# Heterogeneity Aware Fault Tolerance for Extreme Scale Computing

by

**Zaeem Hussain**

Masters in Computer Science, Lahore University of Management Sciences, 2014

Submitted to the Graduate Faculty of
the Department of Computer Science in partial fulfillment
of the requirements for the degree of

**Doctor of Philosophy**

University of Pittsburgh

2020

UNIVERSITY OF PITTSBURGH

SCHOOL OF COMPUTING AND INFORMATION

This dissertation was presented

by

Zaeem Hussain

It was defended on

July 17th 2020

and approved by

Rami Melhem, Department of Computer Science

Taieb Znati, Department of Computer Science

Daniel Mosse, Department of Computer Science

Balaji Palanisamy, School of Computing and Information

Dissertation Advisors: Rami Melhem, Department of Computer Science,

Taieb Znati, Co-advisor, Department of Computer Science

**Heterogeneity Aware Fault Tolerance for Extreme Scale Computing**

Zaeem Hussain, PhD

University of Pittsburgh, 2020

Upcoming Extreme Scale, or Exascale, Computing Systems are expected to deliver a peak performance of at least $10^{18}$ floating point operations per second (FLOPS), primarily through significant expansion in scale. A major concern for such large scale systems, however, is how to deal with failures in the system. This is because the impact of failures on system efficiency, while utilizing existing fault tolerance techniques, generally also increases with scale. Hence, current research effort in this area has been directed at optimizing various aspects of fault tolerance techniques to reduce their overhead at scale. One characteristic that has been overlooked so far, however, is heterogeneity, specifically in the rate at which individual components of the underlying system fail, and in the execution profile of a parallel application running on such a system. In this thesis, we investigate the implications of such types of heterogeneity for fault tolerance in large scale high performance computing (HPC) systems. To that end, we 1) study how knowledge of heterogeneity in system failure likelihoods can be utilized to make current fault tolerance schemes more efficient, 2) assess the feasibility of utilizing application imbalance for improved fault tolerance at scale, and 3) propose and evaluate changes to system level resource managers in order to achieve reliable job placement over resources with unequal failure likelihoods. The results in this thesis, taken together, demonstrate that heterogeneity in failure likelihoods significantly changes the landscape of fault tolerance for large scale HPC systems.

## Table of Contents

## List of Tables

## List of Figures

## 1.0 Introduction

Extreme Scale, or Exascale, Computing within the supercomputing domain aims to deliver a thousandfold speedup over petascale ($10^{15}$ FLOPS) supercomputers. Several post-petascale and pre-exascale systems have already been deployed over the last few years, and their successors, Exascale Systems proper, are expected to become functional by year 2022[62]. While some of the increased speedup for these projected exascale systems is expected to come from improvement in the speeds of individual nodes, the major push is expected to come from a much larger number of system components, when compared with current petascale systems[8]. Due to their massive scale, a major theme in the research and development of Exascale systems is resilience[10]. This is because the failure of a single node being used by a large scale parallel job can bring the entire job to a halt, severely impacting the efficiency of the system. Since any resilience scheme comes with it own costs and overheads, providing fault tolerance with low overhead while maintaining a high performance under failures has been the holy grail for research efforts in this area.

### 1.1 HPC Fault Tolerance Landscape

The classical fault tolerance technique for HPC is coordinated checkpoint-restart[37]. This involves storing the application state, or *checkpoints*, during execution so that the application can be restarted from the last checkpoint if a failure happens. Like any other resilience mechanism, checkpointing provides fault tolerance at a cost, which is the time it takes to write a checkpoint. When failures are infrequent, the checkpointing and recovery cost constitute a small proportion of the total job wall clock time, on average.

As the number of nodes in the system increases, causing an increase in system failure rate, the number of checkpoints and reexecutions needed also increases. This causes a degradation in the efficiency of a system employing checkpointing as the primary fault tolerance technique. For this reason, replication[29] has been proposed as an alternative to coordinated

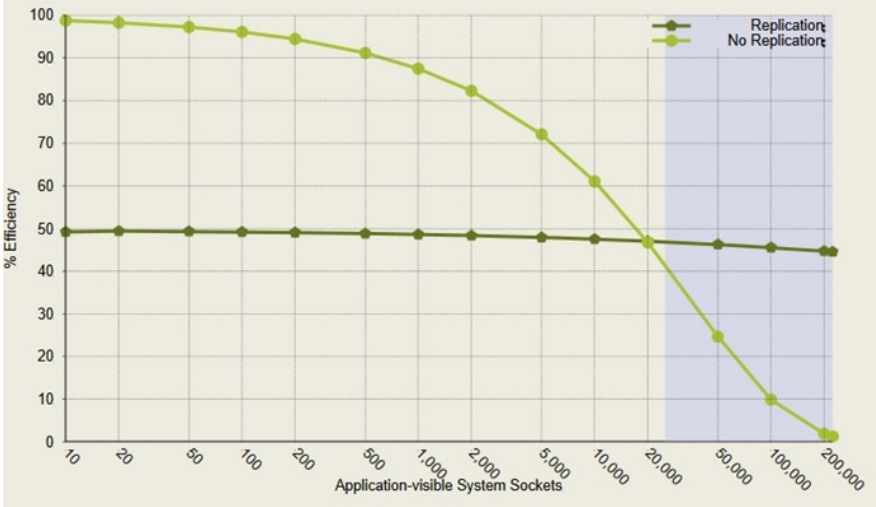



Figure 1: Traditional Coordinated C/R vs Pure replication[29].

checkpoint-restart (C/R). Replication involves duplicating the work on redundant hardware so that the failure of an individual node does not affect the execution of an application as long as a replica of the failed node is alive. This allows replication to increase the mean time to interrupt (MTTI) of a system [11]. This increase, however, is achieved at the cost of extra system resources and power. Thus, pure replication is only viable as a primary fault tolerance technique when the efficiency of coordinated C/R drops much lower, usually even below 50%, as can be seen in Figure 1. Moreover, infrequent checkpoints and rollbacks are still needed for the rare occurrence when both the original node and its replica fail.

The poor scalability of coordinated checkpoint-restart and the 50% efficiency barrier of pure replication have both resulted in resilience being identified as a key challenge facing Extreme Scale Computing systems of the near future. To some extent, the loss of efficiency with increasing failure rates is unavoidable, since a cost, either in terms of system resources or processing time, has to be paid in order to overcome failures. However, another source of the inefficiency of these two techniques stems from the fact that these techniques have traditionally been applied while assuming a simplified behavior of the underlying system when it come to failures. This thesis aims at exploring how a more refined understanding of the underlying system's failure characteristics as well as of the execution pattern of the

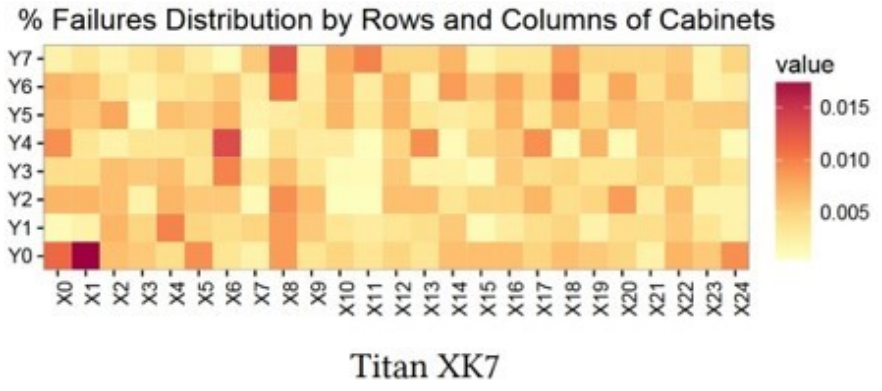



Figure 2: Spatial distribution of failures in the Titan Supercomputer[35].

application can be used to provide more efficient fault tolerance at large scales typical of the predicted Exascale systems of the future.

