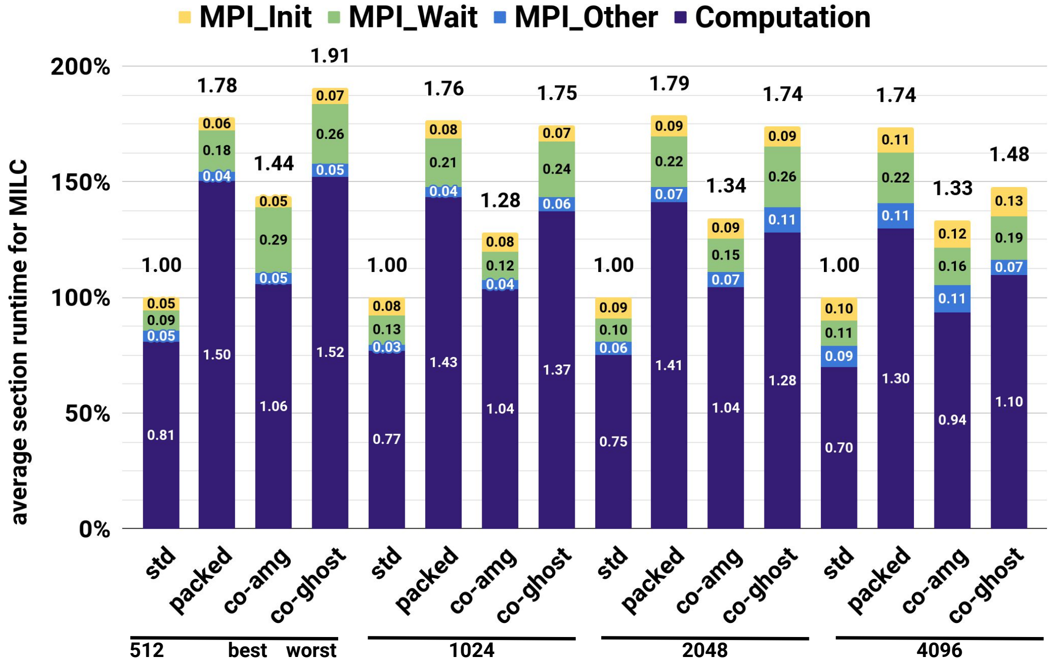

Fig. 2.12: Average runtime of MILC functions relative to STD.

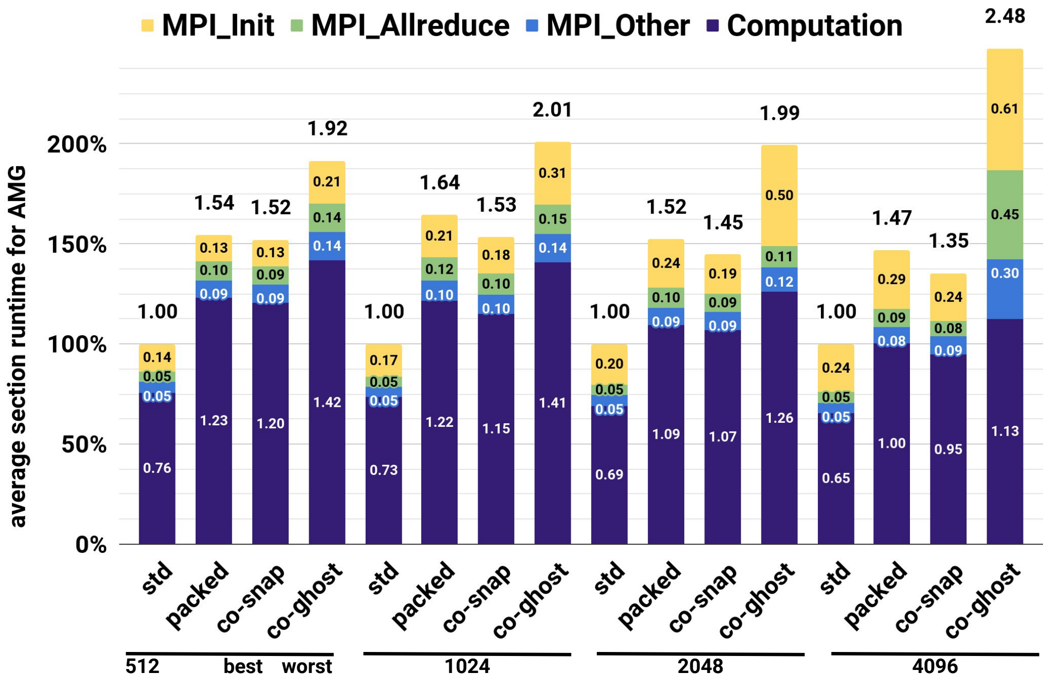

Fig. 2.13: Average runtime of AMG functions relative to STD.

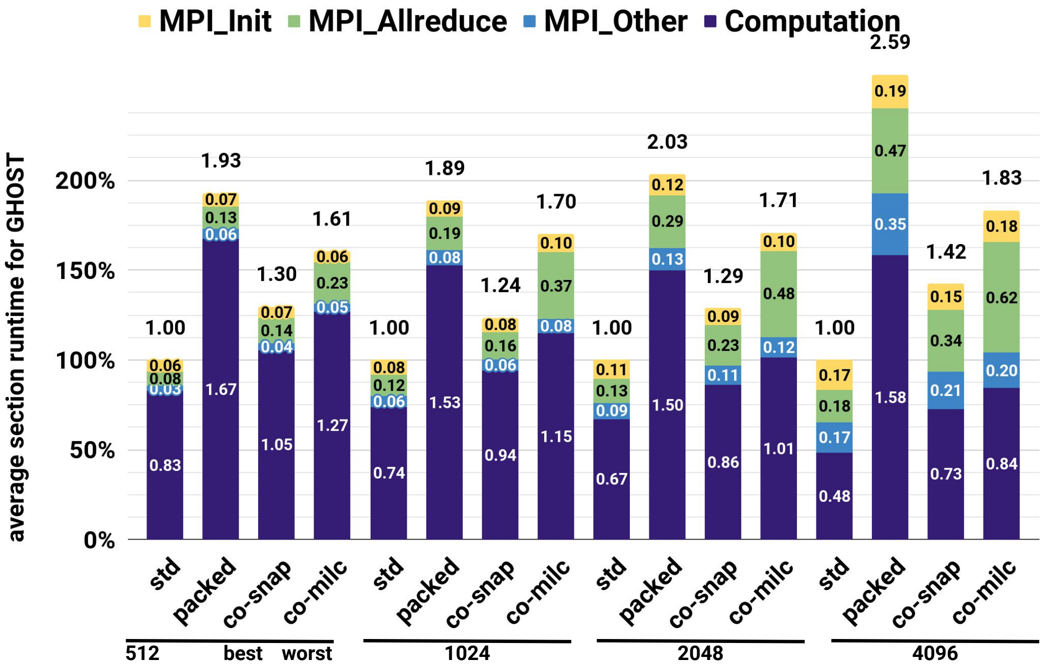

Fig. 2.14: Average runtime of GHOST functions relative to STD.

the profiling figures it can be seen that packing an application will be worse or at best similar to the best co-allocated alternative. For this reason the no-twins heuristic can be used to enforce heterogeneous computational workloads to saturate computational resources. The main takeaway is that co-allocated applications experience a slowdown by a factor smaller than two when paired with the best possible alternative partner.

2.3.4 Discussion of the results

This chapter performed parallel co-allocation experiments of MPI applications of up to 128 nodes. The experiments placed applications with equal amount of ranks on the same set of nodes and shared cores using one rank per HT. The experiments included profiling of the compute and networking sections of each application and also performed batch systems tests to show the overall HPC system throughput improvements. After investigating the benefits of co-allocation, showed that most heterogeneous co-allocation pairs used fewer resources (core hours) when compared to simple application packing or even standard allocation. Application packing was done by oversubscribing resources with more ranks from the same application and running it alone. This lead to slowdowns that were greater than simply running the application with one rank per core, without any throughput benefits. The benefits of (heterogeneous) interleaved hardware utilization discussed in Section 2.1.2 can be referenced as the source of the savings from co-allocation of distinct applications.

The experiments were done using five applications and showed that less cour-hours are needed to complete equivalent workloads when using co-allocation. More applications can give co-allocation aware schedulers more opportunities to find good pairings and also to have more options to avoid bad pairs.

Both *Backfill* and *First-fit* batch scheduling algorithms can improve upon exclusive scheduling when the scheduler is allowed to use co-allocation. Both were setup to use a simple naive (all pairs allowed) and a no-twins co-allocation heuristic that prohibits the co-allocation of two applications of the same type. The amount of resources needed to complete application runs is reduced when choosing good co-allocated pairs, which in turn benefits scheduling algorithms by using less resources to solve queues of jobs.

It was also described that the *First-fit* scheduler performed better than *Backfill* in terms of scheduling efficiency when the job queues are smaller and less varied. Nevertheless, the testing scenario accounted for this bias with a huge queue of jobs so that *Backfill* could be able to find a job to perform the backfilling tasks.

Overall the benefits of co-allocation within both scheduling algorithms were evident and more pronounced when only good pairs were allowed. It is possible that more sophisticated heuristics (e.g. being based on resource usage characterizations) can be implemented within co-allocation algorithms of batch schedulers to make better pairing choices on the fly. Matrices of good and bad pairs could also be dynamically generated during normal scheduling operation by allowing applications to share nodes without limits. After multiple runs, the matrix could be updated with the latest metrics recording core-hours needed to solve the problems.

3

Failure probability aware checkpoint intervals

The mitigation of hardware related failures is becoming more important as the failure rates and amount of components in big HPC systems increases. High component counts increase the risk of at least one hardware element failing, which can translate into software failures for HPC applications. HPC users have the opportunity to use checkpoints to save the state of their applications in case a failure and crash occurs. The decision of when to create a checkpoint is also critical for users who have quotas that restrict the amount of resources they can use in HPC systems. Creating a checkpoint is not free, since it usually requires pausing computations and spends time creating a save state. Too many indiscriminate checkpoints would require the allocated compute time to be used in creating them, instead of doing actual computations. This indiscriminate checkpoint time is wasted when no failures happen at all, and when failures do happen, restarting could be cheaper than doing too many checkpoints.

It is possible to perform checkpoints in static intervals to have save points at various points in time. This has the added benefit of reducing the average cost of checkpointing when considering lost computations and restarting. To benefit from interval based checkpoints and reduce cost, the interval has to be chosen optimally to not outweigh the expected costs of lost computations and restarts. The traditional checkpoint intervals that have been defined in literature are meant to be used for extremely long running applications in systems where failures are highly probable. In practice this assumption rarely holds since applications have shorter runtimes and fewer nodes than would be necessary to increase the probability of failures occurring. By accounting for job runtimes and node counts it is possible to optimize checkpoint intervals and reduce resource waste associated with checkpointing overheads.

To have an example overview of the type of HPC jobs that make up batch queues we showcase here the system statistics of the MOGON II HPC system at the Johannes Gutenberg University Mainz. The jobs in these batch systems could benefit from custom checkpoint intervals not defined in previous works. Over a period of two years we analyzed four million user jobs and was able to determine that ∼1.59% of submitted jobs crashed from non user related hardware or system software failures.

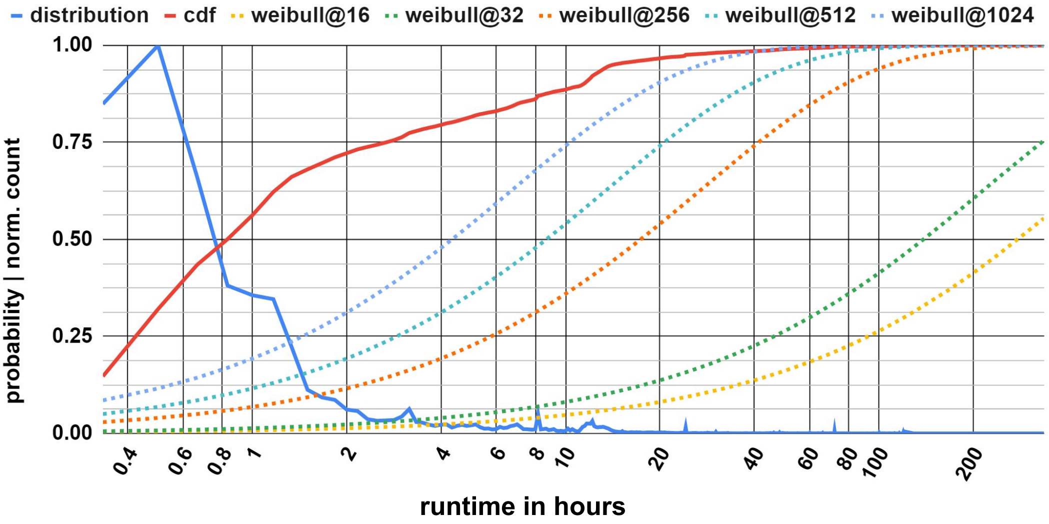

Fig. 3.1: Distribution of the runtime from user jobs within the MOGON II HPC system (blue). The red curve is the cumulative distribution function of runtimes and the dotted curves the failure probability distributions for five node counts.

Furthermore, we were able to determine the amount of resources used by the HPC jobs. Figure 3.1 shows a normalized distribution (count of jobs) of runtimes for jobs in the Mogon II system. The Figure also shows the equivalent cumulative distribution function (CDF) for the runtimes and an extra five Weibull failure probabilities (plotted to their own probability Y axis) for jobs with node counts up to 1,024. The dotted curves are the failure probabilities for the jobs with shown runtimes (X axis) and 16, 64, 256, 512 and 1,024 nodes. The shown failure probability of each job curve (dotted) depends on the runtime, node count and the MTBF of the system. Note how the probability of failure increases with the number of nodes used by a job so that a job with 512 nodes and 8 hours of runtime can have a failure probability of 50%, while the same job runtime for 1,024 nodes has a 65% failure probability. It is noteworthy that most jobs are shorter (3.91h) than the system MTBF of 5.71h.

Figure 3.2 shows how the job node size distribution of the Mogon II system looks like. Similar failure probability curves are shown as before but based on runtimes of 1, 5, 10, and 50 hours. It is again noteworthy that the amount of nodes used by jobs is commonly small and that they are mostly in powers of two. This reinforces this observation since it shows that at least 50% of the job node counts are smaller than 26 nodes, 75% smaller than 54 nodes and 90% smaller than 104 nodes. A job has to use at least 74 nodes for a runtime of over 50 hours for its failure probability

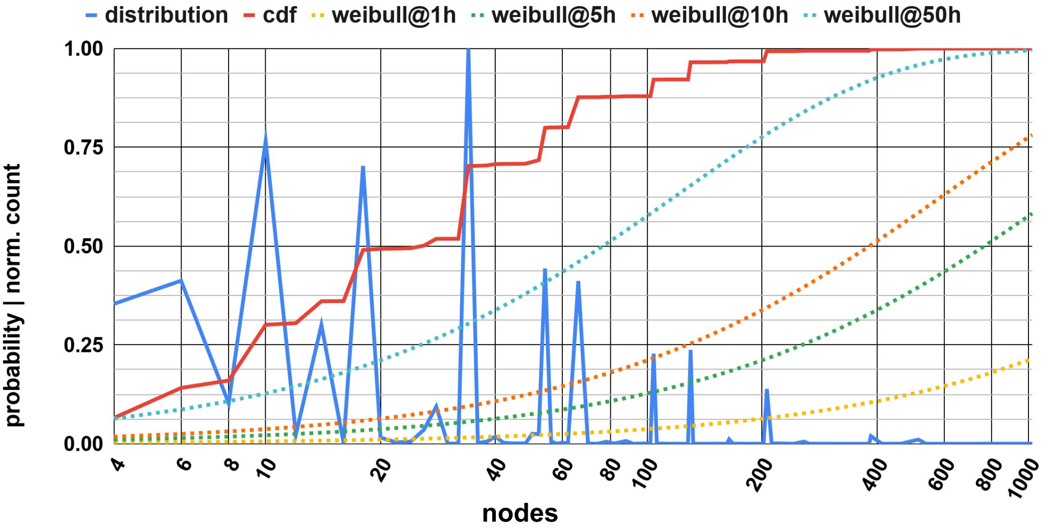

Fig. 3.2: Node distribution of user jobs within the Mogon II HPC system (blue). The corresponding red curve indicates the CDF and the dotted curves the failure probability distributions.

to reach coin toss odds of 50%. Shorter jobs running for 1 hour would need to use 3,826 nodes to fail with a probability of 50%. This data set shows that many jobs have probabilities of failure, which are quite low.

Browne *et al.* [22] found within the *Lonestar4* and *Ranger* systems, that their node and runtime distributions are also skewed to lower runtimes and node sizes. This characteristic is common and Amvrosiadis *et al.* [72] reported that in the LANL Trinity and LANL Mustang traces 80% of the jobs lasted less than 3 hours for the Mustang and 6 hours for the Trinity system. The LANL traces show that MPF jobs are also very common and that around 90% of their jobs use less than 128 nodes.

Checkpoint intervals calculated using the assumption that jobs have high failure probabilities can cause significant resource waste. The jobs from various HPC systems listed previously have HPC jobs that do not conform with the high failure probability assumption. The checkpoint intervals can be improved by considering the individual failure probability of a job by accounting for the number of nodes used and expected runtime. The job failure aware checkpoint method proposed further in this chapter will use a novel cost function that is independent of the failure distribution of HPC systems. This aware method weighs the cost of checkpoints with the generalized failure probability and gives two examples using Weibull and Poisson distributed failures. The method we propose enables users of common medium jobs to reduce the average overhead of checkpointing and therefore lower the number of required core hours to compute workloads.

3.1 Related work

The main resilience measure used in HPC systems is checkpointing. Checkpointing regularly saves the state of an application with a checkpoint to restart computations after a system failure. To minimize the overhead of continuous checkpointing, the choice of interval between checkpoints cannot be left to arbitrary values.

The main theoretical framework to model the overhead of checkpoint and restart methods and to determine the optimal interval was developed by Young [31] and Daly [32].

Young developed a first order approximation for optimal checkpoints for long running applications with exponential failure distributions with high failure probabilities. His proposed optimal interval is fixed and equals $\sqrt{2Mt_c}$ where M is the mean time between failures and t_c the checkpoint cost. Daly continued Young's work by taking into account restart costs, leading to the higher order approximation of the same fixed interval for exponential failures $(\sqrt{2Mt_c} + t_c)$. His derived interval showed that the restart times did not contribute to the interval determination, even after considering them in the derivation. We expand on previous works based on interval based checkpoints by considering jobs with low probabilities of failure. The works derived from the Daly / Young failure model also fail to consider jobs with lower failure probabilities. These types of jobs tend to have either shorter runtimes, less nodes or run on very stable systems with low failure probabilities.

Bouguerra *et al.* proposed such a method that searches for an interval that minimizes the average completion time of jobs in systems with Weibull failure distributions[73]. The authors also showed that the traditional periodic checkpointing strategies are optimal when all nodes rejuvenate to failure free states with no previous history of failures. This is often referred to as rejuvenation and is a very restrictive requirement in practice. For rejuvenation to apply in practice, an HPC system would need new nodes with no previous history of failures that might affect future behavior. The work presented here does not model its failures using component rejuvenation and instead assumes that available nodes within the HPC system will be used to run re-queued jobs after a failure. The approach of this thesis is also compatible with Weibull distributions and differs from the work by Bouguerra by searching through intervals to minimize checkpointing costs.

Bougeret *et al.* proposed a heuristic called DPNextFailure to determine checkpoint intervals in multi-node systems. The heuristic maximizes the work done before a failure instead of minimizing waste or makespan [74]. Their method is very computationally intensive because it recursively calculates probabilities for all possible intervals. Their approach does consider failure probabilities, but does not account for the total runtime of an application in the probability calculations. It instead calculates failure probabilities for intervals, making it inaccurate for MPF jobs. The recursive algorithm also makes their method very computationally expensive with cubic complexity. Although their results are suitable for their modeling of rejuvenated failures, their approach lacks the possibility to accurately determine checkpoints for jobs with lower failure probabilities.

Jin *et al.* [75] expanded Young's interval approximation by considering multiple nodes and spare nodes. Jin arrived at a first order approximation that matches Young's when the recovery cost is zero. As previously mentioned, the methods shown in this thesis relies on existing nodes within a batch system and does not use new spare idle nodes. Jin's work also does not account for jobs with lower failure probabilities.

Subasi *et al.* [76] proposed several analytical formulations for arbitrary failure distributions making their calculated intervals usable for many failure characteristics. Their method is however still inadequate for jobs with lower probabilities of failures. Our approach also models arbitrary failure distributions, as long as density functions and truncated moment formulas are available.

Bessho *et al.* [77] used in their intervals a scaled MTBF for multi-node systems, where only the amount of nodes that the application uses were considered. This aspect is part of the method presented in this thesis. We perform simulations of batch systems by scaling the MTBF according to the node count of used by jobs and by treating the failures of distinct nodes as independent. This assumption was explored and validated by Aupy *et al.* [78] and is reasonable for failures of individual components. Failures that cause multiple components to fail, i.e: network switch failure, would have to be accounted for in the modeling of the failure distribution. This task is difficult and requires empirical data of how correlated failures behave. This scenario is therefore not considered in this work.

Herault *et al.* [79] explored how checkpointing is influenced by I/O resource contention. Their approach is similar to the interval approximation of Daly but expands it by including modeling of I/O utilization by applications. Our approach requires the I/O overhead to be modeled within the cost of checkpointing.

Tiwari *et al.* [80] explored a fixed interval method that takes into account an estimate for the average fraction of lost work in their checkpoint formula. Our method also considers this lost work after a failure but unlike Tiwari, we do not use a heuristic estimate. We instead calculate accurate statistical values using truncated moments of the failure distributions from the HPC system. The use of truncated moments is one of the main differences and contributions that the method presented here contributes to previous works.

El-Sayed *et al.* [81] concluded that for their LANL systems the extra overhead of variable checkpoint intervals was not justified. In accordance with this assessment we focus on fixed checkpoint intervals for applications. Nevertheless, the checkpointing interval computed for a specific job can be updated using new residual runtime left. This is also possible after restarting the job from a failure, because our approach can compute an interval for the reminder runtime of the job.

Jayasekara *et al.* [82] included the time needed to detect a failure in their interval determination method. They managed to solve optimality using a Lambert function and by not assuming that all single node failures are fatal. This assumption is not shared by most HPC checkpointing approaches. The method we present here considers the first encountered failure to be fatal, since protecting against fatal errors is the main goal of using a checkpoint and restart method.

Gupta *et al.* [83] showed that some failures can be correlated based on physical proximity and that these types of errors need to be modeled differently to account for failures occurring on multiple components. The method we present later can support such locality of failures when they are modeled within the failure distribution. This is however not trivial, as it requires empirical data to model the distribution correctly. We also require the failure distribution to model the whole HPC system and not just sub-components like a node or disks. The failure distributions are therefore preferably derived from real logs or other types of empirical data.

Finally the model used for the interval method we propose does not need to consider multiple failures fatally interrupting an application. This follows from how parallel MPI applications behave in real systems when they fail. A parallel application with intercommunicating processes fails catastrophically upon encountering the first failure and any subsequent failures of the used components affect unused nodes because the application already failed or affects nodes being used on a new scheduled application. Furthermore, there is no need to consider any sequence of non-fatal failures that do not invalidate computations. It is therefore our stance that multiple failures do not change interval determination before encountering the first fatal failure.

3.2 Problem Formulation

The main goal of this checkpoint aware chapter is to optimize the checkpoint interval length τ for each HPC job by taking into account their individual failure probabilities. Optimal checkpoints need to statistically balance the checkpoint creation overhead and the wasted computations caused by a failure after the last checkpoint.

Figure 3.3 depicts the runtime of an application, the checkpoints being performed at intervals and a failure crashing the execution. The value t_c is the time needed to create a checkpoint every τ time units. For an estimated base runtime of t_b of the job as given by the user, the total runtime of the job becomes longer due to the amount of performed checkpoints. The final runtime is then $t = t_b + nt_c$, where n checkpoints are performed and no failure occurs. When a failure happens at t_f, the computations w after the last checkpoint are wasted and the application needs to be reloaded for a cost l.

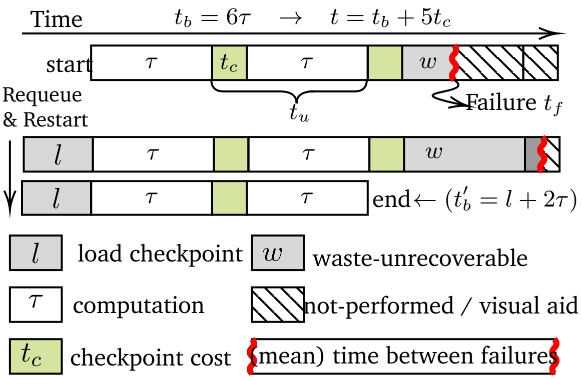

Fig. 3.3: Diagram showing the cyclical processes of periodic checkpoints between computation phases. In the event of a failure the application is re-queued and restarted by loading the checkpoint.

To optimize the checkpoint interval τ the following Assumption about the runtime of applications 1 needs to be made:

Assumption 1. *The base runtime of an application running without checkpoints can be estimated and provided by the user or some other predictive system.*

Users already provide an upper bound for the submitted job's runtime to the batch system making this requirement possible to fulfill. The accuracy of a users estimate could be further improved using past history of successfully completed instances.

Furthermore, a random variable $\mathbb{F}$ can be used to model the occurrence of failures within an HPC system so that the realization of $\mathbb{F}$ can determine a failure time $0 \leq t_f$

of when a failure occurs. It becomes possible to establish whether a job fails for $t_f < t$, otherwise the job succeeds.

Equation 3.1 shows the expected average cost $\hat{S}$ of performing checkpoints for failure times at t_f. The cost $\hat{S}$ can then be simulated by performing periodic checkpoints and F failure events for a job runtime t and interval τ. Any runtime within $\hat{S}$ is considered a cost because it is not used for the original computations of the application. The first term (3.1a) of Equation 3.1 is the total cost of performing checkpoints $\lfloor \min(t, t_f)/(\tau + t_c) \rfloor$ in each simulated event. This cost is always paid to checkpoint a job even when the job does not fail.

$$\hat{S} = \frac{1}{F} \sum_{f=1}^{F} t_c \lfloor \min(t, t_f)/(\tau + t_c) \rfloor \tag{3.1a}$$

$$+ t_f \bmod (\tau + t_c) \quad \text{if } t_f < t. \tag{3.1b}$$

The term 3.1b continues by adding the wasted time that was not covered by a checkpoint. This time is the runtime after the last successful checkpoint ($t_f \bmod (\tau + t_c)$) in case a failure occurs at time $t_f < t$. The cost function $\hat{S}$ is discontinuous due to the floor functions that models the actual number of checkpoints that occur at lower failure probabilities.

In the experimental Section 3.4 we will explore how Equation 3.1 converges to the traditional checkpoint costs when the failure event $t_f < t$ occurs with high probability. It will also be shown how the optimal interval developed in Section 3.3 expands the traditional methods with the case of $t_f > t$ for jobs with medium probability of failure (MPF jobs). Additionally we will showcase how the stepped effects of the floor function are smoothed out at failure probabilities close to one.

The cost function $\hat{S}$ can work with arbitrary failure distributions $\mathbb{F}$ if the truncated moments (cf. Eq. (3.12)) exist and if the failure assumptions hold. We will also provide working examples with Weibull failure distributions for their flexibility [84] and Poisson for its use in previous work [31, 32].

HPC jobs are submitted to a batch system that queues the jobs based on priorities and then proceeds to wait for resources before executing said user programs. When a job fails with a valid checkpoint available, it can be re-queued into the batch system and eventually restart as a new job with a new base runtime t_b' and with nodes from within the already existing HPC system. Re-queued jobs that restart can be checkpointed since they will be susceptible to failures again. In the modeling and simulations we ignore the waiting/queuing time of jobs within the batch system, because waiting consumes no system resources. This waiting time is traditionally also not taken into account by any other previous work. For the additional time needed to restart from the last checkpoint, the following Assumption 2 is made in accordance with previous works that show no contribution from the restart time [32, 79, 80, 75, 82]:

Assumption 2. *The restart time for a failed and re-queued job is known and is part of the new base runtime* t'_b.

A re-queued job is considered a new job within batch systems, which allows for a new checkpoint interval τ' to be computed for the new base runtime t'_b when it restarts. Note that all meaningful failures are considered catastrophic.

The following Assumption 3 accounts for failures being related through spatial locality and for multiple failures affecting the same application. It also requires that the failures of a system are modeled as a whole for the HPC system instead of modeling failures per node (component rejuvenation).

Assumption 3. *An application fails catastrophically upon encountering the first node failure that stops computations. Multiple correlated failures can affect newly re-queued jobs and should be modeled within the failure distributions of the whole system.*

In the next section we will derive the mathematical model of the problem formulation described previously. We will then show in Section 3.4 that the MPF aware checkpoint intervals offer lower expected checkpointing costs for MPF jobs compared to other methods.

3.3 Checkpoint Algorithm

Determining an analytical solution of an optimal checkpoint interval length based on Equation 3.1 is difficult because of the floor functions present. Another issue with a fixed analytical formulation is that it would be tied to a single failure distribution and would not permit to model arbitrary failure characteristics. Instead of an analytical solution for a specific failure distribution, we will provide an iterative algorithm that can evaluate which values of τ offer a minimal cost given an expected application failure probability. The iterative algorithm will not necessarily determine the optimal analytical value of τ in a mathematical sense, but will instead provide fast computation method accurate in practice that can be implemented in production.

3.3.1 Cost function

It is important to combine the job and system characteristics needed to model checkpointing as a vector s:

$$s = (t_c, \tau, M, t, t_f),$$

Within vector s the time t_c is the cost required to perform a single checkpoint, τ is the computing time between two checkpoints (end of a checkpoint and start of another), M is the MTBF of the whole HPC system (related to the compute nodes), t the runtime of the application to be checkpointed and t_f is the time at which a failure happens.

Previous works that compute optimal checkpoint intervals perform the assumption that jobs encounter failures almost certainly. This translates into high failure probabilities due to long running applications compared to the MTBF of the underlying system. The approach presented here removes this limitation by considering the failure probability of each job individually. The approach also weighs the costs related to the job failing or succeeding with each respective probability of occurring. Considering a single job with total runtime t and a possible failure occurring at t_f, the following two part cost function can be defined:

$$C(t_f, t) := \begin{cases} C_{\text{ok}} & \text{if } t_f > t \\ C_{\text{fail}} & \text{if } 0 \leq t_f \leq t \end{cases} \tag{3.2}$$

Furthermore, the individual costs of each possible event can be defined as:

$$C_{\text{ok}} := \left\lfloor \frac{t}{\tau + t_c} \right\rfloor t_c, \tag{3.3a}$$

$$C_{\text{fail}} := t_f - \left\lfloor \frac{t_f}{\tau + t_c} \right\rfloor \tau. \tag{3.3b}$$

The cost function presented here accounts for the cost that needs to be paid if the job terminates successfully, plus the wasted cost in case of a failure. The sub-cost function C_{ok} accumulates the time spent creating the checkpoints, which is considered a cost and is independent of the failure time t_f. The sub-cost function C_{fail} represents the traditional checkpointing cost of previous works. The function calculates the difference of the time t_f at which the failure occurs and the successfully completed full computation intervals $t_u = \tau + t_c$ until t_f. Anything left from the subtraction is the cost of performing checkpoints plus the wasted computations that were not saved after the last checkpoint. In this failure case the computation progress is accounted for by the computation chunk τ and the rest is wasted work that constitutes a cost. If no checkpoints would be performed, all computations would be wasted in the case of a failure. In both scenarios the total time $t = t_b + nt_c$ depends on the value of the time interval τ between two checkpoints.

Both cases in 3.2 use floor functions to determine checkpoint counts, which makes this approach different from previous works. The floor function computes the number of checkpoints by producing a whole number (*e.g.* $4.35 \mapsto 4$) that accounts for only the cost of checkpoints that are actually performed. The main drawback is that the floor function produces a non-continuous cost function for MPF jobs For traditional jobs where the probability of failure approaches 1 the cost function retains relative smoothness. This will be shown experimentally later.

In summary, the cost function C consists of two cases that account for the cost of checkpointing and wasted computations when failures occur. Restart and resume costs are conceptually modeled within batch system simulations and the job requeueing after catastrophic failures.

Substituting the random variable $\mathbb{F}$ that models the occurrence of failures (see examples in Section 3.3.3), into the cost function (3.2) yields the new random variable $C(\mathbb{F}, t)$ with an expected value

$$E_C := E[C(\mathbb{F}, t)] \tag{3.4}$$

that represents *the expected cost checkpointing a job*. This expected cost in Equation (3.4) is a function of τ, where the (mathematical) optimal value of τ is the minimum of the expected cost (3.4). Determining the optimal τ with their own smooth cost functions was the approach taken by Young/Daly and Bougeret.

By definition, the expected cost E_C (3.4) may be written as

$$\begin{aligned}
E_C &= \int_0^t C_{\text{fail}}(x,t)p(x)\,\mathrm{d}x + \int_t^\infty C_{\text{ok}}(x,t)p(x)\,\mathrm{d}x && \text{(3.5a)}\\
&= \int_0^t C_{\text{fail}}(x,t)p(x)\,\mathrm{d}x + \int_t^\infty \left\lfloor \frac{t}{\tau + t_c} \right\rfloor t_c p(x)\,\mathrm{d}x && \text{(3.5b)}\\
&= \int_0^t C_{\text{fail}}(x,t)p(x)\,\mathrm{d}x + \left\lfloor \frac{t}{\tau + t_c} \right\rfloor t_c \int_t^\infty p(x)\,\mathrm{d}x && \text{(3.5c)}\\
&= \int_0^t C_{\text{fail}}(x,t)p(x)\,\mathrm{d}x + \left\lfloor \frac{t}{\tau + t_c} \right\rfloor t_c(1 - P(t)) && \text{(3.5d)}
\end{aligned}$$

where the Cumulative distribution function (CDF) $P(t) = \int_0^t p(x)\,\mathrm{d}x$ represents the probability of a failure occurring before time t. The fail part in (3.5) can be expanded using the definition in 3.3b:

$$
\begin{aligned}
&\int_0^t C_{\text{fail}}(x,t)p(x)\,\mathrm{d}x \qquad\qquad ; x = t_f && \text{(3.6a)}\\
&= \int_0^t \left(x - \left\lfloor \frac{x}{\tau + t_c} \right\rfloor \tau \right) p(x)\,\mathrm{d}x && \text{(3.6b)}\\
&= \int_0^t xp(x)\,\mathrm{d}x - \tau \int_0^t \left\lfloor \frac{x}{\tau + t_c} \right\rfloor p(x)\,\mathrm{d}x && \text{(3.6c)}\\
&= \mu^t - \tau \hat{n}_f && \text{(3.6d)}
\end{aligned}
$$

Here $\mu^t = \int_0^t xp(x)\,\mathrm{d}x$ is the first right truncated moment of the failure distribution. The first right truncated moment is also known as the partial average of the distribution from $\mathbb{F}$ [85]. Using the truncated moment enables the modeling of average runtimes before a failure for MPF jobs and is new among checkpoint determination techniques. In Equation (3.6c) τ can be taken out of the integral since it is independent of t, leading to a factor that represents the mean amount of checkpoints in case of failures $\hat{n}_f$. Equation (3.6c) may be interpreted as the mean runtime of applications that fail, minus the mean time spent doing actual computations. Effectively, this term represents the expected time spent doing checkpoints plus the wasted computations that need to be redone upon a failure.

The expected number of checkpoints $\hat{n}_f$ before a failure at t_f in (3.6d) can be numerically calculated using a finite sum based on t:

$$
\hat{n}_f = \sum_{i=0}^{t/t_u} i \cdot [\, P((i+1)t_u) - P(it_u)\,]\,, \tag{3.7}
$$

This form of $\hat{n}_f$ can be obtained by splitting up the second integral from Equation (3.6c) into intervals where the integrand is smooth. Within the sum, each count i is weighted by the probability of a failure occurring in the particular interval. To our knowledge, this is also the first formulation of it.

The expected costs E_C can therefore be summarized in Equation (3.8) as the sum of the partial costs for failures (3.8a) and successes (3.8b):

$$
\begin{aligned}
E_C = \quad & \mu^t - \tau \sum_{i=0}^{t/t_u} i \cdot [\, P((i+1)t_u) - P(it_u)\,] && \text{(3.8a)} \\
& + \left\lfloor \frac{t}{\tau + t_c} \right\rfloor t_c(1 - P(t)) && \text{(3.8b)}
\end{aligned}
$$

The final Equation (3.8) is used for Algorithm 1 to find the value of t_u that minimizes it.

3.3.2 Iterative Algorithm

Algorithm 1: Compute t_u for which E_C is minimal.

```
Data: system $s$ with input $t_b$.
Result: $t_u$ with minimum expected cost.
$\mathbb{U} = [\frac{1}{60}, \frac{2}{60}, \frac{3}{60} \dots, t - t_c]$ ;  $t_u \in \mathbb{U}$ in hours
$c_{\min} = maxflag$ ;
$u_{\min} = undefined$ ;
foreach $t_u$ in $\mathbb{U}$ do
    $c = E_C(t_b, t_u)$;  Eq.  3.8
    if $c < c_{\min}$ then
        $c_{\min} = c$;
        $u_{\min} = t_u$;
return $u_{\min}$
```

Algorithm 1 uses Equation (3.8) to search for an approximated optimal checkpointing interval τ. It iterates over the set $\mathbb{U}$ of t_u choices for $t_u = \tau + t_c$ with a resolution of one minute and then computes the associated expected cost using Equation 3.8. Finally it keeps track of the minimum cost to select the value of t_u, which maintains the waste at a minimum.

3.3.3 Poisson and Weibull examples

The random variable $\mathbb{F}$ is usually modeled with a Poisson and Weibull failure distribution [32, 83, 86, 84]. These two distributions are shown here as examples, but Equation 3.8 and Algorithm 1 is derived independently of them. Algorithm 1 requires the properties of $\mathbb{F}$ to be known and available, which for Weibull includes the scaling factor w. The probability density function of an example random variable $\mathbb{F}$ that models both Weibull and Poisson is

$$p(x) = \frac{w}{\lambda_w}\left(\frac{x}{\lambda_w}\right)^{w-1} e^{-(\frac{x}{\lambda_w})^w} \qquad \text{for } x \geq 0 \tag{3.9}$$

with the parameter

$$\lambda_w = \frac{M}{\Gamma(1+\frac{1}{w})} \tag{3.10}$$

where $w = 1$ gives the Poisson distribution and $w \neq 1$ the Weibull distribution. Each of these distributions also has a first right truncated moment μ^t (the incomplete mean) as given by, e.g. [87],

$$\mu^t = \int_0^t xp(x)\,\mathrm{d}x \tag{3.11}$$

$$= \begin{cases} M\gamma\left(\frac{1}{w}+1, \left(\frac{t\Gamma(\frac{1}{w}+1)}{M}\right)^w\right) & \text{if Weibull} \\ \\ M(1-(\frac{t}{M}+1))\exp(-\frac{t}{M}) & \text{if Poisson} \end{cases} \tag{3.12}$$

where $\Gamma(\cdot)$ is the one parameter gamma function, $\gamma(\cdot,\cdot)$ is the two parameter gamma function and w the Weibull factor that models Weibull failure distributions [85]. The use of truncated moments for checkpoint determination allows for the modeling of relevant parameters to be done numerically.

Finally the failure probability until time t is

$$P(t) = \begin{cases} 1-\exp\left(-\left(\frac{t\Gamma(1+\frac{1}{w})}{M}\right)^w\right) & \text{if Weibull} \\ 1-\exp(-\frac{t}{M}) & \text{if Poisson} \end{cases} \tag{3.13}$$

3.3.4 Usability

The statistics mined from the MOGON II HPC system show that MPF jobs are common. Furthermore, using Equation 3.8 and Algorithm 1 system administrators could deploy an implementation of an FPS for their users. Such a program can provide the best aware checkpointing interval to applications. Users need to only provide the expected runtime and node count of their applications. These parameters are already present in batch system submission scripts. The program can then compute an interval for the specific runtime of the application allowing the user to tune their checkpointing method. The result is less expected costs from using interval based checkpoints.

3.4 Simulation and Results

This section shows the performed simulations of HPC jobs to compare the proposed MPF aware algorithm to multiple traditional algorithms, specifically: Daly's (2006) second order estimate [32], Bouguerra's (2009) Weibull approach [73], Subasi's (2017) general method [76], Jayasekara's (2019) method considering failure detection costs [82] and Tiwari's (2014) waste-aware (OCI) approach [80].

In the simulations there is a brief comparison to Bougeret's (2011) heuristic DP-NextFailure (DPNF) [74]. This Heuristic is computationally expensive due to its $\mathcal{O}(t^3)$ complexity. Additionally the DPNF method does not provide significant improvements in MPF scenarios compared to the other five methods. The checkpoint intervals based on Algorithm 1 are called "**aware intervals**" because they account for the job failure probability.

Algorithm 1 was implemented single-threaded with Python 2.7 using the NumPy 1.16 and SciPy 1.2 libraries and an Intel Xeon E5-2650 CPU @ 2.00GHz. The implementation of Equation 3.8 can evaluates 25,000 costs per second and Algorithm 1 can compute at least 400 aware intervals per second.

The simulations shown here will model the failure characteristics on an HPC system using a synthetic Weibull failure distribution with a weight factor of 0.8. Weibull is a good fit for the LANL ATLAS system failures as shown by Yuan *et al.* [88]. From now on we will use mainly the Weibull distribution, but will also provide summarized results with the Poison distribution.

The MTBF used in the failure distribution represents an entire HPC system. For a job using partial node counts the MTBF of the system is then scaled to the node count of each job through

$$\mathrm{MTBF}_{\mathrm{job}} = \mathrm{MTBF}_{\mathrm{machine}} \cdot \frac{N_{\mathrm{machine}}}{N_{\mathrm{job}}} \tag{3.14}$$

where N_{machine} and N_{job} denote the node number of the whole machine and the job, respectively. This scaling of MTBF is consistent with modeling in previous works [83, 79]. For systems where this cannot apply, the MTBF of the specific nodes would have to be determined differently. This nevertheless, does not invalidate the use of the adapted MTBF's within the checkpointing algorithms.

A total of $F = 10,000$ failure events were sampled from the failure distributions and used in a simple monte-carlo simulation to calculate the average cost $\hat{S}$ from Equation 3.1. The ratio of successes and failures to the total F coincides with the failure probability of the job. The average simulated checkpoint cost ($\hat{S}$) from Equation 3.1 includes the cost of checkpointing and the cost of computations lost from failures. This cost is used to compare the various methods.

3.4.1 Interval and cost function simulation

The first set of experiments compare all algorithms against the simulated average checkpoint costs of Equation 3.1. It uses F failure events at time t_f.

Figures 3.4 to Figure 3.6 show the simulated (blue) and the estimated (dotted red) costs (using Equation 3.8) of four distinct application runtimes t 18.99h, 116.54h, 6.59h, 22.51h as a function of the checkpoint interval τ (x axis). The τ based suggested interval value of each algorithm is marked in the full final cost for the simulation (blue). All panels shown run on a system with a MTBF of 24 hours and with a checkpoint cost of half an hour. Further configurations are shown in later experiments.

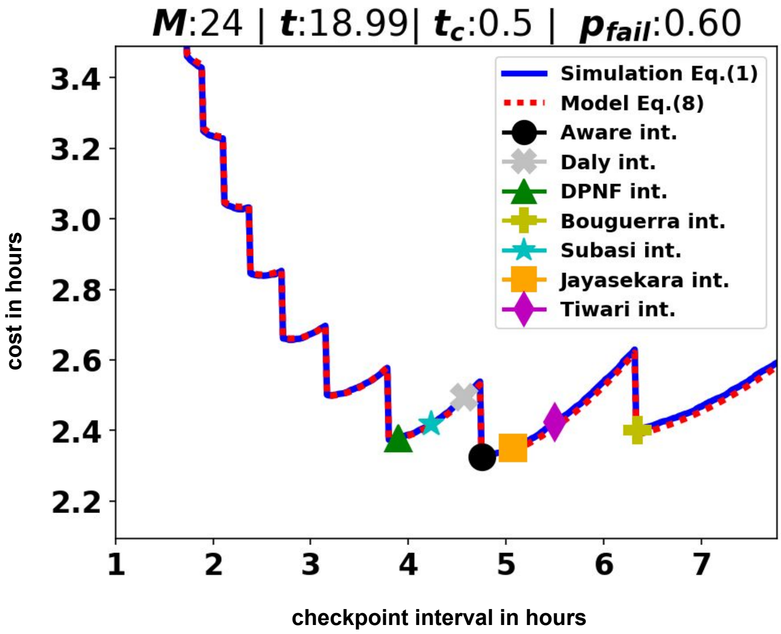

Fig. 3.4: Simulated and aware-algorithm costs of checkpointing for jobs with 60% probability of failing. The aware algorithm is shown in a black circle.

The first noteworthy feature in Figure 3.4 is the good accuracy of the model in red with relation to the stochastically simulated events in blue. Numerically, the standard error of costs for the proposed model is consistently below 10^{-2} or an equivalent interval value of 1.6 minutes. Qualitatively, the cost curve in blue closely follows the simulated curves across the entire range of checkpointing intervals.

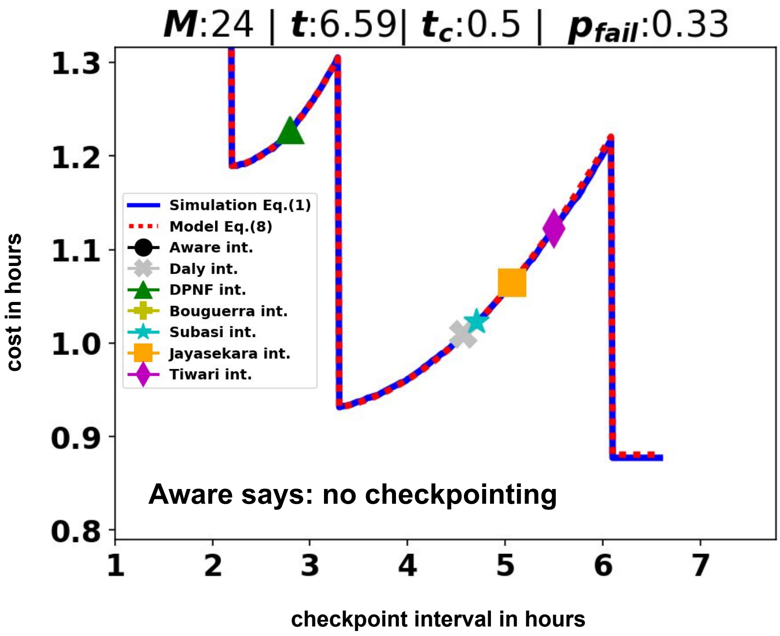

Fig. 3.5: Simulated and aware-algorithm costs of checkpointing for jobs with 33% probability of failing.

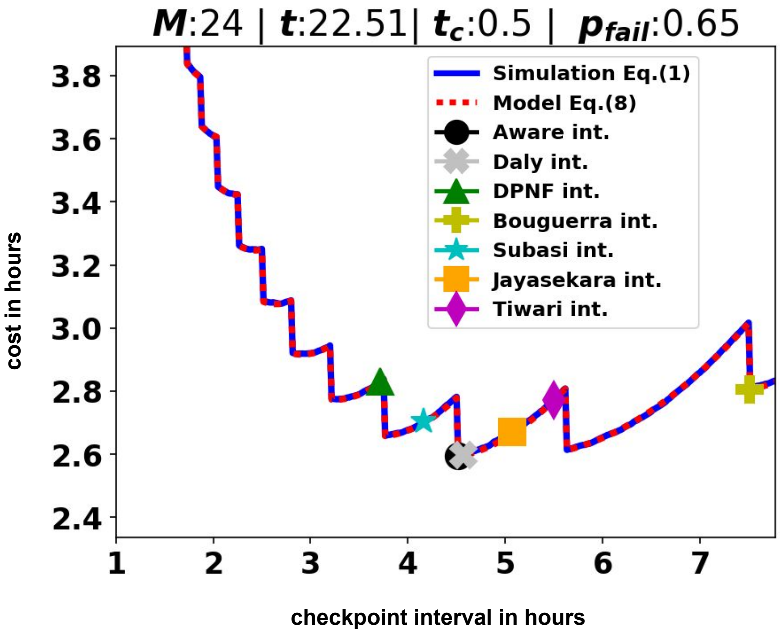

Fig. 3.6: Simulated and aware-algorithm costs of checkpointing for jobs with 65% probability of failing.

The cost function in Figures 3.4, 3.5 and 3.6 showcases a stepped shape that is pronounced for these jobs with low and medium failure probabilities. The sharp steps arise from the floor function within the cost function. These steps cause inaccurate cost estimation in the alternative algorithms, since they do not account for the full checkpoint count. Accounting for these steps is necessary at medium to low failure probabilities.