### 1.2 Heterogeneity in System Failure Likelihoods

The traditional assumption when analyzing fault tolerance techniques for HPC systems is that all the homogeneous nodes in the system also have independent and identical (iid) failure distributions. Hence, the analysis usually proceeds by using estimates of the parameters of the failure distribution of a single node and calculates system level parameters as if the same distribution describes each node's failure likelihood.

While the methodology outlined above does simplify the theoretical analysis, there is no experimental evidence to suggest that this assumption holds true in real life large scale HPC systems. In fact, several studies[25][36][35][23] on failures experienced by supercomputing systems have concluded that failures are spatially non-uniform. A visual example of such heterogeneity is shown in Figure 2. The source of this heterogeneity comes not just from the slight variations in the manufacturing of the individual components, but also from external events that affect the components in different ways during the life cycle of a large scale system. For example, some nodes in the system might be replaced over time, leading to

a disparity in the age of those components, which might result in different classes of node reliability[84].

Based on the above discussion, the first step towards more realistic system failure models would be to do away with the iid node failure assumptions. Removing the iid node failure assumption leads to a more sophisticated model for system failures. The simplest such model would assume the same kind of distribution for each node, albeit with parameter values that are not necessarily equal. Such a model still retains the assumption that failures of different nodes are independent, but allows for different nodes to have different probabilities of failures. In this dissertation, we use this heterogeneous model of node failure likelihoods to analyze the implications for HPC fault tolerance.

## 1.3 Research Overview

The primary goal of this dissertation is to investigate the implications, for fault tolerance in large scale HPC systems, of heterogeneity in failure likelihoods and of the imbalance in parallel computations. To that end, this dissertation studies the following research questions:

**RQ1** How can a deeper understanding of failure characteristics of a large scale system be used to inform and improve current fault tolerance mechanisms?

**RQ2** Can the imbalance in parallel workloads be used to provide more efficient fault tolerance for such workloads?

**RQ3** How can HPC job schedulers and resources managers use heterogeneity in failure likelihoods of individual resources to achieve reliable job placement within the system?

A visual depiction of the scope of these research questions is shown in Figure 3. I further elaborate on these research questions below:

### 1.3.1 Improving the State of the Art in Fault Tolerance for HPC Systems

For coordinated checkpoint-restart with and without dual replication, the default assumption has always been that all nodes in the system have the same likelihood of fail-

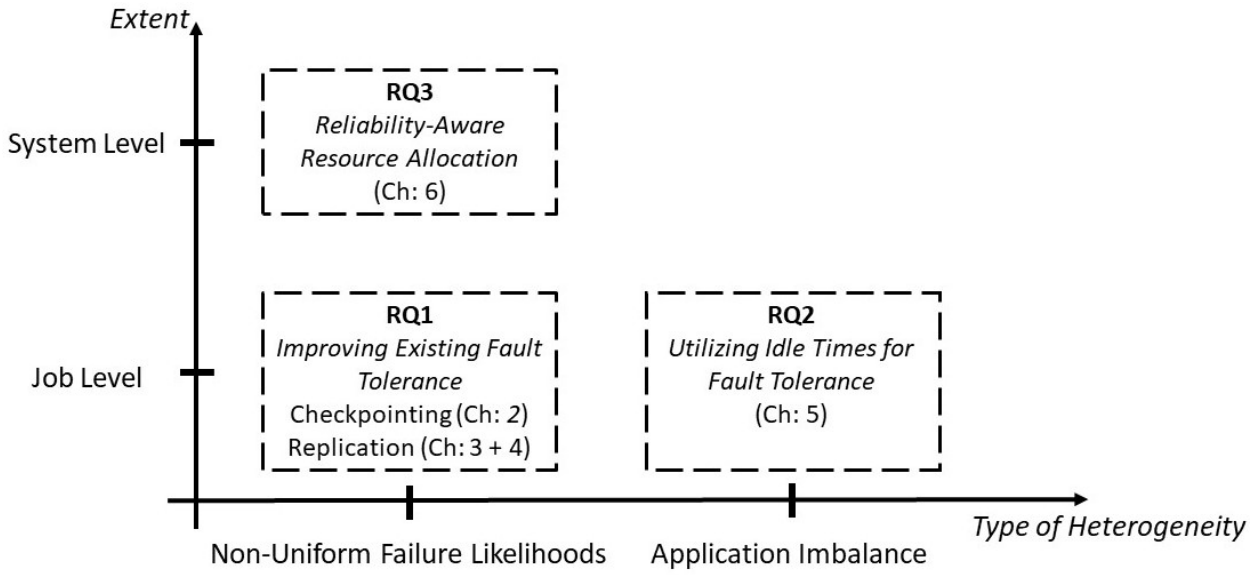



Figure 3: Scope of this dissertation.

ure. Thus, both of these techniques in their implementation are typically oblivious of any variations that may exist among individual node reliabilities and do not present an ideal opportunity to fully realize the benefits that could be derived from such heterogeneity. Nevertheless, an awareness of such heterogeneity does bring up some interesting possibilities and research questions for both simple checkpointing and checkpointing with replication, which I will briefly discuss in the following subsections.

**1.3.1.1 Checkpointing** In recent years, in-memory, or diskless[65], checkpointing has increased in popularity because it significantly improves the time to take a checkpoint. It is accomplished by either placing the entire copy of a process' state on the local memory of another process, or by encoding the states of a group of processes and distributing the encoded checkpoints over the local memories of all the processes in, either the same or another, group. However, in-memory checkpointing may not tolerate all kinds of failures. One such *catastrophic failure*[58] happens when a node and the node containing its checkpoint both fail, either simultaneously or in quick succession. The optimal placement of in-memory

checkpoints would thus select a placement strategy that minimizes the likelihood of such catastrophic failures. Finding this optimal strategy is non-trivial when all nodes do not have the same likelihood of failure, but it can lead to significant improvement in the capability of in-memory checkpoint-restart to tolerate a larger number of failures. This dissertation makes substantial contributions towards finding optimal placement strategies for the most common variants of in-memory checkpointing under a heterogeneous model of system failure likelihoods[44].

**1.3.1.2 Replication** The projection in Figure 1 shows that replication is more efficient than no replication when the application is running at a sufficiently large scale. However, this does not justify the use of such a large scale for a single application in the first place. This is because *reliability-aware speedups*[81] (i.e. the speedups under failures) do not always improve with increasing scale, unlike the failure free speedup under Amdhal's law. I show in this dissertation that the peak speedup under failures with replication is much higher and occurs at much larger scales than the peak reliability-aware speedup without replication[45]. This provides a definitive justification for the use of replication at larger scales in order to achieve improved performance under failures.

After demonstrating the superior scalability of application speedup with replication in the presence of failures, I investigate in this dissertation the consequences of having heterogeneity in failure rates when using replication, and show that heterogeneity is actually the key to the feasibility of *partial replication*[26]. As the name suggests, partial replication allows for replicating only a subset of the processes. The idea of partial replication has been discussed in the literature because it is not necessarily bounded above by 50% efficiency. However, when compared with pure replication and no replication, partial replication has never been shown to be the most efficient scheme at any system scale with iid node failure distributions. I show in this dissertation that this conclusion changes if the large scale system has non-identical node failures. Thus, I again use a heterogeneous failure likelihood model and first determine the best way of associating nodes with replicas, since nodes have different failure probabilities. I then compare the resulting configuration with full and no replication to show that partial replication can yield the most efficient fault tolerance scheme under

heterogeneous failure likelihoods[43].