Figure 3.6 also shows how the alternative methods might report intervals with minimal cost due to chance, since the location of the minimum interval will change depending on the parameters t, t_c, M, although only the aware method considers t. The Jayasekara and Daly methods were close to the aware interval, with the first having a lower cost than the former. Although DPNF and Tiwari provided intervals further apart from ours than Daly, they obtained lower costs than Daly by sitting in lower parts of the steps. This is an example of how alternative methods might have increased checkpoint costs although they have comparable intervals to the optimal. This inexact behavior is common for jobs with lower failure probabilities and is present regardless of the distribution or failure parameters.

Furthermore, Figure 3.5 shows an example of how the aware interval method can recommend to not perform checkpoints at all. In this scenario the aware method recognizes that the average cost of checkpointing is greater than that of losing application progress. In this case users would benefit from not performing checkpoints at all.

At the larger failure probabilities like shown in Figure 3.7, the steps smooth out making the traditional methods consistent with their own modeling algorithms. In this scenario the failure events overtake the non failure ones, and the average cost becomes dominated by the term from Equation .

Qualitatively the cost curve is smooth and concave in the limit $P \to 1$, thus a minimum can be analytically derived in the classical scenario as performed by Daly and Young. Figure 3.7 shows how the cost difference between the aware method and the alternatives consistently shrinks as $P \to 1$, because at that point the aware algorithm achieves the traditional assumption that jobs will fail with high probability. At high probabilities the cost among methods becomes smaller and the choice of method can be left to special requirements that match the specialization of the alternative methods.

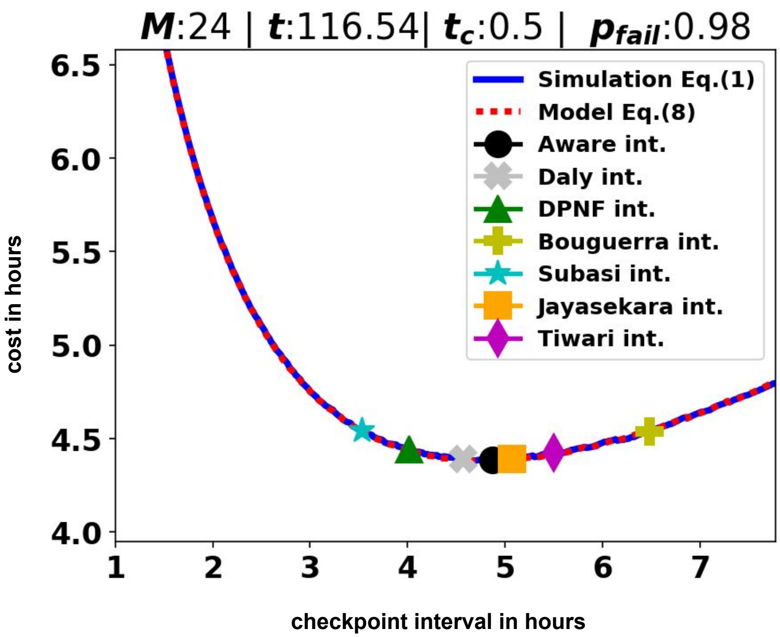

Fig. 3.7: Simulated and aware-algorithm costs of checkpointing for jobs with 98% probability of failing.

3.4.2 Checkpoint cost overhead

Figures 3.8 and 3.9 show the relative extra checkpoint cost for a range of application runtimes as a function of their failure probability $0 < P(t) < 1$. Each point in the curve shows the relative checkpoint costs for using the computed interval of the alternative methods in relation to the aware method. The range of ratios shown is between 1.0 to under 2.0, where values greater than 1.0 show a increased overhead from the alternative when compared to the proposed aware method.

Figure 3.8 shows the same step-wise cost change seen in the previous curves. This step change is evident in the diverging costs between the methods that occur regularly and look like peaks. These ratios oscillate between 1.0 and continuously decreasing maximum. The decrease in maximum continues as the probability of failure increases. From the Figure it can be seen that the stepped cost overhead of the Subasi, Jayasekara and Tiwari methods start at around 60% and lowers rapidly to 20% ending at minimal overhead for high $P \to 1$.

In Figure 3.9 the ratio of all but one method (Bouguerra) also approaches one for $P \to 1$. This shows Bouguerra and DPNF having the highest cost overhead at low probabilities and that the Daly interval can produce 80% more overhead than the new aware method.

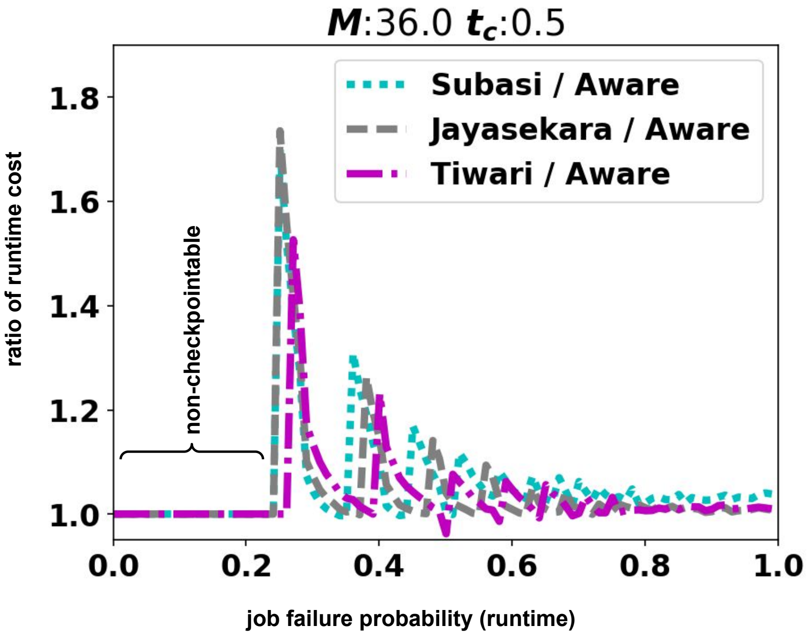

Fig. 3.8: Relative cost between alternative methods and the new aware checkpointing approach for applications with runtimes corresponding to failure probabilities of $0 < P < 1$.

From both Figures we can see again that the non-aware methods offer similar costs to the aware approach for $P \to 1$, but incur significant overheads at low and medium failure probabilities.

3.4.3 HPC system traces

In this section we present synthetic benchmarks with queues of jobs sampled from four different HPC systems: Mogon II (2,000 nodes) [1], LANL Trinity (9,408 nodes) and LANL Mustang (1,600 nodes) [89] and TGCC Curie (5,544 nodes) [90]. For each system there is size information available for each user submitted batch job. The size data consist of the applications node count and expected runtime as given by the users to the batch system on job submission. The collection of these two values for each user job, makes up a system trace of batch jobs. Using batch traces allows us to have an overview of potential cost savings for entire systems. From the traces used in experiments, the Mogon II system showed the most MPF jobs and Curie the least. The results use only the checkpointable jobs (MPF and non-MPF) from the traces (where $t_c + \tau < t$) and explore the MTBF-values $M \in \{24, 36\}$ hours and the checkpoint times $t_c \in \{6, 15, 30\}$ minutes. The savings reported are for the aware-method relative to the alternative methods considering the cumulative

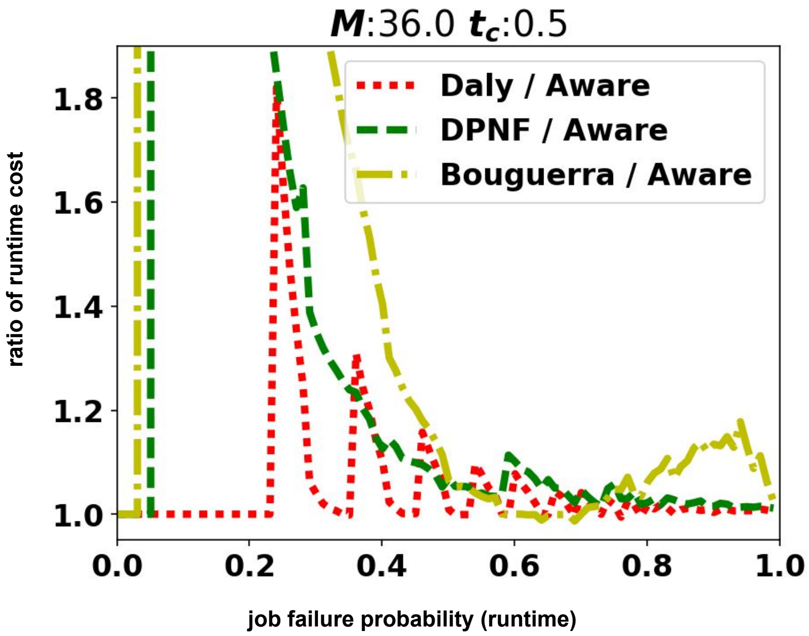

Fig. 3.9: Relative cost between alternative methods and the new aware checkpointing approach for applications with runtimes corresponding to failure probabilities of $0 < P < 1$.

runtime hours of all jobs, re-queueing and checkpointing of jobs. In addition to the various MTBF values and checkpoint costs, the experiments show four versions of Weibull failures with $w = 0.8$, a Poisson failure distribution and two system where the MTBF is uncertain and off by -20% and +20% (Weibull). This is a total of 480 different trace simulations that perform 10,000 failures for each job in the batch traces, where the average checkpoint cost per job is summed and then used to calculate the checkpointing cost savings.

Weibull

Table 3.1-a (bellow the system rows) lists the summarized total checkpoint savings when Weibull failures occur after considering all jobs, including MPF and non-MPF. The summarized row shows at least 7.1% average total savings across all systems compared to the Jayasekara and Daly methods. These two methods can be considered the best performing alternatives for the traces examined. On the other hand, the average checkpoint cost savings compared to Tiwari, Subasi and Bouguerra are 18.2%, 25.7% and 16.2% respectively.

For the Mogon II HPC system the aware algorithm saves between 11.3% and 19.6% compared to Tiwari and between 4.1% and 5.0% for the Curie HPC system. Against

Weibull $w = 0.8$			Total checkpoint cost savings of all jobs (MPF and not) of Alg. 1 compared to method...					e)
System ↓	Parameters							
	MTBF	t_c	Tiwari	Subasi	Jayasekara	Daly	Bouguerra	← Avg.
MUSTANG	36h	0.50h	7.1%	6.2%	5.6%	5.6%	16.6%	13.0%
		0.25h	7.3%	25.0%	5.5%	5.4%	19.0%	
		0.10h	7.0%	57.6%	5.3%	5.2%	22.0%	
	24h	0.50h	6.2%	5.5%	5.3%	5.3%	15.5%	
		0.25h	6.4%	23.2%	4.5%	4.7%	17.8%	
		0.10h	6.8%	56.8%	5.1%	5.0%	21.0%	
ATLAS	36h	0.50h	8.0%	6.6%	6.2%	6.5%	6.5%	9.0%
		0.25h	7.2%	15.6%	5.5%	7.0%	8.4%	
		0.10h	5.8%	44.7%	5.7%	6.0%	13.1%	
	24h	0.50h	6.2%	7.3%	7.5%	8.0%	2.4%	
		0.25h	6.6%	15.4%	5.8%	5.3%	8.5%	
		0.10h	6.1%	42.9%	4.6%	4.1%	14.5%	
MOGON II	36h	0.50h	11.3%	10.5%	9.4%	10.4%	18.4%	18.1%
		0.25h	11.9%	30.3%	9.1%	9.5%	19.8%	
		0.10h	17.0%	39.3%	13.0%	13.0%	26.7%	
	24h	0.50h	11.6%	10.8%	10.0%	10.1%	16.0%	
		0.25h	16.8%	24.2%	24.4%	22.4%	20.1%	
		0.10h	19.6%	39.9%	18.8%	18.5%	30.7%	
CURIE	36h	0.50h	5.0%	5.1%	4.0%	4.0%	13.4%	10.5%
		0.25h	4.7%	18.9%	3.6%	3.4%	16.1%	
		0.10h	4.8%	55.3%	3.3%	3.1%	18.7%	
	24h	0.50h	4.3%	4.7%	3.4%	3.4%	12.5%	
		0.25h	4.1%	17.0%	3.0%	2.8%	14.7%	
		0.10h	4.2%	54.5%	2.8%	2.6%	16.9%	

a) ↑ This Weibull	8.2%	25.7%	7.1%	7.1%	16.2%
b) With Poisson	27.5%	25.1%	7.3%	7.7%	10.9%
c) Same Weibull -20% MTBF	11.1%	24.9%	6.0%	6.0%	15.9%
d) Same Weibull +20% MTBF	12.5%	26.6%	7.5%	7.5%	16.3%

Tab. 3.1: Savings when using our approach for four HPC systems with a combination of two MTBF values $24h, 36h$ and two checkpoint costs $0.25h, 0.5h$. The simulations of a), c) and d) are done using Weibull failures with $w = 0.8$. Poisson failures are used for b).

the Subasi method in the Mustang system the aware algorithm saves 56.8% of the cost for configurations with $0.10h$ checkpoint costs. This low performance for Subasi originates from its bad performance at low checkpoint costs on MPF jobs. The savings over Jayasekara and Daly are between 9.1% and 24.4% for the Mogon II system and around 3% to 5% for other systems.

Against the Bouguerra method the aware algorithm sees better savings between 12.5% and 30.7% for the Mustang, Mogon II and Curie traces. The good savings on Mogon II against Bouguerra can be attributed to the amount of MPF jobs within the Mogon II system and the savings on Curie can be attributed to the degrading performance of Bouguerra on big probabilities of failure as seen in Figure 3.9. Bouguerra performs best on the Atlas system with the aware method having only 2.4% savings for the 24h MTBF and 0.50h checkpoint cost. For the Curie system the aware method shows consistent savings under 5% against the Tiwari, Jayasekara and Daly methods and between 13.4% and 18.7% compared to the Bouguerra method.

In Table 3.1-e the aware algorithm offers average savings of 13.0% across all methods for the Mustang traces. The Mogon II system sees the most savings across all methods with 18.1%. Finally the Atlas and Curie traces see 9.0% and 10.5% savings.

Poisson

In Sub-table 3.1-b we present the savings across all parameters and all system configurations when the failures use a Poisson failure distributions. The average cross system savings for Poisson failures increase to 27.5% against Tiwari and remain similar for Jayasekara and Daly at 7.3% and 7.7% respectively. On the other hand, the savings of the aware method over Subasi and Bouguerra decrease from 14.1% to 12.5% and 10% respectively while the savings remain at an average of over 7% against all methods. These Poisson experiments show that the aware method also provides savings regardless of the failure distribution. It is expected that other failure distributions could be used with the aware checkpoint method.

MTBF Uncertainty

This section investigates how the savings change against the alternatives if the MTBF is not accurately known. This is a common scenario in HPC systems that have not yet collected enough data to determine their own MTBF and would therefore have to extrapolate MTBF guesses from external data. This scenario can be studied by computing the optimal checkpointing intervals with an uncertain value for the MTBF that is perturbed by factors -20% and $+20\%$. The simulations are then done using the "real" MTBF values of 24 and 36 hours for Weibull failures. The aware and alternative checkpointing methods use the uncertain MTBF to determine checkpoint intervals while the simulations of “real” costs are done with the accurate values.

The savings are above 6.0% and 7.5% in Tables 3.1-c and 3.1-d for uncertain MTBF simulations. Savings over the Tiwari method increase slightly to 11.1% and 12.5% for an inaccurate MTBF of -20% and $+20\%$ respectively. The savings over Jayasekara and Daly diminish by 1.1% for the under estimation and increased by 0.4% for the over estimation.

As a side note, performing no checkpointing at all could be an alternative approach by not wasting resources saving the checkpoints. This could save the entire overhead of checkpointing, but computations are lost every time a failure happens, incurring cost regardless. Initial experiments showed that the aware approach saves 49.39% of the failure costs on average compared to no checkpointing at all on average against all HPC systems.

3.4.4 Discussion of the results

The simulations performed show that the failure probability aware optimal checkpointing method presented in this chapter can indeed provide significant savings for entire HPC workloads that include MPF and non-MPF jobs. The aware approach can obtain average savings of 7.1% against the best alternative methods. Furthermore, the aware approach can still obtain average cost savings of at least 6.0% in scenarios where the MTBF is not accurately known. The savings are more significant for systems like Mogon II where MPF workloads are very prominent.

These savings are possible because the aware method considers the cost of checkpointing with the probability of the jobs failing and not failing. The gains of the aware method over the alternatives come from the assumption they make about HPC jobs failing with high probability. This assumption does not hold for the majority of jobs. By accounting for the runtime and node count of individual jobs, the aware method optimizes the checkpoint intervals and reduces resource waste associated with checkpointing overheads.

The best alternatives to the aware checkpoint algorithm are the Jayasekara and Daly methods. At high failure probabilities $p_{fail} \to 1$ other checkpoint methods close the savings gap and might be a better choice in order to adopt their modeling. An example is the Jayasekara method that allows to model the time needed to detect a failure in their interval determination method. The aware method also converges to Daly's for jobs with probabilities of failure close to one such that $p_{fail} \to 1$. Since the Daly method is so cheap to compute, it would be a very good alternative in performance strained scenarios that do not require complex modeling of the failures or system.

Further gains over Daly and Jayasekara come from these two methods not considering Weibull failures at all. Additional gains over the Subasi and Tiwari approaches arise from their simple mathematical form that slightly deviates from Daly's by using a hazard function and work-lost factor respectively. Gains over the Bouguerra interval method can be attributed to the its bad performance at large probabilities of failures.

Overall the failure aware method proposed in this chapter can be beneficial for HPC systems with workloads of MPF jobs

4

Failure prediction for parallel applications

The rate of hardware related failures can increase with the amount of components in big HPC systems. This in turn increases the risk of at least one hardware element failing, which can cause parallel HPC applications that use thousands of nodes to crash. Periodic checkpoint and restart is a common preventative resilience method in practice. An alternative resilience method to creating checkpoints periodically is to react after a predicted failure warning with a checkpoint on demand or process migration. The reactive procedure must be triggered by a failure prediction system that monitors the state of the hardware and decides whether a failure could happen.

Prediction of incoming failures is traditionally performed for specific components using available health metrics from sensors. An HPC job can use thousands of nodes and at large scale the predictions of all nodes need to be aggregated to evaluate whether a job will crash due to any failing node. Therefore the preemptive measures need to be performed whenever an incoming failure alarm is raised, because a single failing node can crash an entire HPC job. This necessary reaction to errors creates unnecessary overhead when too many failure warnings are false and are not actually followed by a real failure.

These false alarms are referred to as false positives in machine learning terms [33, 34, 35] and are a performance metric of the machine learning classifiers that are used to predict failures. The failure predictors are typically implemented using some version of a machine learning classifier that uses logs or sensors as inputs and classifies them as either being associated with a failure or not. The sensor data classification can therefore be made into a binary classifier [34] that needs to be trained to recognize data sets tagged with both failure and non-failure events. Every time the classification of data is performed, the predictor will report a failure warning or an "ok" state. Failure warnings are then followed with a resilience measure like checkpoints. False positives are the failure warnings from the predictor that are not followed by an actual failure event but still produced a resilience measure anyway. False positives then trigger costly and unnecessary mitigation actions.

The failure predictor shown in this chapter will use multiple neural networks [35] in an ensemble to reduce the amount of false positive events. Predicting failures using sensor data and ensemble machine learning has successfully been done to classify time series [91, 92] and used in other fields [93, 94, 95]. Cloud systems [96] and storage nodes [97] have also used ensemble learning successfully to predict failures. Ensemble learning is also a very robust method for binary classification [98, 99, 100], but it has not been used to achieve the low enough false positive rates needed in HPC failure prediction. This HPC specific scenario has thousands of components predicted on, where a single false positive can trigger a costly counter measure.

This chapter will explain the False Positive Prediction problem (FPP) on HPC systems in more detail. It also gives an overview of the classical prediction metrics, an example of the metrics used in previous works and a new proposed metric to better evaluate the FPP called the "the probability of unnecessary checkpoints" or UC for short. Finally the chapter will provide an ensemble method that uses multiple neural networks to predict failures related to hardware anomalies via two voting methods. The final proposed voting failure predictor will obtain a low false positive rate and will also lower the amount of unnecessary checkpoints to make the FPS usable at thousands of nodes.

4.1 Related Work

Failures of HPC components cause significant resource waste by forcing users to re-run their computations lost during a crash. The DARPA white papers on system resilience by Cappello *et al.* showed that large scale HPC systems suffer from 20% computing resource waste including failures and recovery [14, 15]. The goal of failure prediction within HPC environments is to preemptively react to a possible application crash by producing checkpoints on demand or by migrating tasks that might be at risk failing.

Previous works have also shown that to reduce the impact on HPC system performance, careful consideration needs to be given to the method of logging and data collection [101, 102]. Frequent sampling of data for prediction can increase the system overhead by having to store too many values and fewer data samples hinder the creation of training sets for failure prediction. The failure prediction method showcased in this chapter requires high granularity of data sampled. This could have an overhead impact in terms of collection and storage. This high sampling requirement is nevertheless needed to attain usable prediction metrics with enough warning time. Storage overhead could be reduced by using specialized file systems meant for highly parallel environments in HPC. Vef, Brinkmann, *et al.* proposed various distributed file systems that are capable handling high throughput of write requests [103, 104, 105]. The data sets shown in this work use a fifteen second sampling interval. The data is manually mined and labeled for failures or ok events using the unacceptable thresholds for sensors shown in Figure 4.2. The manual labeling of events from data collected could be automated with further machine learning techniques. Gainaru *et al.* labeled the normal and faulty behaviors of nodes by using signal analysis concepts that makes automating and defining failure prediction easier [106]. The data is also collected across an independent management network that does not collide with the computing network infrastructure.

Das *et al.* used LSTM (long short-term memory) networks to predict node failures with a three minute lead time, 85% recall, and 83% accuracy [107]. We use the same LSTM networks for a voting procedure on our second failure prediction system. Klinkenberg *et al.* used a data set collected in five minute intervals and showed adequate prediction rates [108]. Their classifier can identify failures in nodes with a precision of 98% and a recall of 91%. This shows that high record rates are important for good prediction rates. The method shown in this chapter is capable of attaining competitive prediction rates with previous works, but uses a different machine learning methods. The method uses multiple neural networks (ensemble) to reduce false positives after a voting procedure is performed.

Guan *et al.* proposed an ensemble Bayesian classifier for cloud failures that used unsupervised learning to categorize health data [96]. Their method was exploratory and requires administrators to manually label events after the fact. After labeling they use supervised learning with a decision trees classifier to predict failures. They achieved usable prediction of failures that had a false positive rate of 0.008. Our work differentiates itself by doing the labeling by hand directly and using the ensemble for the binary classification. This allows the ensemble of neural network classifiers to achieve lower false positive rates.

Class imbalance is also of relevance in classification. It is usually undesired [109, 110, 111], since it skews the classifier to learn the higher represented class better. We use this to skew the failure predictor into reducing the false positive rate to our advantage as done in other unrelated fields [112, 113].

Hadi *et al.* used an ensemble decision tree method to predict storage failures [97]. They showed that the recall rate was better with lead-up times not too long between prediction and failure. They showed that using neural networks instead of decision trees offers better prediction rates in their use case. The work done in this chapter assumes that sensor data close to the failure events (within 2-7 minutes) is available and also uses an ensemble method to combine multiple classification through voting.

Gainaru *et al.* developed a decision tree based failure prediction method that used data mining event of logs and processed them using signal processing [114]. They recognized that failures within HPC systems did not have a consistent representation in logs and that signal processing helped to improve the results obtained using logs for prediction. The major drawback from their method is that their recall was of around 50%. From their work we corroborated that the available textual logs were not enough to define or predict failures with.

Moraes *et al.* [115] investigated how to reduce false positives for image recognition (applied to medical diagnosis) with support vector machines (SVMs) by using a second classifier. They focused on treating misclassified samples around the decision boundaries. Although we do not explore SVMs, we expand upon their work by using extra classifiers to reduce false positives in an ensemble technique. We try to further reduce false positives by tuning the decision threshold using ROC plots for the final voting mechanism presented later.

Although these previous methods demonstrated usable prediction rates, Yang *et al.* showed their comprehensive survey of 37 methods, that the false positive performance metric was not sufficient to perform preemptive actions only when necessary [116]. Their analysis also show deficiencies in metric reporting, with most works failing to report multiple metrics or analysis like false positive rates or ROC plots. The goal of our approach is to reduce the false positive prediction rates and amount of unnecessary checkpoints that occur when false alarms are triggered. Some previous works have discussed the shortcomings of the classical performance metrics related to binary classification and therefore failure prediction. Jauk *et al.* [116] highlighted that the negative effect of high false positives was under-investigated, Salfner [117] *et al.* pointed out that the accuracy metric is less useful in failure prediction because failure events are rare and Zheng *et al.* [118] proposed a refined definition for *TP, FN, FP* that accounts for lead time and the locality of predicted failures. Taerat *et al.* [119] also remarked that existing statistical metrics were insufficient due to a disconnect with the mitigation effects and introduced a metric that accounts the time used for a mitigation action. We contribute to traditional metrics by proposing a new value to measure the probability of triggering unnecessary checkpoints (UC) caused by the false positive rates of the prediction systems.

The failure prediction method shown in this work focuses in solving this shortcoming by providing high prediction rates and low false positive predictions. Low false positives allows preemption systems to trigger only measures when required. High rates of false alarms would cause the system to waste resources in unnecessary migrations or checkpoints.

The specifics of how the migration and checkpointing methods work are left out from this thesis. These mitigation methods can nevertheless still react to a failure prediction and could be integrated into the proposed prediction framework presented here.

Pickartz *et al.* compiled an overview of suitable migration methods that could be used to increase resilience after a failure prediction is detected [120, 121]. Gad *et al.* provided a method to reduce VM migration time, making it possible to migrate applications in short enough lead-up times after a failure prediction [122, 123]. Wang *et al.* showed that live process migrations could take ead-up times of less than ten seconds and that entire virtual machine migrations could take up to half a minute [124]. All these approaches offer short enough migration times, that they could be performed in the lead-up time between failure prediction and an eventual failure of the respective component.

4.1.1 Supervised binary classifier for failure prediction

The goal of a hardware failure prediction system for parallel applications is to warn about incoming failures using existing observable data with as little false alarms as possible. We utilize a combination of existing known methods and custom approaches to achieve low false alarms. We exploit the inherent class imbalance present in the data available where failure data is less common and non failure events are plentiful. By training the classifiers with more non-failures events the resulting classifier will focus on non misclassifying the non-failure vents as failures. The classifiers can also be tuned using operating characteristic curves to choose a decision threshold that favors reducing false alarms over predicting more failures. To reduce the amount of false alarms further we also use an ensemble of individual predictors that must a vote on a final prediction. Our final voting procedure uses a recurrent neural network to learn the best voting combination from the sample classifiers and reduce false alarms even further.

Since we are interested in a distinction between a failure and an "ok" event, this rules out the use of classification using regression models for continuous variables. The failure prediction problem can therefore be translated into a binary classifier that decides whether sensor data is a sign of an incoming failure or not. In essence the binary classifier should decide between sorting data into failures and non-failure events.

To accomplish this, the binary classifier must learn from existing sensor data that is associated with logged job failure events. Supervised learning was therefore chosen to train the failure predictor. Supervised learning is suitable for this case, because we can produce class tags (failure and ok) for each time-series of sensor data. This is done by mining data of existing logs for relevant application crashes and associating the hardware sensor data with the failure.

The last aspect to consider is the type of machine learning approach to use for classification. For training we will have access to a large training data set where each dimension is a continuous variable. We will also want to use multiple variations of the classifier in an ensemble to reduce the amount of false positives. This is can be done using an ensemble of classifiers where each instance votes with its individual classification. A multi-layer neural network suits these previously mentioned scenarios well and we can easily train multiple neural networks (NN) to create the ensemble. Each neural network can reach its own classification within the ensemble by randomizing the initial weights of the NN and by using different training sets shifted in time for each NN. The main drawback of using neural networks is the difficulty of obtaining knowledge about how the classification decisions are made. To obtain such insight extensive time needs to be spent analyzing the relevant input variables, doing sensitivity analysis and building prototypes that might help reveal inner workings of NN models [125, 126, 127]. An alternative would be to perform decision trees as used by [108] to obtain decision rules directly from the tree structure. This was not done due to the high false positive rate achieved by their method.

4.1.2 Classical Metrics on Classifier Performance

The following basic machine learning metrics describe the quality of a binary classifier that decides between failures and ok events:

- *True Positive (TP)* refers to correctly classified failures
- *True Negative (TN)* refers to correctly classified non-failure (in the following: ok) events

- *False Negative (FN)* refers to misclassified failures as ok
- *False Positive (FP)* refers to ok events that are misclassified as failures.

A binary classifier can predict events to be positive (failure, $(+)$) or negative (non-failure, ok, $(-)$). False positives are false alarms generated by the failure predictor. These false alarms would trigger a checkpoint even when no failure is imminent. The cost of checkpoints would therefore raise in scenarios where false positives are prominent. The False Positive Prediction problem occurs when a prediction system predicts false failure alarms for thousands of components at extremely high rates.

Existing metrics fail to account for this specific scenario and established literature evaluates failure prediction using the following traditional metrics [128]:

- *Precision:* the probability of a positive prediction to be correct, also expressed as a the positive predictive value (**ppv**) calculated with $\frac{TP}{TP+FP}$. In failure prediction this represents the ratio of how many positive failure predictions are followed by an actual failure. The higher the value, the less false positives there are.
- *Recall:* the probability of correctly classifying a positive event, or true positive rate (**tpr**) calculated with $\frac{TP}{TP+FN}$. In failure prediction this represents how many of all failures are correctly predicted. The higher the ratio, the more failures can be preemptively handled.
- *Fall-Out:* the probability of misclassifying a positive event as a negative event, also expressed as false positive rate (**fpr**) calculated with $\frac{FP}{FP+TN}$. In failure prediction this ratio represents how many actual non-failure events are miss-predicted as incoming failure alarms. A low value traditionally represents a good performance metric where false alarms are low.

Although these metrics are adequate to measure the quality of predictions on single components, they fail to capture the performance when aggregating nodes for HPC jobs. Researchers have noted how these metrics do not account for the loss of compute time nor the overhead for migrations and checkpointing [117, 119, 118, 129].

Although a high false positive rate indicates that unnecessary checkpoints will be triggered [129, 116], a low value of this metric fails to assure the opposite (that no unnecessary checkpoints will be triggered) due to the FPP.

This happens because the calculation of the **fpr** is usually based on the **probability to misclassify a non-failure (ok) event** for a single component. Since the fpr is for continuous predictions of the same individual component, the problem significantly increases with higher node counts [116].

Figure 4.1 illustrates how the $\boldsymbol{fpr}$ can influence the false prediction problem. It shows the probability – for various $\boldsymbol{fpr}$ values – that at least one FPS in n nodes (x-axis) miss-predicts a failure (that is actually a non-failure event) causing an unnecessary checkpoint for a job running on the n nodes. This probability of causing unnecessary checkpoints $\boldsymbol{UC}$ will be formally defined in Section 4.1.4, but for now it helps to understand that false positives of individual nodes have negative effects when predicting many nodes. In this scenario a failure predictor nearly always predicts at least one node failure. It follows that predicting failures for each component can trigger too many unnecessary checkpoint events at low **false positive rates** of 0.01. An $\boldsymbol{fpr}$ of 0.00156 (with a $\boldsymbol{tpr}$ of 0.9158) is already low and is traditionally considered a great value in prediction systems (see [108] in Table 4.1). Any FPS with a UC value of higher than 0.5 is worse than a random predictor with coin-toss probabilities. An $\boldsymbol{fpr}$ value of 0.00156 is better than most works, but has much worse $\boldsymbol{UC}$ probability than a random predictor at 750 nodes and will trigger a lot more unnecessary failure preemptive measures for jobs running on thousands of nodes.

It is also the case that many related works do not consider or report the $\boldsymbol{fpr}$ at all, making it harder to evaluate their performance when predicting on high node counts. In Table 4.1, FPS for both HPC and cloud systems are shown. The $\boldsymbol{tpr}$ metric of HPC FPS works in Table 4.1-a) shows very good values above 80% and up to 95% for HPC systems. This suggests that correctly classifying failures with high rates is possible at the cost of false positives for large node counts.

These false positives are of interest because HPC applications using thousands of nodes would fail when any node fails. For this reason a failure prediction method needs to trigger a mitigation event on any positive failure prediction. This would be very costly even for the lowest false positive rate in the table, where 79.78% of the time would trigger a costly checkpoint.

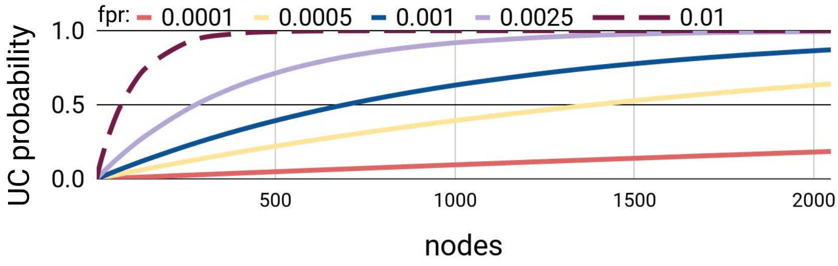

Fig. 4.1: The various curves show the probability of triggering an unnecessary migration or checkpoint due to certain false positive prediction rates in a prediction system.

Tab. 4.1: Example FPS for clusters – the last six entries represent this work for different numbers of neural networks while $UC(1024)$ denotes the probability of unnecessary checkpoints for 1,024 nodes. All the rates are values $\in \{0-1\}$ where 0.80 would mean 80%.

Machine/Reference	tpr	ppv	fpr	$UC(1,024)$
a) HPC Systems				
TITAN[130]	0.89	0.73	NA	NA
TITAN GPU[131]	0.95	0.82	NA	NA
BlueGene/L[132]	0.80	0.4848	0.33	>0.9999
BlueGene/P[133]	0.854	0.823	NA	NA
RWTH Aachen[108]	0.9158	0.98	0.00156	0.7978
Unnamed SC[134]	0.86	0.99	0.23	>0.9999
b) Cloud Systems				
Google[135]	NA	0.80	0.11	>0.9999
Google[129]	0.94	0.95	NA	NA
AmazonEC2[136]	0.859	0.942	NA	NA
Custom Hadoop[137]	0.91	0.93	0.091	>0.9999
Custom Misc.[96]	0.65	NA	0.008	>0.9997
Baidu [97]	0.81	NA	0.0037	>0.9775
c) Unanimous Voting				
MOGON *this@1NN*	0.8461	0.9655	0.00292	0.9498
MOGON *this@2NN*	0.7795	0.9884	0.00088	0.5945
MOGON *this@4NN*	0.7302	0.9944	0.00040	0.3361
MOGON *this@6NN*	0.6918	0.9970	0.00020	0.1852
d) RNN Voting				
MOGON *this@6NN*	0.7448	0.9992	0.00004	0.0417

NA: Data not available.

Unfortunately only half of the examples provide a value of the fpr for evaluating UC probabilities. Even when the fpr value is available, the UC probability makes the method unusable even at 1,024 nodes (see last column). The ROC analysis or AUC values are traditionally also not reported in the HPC works of sub-Table a) and from sub-Table b) only [96] reports them and [97] mentions to use ROC analysis. In this work we perform the ROC plots to decide on good choices for fpr and tpr.

The cloud specific FPS from Table 4.1-b) show similar good tpr for failure prediction but also posses the same bad fpr. The main difference is that the FPS that these cloud systems show is sufficient for cloud applications. The cloud programs are not necessarily machine-parallel and highly interconnected, making the single machine fpr value good for the use case.

The FPS that we propose are shown in Tables 4.1-c) and d). The first four entries show the rates for using 1,2,4 and 6 neural networks in the predictor with an unanimous voting procedure to perform the final prediction. These results will be discussed in more detail in the later Sections. The important thing to note is that the **tpr, fpr** and **UC(1,024)** decrease with the amount of networks. The decrease of tpr is not desired, but it is an effect of adding more networks that need to all agree on failures. This in turn decreases the fpr and **UC(1,024)** to lower values than previous works. Table 4.1-d) shows an improved voting system that uses a recurrent neural network to aggregate and learn a better the voting using all six feed forward neural networks as inputs. This system further improves on the both the tpr and fpr by not having a rigid unanimous voting. We will show how both systems work and perform in more detail in the following Sections.

4.1.3 Modeling Hardware Failures in HPC Systems

Knowing the frequency of application crashes within HPC systems can help us understand how often a failure prediction system is expected to classify actual failures. The less failures occur, the more actual ok (non-failure) states will have to be classified by the FPS. If most classifications over time are ok events, the risk of false positive predictions increases.

The reliability of (HPC) systems is typically modeled using the *Mean Time Before Failure (MTBF)* and exponential distributions for the arrival time of failures [138]. The true failure rate (or MTBF) of production systems is typically kept secret [139] or is extrapolated from old systems. To understand failures in production systems using real data we collected the statistics of a huge set of sensor values over a period of six months for the MOGON I HPC system at the Johannes Gutenberg University. This HPC system is at the end of its usable life and is susceptible to failure events, which makes it easy to evaluate the quality of failure prediction techniques. We are in this scenario interested in the probability of failures and ok events for the whole HPC system.

$$P(E) = \frac{\#failure}{\#all} = 0.0000099653 \quad P(K) = \frac{\#ok}{\#all} = 0.9999900347 \tag{4.1}$$

Equation 4.1 shows the probabilities for failure events and non-failure (ok) events after data mining the sensors and HPC job logs of the MOGON I system. The probabilities were determined by counting all events (including (E)) over the period of six months using the sensor data sampled every fifteen seconds and with job statistics retrieved from the batch system scheduler. All non-failure events are referred to as *ok* (K) events. From these failure probabilities it can be seen that even for such an old system failures are rare and it can be assumed that younger systems will experience even lower $P(E)$ values due to lower failure rates of new hardware.

As discussed previously, an FPS should trigger a checkpoint in response to a predicted failure (positive), but as the $\boldsymbol{fpr}$ value increases, so does the amount of unnecessary checkpoints that are triggered. We introduce the **sample probability of false positives** $S(f_+)$ as a better measure of how false positives are encountered in prediction systems. The values for $S(f_+)$ can be calculated using Equation 4.2, which combines the probability of encountering or sampling an ok (non-failure) event with the false prediction rate $\boldsymbol{fpr}$. The term $P(K)$ is the probability of encountering ok events in practice and $\boldsymbol{fpr}$ is the classification rate that comes from a binary machine learning classifier. The probability of an event being a non-failure (ok) $P(K)$ is very high since failures do not always occur. Combining this high sampling rate with the probability of misclassifying the event as a failure, makes the value $S(f_+)$ stay close to the $\boldsymbol{fpr}$. The related terms $S(f_-), S(t_-)$ can be calculated similarly.

$$\begin{aligned} S(f_+) &= P(K) \cdot fpr, \quad & S(t_+) &= P(E) \cdot tpr \\ S(f_-) &= P(E) \cdot fnr, \quad & S(t_-) &= P(K) \cdot tnr \end{aligned} \tag{4.2}$$

Assuming that a “mock FPS” has relatively good prediction rates of $\{fpr = 0.01,\ tpr = 0.99\}$ and failure rates similar to the MOGON I cluster environment, the following values for the false $S(f_+)$ and true $S(t_+)$ positive probabilities can be calculated:

$$\begin{aligned} S(f_+) &= P(K) \cdot fpr = 0.01 \\ S(t_+) &= P(E) \cdot tpr = 0.000009865647 \end{aligned} \tag{4.3}$$

Equation 4.3 shows that most predicted failures $(+)$ would actually be false positives, even when the prediction quality dictated by the recall (*tpr*) is high. With this knowledge, it becomes evident that the sampling probabilities $P(E)$ and $P(K)$ must be taken into account when evaluating failure prediction methods.

4.1.4 Unnecessary Checkpoints

To prevent an application from crashing due to a node failure, an FPS must be able to trigger a mitigation mechanism after prediction and before such failure occurs. Checkpointing is a typical mechanism that applications can use to save a stable state of their computations to then later restart on a new set of failure-free nodes [140, 30, 141]. Alternative approaches like process [124, 142] and virtual machine migration [143] can reduce the overhead of each triggered mitigation.

MPI applications running on n nodes can fail when any of their nodes encounter a failure and provokes a crash of the programs processes. When dealing with an FPS that can make continuous predictions per node, consider this new metric UC as the **probability of triggering an unnecessary checkpoint** for n. This $UC(S(f_+), n)$ probability is modeled as a two-factor-multinomial experiment [144] characterized by the cumulative distribution function (CDF) shown in Equation 4.4.

$$UC(S(f_+), S(t_-), n) = \sum_{x=1}^{n} \binom{n}{x} \cdot S(f_+)^x \cdot S(t_-)^{(n-x)} \tag{4.4}$$

The model in Equation 4.4 is used to describe the $\boldsymbol{UC}$ probability under the assumption that each job runs on n identical nodes and that the FPS of each node has a probability $S(f_+)$ of encountering a false positive and a probability of $S(t_-)$ for true negatives. The $\sum_{x=1}^{n} \binom{n}{x}$ sum adds all possible outcomes $x \in [1, n]$ in which the predictor can report a false alarm event $S(f_+)$, **and** no real failure was expected. The entire equation represents the probability of unnecessarily triggering checkpoints given a false positive rate.

Using Equation 4.4 on the "example FPS" with a $\boldsymbol{fpr}$ of 0.01, the $\boldsymbol{UC}$ value for a job using 256 nodes reaches $UC(0.01, 256) = 0.9236$. *At this value it is almost certain that every prediction for such a job will trigger an unnecessary checkpoint.* The

UC value increases as the number of nodes (n) increases, showing that established metrics such as tpr and **ppv** are insufficient at jobs with high node counts. When considering $1,024$ and a fpr of 0.00035, the UC probability is reduced to usable levels $UC(0.0001, 1024) = 0.0973$ where only a tenth of predictions trigger an unnecessary checkpoint.

The false positive prediction problem can therefore be expressed as follows:

A failure prediction system in HPC, which serves parallel jobs require thousands of nodes, will continually trigger unnecessary checkpoints (or other mitigation) if the false positive rate is not low enough such that UC stays bellow a usable threshold.

We propose that Equation 4.4 is a better metric to evaluate failure prediction systems and their sensitivity to the false positive prediction problem for jobs with thousands of nodes. This metric is used in the prediction system presented later in the thesis.

4.2 Data processing and prediction system

This section describes the data mining process from hardware log gathering to the creation of a failure prediction system. The final goal is to predict batch job failures for nodes using hardware sensor and operating system data with sufficient lead-up time that mostly only necessary preemptive measures like checkpoints are triggered. The presented failure events and trained machine learning classifiers describe only the Mogon I cluster at the Johannes Gutenberg-Universität Mainz, but the principles of the ensemble machine learning approach should be transferable to other HPC systems, given that usable sensor data is available.

Hardware and OS sensors will be used to define anomalies and their timestamps will be matched with failing HPC jobs to create "failure" events. The hardware sensors and OS logs from these events will then be used to create a machine learning classifier at the individual compute node level. The resulting classifiers will represent the collected data of all nodes and might be consider to represent a generic compute node.

4.2.1 Available data and statistical anomalies of hardware problems

Consistent hardware failure logs that contain accurate timestamps are difficult to obtain. Failure logs consist usually of software system logs that crash at some point after the hardware has already failed [114]. We are interested in hardware failures that have accurate timestamps of the time of failure. In our experience, system Linux logs from the Mogon I cluster at the Johannes Gutenberg-Universität Mainz contained no hardware specific logs with timestamps. These timestamps are important since they allow us to mine sensor and OS data preceding the failure event.

Due to limited timestamp availability, hardware failures related to *performance degradation* and *extreme hardware thresholds* were used for predicting "hardware-related" HPC job failures (i.e: the batch job failed) that followed the data anomalies. This results in two types of meta-events (failure and ok/non-failure) that were mined from sensor data of the type presented in Table 4.3, where expert knowledge was used to determine thresholds at which data is deemed intolerable for a production system. Expert knowledge is used instead of another machine learning mechanism, because these are vague properties that are specific to HPC center policies of acceptable working parameters. If other HPC systems better logged failure data with usable timestamps, the learning of such failure events would also not be necessary. Batch job failure logs were then used to keep only HPC jobs, which correlated with anomalous behaving hardware. Software failures from global file system crashes or application miss-configurations are of no interest to this hardware related prediction and were filtered away. These "non-relevant" events where filtered out by failure code, logs and by removing events that had software stop extremely close to the start, i.e: up to 3 minutes. This approach limits the sample scope of the proposed predictor to node-level HPC job failures that correlate to anomalous hardware behaviour. This is done because we are interested in failed jobs that can recover and complete with a checkpoint, which software errors will not make possible.

The data was mined from the *MOGON I* HPC system for six months between 2017 and 2018. The HPC system (now out of service) consisted of over 512 homogeneous nodes running AMD Opteron 6272 CPUs. Each node had four sockets, 64 physical cores, a minimum of 128GB memory, and an Infiniband high performance interconnect. The system was at the end of its life and had continuous applications failing. The homogeneous nature allows us to model the failures of multiple nodes into a machine learning classifier. Our work does not deal with heterogeneous systems although the work could be replicated for each type of hardware.

Table 4.3 lists the data from the compute nodes used in this work. The software data is collected from Linux `proc` metrics and IPMI logs through a management Ethernet network. Due to privacy regulations, no user application data was collected, making our approach application-agnostic. Many other possible logs were omitted

Tab. 4.2: Table showing the data collected every 15 seconds for all compute nodes.

Hardware & OS	Data
CPU Temperature	CPU User
CPU Fan RPM	DISK Usage
CPU Voltage	MEM Usage
PSU Temperature	Bytes In
1Min Load	Bytes Out

due to a strong correlation between various fields, e.g., packets in/out correlates with bytes in/out. All data is stored every 15 seconds in round-robin databases with corresponding timestamps. These timestamps are necessary to match the log values to a specific type of event. The *(value, timestamp)*-pairs are the **necessary data condition** needed by any FPS in order to predict an incoming event with some lead-up time.

We selected the shown hardware sensors for their vital properties in the correct functioning of a node. The sensor data selection was also reduced from a bigger set of metrics by removing correlated data like duplicated temperature sensors, fan RPMs, among others. We plotted some logs as histograms on Figure 4.2 to show correct functioning thresholds and highlight outliers. This information was used to determine whether a failed batch job contained enough extreme values resulting in a failure. As we mentioned previously, we are interested in failed jobs with related hardware anomalies that can recover and complete with a checkpoint. By determining these hardware anomalies within failed jobs, it is possible to later predict job failures based on the data available in Table 4.3. For these and only for these jobs, a migrated or checkpointed application may resume successfully on alternative hardware. This method can be extended to any type of hardware anomaly if the corresponding data is timestamped.

Figure 4.2 show the histograms for the values of some of the available sensors and the thresholds that are of interest to us. E.g., low fan RPMs could be caused by an under-performing fan that is struggling to spin, lading to higher temperatures and thermal throttling by components. High RPMs could signal an imminent failing fan that is performing out of specification.

This allows us to define a failure event relevant to HPC jobs as hardware anomalies that match an actual HPC job failures using batch system data. These failure definitions could be extended by being more strict, e.g. with trends of these values over time, such as, oscillating RPM values. For the sake of simplicity, complex definitions of thresholds or failure modes are not covered in this work. Alternatively,

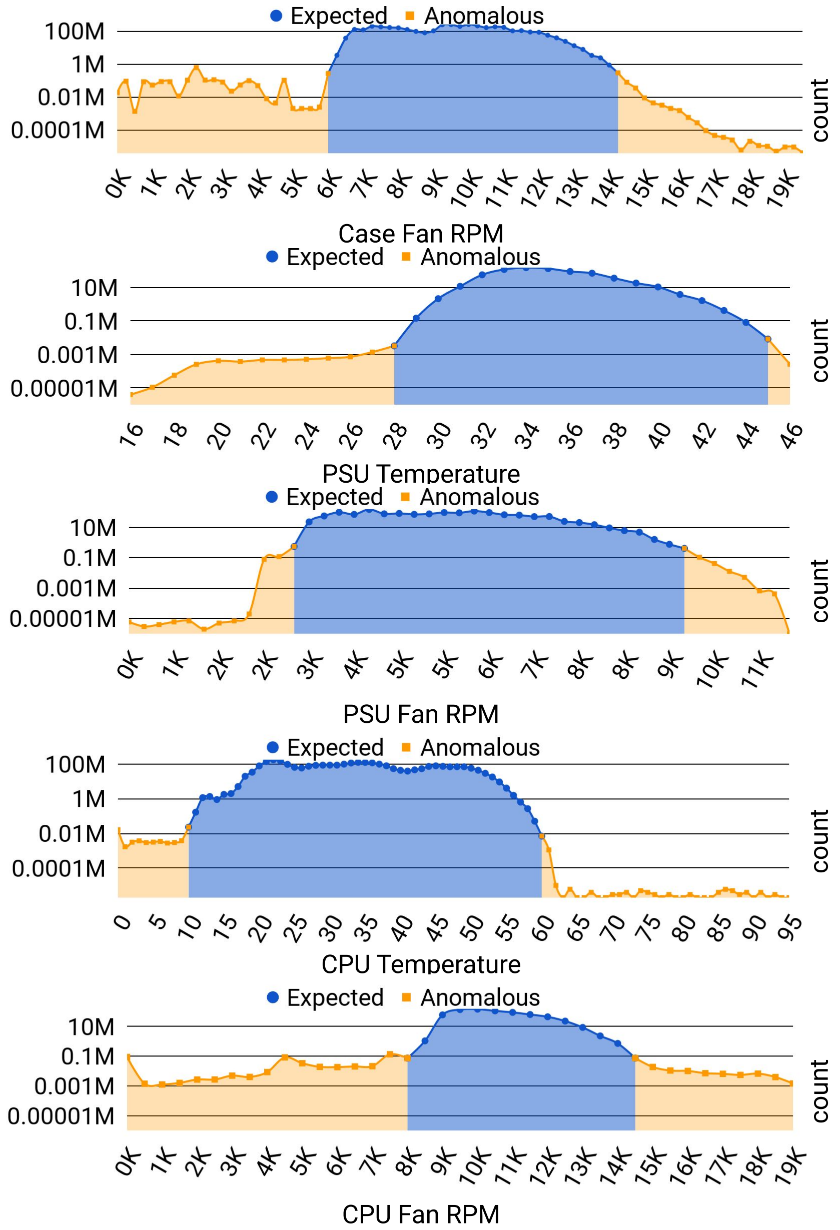

Fig. 4.2: Histograms of sensor data where orange shows anomalous values and blue expected ranges.

this process could be replaced with proper failure logs if those are available with timestamps too. A machine learning approach could be performed to obtain more data, but we found this to be unnecessary and beyond the scope of reducing false positives. Failure events used for the prediction of MOGON I we can therefore be defined as: ***hardware anomalies that are followed by an application failure.***

4.2.2 Data processing

The first step was to extract batch job traces from the SLURM batch system in MOGON I using the `sacct` API[1]. The jobs data was mined within a six month-long time frame and the resulting traces were classified based on their exit codes either as ok for successful job completion or failed jobs for non-user related failures. The data was then filtered to remove failed jobs that crashed within two minutes after starting, as these would mostly be application failures like missing binaries, miss-configurations, among others. To avoid job type bias the amount of events considered from each application type was limited to a chosen threshold of ten. Considering too many failures of the same HPC job might lead the machine learning method not generalizing the failure types that other jobs suffered and having a bias. Too many failed jobs of a single type could also hint the failure being an application problem.