### 1.3.2 Leveraging Application Imbalance for Fault Tolerance

Despite the tremendous engineering effort in scaling HPC workloads and in minimizing idleness in computational resources, there are several HPC applications that, by their very nature, exhibit a considerable degree of imbalance leading to idle time during their execution. In cases where such imbalance cannot be mitigated, the next best option is to somehow leverage that imbalance in optimizing one of the other desirable objectives within the system. An example in this direction is [32] where the authors use the idle time resulting from imbalance in MPI applications to adjust power consumed by individual cores/processors, leading to overall reduction in energy/power consumption. Yet another example is in-situ workload performance improvement[78]. The potential of such imbalance to improve performance in case of failures, however, has remained unexplored. Since such an imbalance is also a type of heterogeneity, albeit at the application level and different from the heterogeneity in failure likelihoods in the system, I explore in this dissertation whether this imbalance could, in fact, be utilized for better fault tolerance.

I first identify the challenges that are unique to fault tolerance when it comes to capitalizing on the idle time during application execution. I then argue in light of those challenges that the natural approach towards fault tolerance in this context would be to use the recently proposed Shadow Computing model[57][16]. The basic idea in this model is to associate with each process a *shadow* process which, like a replica, duplicates the work of its original process, or the *main*, but usually executes at a slower speed than the main. The flexibility in execution speed allows the shadow computing model to tailor its implementation to the specific requirements of the computing environment, such as power and QoS constraints[17][14]. For HPC message passing applications, slower shadows can be used for recovery in case of failures by employing message logging during normal operation[41]. While the original implementation of this model placed shadows on separate hardware from the original application, in this dissertation we interleave the placement of the shadow processes on the same nodes as the main processes, so that the shadows can execute during the idle time of the

main. We then evaluate the overheads and performance of this scheme in case of failures in order to assess the feasibility of utilizing application imbalance for fault tolerance.

### 1.3.3 Heterogeneity Aware Resource Managers

The two research questions discussed above are primarily focused on improving the performance under failures of a single job. In the presence of heterogeneity in failure likelihoods, I contend that there are additional opportunities at the system level to further optimize the efficiency/performance in the presence of failures. In the last part of this dissertation, I study one example of such an opportunity for resource assignment to multiple jobs, and show that information about the heterogeneity in failure rates can be used to perform more efficient resource assignment. Further background on the specific problem that I study as part of this research question is discussed below.

One of the roles of job schedulers in large scale clusters is to assign the available computational resources to the set of jobs ready to be scheduled. Traditionally, such assignments are done without considering their impact on the reliability of jobs, because the default assumption is that the compute nodes have identical failure distributions, rendering all the assignment candidates at a given time equivalent in terms of their reliability. However, under heterogeneity in failure likelihoods, allocating resources to jobs while disregarding the differences in reliabilities of those resources may end up increasing the waste incurred by the system. I formulate in this dissertation objective functions that allow us to compare, in terms of reliability, different possible assignments of resources to jobs. I find reliable placement schemes in light of those objective functions and demonstrate their potential to reduce system waste in case of failures.

### 1.3.4 Thesis Statement

By distilling the key findings from my investigations into the research questions described above, I claim the following:

*"An understanding of the heterogeneity in system reliability can lead not only towards significant improvement in current fault tolerance techniques at the job level but also towards*

*additional opportunities at the system level to deliver efficient fault tolerance for future extreme scale computing systems."*

## 1.4 Contributions

In summary, I make the following contributions in this dissertation:

- I provide the optimal strategy for in-memory checkpoint placement under heterogeneous node failure likelihoods for both full and group-encoded checkpoints. [Chapter 2]
- I extend reliability-aware speedup models by adding replication as an additional fault tolerance mechanism in those models. I provide results on the speedup profile of applications with replication and contrast them with the speedup without replication to show that, at scale, replication will be required to improve on the optimal speedup possible through checkpoint-restart alone. [Chapter 3]
- I identify the optimal partial replication configuration of nodes with non-identical reliabilities. We use this configuration to demonstrate the feasibility of partial replication versus full and no replication under heterogeneous node failure likelihoods. [Chapter 4]
- I implement co-located shadows and provide results on their scalability and performance under failures. [Chapter 5]
- I study the problem of allocating compute resources to jobs in a cluster comprising of nodes with non-identical failure rates, such that the waste due to failures is minimized. I formulate and study two objective functions that capture the reliability of jobs to be scheduled onto the resources, which lead to two heuristics for reliable resource allocation. I further extend the analysis by providing theoretical results for the case of replicated jobs. [Chapter 6]

### 1.4.1 Organization

The remainder of this document is organized as follows: Chapter 2 discusses placement of in-memory checkpoints, Chapter 3 studies the speedup of replication, Chapter 4 dis-

cusses partial replication of non-identically reliable nodes, Chapter 5 discusses the design and implementation of co-located shadows, Chapter 6 discusses resource allocation under heterogeneous failure likelihoods, and Chapter 7 concludes. Each of chapters 2 to 6 includes a related work section that discusses prior work related specifically to the problem and techniques discussed in that chapter.

## 2.0 Optimal Placement of In-Memory Checkpoints under Heterogeneous Failure Likelihoods

This chapter studies how the assumption of heterogeneous failure likelihoods can be used for optimizing coordinated checkpoint-restart. Specifically, it looks at the problem of finding the best placement strategy for in-memory checkpoints under this new assumption[1].

### 2.1 Introduction

In-memory checkpointing is accomplished by either placing the entire copy of a process' state on the local memory of another process, or by encoding the states of a group of processes and distributing the encoded checkpoints over the local memories of all the processes in either the same, or another, group. Although it reduces the overhead, in-memory checkpointing may not tolerate all kinds of failures. One such *catastrophic failure*[58] happens when a node and the node containing its checkpoint both fail, either simultaneously or in quick succession. In such a scenario, recovery from in-memory checkpoints becomes impossible and either a full application restart or a recovery from a checkpoint stored on disk is required.

Although catastrophic failures are relatively infrequent occurrences, with their share of the total number of failures typically considered to be around 4%[59][1], their impact on the performance of an application can be significant. This is because of the huge cost of writing checkpoints to the file system and larger recovery times. It can be seen from Fig 4 that, for projected exascale systems with a cost to checkpoint to file system of 18 minutes[1], average overhead is upwards of 40%. Fig 4 also highlights that if the likelihood of catastrophic failures is reduced, say from 4% to 2% (or even 3%), their impact on the overhead can also be brought down significantly. One way to reduce catastrophic failure likelihood is by optimizing the placement of in-memory checkpoints.

When all nodes are equally likely to fail, depending on the type of in-memory checkpoint-

[1]This work appeared in IPDPS 2019.

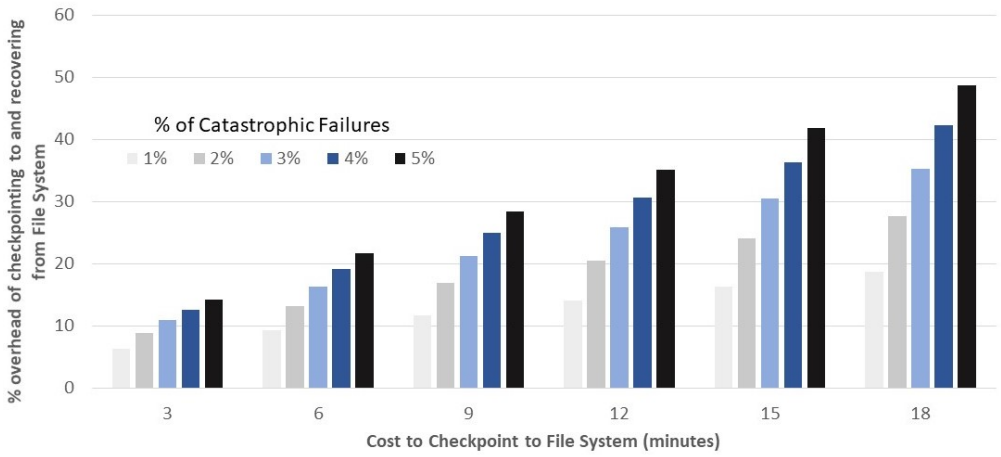



Figure 4: Average overhead due to catastrophic failures, based on the multilevel checkpoint model[24] using 2 levels. Projected exascale system paramters (taken from [1]): Number of nodes = 100,000, Node MTBF = 5 years, In-memory (Level 1) checkpoint cost = 9 seconds.