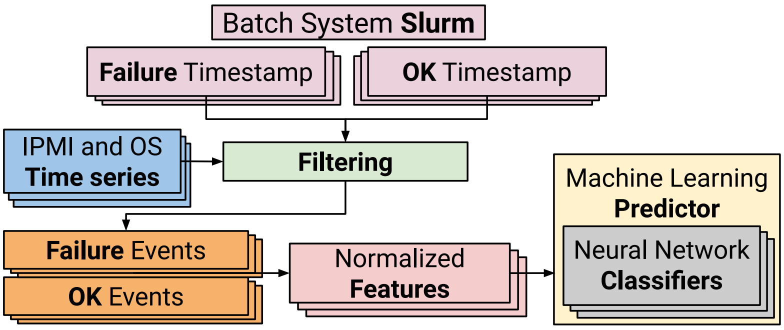

Fig. 4.3: This pipeline depicts the source of the data mine, the filtering of said data and the creation of a failure prediction system through a machine learning classifier.

Figure 4.3 depicts how the job data is gathered from SLURM, manually classified as *ok* and *failure* events, filtered and combined with sensor data and finally used in a predictor, which is made of several neural network classifiers. We was able to gather 13,868 pre-filtered failure events with available relevant sensor (IPMI+OS) time-series information. The timestamps of the events were mapped to their time-series sensor data and the relevant values were extracted within a useful time frame leaving enough lead-up time to perform checkpoints between prediction and the possible failure.

As defined previously, the failure events were defined as a combination of having failed HPC jobs and having any sensor value with enough anomalous threshold values that were followed by an HPC application failure. A more accurate event log about complete hardware failure would be proffered, but this limitation instead lets

---

[1] `https://slurm.schedmd.com/sacct.html`, last accessed in June 2019

the machine learning method generalize the relation between sensor data and the logged application failures.

The ok events (batch jobs that completed successfully) were mined with sensor samples from within successful jobs. The filtering of events is necessary to keep only those with available timestamps and valid sensor data. The goal is to keep data from jobs, which can be resumed after a checkpoint due to hardware anomalies. After filtering only 59,972 ok events and 5,872 failure events remained. Machine learning methods might traditionally require balancing of training classes to avoid bias. Balancing was not done due the limited availability of failure events and the focus of this work on reducing false positive predictions. An unbalanced set of classes to the positives will favor the reduction of false positive predictions.

In a practical real system there will be more events that have non-failures than there are failure events. We embrace the inherent class imbalance present in the data itself to provide the machine learning method with as many non-failure events as possible. This will allow the NN to skew its classification rates towards properly identifying negative (non-failures) and reduce the fpr (negatives miss-identified as positives) in the process. For this reason a ratio of $10:1$ in favor of non-failure (ok, negative) events is chosen. If we were to reduce the amount of negative events to be in equal proportion to positives, the system would tend offer better tpr, but at the cost of fpr. This method alone will not be sufficient to offer low tpr, but will certainly help to achieve it.

After obtaining events, the gathered sensor values are transformed into normalized features from the available time series. The feature normalization is done for a set of seven basic features that showed promised in early data exploration (see Table 4.3). No more were added to keep the amount of dimensions and inputs to the NNs low, since each additional feature would add 10 more inputs to the NN (one per data log). The features where not automatically learned in order to reduce the training time of the NNs. If features are learned automatically, the NNs would need to be deeper to learn them and have longer training times. The basic set of features listed here also proves to be sufficient in the results.

Each feature is calculated once per time series for a total of 70 normalized values per compute node.

It is possible to have different feature values by changing the time frame of sensor data used to calculate them. As time goes on, old sensor values can be discarded and replaced with newer ones. This creates a sliding window of sensor data that can

Tab. 4.3: Table showing the features used for each data sample.

Feature
median
minimum
standard deviation
sum of values
variance
maximum
mean

be used to calculate the features in sequence that are fed into the different neural networks.

4.2.3 Failure Prediction with Neural Networks Ensemble

The main goal of this section is to address the FPP problem by proposing a failure predictor that can predict as many failure events as possible while maintaining lower false positive rates than previous works. Machine learning methods with good tpr rates already exist, but these methods do not solve the false positive problem due to high false positive rates (see Table 4.1) when combining the prediction of thousands of nodes. The FPP problem is addressed here by using multiple neural networks (NN's) and testing two aggregate voting system to create a failure prediction. Two aggregate voting systems for the FPS are tested, a manual method where all networks must agree to produce a positive prediction (unanimous voting), and a machine learned (**RNN**) system that classifies all individual **NN** classifications to predict a failure. The aggregation of the ensemble classifications into a final prediction allows the FPS to attain low enough $fpr's$ so that false alarm predictions trigger less unnecessary checkpoints. The benefit is a reduced overhead from mitigating potential failures by reducing the amount of unnecessary ones.

By using this ensemble technique we can train multiple neural networks using different initial random weights and with different training data sets from the time shifted features discussed in the previous sub-section. The different datasets are produced with a sliding window that uses new sensor values as time goes on and discards old ones. These changing values are then used calculate the features that become the inputs to the neural networks. This makes it so that each neural network (NN) in the proposed predictor is trained with samples mined at different lead-up times (relative to the failure event) in a sequential process.

The choice of using an ensemble of networks trained for different lead-up times was done to solve the *False Positive Prediction (FPP) problem* by having more classifications. The six different networks are trained to predict failures at lead-up times of 2, 3, 4, 5, 6, and 7 minutes. Each NN consists of 70 input neurons that map to the seven features of the 10 data logs. The NN also have 52 $tanh$ hidden neurons, and two output neurons for both prediction classes, failure and ok events (non-failure). The NNs are therefore constructed similarly to mock network in Figure 1.11.

At each prediction the NNs receive a total of four minutes of time series window data transformed into normalized features (i.e: mean). Every individual network will attain a good $\boldsymbol{fpr}$ and $\boldsymbol{tpr}$ score by itself, which are competitive with Table 4.1 previous works. Unfortunately individual networks will fail to achieve low enough probability of triggering an unnecessary migration, as defined in Equation 4.4. This will be discussed in more detail in Section 4.3. The $\boldsymbol{UC}$ is expected to be improved when using NN predictions using a voting approach where multiple networks must all agree whether a failure is imminent. This is an ensemble of NNs that should reduce the $\boldsymbol{UC}$ probability to more manageable ranges.

Unanimous voting FPS

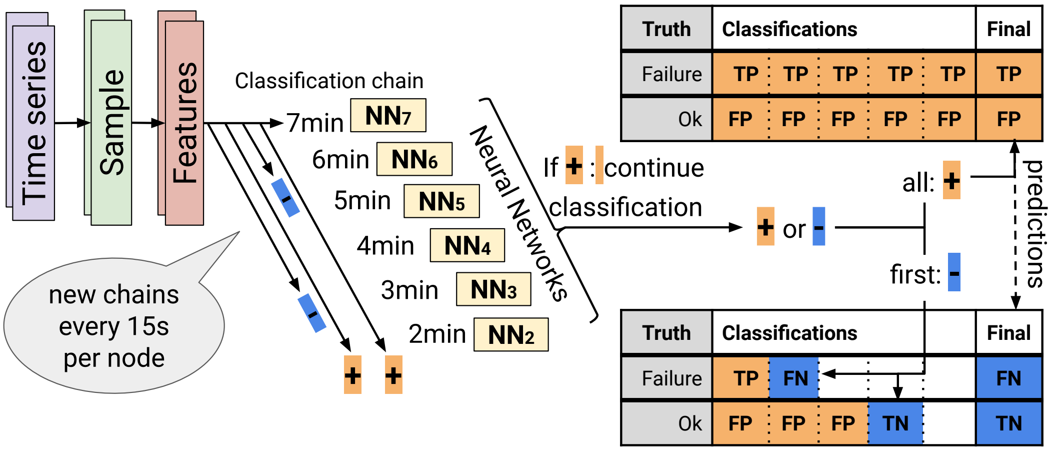

Fig. 4.4: The first predictor consists of sequential classification NN with up to six networks using feature-vectors with incremental lead-up times and an all agree voting system.

Figure 4.4 illustrates the layout of the sequential neural network classification with unanimous voting to corroborate whether a predicted failure is not a false positive when an unanimous decision is made. The voting in this example has to be unanimous before a positive prediction can be made. It is therefore important to distinguish between intermediate classifications made by individual neural networks and the final prediction made by the FPS after voting. A classification sequence starts by utilizing feature samples (four minutes of original sensor data), which goes through a network with a lead-up time of seven minutes. Classification continues by rotating sensor data (add new and dropping old) and feeding it to the next NNs in the sequence. A new sequence can be started at a rate of 15 seconds and this can be modified to how far apart final predictions should be. The classification times on already trained individual networks takes no more than a few hundred milliseconds, making final predictions within the lead-up times possible.

For a prediction all networks must classify a positive (incoming failure) and effectively unanimously consent on predicting a failure. If at least one network classifies a negative (no failure detected), the classification sequence is stopped and a final prediction is emitted indicating that no failure will occur. This way, the number of false positives is reduced at the cost of true positives. Therefore multiple NN combine distinct positive classifications into a single positive prediction. After each final positive prediction of a failure, a total of two minutes of lead-up time (using six networks) for checkpointing is available for mitigation procedures like checkpoints. This lead-up time is enough to trigger and perform a checkpoint according to relevant works [122, 124]. The unanimous voting mechanism might be too strict of a heuristic, so a learned approach using a recurrent neural network is tried next.

Machine Learned voting FPS

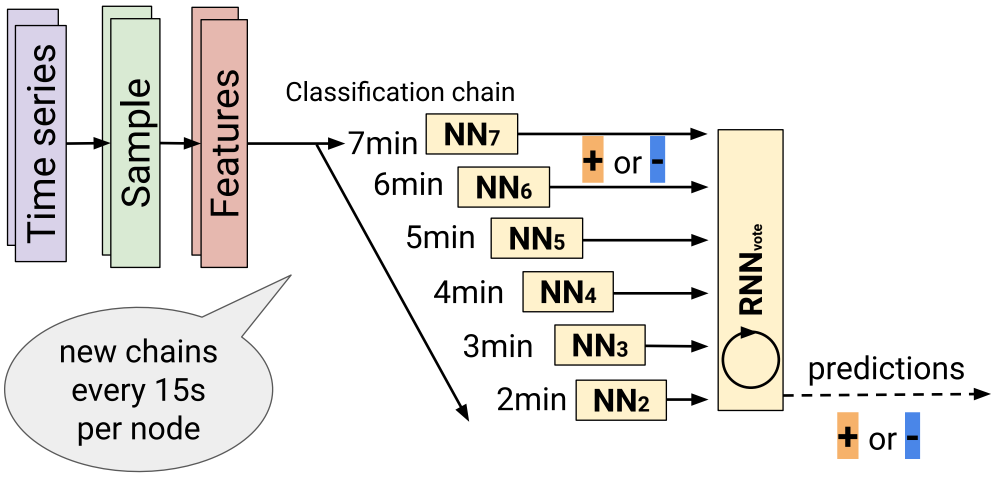

Fig. 4.5: The second predictor consists of sequential classification NN with up to six networks and a final recurrent neural network that aggregates the results from previous neural networks for a final prediction. This method is meant to improve the classification rates.

Figure 4.5 illustrates the layout of the same previous ensemble predictor, but with a final aggregate recurrent neural network that learns the sequence of positive classifications from previous networks and does a final prediction. This final RNN replaces the unanimous consensus voting system and is better able to aggregate the individual classifications of time shifted data and produce better classification rates. Unlike the previous FPS with unanimous voting where less networks can be used, this learned voting needs all six networks as inputs for the RNN. The RNN is trained with the outputs of all the NN_i networks as samples and the expected event type (ok, failure) as labels. This allows the RNN to learn which NN_i are mistaken with false positives and produce a better voting system with better final rates.

We emphasize that no specific binary classification method is required for the individual classification steps in the sequence. Different methods tailored for different input data types could be used to neural networks shown here. This could include ML methods tailored for ECC errors or sensors of accelerators. A mixture of methods could also be used at different lead-up times. This work uses neural networks out of convenience and ease of implementation with standard tools. Other ML methods might have worse or better performance in terms of traditional metrics, but neural networks allows us to have an ensemble where each NN has different training sets.

It is also important to note that the specific failure prediction results presented in this work are representative only of the MOGON I HPC system. Nevertheless the method is not restricted to the underlying hardware and could be applicable to other types of failures or HPC systems, but the final trained networks are only predicting failures for the system from which the data was sampled.

Usage Example

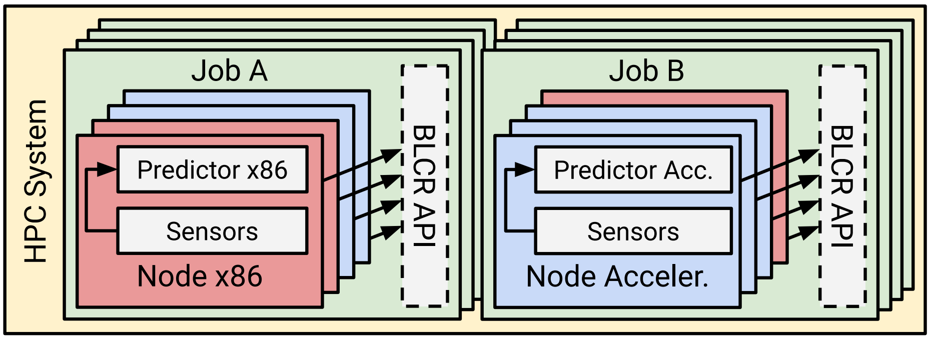

Fig. 4.6: Example deployment of the NN predictor warning of node-level failures in an HPC system running multi-node user jobs.

Figure 4.6 shows how to deploy one predictor on individual nodes of an HPC system. Two distinct predictors are shown for an x86 system and an accelerator system. The figure shows two distinct jobs running on their own set of nodes. Checkpoint frameworks like LAM/MPI, Berkeley (BLCR) and Scalable Checkpoint/Restart (SCR) [145, 146, 147] can be invoked to produce a checkpoint when an imminent failure is predicted at a node. A positive prediction on any node of a job will trigger a checkpoint. The number of nodes in the system or job does not limit the ability to predict failures for each independent node, but does affect how many events are triggered based on the $\boldsymbol{fpr}$ and $\boldsymbol{tpr}$ as described by the UC probability.

4.3 Evaluation and results

All the used time series data and job traces were processed using the RRDtool [148] version 1.5 and python 2.7 with the libraries tsfresh, numpy, and pybrain [149]. The failure prediction system was created using the layouts shown in Section 4.2.3 with six networks classifying event for lead-up times of two to seven minutes. Two voting systems are tested, unanimous voting, and machine learned voting. From the MOGONI HPC system at the JGU [150], 59,972 ok events and 5,872 failure events were mined. The data sets were split into a training, validation, and final testing sets. The data sets are class imbalanced in favor of ok events to favor their correct identification and diminish false positives. For each corresponding lead-up time (see Figure 4.4), the corresponding training set was used to train the networks and the performance of each network was evaluated with the validation set. Monte Carlo (MC) cross-validation (see k-fold cross-validation [34, 35]) was used in the training of the neural networks.

All feed forward networks were trained in a supervised fashion using the data samples and their labels. All neural networks were trained independently, meaning that their training sets originate from the sliding window of features, making the data different. The neural networks for different lead-up times are basic feed forward networks and the learned neural network voting system uses an LSTM neural network. The base feed forward neural networks consist of 70 input neurons, 52 hidden neurons, and two normalized outputs where the probability of failure and non-failure add up to 1. These base NNs are used in both the all-agree voting predictor and as inputs for the aggregate LSTM voting neural network predictor. The LSTM therefore consists of six inputs (one per base NN), 4 hidden neurons and two output neurons. The base NNs where trained using back propagation (BP) and the recurrent LSTM networks using back propagation through time (BPTT) as provided by pybrain. The LSTM RNN used 60% of the samples (NNs outputs are the inputs) for training and 40% for validation. These are all standard training algorithms that were used as present in the pybrain platform. The implementation is not dependent on the machine learning platform and could be re-done in other architectures like Tensorflow [151], Pytorch [152] or Keras [153].

As mentioned in Section 4.2.3, the **prediction** of failures is done at the final prediction made by the FPS and **classification** is only the intermediate result produced by each NN.

4.3.1 Failure prediction metrics

In this section we present the classical prediction performance of the unanimous voting FPS as the number of NN's used in the classification increases to six. We also explore the learned voting mechanism to improve the rates and UC probability. The predictors use feed forward neural networks to classify samples of features into failure and ok events. The features correspond to time series data offset with a different lead-up time. The voting mechanism then takes the classifications and performs a final prediction.

Figure 4.7 shows three sample logarithmic ROC curves for the first, third, and last neuronal networks and the classification rates of the $argmax$ decision function for the first network too. Each NN was set to classify using an $argmax$ decision function from the two outputs neurons. This decision function offers only a fixed set of rates unlike the hand tuned rates that can be achieved using the ROC curve. The $argmax$ decision function offered better false positive rates and usable true positive rates in comparison to any possible choice in the ROC threshold curve. This is particular to these trained NNs and not a general case. In fact, the LSTM recurrent neural network used for voting later will use ROC tuning to choose a threshold for classification. The ROC threshold curves show that to achieve a $\boldsymbol{tpr}$ over 0.75, a $\boldsymbol{fpr}$ of at least 0.01 has to be paid as a consequence. In comparison a basic $argmax$ of both output neurons improves the classification to an $\boldsymbol{fpr}$ of 0.0027 with a $\boldsymbol{tpr}$ of 0.8461 for the first networks. All other feed forward networks have the same be behavior and use the $argmax$ decision method for classification before doing a final prediction through voting. The quality of these networks is good enough that they can be used as part of the ensemble for voting on a final prediction. Next we discuss how the rates change by voting through an increased amount of networks.

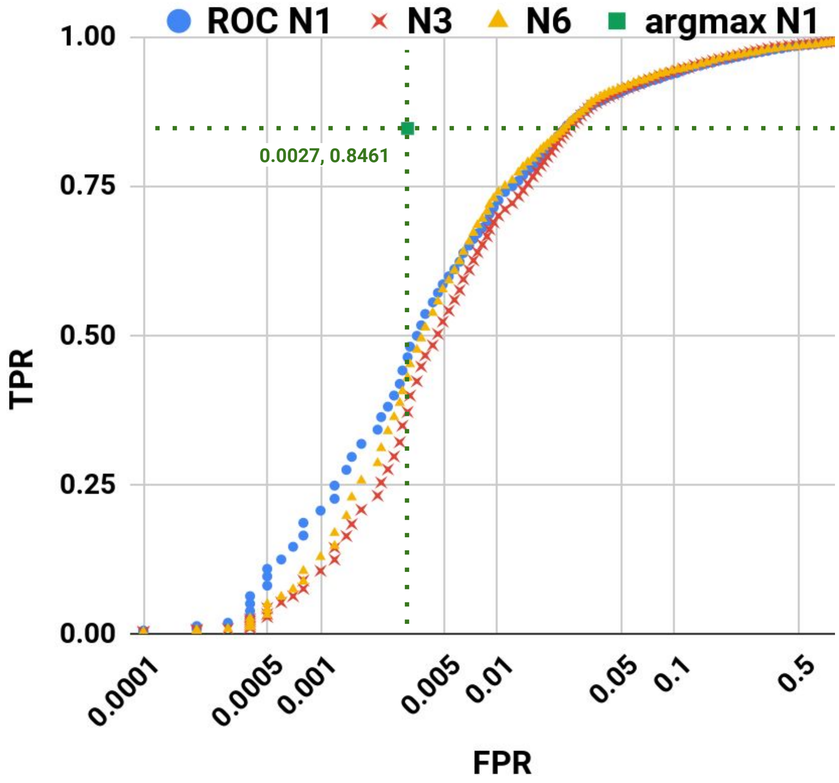

Fig. 4.7: ROC plots for networks 1,3,6 and the classification rates for network 1 when using $argmax$ as a classification decision function.

Unanimous voting FPS

Figure 4.8 shows how the classical prediction metrics change for an increasing number of NNs. Each rate value represents the prediction rate for a sequence of n networks (up to six) where all networks must agree that a classification is a true failure before final prediction of the failure. The values for a single (1*) network are equivalent to a traditional approach binary classification using one neural network (non-ensemble). Adding more networks to the sequence requires more positive classifications for a final positive prediction. The $\boldsymbol{fpr}$ term is important for the reduction of false positives that influence unnecessary checkpoints. The $\boldsymbol{fpr}$ value is reduced by an order of magnitude when using six networks compared to a single network, which allows the FPS to better cope with the FPP problem better.

The **recall** rate ($\boldsymbol{tpr}$) determines how many failures are correctly predicted, and in this approach it shows a reduction from 0.8461 to 0.6918 for six networks. This is expected, as the ability to correctly predict failures declines as more networks are added and required to agree. Not all networks agree because they were each trained with a unique time shifted set of sensor values. The reduction in $\boldsymbol{tpr}$ is

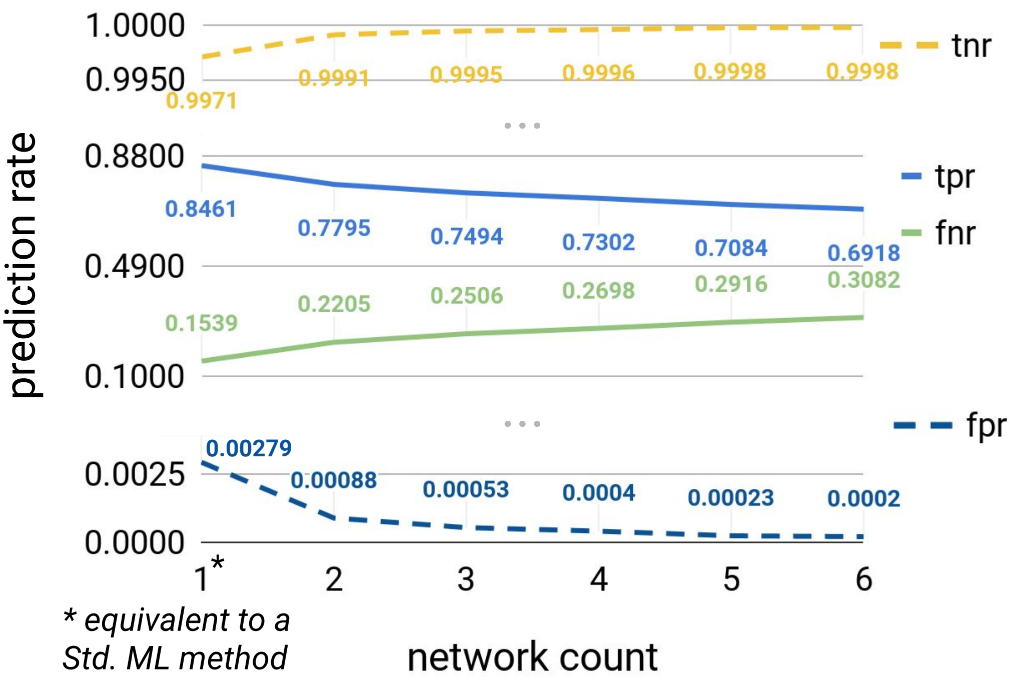

Fig. 4.8: Prediction rates fpr, tpr, tnr, fnr after evaluating all networks (up to six) using the proposed prediction procedure.

the result of the unanimous consensus requirement because the more networks are required to classify their sample as positive, the lower the probability that all will do so. This reduction can be controlled by limiting the amount of networks used. The FPS does a final negative prediction as soon as an intermediate network in the sequence classifies a negative, regardless of how many other positives classification happened, which in turn increases the negative rates tnr and fnr. These tnr increases inversely to the fpr and the fnr increases from 15.39% to 30.82% as the tpr decreases.