ing being used, either the optimal placement can easily be identified through combinatorial counting, or all placement schemes have identical reliability. If, however, all nodes do not experience failures with the same likelihood, finding the best checkpoint placement that minimizes catastrophic failures is non-trivial. I formulate this placement problem by assuming that each node has an independent probability of failure which may be different from other nodes. This chapter will describe the solutions I found to this placement problem for two of the most common in-memory checkpointing techniques: full in-memory checkpoints and grouped XOR encoded checkpoints. Since the goal of this work is to assess the reliability to catastrophic failures of different placement schemes, I make the simplifying assumption that the cost of sending checkpoints to different nodes is uniform. In practice, the cost depends on the locations of the source and destination nodes within the network. A more advanced analysis that does take into consideration the cost of different placement schemes and weighs that against their reliabilities is left for future work.

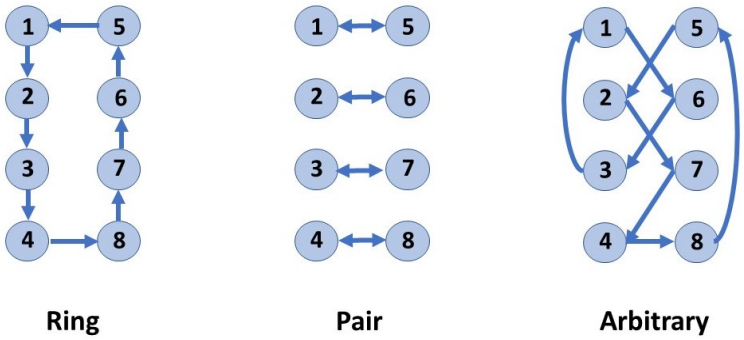



Figure 5: Some examples of full in-memory checkpoint placement schemes over 8 nodes. An arrow starts at the node whose checkpoint is to be stored in another node and ends at the node where that checkpoint is placed.

## 2.2 Full In-Memory Checkpoints

This section deals with the case of full in-memory checkpoint placement, where each node stores a full copy of another node's checkpoint in its local memory. Fig. 5 shows some possible placement schemes for full in-memory checkpoints. Ring based and paired placement are the more well known schemes. While simple to understand and visualize, these are not the only possible options. In fact, it is possible to use any arbitrary checkpoint placement scheme (Fig 5, right). Each placement scheme can be thought of as a directed graph where each node (or vertex) has one incoming edge (for the remote checkpoint it is hosting) and one outgoing edge (for its own checkpoint). A catastrophic failure thus occurs when any two nodes that are neighbors of each other in the directed graph fail.

### 2.2.1 IID Node Failures

When all nodes are independent and equally likely to fail, the question of optimal checkpoint placement boils down to selecting between a ring based or a paired placement. This is because the arbitrary placement scheme can also be thought of as a collection of disjoint rings. For the example in Figure 5, the arbitrary scheme consists of two rings, one over

nodes 1, 3 and 6, and the other over the remaining nodes. The only thing to determine, then, is which of the two (ring or pairing) is more resilient, which can simply be determined by combinatorial counting of the number of multiple node failures that will lead to a catastrophic failure. This was done in [13] and it was shown that pairing is better than a ring based placement.

### 2.2.2 Non-Identical Node Failures

When individual nodes in the system have distinct failure likelihoods, it is no longer true that any paired scheme is better than a ring based or arbitrary scheme. Looking at Fig. 5, suppose, for example, that nodes 1 to 4 have all reliability of 1 whereas nodes 5 to 8 all have a reliability of 0.5 and all nodes are independent. By enumerating all possible catastrophic failures for each scheme and calculating their likelihood, we find that the reliability of the ring on the left is 0.5, the reliability of the pairing scheme in the middle is 0.5625 while the reliability of the arbitrary scheme is 0.75, the highest of the three schemes. This is because, of the nodes that have a non-zero likelihood of failure (nodes 5 to 8), only nodes 6 and 7 are connected in the arbitrary scheme, while the ring and pairing schemes both have more connections among the less reliable nodes. Thus we can see that finding the optimal placement for non-iid node failures cannot be done simply by combinatorial counting, as for iid failure likelihoods. Our main theoretical result of this section describes this optimal mapping.

Before describing the main result of this subsection, we first need to define some notations. Consider a job running on $N$ nodes where each node $i, i \in \{1, \ldots, N\}$ has a likelihood of survival within a specified time interval (or reliability) given by $p_i$. The failure likelihood of that node will thus be given by $1 - p_i$. Further, we assume that the relative ordering of the nodes based on their reliability is known and given, in increasing order of reliability, by the permutation $\sigma$, such that $p_{\sigma(i)} \leq p_{\sigma(i+1)} \forall i \in \{1, \ldots, N-1\}$. We now provide the main result of this subsection:

**Theorem 1.** *The optimal checkpoint placement that minimizes the likelihood of catastrophic failures is achieved by a paired placement scheme such that pair $i$ consists of nodes $\sigma(i)$ and*

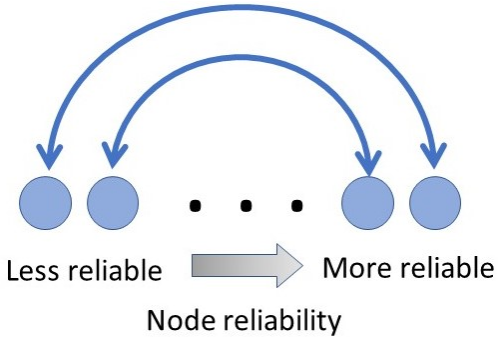



Figure 6: Optimal checkpoint placement scheme for nodes with different reliabilities.

$\sigma(N-i)$, $i \in \{1, \ldots, N/2\}$.

Such a placement scheme is pictorially depicted in Fig. 6 and is basically constructed by pairing the least reliable node with the most reliable node, and so on. Further, it does not require knowledge of values of individual failure likelihoods but only the relative ordering of the nodes based on their failure likelihoods.

I will prove the above theorem through a series of lemmas, starting with the earlier observation that any arbitrary checkpoint placement scheme consists of one or more mutually exclusive rings or cycles. Since all nodes are assumed to be independent, the likelihood of a scheme avoiding a catastrophic failure will be given by the product of the individual rings' probability of avoiding a catastrophic failure. If, therefore, a ring within a scheme is modified without changing anything else such that the reliability of the set of nodes from the original ring is improved, the reliability of the new scheme will also be higher than the original scheme.

**Lemma 1.** *Any ring of size $R \geq 4$ can always be broken into two rings of sizes $R-2$ and 2 respectively, such that the likelihood of avoiding a catastrophic failure is improved.*

*Proof.* I prove this lemma by showing that there is always a pair of neighboring nodes in the ring which can be taken out, as shown in Fig. 7, such that the new configuration will have either the same or better reliability than the original ring.

Let the nodes be labeled as shown in Fig. 7. Note that the reliability of the new scheme

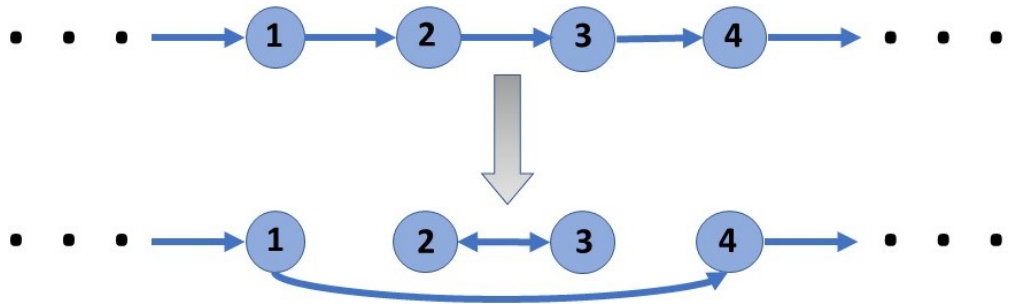



Figure 7: Decomposing a larger ring into a pair and a ring of smaller size.

obtained by the decomposition differs from the original ring in exactly three scenarios: i) failure of nodes 1 and 2 does not always lead to a catastrophic failure, unlike in the original ring, similarly, ii) failure of nodes 3 and 4 does not always lead to a catastrophic failure, and iii) failure of nodes 1 and 4 always leads to a catastrophic failure, which was not always the case in the original ring. Based on this observation, I write the difference between the reliability of the new scheme and the original one as:

$$\delta = p_1p_2(1-p_3)(1-p_4)\alpha_4 + p_3p_4(1-p_1)(1-p_2)\alpha_1 - p_2p_3(1-p_1)(1-p_4)\alpha_{1,4} \tag{2.1}$$

where $\alpha_4$ is the likelihood of all outcomes of the rest of the nodes in the ring which will not lead to a catastrophic failure when node 4 fails but 1 does not fail. $\alpha_1$ is defined similarly and captures the case when node 1 fails but 4 does not. Finally, $\alpha_{1,4}$ is the probability of the rest of the nodes avoiding a catastrophic failure when both nodes 1 and 4 fail.

We now need to show that there exist four consecutive nodes in the ring for which $\delta \geq 0$. We first note that $\alpha_1 \geq \alpha_{1,4}$ and $\alpha_4 \geq \alpha_{1,4}$. This is because all possible outcomes, of the rest of the nodes in the ring, that do not lead to a catastrophic failure when nodes 1 and 4 both fail will also not lead to a catastrophic failure when one of those two nodes does not fail, and so their likelihood is also counted in both $\alpha_1$ and $\alpha_4$. Thus, it is enough to show that there exist four consecutive nodes for which $\tilde{\delta} \geq 0$ where

$$\tilde{\delta} = p_1p_2(1-p_3)(1-p_4) + p_3p_4(1-p_1)(1-p_2) - p_2p_3(1-p_1)(1-p_4) \tag{2.2}$$

By simplifying and rearranging, we get the following expression for $\tilde{\delta}$

$$\tilde{\delta} = p_1p_2 + p_3p_4 - p_2p_3 - p_1p_4(1 - (1 - p_2)(1 - p_3)) \tag{2.3}$$

Since $0 \leq (1 - (1 - p_2)(1 - p_3)) \leq 1$, $\tilde{\delta} \geq 0$ whenever

$$p_1p_2 + p_3p_4 - p_2p_3 - p_1p_4 = (p_1 - p_3)(p_2 - p_4) \geq 0 \tag{2.4}$$

which holds in two cases:

$$\text{Case 1:} \quad p_1 \geq p_3 \text{ and } p_2 \geq p_4$$

$$\text{Case 2:} \quad p_1 \leq p_3 \text{ and } p_2 \leq p_4$$

I argue that there always exist four consecutive nodes in a ring that satisfy one of the two cases above. Let the least reliable node in the ring be $i$, which implies that $p_{i-2} \geq p_i$ and $p_i \leq p_{i+2}$. Now, if node $i-1$ is more reliable than node $i+1$ (i.e $p_{i-1} \geq p_{i+1}$), then the nodes $i-2$, $i-1$, $i$ and $i+1$ are four consecutive nodes that satisfy Case 1 above. If, however, node $i-1$ is less reliable than node $i+1$ (i.e $p_{i-1} \leq p_{i+1}$), then nodes $i-1$ to $i+2$ are four consecutive nodes that satisfy Case 2 above. □

Lemma 1 means that, given any arbitrary checkpoint placement scheme, one can get a more reliable (or as reliable as the original) scheme by taking out pairs of nodes from larger rings. The main consequence, therefore, is that the optimal scheme with maximum reliability to catastrophic failures will not contain a ring of size larger than 3 nodes. Our next result deals with rings of 3 nodes.

**Lemma 2.** *Two rings of 3 nodes each can always be transformed into three pairs of nodes such that the reliability of the new scheme is improved.*

*Proof.* We break both the 3-node rings by taking one node from both of them and joining the two nodes to form a pair, as shown in Fig. 8. It can then be shown, by comparing the overall reliability of the two original rings with that of the three newly formed pairs, that the three pairs will have a higher reliability as long as both of the two nodes taken from each of the original rings (Nodes 3 & 4 in Fig 8) are not the least reliable nodes in their respective rings (one of them can still be the least reliable in its ring). We omit the detailed expressions for brevity. □

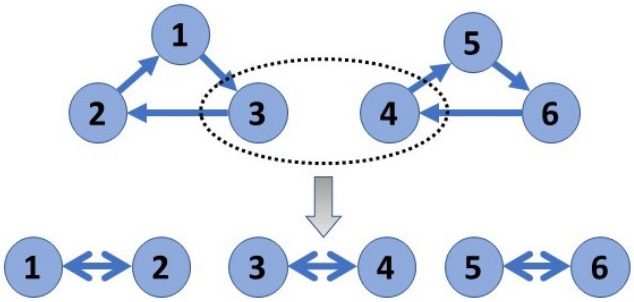



Figure 8: Breaking two rings of 3 nodes each into three pairs of nodes

Using lemma 1 we had inferred that the optimal checkpoint placement consists only of pairs and rings of 3 nodes. Using lemma 2, we can say that the optimal checkpoint placement scheme consists only of pairs of nodes when the total number of nodes is even. If total number of nodes is odd, the optimal placement will contain one ring of three nodes. The optimal placement of the 3-node ring cannot be determined from ordering of node reliabilities alone, but rather requires a search using the actual values of node reliabilities. In the remainder of this chapter, I will assume that the total number of nodes is even, so henceforth we will focus solely on pairs of nodes.

*Proof of Theorem 1.* We have already established, using lemmas 1 and 2 that the optimal placement of full checkpoints is accomplished by some paired scheme. What remains is to show that that pairing follows the arrangement mentioned in the theorem statement (and as shown in Fig 6). In the chapter on partial replication in this dissertation (Chapter 4), I show that the optimal pairing of nodes with replicas follows the arrangement shown in Fig. 6. The expression for the likelihood of avoiding a catastrophic failure in a paired checkpoint scheme is the same as the expression for the likelihood of avoiding a node-replica pair failure when using the same mapping scheme for replication. We can thus conclude that the optimal checkpoint pairing that minimizes the likelihood of catastrophic failures is achieved by pairing the least reliable unpaired node with the most reliable unpaired node and continuing in this manner recursively, resulting in the arrangement of Fig. 6. This concludes the proof of Theorem 1. □

Looking back at Fig. 5, and using the reliability values of the nodes mentioned in the beginning of this subsection, we can apply Theorem 1 to find the optimal placement scheme. Such a scheme will pair a node from nodes 1 to 4 with one of nodes 5 to 8, and will result in an overall reliability of 1. In fact, we do not even require knowing the actual values of node reliabilities. It would be sufficient, for example, to know that nodes 1 to 4 all have the same reliability and nodes 5 to 8 all have the same reliability but less than that of the first 4 nodes, and we could still apply Theorem 1 to obtain the optimal grouping.