The next rates of interest are the precision and the false discovery rate (fdr) ($1 - precision$) from Figure 4.9. The precision increased from 96.55% to 99.70% at six networks and the fdr decreased by an order of magnitude to 0.30%. This shows that adding more networks leads to reduced false positive rates. This precision value is also competitive against the ML method [108] of Table 4.1.

Any event classified as negative within any network immediately drops out of the classification sequence because a single negative vote is enough to not consider the final prediction as positive. This is done to reduce the amount of false positives. The probability of **ok** events still being considered in the network sequence decreases after each classification. This is consistent with the fdr rate in Figure 4.9.

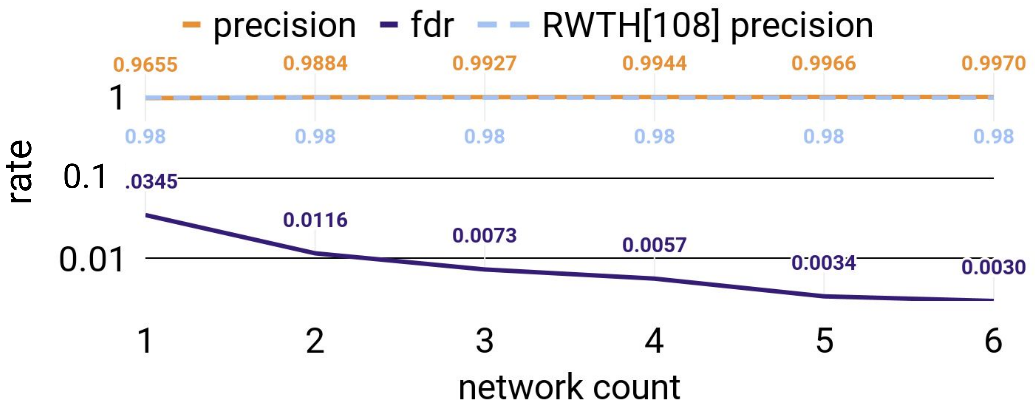

Fig. 4.9: Precision and false discovery rate $fdr = (1 - precision)$ after sequence lengths of up to six networks.

As discussed previously, the classical metrics shown above are not enough to evaluate the quality of an FPS for big HPC jobs due to the amount of false positives that would be triggered when applying a predictor to each HPC node in a parallel application. To account for this, the $\boldsymbol{fpr}$ and $\boldsymbol{tnr}$ need to be combined using Equations 4.4 and 4.2.

Unanimous voting UC metric

Here we evaluate the $\boldsymbol{UC}$ metric for HPC failure prediction systems as introduced in Section 4.1.4. This metric shows the theoretical probability of triggering an unnecessary checkpoint for a specific job size given an FPS. The more nodes, the more FPS constantly predict. If an HPC job uses thousands of nodes, the probability of at least one FPS producing a false alarm increases. The $\boldsymbol{UC}$ metric accounts for at least one FPS in n nodes producing a false positive from the encountered data values. The false positive $P(f_+)$ and true negative $P(t_-)$ probabilities are set according to the results already shown and we present the UC value according to the encounter probabilities from Equation 4.1. The values are estimated using the $\boldsymbol{fpr}$ from the previous section and the $P(K)$ and $P(E)$ of Section 4.1.

Figure 4.10 shows the decreasing $\boldsymbol{UC}$ probabilities for up to 6 networks where each count in the x-axis requires all previous networks to agree on failure predictions. The work from Klinkenberg *et al.* [108] at the RWTH is shown in dashed lines for comparison; it has a constant false positive rate and therefore a constant $\boldsymbol{UC}$ probability. The first notable feature is that the $\boldsymbol{UC}$ probabilities decrease with the amount of networks in the ensemble. When only one network is used, all node counts shown have $\boldsymbol{UC}$ probabilities higher than 50%. The 256 node count

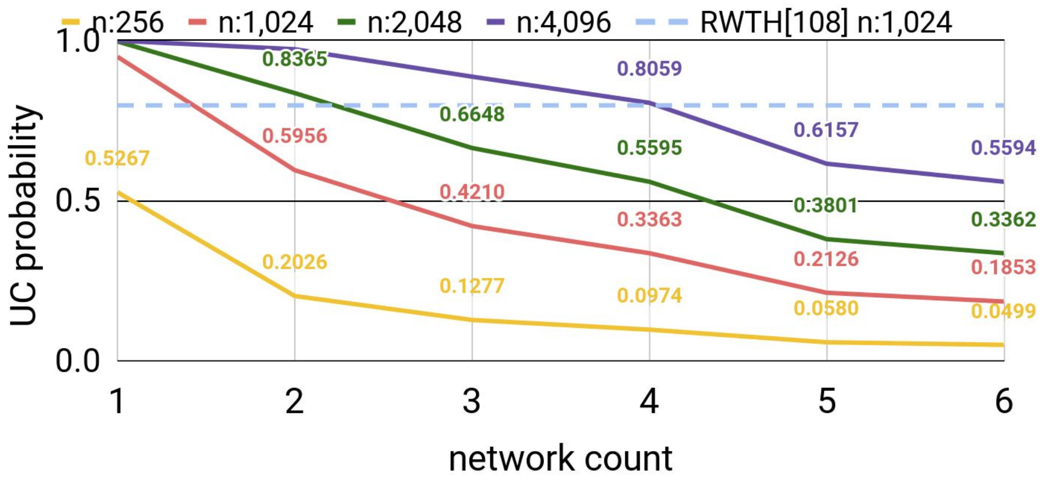

Fig. 4.10: Calculated UC probability for the ensemble NN predictor of up to six NNs.

is reduced to 20.26% at two NNs and 1,024 is reduced to 42.10% at three NNs. Probabilities bellow 50% have unnecessary checkpoints for less than half of all predictions. The 2,048 line is the highest node count that is bellow the 50% is line at a UC probability of 33.62% and with six NNs. The 256 node count has only a 4.99% UC probability, making it very good for failure prediction at the same NN amount. Unfortunately the 4,096 node count can only be reduced to 55.94% when using six neural networks. Although the ensemble method is able to reduce the UC value with more networks, the method hits a limit after 2,048 nodes (see 4,096).

Overall, the more nodes in failure prediction, the higher the UC value across the network counts becomes. It is also the case that having more networks lowers the probability of unnecessarily triggering checkpoints. The FPS is then able to reduce the UC probability of up to 2,048 nodes to a value of 33.62% with six networks.

Machine learned voting FPS

This second FPS replaces the unanimous voting with a recurrent long short term memory (LSTM) network. Changing the all-agree voting method in the predictor to a recurrent neural network can improve the UC value to usable rates from 2,048 to 8,192 nodes. Unlike the previous FPS, the RNN requires that all input NNs be present. Otherwise a new RNN would have to be trained for every permutation of basic NNs. The ROC and UC plots shown here are representative of an RNN with six NN inputs.

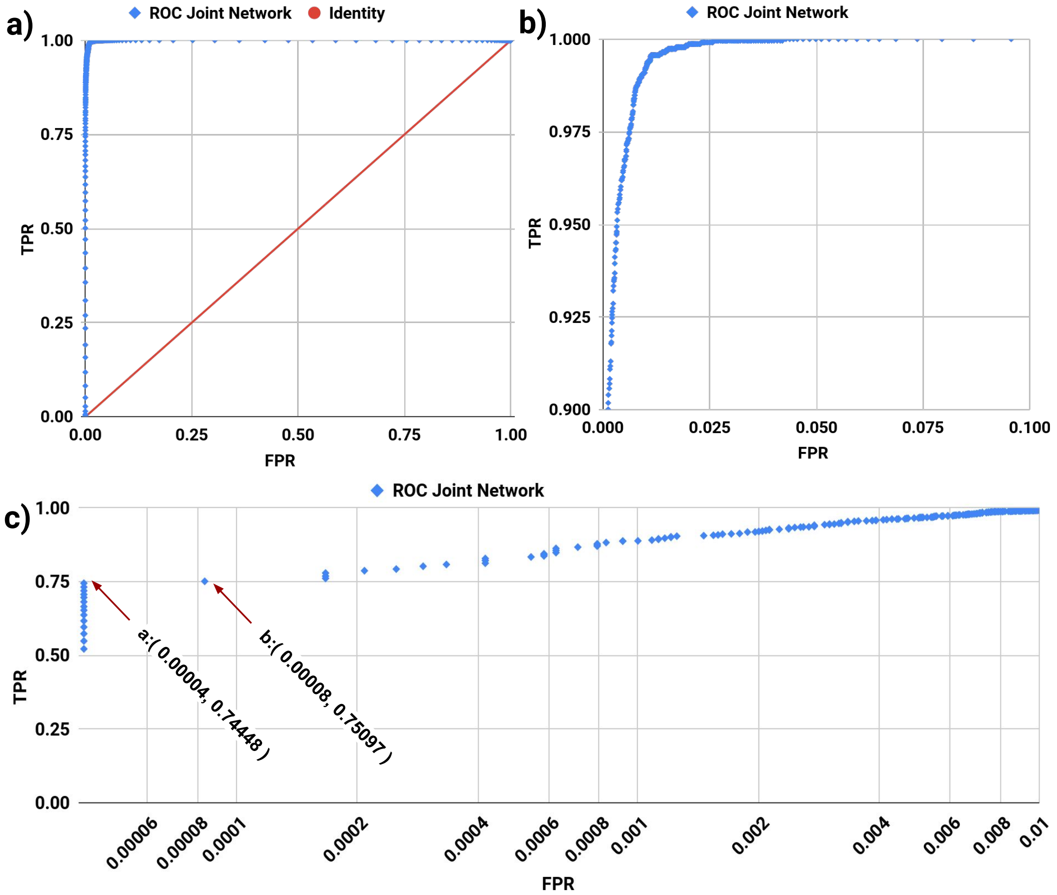

Fig. 4.11: ROC curves for an RNN voting FPS with six NN as inputs. Subplot a) shows a traditional ROC plot, b) an area of interest where the slope changes, and c) a logarithm scale version.

Figure 4.11 shows the ROC curve for the RNN voting FPS in three different ways. The good quality of the RNN is clear from sub-Figure **a)** where the curve has a sharp climb to 1.0 at low fpr. Zooming in with sub-Figure **b)** the curve change becomes more clear. An optimal classifier would have an ROC curve with a straight sharp right angle corner from $0 \longrightarrow 1.0$. Sub-Figure **c)** can be used to better help choose an optimal threshold in the ROC curve. The separation between the very low fpr values is more clear and the step wise change in them is also more pronounced. The steps are caused by the decreasing threshold (towards the right of the x-axis) that allows more events to be considered as positive, the further it is decreased. As it changes, more samples are considered positive, both false and true positives, increasing the fpr and tpr too. The first set of points that climb vertically around 0.00004 shows that the initial threshold reduction adds more true positives and no false positives. Only after jumping horizontally from the set of points at 0.00004 to the point 0.00008, false positives start being included by the threshold decision function. Based on this information, the decision threshold of the voting LSTM RNN is chosen so that the fpr and tpr are 0.00004 and 0.7448 respectively. Choosing the first point in the curve of **c)** would yield too low of a tpr for the same fpr, and choosing the second point would increase the tpr, but also increase the fpr to 0.00008.

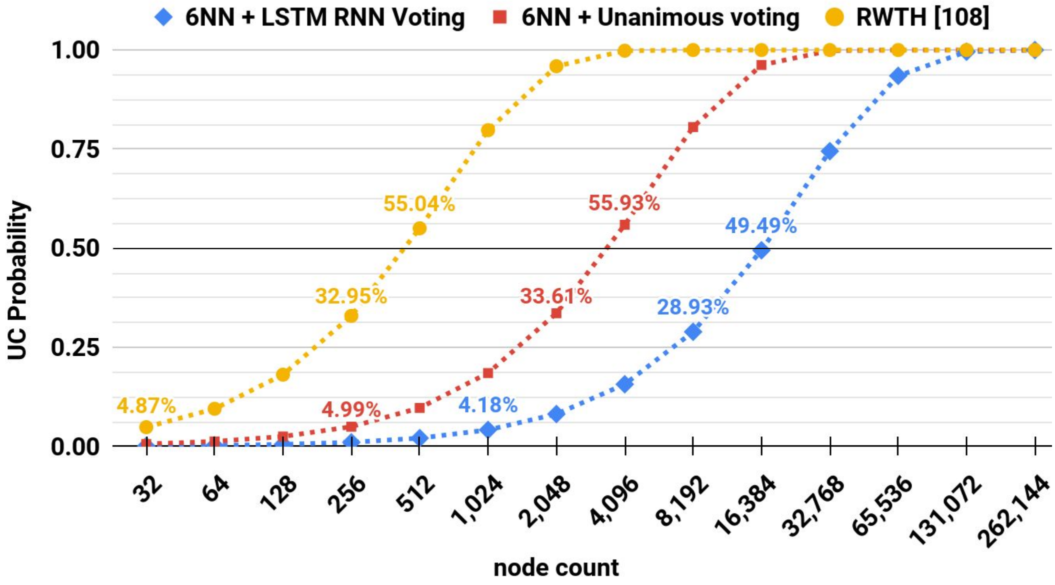

Fig. 4.12: UC probability for the FPS using an ensemble of six NNs and unanimous voting, the FPS using the LSTM RNN voting, and the best HPC work ([108]) from Table 4.1.

Figure 4.12 shows the probability of unnecessary checkpoints (UC) for three systems, the unanimous voting FPS, the LSTM RNN voting FPS, and the best alternative HPC work from Table 4.1. The Figure depicts the continuously increasing UC probability for node counts up to 2^{18}. Node counts are shown in powers of two because they are the most common HPC allocations at the MOGON HPC systems. For each system three data labels are shown at bellow 5%, around 30% and around 50%. The first set of labels at 5% shows the node count at which the FPS are very effective and with low unnecessary checkpoints, the second around 30% the point where it is still usable (one third of predictions being false positives), and the 50% shows where half of predictions are wrong and the system might be considered unusable. The RWTH system is effective at 32 nodes and quickly raises in UC probability to 32.95% at 256 nodes and becoming unusable at 512 nodes. In contrast the unanimous voting FPS keeps bellow 5% up to 256 nodes and only reaches 33.61% at 2,048 nodes. It becomes unusable at twice the node amount of 4,096. The LSTM RNN voting FPS improves the UC probability further by staying bellow 5% up to 1,024 nodes, only reaching 28.93% at 8,192 nodes, and being unusable at 16,384 nodes.

The reduction of the fpr clearly improves the UC probability and allows FPS to be used a much bigger systems. Unfortunately node counts above 2^{14} still remain unusable with an FPS at these rates. Alternative approaches can be used to further improve the UC probability across all nodes within an HPC job. One could screen nodes for failures first, and only deploy the FPS after at least one failure is recorded. This might reduce the amount of FPS needed and allow for greater node counts, but would only work at the beginning lifetime of the HPC system. As more nodes tends to fail, more FPS will be needed to avoid further failures and more nodes will need an FPS, reaching high node counts in the process. Overall the presented ensemble based FPS could be used at node counts up to 8,192 by itself.

4.3.2 Discussion of the results

High Performance Systems keep growing with an increasing amount of components and a higher risk of failures as a result. Machine learning classifiers can use hardware data and past failure events of HPC jobs to predict incoming failures. This chapter introduced the **failure prediction problem** and showed how HPC **failure predictors systems** struggle to attain low enough false positive rates. This is important for **FPS**’s that predict on thousands of nodes, where a single predicted failure warning should trigger a mitigation event like checkpoints on demand. If the fpr is not low enough, too many mitigation events would be triggered, leading to a waste of computational resources.

The chapter also proposed two **FPS** capable of predicting HPC job failures with usable **true positive** and **false positive** rates. The failure predictors are successful in classifying normalized time series of sensor data into failure and ok-state events of batch system jobs. The training data of failing events was mined by combining hardware sensors and job failure logs. The predictors manage to use this training to classify the events by using an ensemble of sequential neural networks that perform a voting procedure. Each network is trained with a different sliding window of data and the ensemble manages to reduce the false positive rates. The first FPS uses unanimous voting and the second FPS a recurrent neural network for voting. The second predictor is our proposed solution to reducing unnecessary checkpoints at nodes up to 8,192.

The tpr of the second failure predictor with RNN voting is competitive with existing ML approaches (see Table 4.1) and it can significantly outperform the existing methods based on the fpr and the UC probabilities. The main drawback of such an ensemble methods is that the tpr (recall) decreases alongside with the desired

fpr, if more NNs are used. As more NNs are used, more positives votes are required to predict a failure. This lowers the desired positive rates given by tpr and fpr. A better system would maintain a high true positive rate tpr to predict as many failures as possible. The RNN voting predictor achieved good UC values of 28.93% for 8,192 nodes, but was insufficient for 16,384 nodes at a UC value of 49.49%. All UC values are all nevertheless better than existing previous works. The tpr was good at 74.48%, which allows it to still predict a good amount of failures.

The RNN voting FPS presented in this chapter can be used to predict incoming failures with lead-up times big enough to trigger mostly necessary checkpoints. The ensemble of neural networks used could be complemented with extra and distinct classifiers for each lead-up time. This could reduce false positive rates even further. Other approaches like only using FPS in failure prone hardware could also be used to further improve the UC values.

5

Conclusion and future work

This thesis proposed three different approaches to improve utilization of resources in HPC systems by reducing waste. The first method uses a scheduling heuristic to co-allocate good pairs of parallel user applications to the same set of compute nodes. This in turn reduces the time needed to complete equivalent computational workloads by utilizing otherwise unused resources and saturating computational pipelines. The second technique proposes a novel algorithm that computes optimal checkpoint intervals for user programs with medium to low failure probabilities. These are job aware checkpoint intervals that reduce the overhead of performing checkpoints, leading to saved computational resources. The third and final approach presents a failure prediction system of incoming failures that can be used to improve the rate of triggering only necessary preemptive checkpoints. This failure predictor manages to offer good prediction rates, while keeping the amount of false failure alarms lower than previous works. The reduced false alarm rates save resources that would otherwise be used to perform unnecessary checkpoints.

Each Chapter can be summarized in more detail as follows:

Co-allocation: To study the benefits of co-allocation in Chapter 2 we tested various HPC job co-allocation techniques and implemented them in the SLURM batch scheduling algorithms to study the effects on application and batch system performance. We used five parallel MPI applications of the NERSC application set and a modified SLURM batch system for the scheduling experiments using backfill and a first-fit approach. The evaluation showed that batch scheduling combined with a co-allocation heuristic yields better computational and scheduling efficiencies than traditional exclusive allocation. On average, the computational efficiency improved by 13% and 19% and the scheduling efficiency of 22.5% and 25.2% for backfill and first fit respectively. Another finding was that the co-allocation effects are dependent on the pairing of applications, with specific ones yielding significant improvements. This knowledge was used to create a co-allocation heuristic that allows distinct applications to share nodes.

For future work the co-allocation heuristic showed evidence that a pairing matrix could be generated with a multi-class machine learning classifier that would associate potential pairs with discrete interference classes. Further studies into co-allocation of heterogeneous resources like GPUs and CPUs could also be done.

Aware checkpoint intervals: In Chapter 3 we proposed a new checkpoint interval method for jobs with medium to low failure probabilities (MPF) that was able to reduce the loss of computations in case of failures. We discussed how the runtime overhead of checkpointing depends on the interval used and how previous optimization studies have focused on jobs with a high probability of failure, e.g., long-running jobs that use a large part of the HPC system. The observations in the chapter showcased that jobs with a medium probability of failure (MPF) are common and that current approaches to interval determination fail to consider them. We therefore proposed an MPF job aware method for finding the checkpointing interval that minimizes the expected cost of checkpointing for jobs with a medium probability of failure. The aware method also arrives at the classical solutions of Daly for jobs with a high probability of failure. Another benefit is that the approach is independent of a specific failure distribution in the HPC system. The method improved previous works by using truncated moments and weighted costs to achieve accurate estimates of costs. The experiments shown use real job traces of four HPC systems, five alternative checkpoint methods, two failure distributions and six failure parameters. The savings for individual MPF jobs reached up to 40% of the checkpointing cost average. The four HPC systems studied showed cumulative checkpoint savings (of all jobs) from 7.1% to 25.7% when considering Weibull failures and between 7.3% to 27.5% for Poisson failures. The Mogon II HPC system was shown to experience average savings across methods with 18.1% checkpoint cost savings followed by the LANL Mustang system with 13.0% savings. The LANL Atlas and Curie HPC systems had 9.0% and 10.5% savings respectively. Finally the proposed method had savings between 6.0% and 26.6% when using inaccurate MTBF values, a scenario most common where accurate failure data is unavailable.

For future work it would be possible to extend the MPF aware checkpoints with knowledge of failure predictor systems. This would produce checkpoint intervals that guard applications together with FPS by taking into account failure and prediction probabilities. Furthermore, more data could be gathered about the failure characteristics of individual job types to incorporate it into more specialized checkpoint intervals.

Failure prediction: In Chapter 4 we improved upon failure prediction systems in HPC by reducing the high false positive rates that would otherwise produce excessive unnecessary overhead from checkpoints. This unnecessary checkpoint overhead was described with the false positive prediction problem and represented with a new metric UC that represents the probability of unnecessarily triggering checkpoints. The proposed failure prediction system uses a series of independently trained neural networks that issue a vote for failure prediction. Two voting systems were tested, first with unanimous voting by all networks, and a second machine learned recurrent neural network voting system for better rates. The UC metric is significantly reduced in comparison to previous works by forcing each network to agree using the first consensus method. The second approach further reduces the UC metric to thousands of nodes. The methods were trained and evaluated using a real HPC system with significant failure events. The evaluation showed that the FPS with RNN learned voting is able to reduce the UC probability for 8,192 nodes to 28.93% using six neural networks as input for voting. It also achieves a **lead-up time** of two minutes, a true positive rate of 0.7448, a false positive rate of 0.00004, and a precision rate of 0.9992.

In future work other methods could be tested to reduce the $\boldsymbol{fpr}$ further. Recurrent neural networks could be given the sensor time series data as direct input to learn the features automatically. This would produce a big LSTM network that would need a more powerful framework to train, but could also re-produce the here presented results. Alternatively, the trained networks could be test in a different system to explore the quality of failure prediction when the target hardware is not the same as the original.