### 2.3 Encoded Checkpoint Grouping

A drawback of keeping a full copy of the checkpoint of a process in another process is that it reduces the amount of memory available to the original process. This is especially a concern because each process also needs to keep its own checkpoint in its memory in order to allow it to roll back when another process/node dies[80]. The idea of encoding checkpoints[65] was introduced to reduce this memory footprint by encoding multiple checkpoints in a group and distributing the encoded checkpoints within either the same, or a different, group.

While several different ways of encoding and distributing the checkpoints have been proposed in the literature[65][33][4], we will focus in this work on the simple XOR based encoding, shown in Fig. 9, which is employed by the Scalable Checkpoint Restart (SCR) library[59][58]. Under this encoding, within a group of size $k$, each checkpoint is split into $k-1$ chunks. Each member stores the XOR obtained by taking a chunk from each of the other $k-1$ members of the group. With this arrangement, any two failures within a group of $k$ nodes will lead to a catastrophic failure. We will assume that the size of the group ($k$) is already determined by the programmer (or the system administrator) based on the application's footprint, the checkpoint size and available memory on the node. Our goal, then, is to find the optimal way to group the system nodes such that likelihood of catastrophic failures (more than one failure in a group) is minimized.

Formally, we are provided with a set of $N$ nodes that need to be distributed into groups of $k$ nodes each. The total number of groups required will be $n = N/k$. We denote the nodes

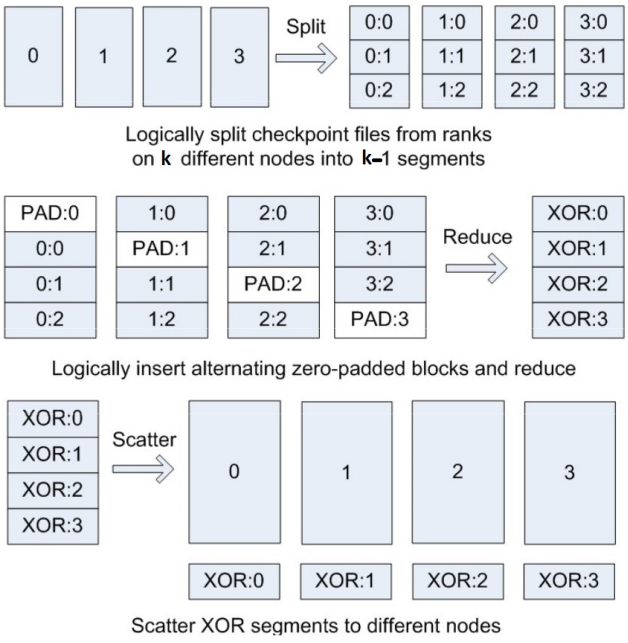



Figure 9: XOR encoded checpointing with a groups size of 4 (Figure taken from [58]).

in a group $g$ by $a_i^g$ where $g \in \{1, \ldots, n\}$, $i \in \{1, \ldots, k\}$ and $a_i^g \in \{1, \ldots, N\}$. The reliability of node $a_i^g$ is given by $p_{a_i^g}$. Our main result, formally stated below, says that the optimal grouping divides the nodes such that the sum of the inverses of reliabilities of all the nodes in a group is as uniform across groups as possible.

**Theorem 2.** *Maximum reliability of the XOR based checkpoint encoding is achieved when the quantity $S$ given by*

$$S = \max_{g,h \in \{1,\ldots,n\}} \left(\sum_{j=1}^{k} \frac{1}{p_{a_j^g}} - \sum_{j=1}^{k} \frac{1}{p_{a_j^h}}\right) \tag{2.5}$$

*is minimum.*

*Proof.* For XOR encoding, the reliability, $r_g$ of a group $g$ of $k$ nodes will be the probability of at most one node failing within the group, and can be written as

$$r_g = \prod_{i=1}^{k} p_{a_i^g} + \sum_{i=1}^{k} (1 - p_{a_i^g}) \prod_{j=1, j\neq i}^{k} p_{a_j^g} = \prod_{i=1}^{k} p_{a_i^g} \left(1 + \sum_{i=1}^{k} \frac{1 - p_{a_i^g}}{p_{a_i^g}}\right) = \prod_{i=1}^{k} p_{a_i^g} \left(\sum_{i=1}^{k} \frac{1}{p_{a_i^g}} - (k-1)\right) \tag{2.6}$$

Since each group is independent of the other groups, the overall system reliability to catastrophic failures will be a product of the reliabilities of the individual groups. Thus, the overall reliability, $r$, will be given by

$$r = \prod_{g=1}^{n} r_g = (\prod_{g=1}^{n}\prod_{i=1}^{k} p_{a_i^g}) \prod_{g=1}^{n} (\sum_{i=1}^{k} \frac{1}{p_{a_i^g}} - (k-1)) \tag{2.7}$$

Since the product $\prod_{g=1}^{n}\prod_{i=1}^{k} p_{a_i^g}$ is simply a product of the reliability of all the $N$ nodes in the system, it does not depend on how the nodes are grouped. Thus, finding a grouping that maximizes $r$ is the same as finding a grouping that maximizes $\tilde{r}$ where

$$\tilde{r} = \prod_{g=1}^{n} (\sum_{i=1}^{k} \frac{1}{p_{a_i^g}} - (k-1)) \tag{2.8}$$

Using the inequality of arithmetic and geometric means (AM-GM)[73], we get

$$\tilde{r} \leq \left( \frac{\sum_{g=1}^{n} (\sum_{i=1}^{k} \frac{1}{p_{a_i^g}} - (k-1))}{n} \right)^n = \left( \frac{\sum_{g=1}^{n} \sum_{i=1}^{k} \frac{1}{p_{a_i^g}} - n(k-1)}{n} \right)^n \tag{2.9}$$

This is useful because $\sum_{g=1}^{n} \sum_{i=1}^{k} \frac{1}{p_{a_i^g}}$ is actually the sum of the inverse reliabilities of all the nodes in the system, which also does not depend on how the nodes are grouped together. This means that $\tilde{r}$ is upper bounded by a constant with respect to the grouping scheme. In order to find the grouping that achieves this upper bound, we use the fact that equality in the AM-GM inequality occurs when all the $n$ numbers are equal. From Eq. 2.8, this implies that $\sum_{i=1}^{k} \frac{1}{p_{a_i^g}} = \sum_{i=1}^{k} \frac{1}{p_{a_i^h}}$ for all $g, h \in \{1, \ldots, n\}$. When it is not possible to have a grouping that achieves strict equality, the optimal grouping will be the one that makes the sum $\sum_{i=1}^{k} \frac{1}{p_{a_i^g}}$ as uniform across all the groups as possible, from which we get the statement of the theorem. □

Theorem 2 means that finding the optimal grouping is actually equivalent to the balanced multi-way number partitioning problem[56]. The $n$-way number partitioning problem[49] seeks to divide a set of numbers into $n$ groups (or subsets) such that the sum of the numbers in each group is the same. The balanced variant of this problem further imposes the constraint that the cardinality of all the subsets (or groups) is the same. Taking the set of numbers to partition to be the inverse node reliabilities ($1/p_i$), based on Theorem 2, finding the most reliable grouping for checkpoint encoding is the same as finding a balanced $n$-way partition for the set of inverse node reliabilities.

Both the multi-way number partition problem and its balanced variant are NP hard, and so several heuristics have been proposed in the literature[75][77]. The most well known heuristic for this problem is the Balanced Largest Differencing Method (BLDM)[75][56]. The BLDM heuristic can be thought of as a generalization of the folding operation used to form pairs in Section 2.2 (and as shown in Fig. 6).

Theorem 2 also helps us to reason about the simpler case of system nodes belonging to a limited set of failure classes, where all the nodes in a failure class have the same likelihood of failure. This is still a useful case to consider in practice because even when there may not be enough information from a real system to distinguish between individual nodes based on their failure likelihoods, it may be possible for system administrators to classify each node into one of a small set of failure classes[42]. In this simplified version of the original problem, I assume that a system of $N$ nodes consists of $C$ failure classes, where each class $i \in \{1, \ldots, C\}$ contains $c_i$ nodes. All the nodes in class $i$ have the same reliability $p_i$ and each node belongs to exactly one failure class, which means that $\sum_i^C c_i = N$. Using Theorem 2 leads us to the following corollary for the simpler problem:

**Corollary 1.** *If $c_i$ is perfectly divisible by $n$ ($c_i/n \in \mathbb{N}$) for all $i \in \{1, \ldots, C\}$, maximum reliability to catastrophic failures can be achieved by placing the same number of nodes of each class ($c_i/n$, for all $i \in \{1, \ldots, C\}$ ) in each group.*

Corollary 1 is simply a statement of the fact that, whenever it exists, a grouping scheme which makes all groups identical by placing the same number of nodes of a particular type in each group will be optimal. Using this, I also formulate a simple grouping heuristic that

works by assigning nodes into failure likelihood classes. Thus, the heuristic first organizes the nodes by placing nodes with similar reliability values in the same failure class such that the number of nodes in each class is divisible by $n$. It forms identical class based groups by placing $c_i/n$ nodes from class $i$ into each of the $n$ groups. Note that if the node failure likelihoods actually had discrete values such that they could exactly be placed into separate failure classes where each class size is divisible by $n$, BLDM and my heuristic would yield groupings with the same reliability. Thus, this class-based heuristic can be considered a further simplification of BLDM, yet performs similarly to BLDM as we will see in the subsequent sections.

## 2.4 Validation

I validate the placement schemes by using the 5-year reliability, availability and serviceability (RAS) logs, collected between 2013 and 2017, of the IBM Blue Gene/Q Mira supercomputer, deployed at the Argonne National Laboratory[22]. Blue Gene/Q Mira contains 49,152 nodes organized into 48 racks of 1024 nodes each. The authors of [22] have made the logs and their analysis tools available at [50] and [21] respectively. Each event entry in the logs contains the time of the event, location and the severity level (INFO, WARN or FATAL). Fatal severity level event "designates a severe error event that presumably leads the application to fail or abort."[51] The authors of [22] processed the logs to filter the fatal events that corresponded to actual failure events in the system. Moreover, they used temporal and spatial similarity analyses to group the fatal event entries that were caused by, or were a manifestation of, the same failure event and obtained 1255 failure events spread over the 5 year duration of the logs. Thus, each of the events obtained from the filters consists of a group of fatal event entries. By looking at the locations of the entries in a group, I determine all the impacted nodes in that failure event. Of the 1255 failure events, 520 ($\approx$ 41%) affected more than one node. Counting the number of failure events in which a node appears, we get the histogram of Fig. 10.

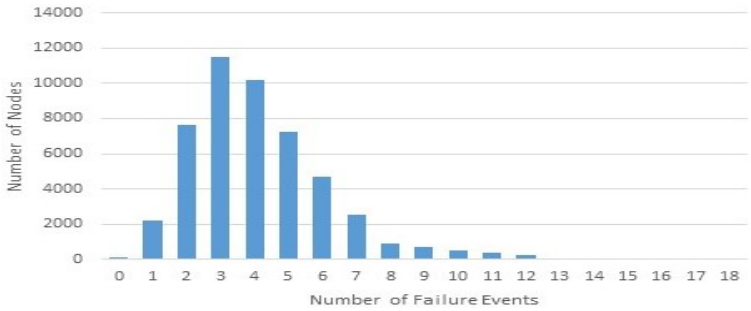



Figure 10: A histogram of the number of failures experienced by nodes in the system.

### 2.4.1 Full Checkpoint Placement

I compare three schemes for full in-memory checkpoint placement in their reliability to catastrophic failures: i) ring formed over a random permutation of all system nodes, ii) random pairing placement and iii) the optimal pairing for non-identical nodes as found in Section 2.2, which we will call Sorted Pairing. For each scheme, we go through each of the 1255 failure events, check if a pair of connected nodes in the scheme both have a fatal event entry within the entries of the failure event, and, if they do, count the failure event as a catastrophic failure event for that scheme.

For Sorted Pairing scheme, I take the number of failures experienced by the individual nodes as a measure of their failure likelihoods. Thus, I sort the nodes based on their failure

Table 1: Catastrophic Failures with Full Checkpoint Placement Schemes

| | Average | Minimum | Maximum |
|---|---|---|---|
| Random Ring | 71.3 | 61 | 82 |
| Naive Optimal (Random Pairing) | 60.3 | 54 | 71 |
| Sequential Pairing | 413 | - | - |
| Sorted Pairing | 31.5 | 29 | 34 |

counts shown in Fig. 10, in order to form the pairs as determined in Section 2.2. I generate 10 random instances of each scheme, and compute the number of catastrophic failures experienced by each of them. The resulting statistics are shown in Table 1. We see that, on average, Sorted Pairing experiences 55.8% less catastrophic failures compared to ring based placement and 47.8% less than a random paired placement scheme. Using the estimate that 4% of failures are usually catastrophic[59][1], a 47.8% reduction would mean that the optimal sorted pairing scheme could bring the likelihood of such failures down to around 2%. Going back to Fig 4 for the projected exascale system, such a reduction translates into the job overhead (caused by file system checkpoints and recovery) going down from 42% to 27%.

### 2.4.2 Encoded Checkpoints Grouping

For group encoded checkpoints, I test 3 grouping schemes: i) random grouping, ii) grouping by the BLDM heuristic[56] using inverse node reliabilities, and iii) approximating nodes into failure classes and then making identical groups based on Corolloray 1 (I refer to this scheme as Class Based Grouping). Since the BLDM heuristic uses the actual values to perform the grouping, I estimate the inverse node reliabilities using their failure counts. I first estimate a checkpoint interval using Daly's formula[20], where I take the checkpointing cost, $C$, to be 1 minute and obtain the system MTBF using the count of 1255 failures over 5 years. I then estimate the reliability of the node within such a checkpointing interval by assuming that the node follows an exponential distribution with rate given by the node's MTBF derived using its failure count over 5 years. The inverse of that likelihood for each node is fed to the BLDM heuristic to obtain a grouping.

Fig 11 shows the number of catastrophic failures experienced by the 3 grouping schemes. For class based scheme, I divided the list of sorted nodes (according to their failure counts) into a number of classes equal to the group size. For example, for a group size of 8, the nodes were also divided into 8 failure classes. For both the BLDM and Class based schemes, we first permuted all nodes with the same failure counts among themselves in the sorted list before applying the grouping scheme. The general trend of increased catastrophic failures with increased group size is expected, since a larger group size trades off memory consump-

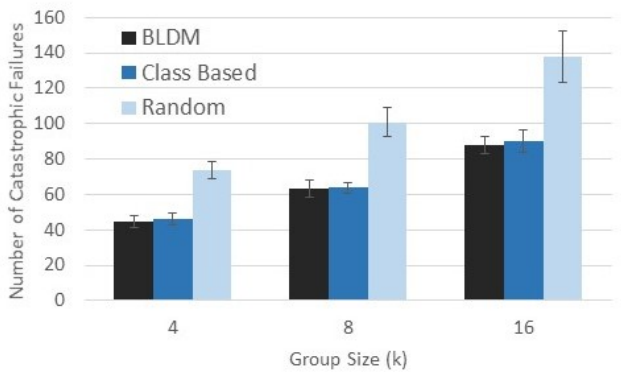



Figure 11: Number of catstrophic failure experienced by the different grouping schemes.

tion for reliability to catastrophic failures. Both the BLDM and Class based schemes that use information of node failure likelihoods experienced about 35% less catastrophic failures compared to a random grouping. Even though BLDM uses the actual values of individual nodes' reliabilities, its performance is comparable to Class Based grouping, which simply groups nodes by picking from failure classes. This could be due to several reasons, such as BLDM itself being a heuristic which may not generate an optimal grouping, and the node reliability estimates not being accurate enough. In conclusion, for HPC systems, the class-based heuristic is sufficient for reducing the likelihood of catastrophic failures.