List of Figures

1.1 HPC system architecture diagram with users accessing the HPC machine. The machine has multiple nodes that are connected with several networks. . . . 4

1.2 Performance of the Top 500 HPC systems over the years (source top500.org [16]). The performance of the HPC systems at position 1 and position 500 has steadily increased with time. . . . 6

1.3 Mock architecture of a two socket compute node with NUMA memory architecture, accelerators and network interconnects through PCIE, two level core cache and quad core CPUs. . . . 7

1.4 Resource utilization in a mock HPC system with one hour node allocation for applications of different node counts and runtimes. The incoming queue is shown at the top with the highest priority job on the right [6,7] and least on the left [1,3]. The first number is the amount of nodes requested and the second number the runtime requested. . . 8

1.5 Example of four applications using four nodes that consist of four CPUs. System a) shows the applications using four nodes without sharing and system b) sets two applications in two shared nodes. . . . 11

1.6 Periodic checkpointing example where a failure occurs and computations not saved are lost. . . . 12

1.7 Failure prediction example over time as a failure is avoided by a prediction that is follow by a restart on a different HPC node. . . . 14

1.8 Diagram of how binary classification sorts real classes into classified classes. The different ratios give the classical ML metrics. . . . 15

1.9 Plot depicting a Sigmoid $S(x) = \frac{x}{1+e^{-x}}$ activation function that classifies between two classes if the activation value c from $S(x)$ is above t. . . 16

1.10 Plot of an ROC curve and AUC value (0.79) from a fictional binary classifier. The horizontal axis shows the false positive rate and the vertical axis the true positive rate. The rates are plotted for all classification thresholds and an arbitrarily good threshold (high $\boldsymbol{tpr}$ and low $\boldsymbol{fpr}$) value is chosen. As the rate is decreased, more data samples are classified as positive. . . . 17

1.11 Diagram of a three layer feed forward neural network with a single hidden layer and two inputs and outputs. 19
1.12 Diagram of a mock arbitrary recurrent neural network with two inputs and one output. The recurrent links to neurons in the same or previous layers are shown in red. 20

2.1 Scheduling steps performed by a *Backfill* algorithm where backfilling is taking place to use idle resources. 29
2.2 Methods to map processes to cores in compute nodes. 31
2.3 Standard mapping, oversubscribed and co-allocation techniques for applications running on a four core node. 31
2.4 Simplified diagram of a hyper-threading pipeline used by two processes P & Q to exploit out-of-order execution. 33
2.5 Figure a) shows applications that share nodes using the same rank count. Unrestricted rank counts are avoided as shown in b) due to possible multi-job interference or non-interference of some ranks. . . . 37
2.6 Applications P and Q run to completion. App. P finishes last while Q generates load after finishing. 38
2.7 Cumulative core hour savings of co-allocation vs. STD. 45
2.8 Efficiency for *Backfill* and *First-fit* algorithms when using standard, naive and no twins. 46
2.9 Percentage of different ranks run by schedulers. 46
2.10 Average runtime of SNAP functions relative to STD. 48
2.11 Average runtime of GTC functions relative to STD. 49
2.12 Average runtime of MILC functions relative to STD. 49
2.13 Average runtime of AMG functions relative to STD. 50
2.14 Average runtime of GHOST functions relative to STD. 50

3.1 Distribution of the runtime from user jobs within the MOGON II HPC system (blue). The red curve is the cumulative distribution function of runtimes and the dotted curves the failure probability distributions for five node counts. 54
3.2 Node distribution of user jobs within the Mogon II HPC system (blue). The corresponding red curve indicates the CDF and the dotted curves the failure probability distributions. 55
3.3 Diagram showing the cyclical processes of periodic checkpoints between computation phases. In the event of a failure the application is requeued and restarted by loading the checkpoint. 59

3.4 Simulated and aware-algorithm costs of checkpointing for jobs with 60% probability of failing. The aware algorithm is shown in a black circle. 68
3.5 Simulated and aware-algorithm costs of checkpointing for jobs with 33% probability of failing. 69
3.6 Simulated and aware-algorithm costs of checkpointing for jobs with 65% probability of failing. 69
3.7 Simulated and aware-algorithm costs of checkpointing for jobs with 98% probability of failing. 71
3.8 Relative cost between alternative methods and the new aware checkpointing approach for applications with runtimes corresponding to failure probabilities of $0 < P < 1$. 72
3.9 Relative cost between alternative methods and the new aware checkpointing approach for applications with runtimes corresponding to failure probabilities of $0 < P < 1$. 73

4.1 The various curves show the probability of triggering an unnecessary migration or checkpoint due to certain false positive prediction rates in a prediction system. 86
4.2 Histograms of sensor data where orange shows anomalous values and blue expected ranges. 94
4.3 This pipeline depicts the source of the data mine, the filtering of said data and the creation of a failure prediction system through a machine learning classifier. 95
4.4 The first predictor consists of sequential classification NN with up to six networks using feature-vectors with incremental lead-up times and an all agree voting system. 98
4.5 The second predictor consists of sequential classification NN with up to six networks and a final recurrent neural network that aggregates the results from previous neural networks for a final prediction. This method is meant to improve the classification rates. 99
4.6 Example deployment of the NN predictor warning of node-level failures in an HPC system running multi-node user jobs. 100
4.7 ROC plots for networks 1,3,6 and the classification rates for network 1 when using $argmax$ as a classification decision function. 103
4.8 Prediction rates $\boldsymbol{fpr}$, $\boldsymbol{tpr}$, $\boldsymbol{tnr}$, $\boldsymbol{fnr}$ after evaluating all networks (up to six) using the proposed prediction procedure. 104
4.9 Precision and false discovery rate $fdr = (1 - precision)$ after sequence lengths of up to six networks. 105

4.10 Calculated UC probability for the ensemble NN predictor of up to six NNs. 106
4.11 ROC curves for an RNN voting FPS with six NN as inputs. Subplot a) shows a traditional ROC plot, b) an area of interest where the slope changes, and c) a logarithm scale version. 107
4.12 UC probability for the FPS using an ensemble of six NNs and unanimous voting, the FPS using the LSTM RNN voting, and the best HPC work ([108]) from Table 4.1. 108

List of Tables

2.1 Runtime and core hours for experiments up to 4,096 ranks using standard, packed, oversubscribed and co-allocation techniques. Darker shades highlight better values. 42

2.2 Cumulative core hours for running a pair of applications under STD, PCK and COA techniques. Darker shades show fewer core hours required. 44

3.1 Savings when using our approach for four HPC systems with a combination of two MTBF values $24h, 36h$ and two checkpoint costs $0.25h, 0.5h$. The simulations of a), c) and d) are done using Weibull failures with $w = 0.8$. Poisson failures are used for b). 74

4.1 Example FPS for clusters – the last six entries represent this work for different numbers of neural networks while $UC(1024)$ denotes the probability of unnecessary checkpoints for 1,024 nodes. All the rates are values $\in \{0 - 1\}$ where 0.80 would mean 80%. 87

4.2 Table showing the data collected every 15 seconds for all compute nodes. 93

4.3 Table showing the features used for each data sample. 97

Abbreviations

Abbreviation	Meaning
HPC	High Performance Computing
MPI	Message Passing Interface
RAM	Random-Access Memory
SSD	Solid State Drive
CPU	Central Processing Unit
RPM	Revolutions Per Minute
HDD	Hard Disk Drive
HPC	High Performance Computing
IB	InfiniBand
OPA	Intel OmniPath Architecture
I/O	Input/Output
PSU	Power Supply Unit
HPJ	High Priority Job
COA	CO-Allocation
SMT	Simultaneous Multi-Threading
HT	Hyper-Threading
STD	Standard
OVR	Oversubscribed
PCK	Packed
JGU	Johannes Gutenberg University
API	Application Programming Interface
IPMI	Intelligent Platform Management Interface
OS	Operating System

Abbreviation	Meaning
FPS	Failure Predictor System
TP	True Positive
TN	True Negative
FP	False Positive
FN	False negative
tpr	True Positive Rate
fpr	False Positive Rate
tnr	True Negative Rate
fnr	False Negative Rate
fdr	False Discovery Rate
ppv	Positive Predictive Value
NN	Neural Network
FPP	False Positive Problem
UC	Unnecessary Checkpoint
MC	Monte Carlo
ML	Machine Learning
MTBF	Mean Time Between Failures
CDF	Cumulative Distribution Function
MPF	Medium Probability of Failure

Bibliography

[1] Alvaro Frank, Dai Yang, André Brinkmann, Martin Schulz, and Tim Süß. "Reducing False Node Failure Predictions in HPC". In: *26th IEEE International Conference on High Performance Computing, Data, and Analytics (HiPC), Hyderabad, India, December 17-20*. 2019, pp. 323–332 (cit. on pp. vii, 72).

[2] Alvaro Frank, Tim Süß, and André Brinkmann. "Effects and Benefits of Node Sharing Strategies in HPC Batch Systems". In: *2019 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2019, Rio de Janeiro, Brazil, May 20-24, 2019*. IEEE, 2019, pp. 43–53 (cit. on p. vii).

[3] Alvaro Frank, Manuel Baumgartner, Reza Salkhordeh, and André Brinkmann. "Improving checkpointing intervals by considering individual job failure probabilities". In: *2021 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2021*. IEEE, 2021 (cit. on p. vii).

[4] Sardar Usman, Rashid Mehmood, and Iyad Katib. "Big Data and HPC Convergence for Smart Infrastructures: A Review and Proposed Architecture". In: ed. by Rashid Mehmood, Simon See, Iyad Katib, and Imrich Chlamtac. Springer International Publishing, 2020, pp. 561–586 (cit. on p. 1).

[5] Shanyu Chen, Zhipeng He, Xinyin Han, et al. "How Big Data and High-performance Computing Drive Brain Science". In: *Genom. Proteom. Bioinform.* 17.4 (2019), pp. 381–392 (cit. on p. 1).

[6] Geoffrey C. Fox, James A. Glazier, J. C. S. Kadupitiya, et al. "Learning Everywhere: Pervasive Machine Learning for Effective High-Performance Computation". In: IEEE, 2019, pp. 422–429 (cit. on p. 1).

[7] Sukeshini, Priyanka Sharma, Mohit Ved, Janaki Chintalapti, and Supriya N. Pal. "Big Data Analytics and Machine Learning Technologies for HPC Applications". In: ed. by Pradeep Kumar Singh, Arti Noor, Maheshkumar H. Kolekar, et al. Springer Singapore, 2021 (cit. on p. 1).

[8] Cheol-Ho Choi, Chaeeun Lee, Jemin Justin Lee, and Kyungho Lee. "Understanding the Deployment Cost of Cloud Computing Services for the Higher Education Institutions". In: IEEE, 2019, pp. 438–443 (cit. on p. 1).

[9] Jack J. Dongarra, Peter H. Beckman, Terry Moore, Patrick Aerts, Giovanni Aloisio, et al. "The International Exascale Software Project roadmap". In: *International Journal of High Performance Computing Applications (IJHPCA)* 25.1 (2011), pp. 3–60 (cit. on p. 1).

[10]Pavan Balaji, Darius Buntinas, David Goodell, et al. "Mpi on millions of Cores". In: *Parallel Process. Lett.* 21.1 (2011), pp. 45–60 (cit. on p. 1).

[11]Al Geist and Daniel A Reed. "A Survey of High-Performance Computing Scaling Challenges". In: *Int. J. High Perform. Comput. Appl.* 31.1 (Jan. 2017), 104–113 (cit. on p. 1).

[12]Guillaume Oger, David Le Touzé, David Guibert, et al. "On distributed memory MPI-based parallelization of SPH codes in massive HPC context". In: *Comput. Phys. Commun.* 200 (2016), pp. 1–14 (cit. on p. 1).

[13]Nosayba El-Sayed and Bianca Schroeder. "Reading between the lines of failure logs: Understanding how HPC systems fail". In: *43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Budapest, Hungary, June 24-27*. 2013, pp. 1–12 (cit. on p. 2).

[14]Franck Cappello, Al Geist, Bill Gropp, et al. "Toward Exascale Resilience". In: *International Journal of High Performance Computing Applications (IJHPCA)* 23.4 (2009), pp. 374–388 (cit. on pp. 2, 80).

[15]Franck Cappello, Al Geist, William Gropp, et al. "Toward Exascale Resilience: 2014 Update". In: *Supercomputing Frontiers and Innovations* 1.1 (Apr. 2014), pp. 5–28 (cit. on pp. 2, 80).

[16]Prometeus GmbH. *The TOP-500 list of the 500 most powerful commercially available computer systems.* "`https://www.top500.org/statistics/`". 2020 (cit. on pp. 3, 5, 6).

[17]Bertil Schmidt, Jorge Gonzalez-Dominguez, Christian Hundt, and Moritz Schlarb. *Parallel Programming: Concepts and Practice*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017 (cit. on pp. 3, 6).

[18]Gene M. Amdahl. "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities". In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. AFIPS '67 (Spring). Atlantic City, New Jersey: Association for Computing Machinery, 1967, 483–485 (cit. on p. 5).

[19]John L. Gustafson. "Reevaluating Amdahl's Law". In: *Commun. ACM* 31.5 (1988), 532–533 (cit. on p. 5).

[20]Andy B. Yoo, Morris A. Jette, and Mark Grondona. "SLURM: Simple Linux Utility for Resource Management". In: *Job Scheduling Strategies for Parallel Processing, 9th International Workshop (JSSPP), Seattle, WA, USA, June 24, Revised Papers*. 2003, pp. 44–60 (cit. on pp. 5, 9, 10).

[21]John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. 5th ed. Amsterdam: Morgan Kaufmann, 2012 (cit. on pp. 6, 7).

[22]Charng-Da Lu, James C. Browne, Robert L. DeLeon, et al. "Comprehensive job level resource usage measurement and analysis for XSEDE HPC systems". In: *Extreme Science and Engineering Discovery Environment: Gateway to Discovery, XSEDE13, San Diego, CA, USA, July 22 - 25*. 2013, 50:1–50:8 (cit. on pp. 9, 55).

[23] Jens Breitbart and Josef Weidendorfer. “Detailed Application Characterization and Its Use for Effective Co-Scheduling”. In: *Co-Scheduling of HPC Applications [extended versions of all papers from COSH@HiPEAC 2016, Prague, Czech Republic, January 19, 2016]*. 2016, pp. 69–94 (cit. on p. 9).

[24] Sergey Blagodurov, Sergey Zhuravlev, Mohammad Dashti, and Alexandra Fedorova. “A Case for NUMA-aware Contention Management on Multicore Systems”. In: *2011 USENIX Annual Technical Conference, Portland, OR, USA*. 2011 (cit. on p. 9).

[25] Andreas de Blanche and Thomas Lundqvist. “Addressing characterization methods for memory contention aware co-scheduling”. In: *The Journal of Supercomputing* 71.4 (2015), pp. 1451–1483 (cit. on p. 9).

[26] Andreas de Blanche and Thomas Lundqvist. “Terrible Twins: A Simple Scheme to Avoid Bad Co-Schedules”. In: *Proceedings of the 1st COSH Workshop on Co-Scheduling of HPC Applications (COSH@HiPEAC), Prague, Czech Republic*. 2016, pp. 25–30 (cit. on pp. 9, 25, 39, 41).

[27] Georg Birkenheuer, André Brinkmann, Jürgen Kaiser, et al. “Virtualized HPC: *a contradiction in terms?*” In: *Softw., Pract. Exper.* 42.4 (2012), pp. 485–500 (cit. on p. 10).

[28] Miguel G. Xavier, Marcelo Veiga Neves, Fabio D. Rossi, et al. “Performance Evaluation of Container-Based Virtualization for High Performance Computing Environments”. In: *21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP), Belfast, United Kingdom, February 27 - March 1*. 2013, pp. 233–240 (cit. on p. 10).

[29] Matthew J. Cordery, Brian Austin, H. J. Wassermann, et al. “Analysis of Cray XC30 Performance Using Trinity-NERSC-8 Benchmarks and Comparison with Cray XE6 and IBM BG/Q”. In: *High Performance Computing Systems. Performance Modeling, Benchmarking and Simulation - 4th International Workshop (PMBS), Denver, CO, USA*. 2013, pp. 52–72 (cit. on pp. 10, 35).

[30] William M. Jones, John T. Daly, and Nathan DeBardeleben. “Application monitoring and checkpointing in HPC: looking towards exascale systems”. In: *Proceedings of the 50th Annual Southeast Regional Conference, Tuscaloosa, AL, USA, March 29-31*. 2012, pp. 262–267 (cit. on pp. 12, 90).

[31] John W. Young. “A First Order Approximation to the Optimum Checkpoint Interval”. In: *Communications of the ACM* 17.9 (Sept. 1974), 530–531 (cit. on pp. 12, 56, 60).

[32] John T. Daly. “A higher order estimate of the optimum checkpoint interval for restart dumps”. In: *Future Generation Computing Systems (FGCS)* 22.3 (2006), pp. 303–312 (cit. on pp. 12, 56, 60, 61, 65, 67).

[33] Simon Parsons. “*Introduction to Machine Learning*, Second Editon by Ethem Alpaydin, MIT Press, 584 pp., ISBN 978-0-262-01243-0”. In: *Knowl. Eng. Rev.* 25.3 (2010), p. 353 (cit. on pp. 14, 16–20, 79).

[34] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of Machine Learning*. Adaptive computation and machine learning. MIT Press, 2012 (cit. on pp. 14, 17, 18, 20, 79, 101).

[35] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. `http://www.deeplearningbook.org`. MIT Press, 2016 (cit. on pp. 14, 19, 20, 79, 80, 101).

[36] Felix A. Gers, Jürgen A. Schmidhuber, and Fred A. Cummins. “Learning to Forget: Continual Prediction with LSTM”. In: *Neural Comput.* 12.10 (Oct. 2000), 2451–2471 (cit. on p. 20).

[37] Sepp Hochreiter and Jürgen Schmidhuber. “Long Short-Term Memory”. In: *Neural Computation* 9.8 (1997), pp. 1735–1780 (cit. on p. 20).

[38] Guillaume Aupy, Yves Robert, Frédéric Vivien, and Dounia Zaidouni. “Checkpointing Algorithms and Fault Prediction”. In: *J. Parallel Distrib. Comput.* 74.2 (Feb. 2014), pp. 2048–2064 (cit. on p. 21).

[39] JSC. *Computing Time on Supercomputers at JSC*. 2020. eprint: `https://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/ComputingTime/computingTime_node.html` (cit. on p. 23).

[40] LRZ. *Access to SuperMUC-NG*. 2020. eprint: `https://doku.lrz.de/display/PUBLIC/Access+and+Login+to+SuperMUC-NG` (cit. on p. 23).

[41] Moritz Lucius Sümmermann, Daniel Sommerhoff, and Benjamin Rott. “Mathematics in the digital age: The case of simulation-based proofs”. English. In: *International Journal of Research in Undergraduate Mathematics Education* (Feb. 2021) (cit. on p. 23).

[42] Alvaro Frank, Diego Fabregat-Traver, and Paolo Bientinesi. “Large-scale linear regression: Development of high-performance routines”. In: *Appl. Math. Comput.* 275 (2016), pp. 411–421 (cit. on p. 23).

[43] Nathan T Weeks, Glenn R Luecke, Brandon M Groth, et al. “High-performance epistasis detection in quantitative trait GWAS”. In: *The International Journal of High Performance Computing Applications* 32.3 (2018), pp. 321–336. eprint: `https://doi.org/10.1177/1094342016658110` (cit. on p. 23).

[44] Zhen-Hao Xu and Markus Meuwly. “Multistate Reactive Molecular Dynamics Simulations of Proton Diffusion in Water Clusters and in the Bulk”. In: *The Journal of Physical Chemistry B* 123.46 (2019). PMID: 31647873, pp. 9846–9861. eprint: `https://doi.org/10.1021/acs.jpcb.9b03258` (cit. on p. 23).

[45] Penporn Koanantakool, Ariful Azad, Aydin Buluç, et al. “Communication-Avoiding Parallel Sparse-Dense Matrix-Matrix Multiplication”. In: *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2016, pp. 842–853 (cit. on p. 23).

[46] James Demmel, David Eliahu, Armando Fox, et al. “Communication-Optimal Parallel Recursive Rectangular Matrix Multiplication”. In: *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. 2013, pp. 261–272 (cit. on p. 23).

[47] Abhinav Bhatele, Andrew R. Titus, Jayaraman J. Thiagarajan, et al. “Identifying the Culprits Behind Network Congestion”. In: *2015 IEEE International Parallel and Distributed Processing Symposium*. 2015, pp. 113–122 (cit. on p. 23).

[48] Mark Stillwell, Frédéric Vivien, and Henri Casanova. “Dynamic fractional resource scheduling for HPC workloads”. In: *24th IEEE International Symposium on Parallel and Distributed Processing (IPDPS), Atlanta, Georgia, USA, 19-23 April 2010 - Conference Proceedings*. 2010, pp. 1–12 (cit. on p. 24).

[49] Qingqing Xiong, Emre Ates, Martin C. Herbordt, and Ayse K. Coskun. “Tangram: Colocating HPC Applications with Oversubscription”. In: *2018 IEEE High Performance Extreme Computing Conference, HPEC 2018, Waltham, MA, USA, September 25-27, 2018*. IEEE, 2018, pp. 1–7 (cit. on p. 24).

[50] Renyu Yang, Chunming Hu, Xiaoyang Sun, et al. “Performance-Aware Speculative Resource Oversubscription for Large-Scale Clusters”. In: *IEEE Trans. Parallel Distributed Syst.* 31.7 (2020), pp. 1499–1517 (cit. on p. 25).

[51] Ruslan Kuchumov and Vladimir Korkhov. “Fair Resource Allocation for Running HPC Workloads Simultaneously”. In: *Computational Science and Its Applications - ICCSA 2019 - 19th International Conference, Saint Petersburg, Russia, July 1-4, 2019, Proceedings, Part IV*. Ed. by Sanjay Misra, Osvaldo Gervasi, Beniamino Murgante, et al. Vol. 11622. Lecture Notes in Computer Science. Springer, 2019, pp. 740–751 (cit. on p. 25).

[52] Tau Leng, Rizwan Ali, Jenwei Hsieh, Victor Mashayekhi, and Reza Rooholamini. “An Empirical Study of Hyper-Threading in High Performance Computing Clusters”. In: *Proc. of the Third LCI International Conference on Linux Clusters: The HPC Revolution*. 2002 (cit. on p. 25).

[53] Florian Wende, Thomas Steinke, and Alexander Reinefeld. “The Impact of Process Placement and Oversubscription on Application Performance: A Case Study for Exascale Computing”. In: *Proceedings of the 3rd International Conference on Exascale Applications and Software*. EASC ’15. Edinburgh, UK: University of Edinburgh, 2015, 13–18 (cit. on p. 25).

[54] Costin Iancu, Steven A. Hofmeyr, Filip Blagojevic, and Yili Zheng. “Oversubscription on multicore processors”. In: *24th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010, Atlanta, Georgia, USA, 19-23 April 2010 - Conference Proceedings*. IEEE, 2010, pp. 1–11 (cit. on pp. 26, 30, 32).

[55] François Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, et al. “hwloc: A Generic Framework for Managing Hardware Affinities in HPC Applications”. In: *Proceedings of the 18th Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP), Pisa, Italy, February 17-19*. 2010, pp. 180 –186 (cit. on p. 26).

[56] Dmitry N. Zotkin and Peter J. Keleher. “Job-Length Estimation and Performance in Backfilling Schedulers”. In: *Proceedings of the Eighth IEEE International Symposium on High Performance Distributed Computing (HPDC), Redondo Beach, California, USA, August 3-6*. 1999, pp. 236–243 (cit. on p. 26).

[57] David Talby and Dror G. Feitelson. “Supporting Priorities and Improving Utilization of the IBM SP Scheduler Using Slack-Based Backfilling”. In: *13th International Parallel Processing Symposium / 10th Symposium on Parallel and Distributed Processing (IPPS / SPDP), 12-16 April, San Juan, Puerto Rico, Proceedings*. 1999, pp. 513–517 (cit. on p. 26).

[58]Joanna Józefowska and Jan Weglarz. “On a methodology for discrete-continuous scheduling”. In: *European Journal of Operational Research* 107.2 (1998), pp. 338–353 (cit. on p. 26).

[59]Adam Janiak, Władysław Janiak, and Maciej Lichtenstein. “Resource Management in Machine Scheduling Problems: A Survey”. In: *Decision Making in Manufacturing and Services* 1.12 (2007), pp. 59–89 (cit. on p. 26).

[60]André Brinkmann, Peter Kling, Friedhelm Meyer auf der Heide, et al. “Scheduling shared continuous resources on many-cores”. In: *26th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), Prague, Czech Republic*. 2014, pp. 128–137 (cit. on p. 26).

[61]Tim Süß, Nils Döring, Ramy Gad, et al. “Impact of the Scheduling Strategy in Heterogeneous Systems That Provide Co-Scheduling”. In: *Proceedings of the 1st COSH Workshop on Co-Scheduling of HPC Applications (COSH), Prague, Czech Republic*. 2016, pp. 37–42 (cit. on p. 26).

[62]Tim Süß, Nils Döring, Ramy Gad, et al. “VarySched: A Framework for Variable Scheduling in Heterogeneous Environments”. In: *2016 IEEE International Conference on Cluster Computing (CLUSTER), Taipei, Taiwan*. 2016, pp. 489–492 (cit. on p. 26).

[63]Ronald L. Graham. “Bounds on multiprocessing timing anomalies”. In: *SIAM journal on Applied Mathematics* 17.2 (1969), pp. 416–429 (cit. on p. 27).

[64]Federico Della Croce and Rosario Scatamacchia. “Longest Processing Time rule for identical parallel machines revisited”. In: *CoRR* abs/1801.05489 (2018). arXiv: `1801.05489` (cit. on p. 27).

[65]Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. USA: W. H. Freeman Co., 1990 (cit. on p. 27).

[66]Chandra Chekuri and Sanjeev Khanna. “On Multidimensional Packing Problems”. In: *SIAM J. Comput.* 33.4 (2004), pp. 837–851 (cit. on p. 27).

[67]Matthias Hovestadt, Odej Kao, Axel Keller, and Achim Streit. “Scheduling in HPC Resource Management Systems: Queuing vs. Planning”. In: *Job Scheduling Strategies for Parallel Processing*. Ed. by Dror Feitelson, Larry Rudolph, and Uwe Schwiegelshohn. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 1–20 (cit. on p. 28).

[68]Simon Hammond, Courtenay Vaughan, and Clay Hughes. “Evaluating the Intel Skylake Xeon processor for HPC workloads”. In: *2018 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE. 2018, pp. 342–349 (cit. on p. 34).

[69]Edgar Gabriel, Graham E. Fagg, George Bosilca, et al. “Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation”. In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 11th European PVM/MPI Users’ Group Meeting, Budapest, Hungary, September 19-22, Proceedings*. 2004, pp. 97–104 (cit. on p. 34).

[70]Katie Antypas, John Shalf, and Harvey Wasserman. “NERSC-6 Workload Analysis and Benchmark Selection Process”. In: (Aug. 2008) (cit. on p. 35).

[71] J. S. Katie Antypas and H. Wasserman. *Nersc-6 workload analysis and benchmark selection process. Technical Report LBNL-1014*. Tech. rep. Berkeley Lab CA., 2008 (cit. on p. 35).

[72] George Amvrosiadis, Jun Woo Park, Gregory R. Ganger, et al. *Bigger, Longer, Fewer: what do cluster jobs look like outside Google?* Carnegie Mellon University Parallel Data Lab Technical Report CMU-PDL-17-104, see `https://www.pdl.cmu.edu/ATLAS`. 2017 (cit. on p. 55).

[73] Mohamed-Slim Bouguerra, Thierry Gautier, Denis Trystram, and Jean-Marc Vincent. "A Flexible Checkpoint/Restart Model in Distributed Systems". In: *8th International Conference on Parallel Processing and Applied Mathematics (PPAM), Wroclaw, Poland, September 13-16*. Vol. 6067. 2009, pp. 206–215 (cit. on pp. 56, 67).

[74] Marin Bougeret, Henri Casanova, Mikaël Rabie, Yves Robert, and Frédéric Vivien. "Checkpointing strategies for parallel jobs". In: *Conference on High Performance Computing Networking, Storage and Analysis (SC), Seattle, WA, USA, November 12-18*. 2011, 33:1–33:11 (cit. on pp. 57, 67).

[75] Hui Jin, Yong Chen, Huaiyu Zhu, and Xian-He Sun. "Optimizing HPC Fault-Tolerant Environment: An Analytical Approach". In: *39th International Conference on Parallel Processing (ICPP), San Diego, California, USA, 13-16 September*. 2010, pp. 525–534 (cit. on pp. 57, 61).

[76] Omer Subasi, Gokcen Kestor, and Sriram Krishnamoorthy. "Toward a General Theory of Optimal Checkpoint Placement". In: *2017 IEEE International Conference on Cluster Computing (CLUSTER), Honolulu, HI, USA, September 5-8*. 2017, pp. 464–474 (cit. on pp. 57, 67).

[77] Noriaki Bessho and Tadashi Dohi. "Comparing Checkpoint and Rollback Recovery Schemes in a Cluster System". In: *12th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP), Fukuoka, Japan, September 4-7*. 2012, pp. 531–545 (cit. on p. 57).

[78] Guillaume Aupy, Yves Robert, and Frédéric Vivien. "Assuming Failure Independence: Are We Right to be Wrong?" In: *IEEE International Conference on Cluster Computing (CLUSTER), Honolulu, HI, USA, September 5-8*. 2017, pp. 709–716 (cit. on p. 57).

[79] Thomas Hérault, Yves Robert, Aurélien Bouteiller, et al. "Optimal Cooperative Checkpointing for Shared High-Performance Computing Platforms". In: *2018 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPS Workshops 2018, Vancouver, BC, Canada, May 21-25*. 2018, pp. 803–812 (cit. on pp. 57, 61, 67).

[80] Devesh Tiwari, Saurabh Gupta, and Sudharshan S. Vazhkudai. "Lazy Checkpointing: Exploiting Temporal Locality in Failures to Mitigate Checkpointing Overheads on Extreme-Scale Systems". In: *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Atlanta, GA, USA, June 23-26*. 2014, pp. 25–36 (cit. on pp. 57, 61, 67).

[81] Nosayba El-Sayed and Bianca Schroeder. “Checkpoint/restart in practice: When ’simple is better’”. In: *2014 IEEE International Conference on Cluster Computing (CLUSTER), Madrid, Spain, September 22-26*. IEEE Computer Society, 2014, pp. 84–92 (cit. on p. 58).

[82] Sachini Jayasekara, Aaron Harwood, and Shanika Karunasekera. “A utilization model for optimization of checkpoint intervals in distributed stream processing systems”. In: *Future Gener. Comput. Syst.* 110 (2020), pp. 68–79 (cit. on pp. 58, 61, 67).

[83] Saurabh Gupta, Tirthak Patel, Christian Engelmann, and Devesh Tiwari. “Failures in large scale systems: long-term measurement, analysis, and implications”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC), Denver, CO, USA, November 12 - 17*. 2017, 44:1–44:12 (cit. on pp. 58, 65, 67).

[84] Bianca Schroeder and Garth A. Gibson. “A Large-Scale Study of Failures in High-Performance Computing Systems”. In: *IEEE Trans. Dependable Secur. Comput.* 7.4 (2010), pp. 337–351 (cit. on pp. 60, 65).

[85] Robert P. McEwen and Bernard R. Parresol. “Moment expressions and summary statistics for the complete and truncated weibull distribution”. In: *Communications in Statistics - Theory and Methods* 20.4 (1991), pp. 1361–1372 (cit. on pp. 64, 66).

[86] Eric Martin Heien, Derrick Kondo, Ana Gainaru, et al. “Modeling and tolerating heterogeneous failures in large parallel systems”. In: *Conference on High Performance Computing Networking, Storage and Analysis (SC), Seattle, WA, USA, November 12-18*. 2011, 45:1–45:11 (cit. on p. 65).

[87] James W. Jawitz. “Moments of truncated continuous univariate distributions”. In: *Advances in Water Resources* 27.3 (2004), pp. 269 –281 (cit. on p. 66).

[88] Yulai Yuan, Yongwei Wu, Qiuping Wang, Guangwen Yang, and Weimin Zheng. “Job failures in high performance computing systems: A large-scale empirical study”. In: *Computers & Mathematics with Applications* 63.2 (2012), pp. 365–377 (cit. on p. 67).

[89] George Amvrosiadis, Jun Woo Park, Gregory R. Ganger, et al. “On the diversity of cluster workloads and its impact on research results”. In: *USENIX Annual Technical Conference (ATC), Boston, MA, USA, July 11-13*. 2018, pp. 533–546 (cit. on p. 72).

[90] Joseph Emeras. *The cea curie log*. `https://www.cs.huji.ac.il/labs/parallel/workload/l_cea_curie`. 2012 (cit. on p. 72).

[91] G. Peter Zhang. “A neural network ensemble method with jittered training data for time series forecasting”. In: *Inf. Sci.* 177.23 (2007), pp. 5329–5346 (cit. on p. 80).

[92] Nikolaos Kourentzes, Devon K. Barrow, and Sven F. Crone. “Neural network ensemble operators for time series forecasting”. In: *Expert Syst. Appl.* 41.9 (2014), pp. 4235–4244 (cit. on p. 80).

[93] Malathy Emperuman and Srimathi Chandrasekaran. “Hybrid Continuous Density Hmm-Based Ensemble Neural Networks for Sensor Fault Detection and Classification in Wireless Sensor Network”. In: *Sensors* 20.3 (2020), p. 745 (cit. on p. 80).

[94] Yang Liu, Xunshi Yan, Chen-An Zhang, and Wen Liu. “An Ensemble Convolutional Neural Networks for Bearing Fault Diagnosis Using Multi-Sensor Data”. In: *Sensors* 19.23 (2019), p. 5300 (cit. on p. 80).

[95] Yongsu Jeon, Beomjun Kim, and Yunju Baek. “Ensemble CNN to Detect Drowsy Driving with In-Vehicle Sensor Data”. In: *Sensors* 21.7 (2021), p. 2372 (cit. on p. 80).

[96] Qiang Guan, Ziming Zhang, and Song Fu. “Ensemble of Bayesian Predictors and Decision Trees for Proactive Failure Management in Cloud Computing Systems”. In: *J. Commun.* 7.1 (2012), pp. 52–61 (cit. on pp. 80, 81, 87, 88).

[97] Anas A. Hadi and F. Fouz. “Detecting Failures in HPC Storage Nodes”. In: *International journal of scientific and engineering research* 7 (2016), pp. 224–229 (cit. on pp. 80, 81, 87, 88).

[98] Mikel Galar, Alberto Fernández, Edurne Barrenechea Tartas, Humberto Bustince Sola, and Francisco Herrera. “An overview of ensemble methods for binary classifiers in multi-class problems: Experimental study on one-vs-one and one-vs-all schemes”. In: *Pattern Recognit.* 44.8 (2011), pp. 1761–1776 (cit. on p. 80).

[99] Omer Sagi and Lior Rokach. “Ensemble learning: A survey”. In: *Wiley Interdiscip. Rev. Data Min. Knowl. Discov.* 8.4 (2018) (cit. on p. 80).

[100] Thomas G. Dietterich. “Ensemble Methods in Machine Learning”. In: *Multiple Classifier Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 1–15 (cit. on p. 80).

[101] Francesco Beneventi, Andrea Bartolini, Carlo Cavazzoni, and Luca Benini. “Continuous learning of HPC infrastructure models using big data analytics and in-memory processing tools”. In: *Design, Automation & Test in Europe Conference & Exhibition, DATE 2017, Lausanne, Switzerland, March 27-31, 2017*. Ed. by David Atienza and Giorgio Di Natale. IEEE, 2017, pp. 1038–1043 (cit. on p. 80).

[102] Chokchai Leangsuksun, Tirumala Rao, Anand Tikotekar, et al. “IPMI-based Efficient Notification Framework for Large Scale Cluster Computing”. In: *Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2006), 16-19 May 2006, Singapore*. IEEE Computer Society, 2006, p. 23 (cit. on p. 80).

[103] Marc-Andre Vef, Nafiseh Moti, Tim Süß, et al. “GekkoFS - A Temporary Burst Buffer File System for HPC Applications”. In: *J. Comput. Sci. Technol.* 35.1 (2020), pp. 72–91 (cit. on p. 81).

[104] Marc-André Vef, Rebecca Steiner, Reza Salkhordeh, et al. “DelveFS - An Event-Driven Semantic File System for Object Stores”. In: *IEEE International Conference on Cluster Computing, CLUSTER 2020, Kobe, Japan, September 14-17, 2020*. IEEE, 2020, pp. 35–46 (cit. on p. 81).

[105] André Brinkmann, Kathryn Mohror, Weikuan Yu, et al. “Ad Hoc File Systems for High-Performance Computing”. In: *J. Comput. Sci. Technol.* 35.1 (2020), pp. 4–26 (cit. on p. 81).

[106] A. Gainaru, F. Cappello, and W. Kramer. “Taming of the Shrew: Modeling the Normal and Faulty Behaviour of Large-scale HPC Systems”. In: *2012 IEEE 26th International Parallel and Distributed Processing Symposium*. 2012, pp. 1168–1179 (cit. on p. 81).

[107] Anwesha Das, Frank Mueller, Charles Siegel, and Abhinav Vishnu. “Desh: deep learning for system health prediction of lead times to failure in HPC”. In: *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing (HPCD), Tempe, AZ, USA, June 11-15*. 2018, pp. 40–51 (cit. on p. 81).

[108] J. Klinkenberg, C. Terboven, S. Lankes, and M. S. Müller. “Data Mining-Based Analysis of HPC Center Operations”. In: *2017 IEEE International Conference on Cluster Computing (CLUSTER)*. 2017, pp. 766–773 (cit. on pp. 81, 84, 86, 87, 104, 105, 108).

[109] Qi Dou, Hao Chen, Lequan Yu, Jing Qin, and Pheng-Ann Heng. “Multilevel Contextual 3-D CNNs for False Positive Reduction in Pulmonary Nodule Detection”. In: *IEEE Transactions on Biomedical Engineering* 64.7 (2017), pp. 1558–1567 (cit. on p. 81).

[110] Grigoris Karakoulas and John Shawe-Taylor. “Optimizing Classifiers for Imbalanced Training Sets”. In: *Proceedings of the 11th International Conference on Neural Information Processing Systems*. NIPS’98. Denver, CO: MIT Press, 1998, 253–259 (cit. on p. 81).

[111] Nitesh V. Chawla. “Data Mining for Imbalanced Datasets: An Overview”. In: *Data Mining and Knowledge Discovery Handbook*. Ed. by Oded Maimon and Lior Rokach. Boston, MA: Springer US, 2010, pp. 875–886 (cit. on p. 81).

[112] R.J. Lyon, J.M. Brooke, J.D. Knowles, and B.W. Stappers. “A Study on Classification in Imbalanced and Partially-Labelled Data Streams”. In: *2013 IEEE International Conference on Systems, Man, and Cybernetics*. 2013, pp. 1506–1511 (cit. on p. 81).

[113] Shan-Hung Wu, Keng-Pei Lin, Hao-Heng Chien, Chung-Min Chen, and Ming-Syan Chen. “On Generalizable Low False-Positive Learning Using Asymmetric Support Vector Machines”. In: *IEEE Transactions on Knowledge and Data Engineering* 25.5 (2013), pp. 1083–1096 (cit. on p. 81).

[114] Ana Gainaru, Franck Cappello, Marc Snir, and William Kramer. “Fault prediction under the microscope: a closer look into HPC systems”. In: *SC Conference on High Performance Computing Networking, Storage and Analysis, SC ’12, Salt Lake City, UT, USA - November 11 - 15, 2012*. Ed. by Jeffrey K. Hollingsworth. IEEE/ACM, 2012, p. 77 (cit. on pp. 82, 92).

[115] Daniel Moraes, Jacques Wainer, and Anderson Rocha. “Low false positive learning with support vector machines”. In: *Journal of Visual Communication and Image Representation* 38 (2016), pp. 340–350 (cit. on p. 82).

[116] David Jauk, Dai Yang, and Martin Schulz. “Predicting Faults in High Performance Computing Systems: An In-Depth Survey of the State-of-the-Practice”. In: *The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC ’19)*. Accepted for publication. Denver, CO, USA: ACM, 2019, t.b.d. (Cit. on pp. 82, 85).

[117] Felix Salfner, Maren Lenk, and Miroslaw Malek. “A Survey of Online Failure Prediction Methods”. In: *ACM Comput. Surv.* 42.3 (Mar. 2010), 10:1–10:42 (cit. on pp. 82, 85).

[118] Z. Zheng, Z. Lan, R. Gupta, S. Coghlan, and P. Beckman. “A practical failure prediction with location and lead time for Blue Gene/P”. In: *2010 International Conference on Dependable Systems and Networks Workshops (DSN-W)*. 2010, pp. 15–22 (cit. on pp. 82, 85).

[119] N. Taerat, C. Leangsuksun, C. Chandler, and N. Naksinehaboon. “Proficiency Metrics for Failure Prediction in High Performance Computing”. In: *International Symposium on Parallel and Distributed Processing with Applications*. 2010, pp. 491–498 (cit. on pp. 82, 85).

[120] Simon Pickartz, Ramy Gad, Stefan Lankes, et al. “Migration Techniques in HPC Environments”. In: *Euro-Par 2014: Parallel Processing Workshops - Euro-Par 2014 International Workshops, Porto, Portugal*. 2014, pp. 486–497 (cit. on p. 83).

[121] Simon Pickartz, Stefan Lankes, Antonello Monti, Carsten Clauss, and Jens Breitbart. “Application migration in HPC - A driver of the exascale era?” In: *International Conference on High Performance Computing & Simulation, HPCS 2016, Innsbruck, Austria, July 18-22, 2016*. IEEE, 2016, pp. 318–325 (cit. on p. 83).

[122] Ramy Gad, Simon Pickartz, Tim Süß, et al. “Zeroing memory deallocator to reduce checkpoint sizes in virtualized HPC environments”. In: *The Journal of Supercomputing* 74.11 (2018), pp. 6236–6257 (cit. on pp. 83, 99).

[123] Ramy Gad, Simon Pickartz, Tim Süß, et al. “Accelerating Application Migration in HPC”. In: *High Performance Computing - ISC High Performance 2016 International Workshops, ExaComm, E-MuCoCoS, HPC-IODC, IXPUG, IWOPH, P^3MA, VHPC, WOPSSS, Frankfurt, Germany, June 19-23, 2016, Revised Selected Papers*. Vol. 9945. 2016, pp. 663–673 (cit. on p. 83).

[124] C. Wang, F. Mueller, C. Engelmann, and S. L. Scott. “Proactive process-level live migration in HPC environments”. In: *SC ’08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. 2008, pp. 1–12 (cit. on pp. 83, 90, 99).

[125] Grégoire Montavon, Wojciech Samek, and Klaus-Robert Müller. “Methods for interpreting and understanding deep neural networks”. In: *Digital Signal Processing* 73 (2018), pp. 1–15 (cit. on p. 84).

[126] “Illuminating the “black box”: a randomization approach for understanding variable contributions in artificial neural networks”. In: *Ecological Modelling* 154.1 (2002), pp. 135–150 (cit. on p. 84).

[127] Pavlo Molchanov, Arun Mallya, Stephen Tyree, Iuri Frosio, and Jan Kautz. “Importance Estimation for Neural Network Pruning”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 2019 (cit. on p. 84).

[128] Tom Fawcett. “An Introduction to ROC Analysis”. In: *Pattern Recogn. Lett.* 27.8 (June 2006), pp. 861–874 (cit. on p. 85).

[129] Nosayba El-Sayed, Hongyu Zhu, and Bianca Schroeder. “Learning from Failure Across Multiple Clusters: A Trace-Driven Approach to Understanding, Predicting, and Mitigating Job Terminations”. In: *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, May 2017, pp. 1333–1344 (cit. on pp. 85, 87).

[130] A. Pelaez, A. Quiroz, J. C. Browne, E. Chuah, and M. Parashar. “Online failure prediction for HPC resources using decentralized clustering”. In: *2014 21st International Conference on High Performance Computing (HiPC)*. 2014, pp. 1–9 (cit. on p. 87).

[131] Bin Nie, Ji Xue, Saurabh Gupta, et al. “Characterizing Temperature, Power, and Soft-Error Behaviors in Data Center Systems: Insights, Challenges, and Opportunities”. In: *2017 IEEE 25th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, Aug. 2017, pp. 22–31 (cit. on p. 87).

[132] Y. Liang, Y. Zhang, H. Xiong, and R. Sahoo. “Failure Prediction in IBM BlueGene/L Event Logs”. In: *Seventh IEEE International Conference on Data Mining (ICDM 2007)*. 2007, pp. 583–588 (cit. on p. 87).

[133] L. Yu, Z. Zheng, Z. Lan, and S. Coghlan. “Practical online failure prediction for Blue Gene/P: Period-based vs event-driven”. In: *2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (DSN-W)*. 2011, pp. 259–264 (cit. on p. 87).

[134] Anwesha Das, Frank Mueller, Paul Hargrove, Eric Roman, and Scott B. Baden. “Doomsday: predicting which node will fail when on supercomputers”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC), Dallas, TX, USA, November 11-16*. 2018, 9:1–9:14 (cit. on p. 87).

[135] Tariqul Islam and Dakshnamoorthy Manivannan. “Predicting Application Failure in Cloud: A Machine Learning Approach”. In: *2017 IEEE International Conference on Cognitive Computing (ICCC)*. IEEE, June 2017, pp. 24–31 (cit. on p. 87).

[136] Mbarka Soualhia, Foutse Khomh, and Sofiene Tahar. “Predicting Scheduling Failures in the Cloud: A Case Study with Google Clusters and Hadoop on Amazon EMR”. In: *2015 IEEE 17th International Conference on High Performance Computing and Communications (HPCC)*. IEEE, Aug. 2015, pp. 58–65 (cit. on p. 87).

[137] Bikash Agrawal, Tomasz Wiktorski, and Chunming Rong. “Analyzing and Predicting Failure in Hadoop Clusters Using Distributed Hidden Markov Model”. In: *Cloud Computing and Big Data - Second International Conference (CloudCom-Asia), Huangshan, China, June 17-19, Revised Selected Papers*. 2015, pp. 232–246 (cit. on p. 87).

[138] Jens Lienig and Hans Bruemmer. “Reliability Analysis”. In: *Fundamentals of Electronic Systems Design*. Cham: Springer International Publishing, 2017, pp. 45–73 (cit. on p. 88).

[139] Earl C Joseph, Robert Sorensen, Steve Conway, and Kevin Monroe. *Analysis of the Characteristics and Development Trends of the Next-Generation of Supercomputers in Foreign Countries*. Tech. rep. 2016 (cit. on p. 88).

[140] Chao Wang, Frank Mueller, Christian Engelmann, and Stephen L. Scott. “Hybrid Checkpointing for MPI Jobs in HPC Environments”. In: *16th IEEE International Conference on Parallel and Distributed Systems (ICPADS), Shanghai, China, December 8-10*. 2010, pp. 524–533 (cit. on p. 90).

[141] James Elliott, Kishor Kharbas, David Fiala, et al. “Combining Partial Redundancy and Checkpointing for HPC”. In: *2012 IEEE 32nd International Conference on Distributed Computing Systems (ICDCS), Macau, China, June 18-21*. 2012, pp. 615–626 (cit. on p. 90).

[142] Chao Wang, Frank Mueller, Christian Engelmann, and Stephen L Scott. “Proactive process-level live migration and back migration in HPC environments”. In: *Journal of Parallel and Distributed Computing* 72.2 (2012), pp. 254–267 (cit. on p. 90).

[143] Simon Pickartz, Carsten Clauss, Jens Breitbart, Stefan Lankes, and Antonello Monti. “Prospects and challenges of virtual machine migration in HPC”. In: *Concurrency and Computation: Practice and Experience* 30.9 (2018), e4412 (cit. on p. 90).

[144] George P Wadsworth. *Introduction to probability and random variables*. Tech. rep. 1960 (cit. on p. 90).

[145] Sriram Sankaran, Jeffrey M. Squyres, Brian Barrett, et al. “The Lam/Mpi Checkpoint/Restart Framework: System-Initiated Checkpointing”. In: *The International Journal of High Performance Computing Applications* 19.4 (2005), pp. 479–493 (cit. on p. 101).

[146] Paul H Hargrove and Jason C Duell. “Berkeley lab checkpoint/restart (blcr) for linux clusters”. In: *in In Proceedings of SciDAC 2006*. 2006 (cit. on p. 101).

[147] LLNL. *SCR: Scalable Checkpoint/Restart for MPI*. 2020 (cit. on p. 101).

[148] Tobi Oetiker. *RRDtool*. accessed on 28.06.2019. 2005 (cit. on p. 101).

[149] Tom Schaul, Justin Bayer, Daan Wierstra, et al. “PyBrain”. In: *Journal of Machine Learning Research* 11 (2010), pp. 743–746 (cit. on p. 101).

[150] *MOGON I and MOGON II HPC Cluster from the ZDV at the Johannes Gutenberg-Universität Mainz*. `https://hpc.uni-mainz.de/high-performance-computing/systeme/`. Last Accessed: 11-11-2021 (cit. on p. 101).

[151] Martín Abadi, Ashish Agarwal, Paul Barham, et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015 (cit. on p. 102).

[152] Adam Paszke, Sam Gross, Francisco Massa, et al. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach, H. Larochelle, A. Beygelzimer, et al. Curran Associates, Inc., 2019, pp. 8024–8035 (cit. on p. 102).

[153] François Chollet et al. *Keras*. `https://keras.io`. 2015 (cit. on p. 102).

Declaration

I hereby declare that this thesis was written by me and is my own original work in accordance with the details on page vii.

Mainz, 12 September 2022

Alvaro Frank

Curriculum vitae

Personal Data

Name:	Alvaro Frank
Date of Birth:	08 October 1986
Place of Birth:	San Jose, Costa Rica

Work Experience

2021	Wissenschaftlicher MA, Interactive HPC RWTH Universität Aachen
2015-2021	Wissenschaftlicher MA, HPC Johannes Gutenberg Universität Mainz
2014	W. MA., Software Engineer, HPC Genomics Helmholtz Center München
2010-2011	Simulation Engineer, ASIC Hardware Hewlett Packard Enterprise

Education

2015-2022	Dr. rer. nat., Computer Science Johannes Gutenberg Universität Mainz
2011-2013	M.Sc., Simulation Sciences RWTH Universität Aachen
2007-2010	B.Sc., Computer Science Universidad de Costa Rica
1992-2005	Abitur & Highschool Deutsche Humboldt Schule, Costa Rica