## 2.5 Related Work

Diskless, or in-memory, checkpointing was proposed in [65]. Examples of implementation of full in-memory checkpoint are [58] and [80]. For full in-memory checkpoint placement over independent and identically distributed (iid) node failure likelihoods, the superiority of paired placement over ring based placement has been discussed in [13] and [55]. The idea of distributing encoded checkpoints was also proposed in [65]. The SCR library[58] provides an implementation of distributed checkpoints as well using simple XOR encoding. All of these techniques can utilize the optimal mapping schemes discussed in this work. Although

other more sophisticated encoding approaches have also been proposed in the literature, an example being the RS encoding discussed in [4], we leave an analysis of the optimal placement schemes under these encoding approaches for future work.

When it comes to considering heterogeneous node failure likelihoods, the only prior work, within the HPC domain, is [61], which performs selective node level duplications based on the heterogeneous failure history of individual system nodes. Outside of HPC, there has been some work[47] on using the heterogeneity in disk reliabilities to optimize the grouping of disks in storage systems. This grouping problem is theoretically similar to the grouping for XOR encoded checkpoints considered in this chapter. However, there are a couple of distinguishing features that make the two problems distinct. The first is that the group size for in-memory checkpoints within an application instance is constant, in order for the memory consumption within each node to be uniform. For disk grouping, however, groups with different sizes are allowed. The second difference is that the work of [47] placed the constraint that disks with different reliabilities cannot be grouped together, unlike the checkpoint grouping problem in this chapter where nodes with different reliabilites may be placed in the same group.

## 2.6 Summary

In this chapter, I studied the optimal placement of in-memory checkpoints when individual node failure likelihoods are not identical. I provided theoretical results on full and group encoded checkpoint placement that minimize the likelihood of catastrophic failures. I further validated my approach on failure logs of a large-scale system and showed that using node-level failure data to place in-memory checkpoint does reduce the number of catastrophic failures.

## 3.0 Enhancing Reliability-Aware Speedup Modeling via Replication

Before investigating the implications of failure likelihood heterogeneity for replication, we need to establish why replication is even considered as a candidate fault tolerance mechanism for exascale. Over the last decade, several studies[29][11] have argued that replication, paired with checkpoint-restart, can provide better performance under failures at exascale than C/R without replication. The crux of the argument in these studies is the fact that, at large scales, the efficiency without replication drops below the efficiency with replication. In this chapter, I add further weight to this argument by showing that the reliability-aware speedup with replication can beat the best possible speedup under failures that is achievable without replication[1]. Simply put, this means that, at large scale, replication can not only outperform no replication at that scale (as argued in prior works), but can actually beat the best performance over all scales that can be achieved without replication. Hence, while the analysis in this chapter assumes identical failure likelihoods, the results of this chapter make the case for studying replication under heterogenous failure likelihoods, which I do in the next chapter.

### 3.1 Introduction to Reliability-Aware Speedups

On a failure free platform, the performance of a parallel high performance computing (HPC) workload always improves with the number of processors if the application speedup follows Amdahl's law. However, if the application is executing on a platform where individual processors are vulnerable to failures, it is no longer true that executing the application over a larger number of processors always results in an improvement in job completion time. This is because the increase in scale also increases the frequency of failures, thus increasing the fraction of time spent in checkpointing and recovery. Eventually, this wasted time starts to outweigh any gains made by further parallelizing the workload and thus adding more

[1]This work appeared in DSN 2020.

processors starts hurting the application performance. This raises the question: what is the optimal number of processors at which a workload can achieve its best possible average speedup, given a platform specific processor failure rate?

To answer the above question, one first needs to develop a speedup model that takes into account the failure rate as well as the type and cost of the fault tolerance mechanism(s) employed. Several past works[12][46][82] have explored such *reliability-aware* speedup models using checkpoint-restart (C/R) as the sole fault tolerance mechanism, with [12] providing theoretical results on the order of optimal processor counts in terms of failure rate $\lambda$. On the other hand, it has been projected in [29] that, at large scales, the efficiency with C/R alone will degrade significantly and that using replication (paired with C/R) will be a more efficient alternative. Thus, it is reasonable to explore whether adding replication can significantly improve reliability-aware speedups at larger scales. In this chapter, I study such reliability-aware speedup of dual replication, obtaining results about the upper bound on the number of processors than can be used with replication as well as results on the contrast between the speedups with and without replication.

## 3.2 Background

This section describes the background information needed to understand the mathematical development in subsequent sections. It should be noted that all the quantities below that depend on $P$ will have different formulae depending on whether replication is employed or not. It should also be noted that $P$ refers to the total number of processors being used, which means that, for dual replication, $P/2$ processors will be replicas of the other $P/2$ processors. Thus, all comparisons between the performance of replication and no-replication are made with the same number of total processors, $P$, used by each technique. This also means that, in cases where replication has better speedup than no-replication, the expected energy cost of replication will be lower, simply because a job using $P$ nodes will finish quicker with replication than without it.

I consider a job model in which the work is distributed among the available processors at

the time of job start, and there is no work stealing. Thus, the number of processors and the work per processor remains fixed throughout an execution and, upon single processor fail, the job recovers from last checkpoint and continues with same number of processors. I assume that each processor has an exponential failure distribution with mean time between failure (MTBF) $\mu$, or equivalently, rate $\lambda = 1/\mu$. For a parallel job using a total of $P$ processors, let $\lambda_P$ denote the resulting failure rate. When no replication is used, $\lambda_P = \lambda P$. With replication, however, not every processor failure interrupts the execution of the workload. The execution is interrupted only when a processor and its replica fail. Thus, for replication, the quantity of interest is the Mean Time To such Interrupts (or $MTTI$), using which we can again define the failure rate of replication as $\lambda_P = 1/MTTI_P$. A general closed form for $\lambda_P$ for replication is not known. For dual replication though, a closed form expression was recently derived in [7] where the authors showed that $\lambda_P = \lambda P(1 + \binom{P}{P/2}/2^P) \approx \lambda\sqrt{2P/\pi}$ (for large $P$), which is the value I will use in the model for the failure rate of dual replication.

I take the checkpointing interval to be equal to Young's[76] first order approximation for the optimum checkpointing interval, given by $\tau = \sqrt{2C\mu_P}$. Here, $\mu_P = 1/\lambda_P$ is the system MTBF of $P$ processors in case of no replication or the MTTI of a dually replicated execution with a total of $P$ processors as discussed above.

I use Amdahl's law[2] to model the failure free speedup. Hence, without replication, we have $S_{norep}(P) = \frac{1}{\alpha+(1-\alpha)/P}$, where $\alpha$ is the sequential fraction of the workload. For dual replication with $P$ total processors, the parallel work is divided over $P/2$ processors only, which means the failure-free speedup will be $S_{rep}(P) = \frac{1}{\alpha+2(1-\alpha)/P}$. We can see that both of these expressions are continuously increasing functions of $P$, which is not the case when failures are taken into account. With failures, we are interested in the average speedup behavior of a workload, and I denote that by $\mathbb{S}_{norep}(P)$ and $\mathbb{S}_{rep}(P)$ for the two cases, respectively. Clearly, $\mathbb{S}_{norep}(P)$ (and $\mathbb{S}_{rep}(P)$) is a function not just of $\alpha$ but other parameters such as the failure rate ($\lambda$), checkpointing cost ($C$), recovery cost ($R$) and the downtime after failure ($D$). When considering the behavior of $\mathbb{S}_{norep}(P)$ and $\mathbb{S}_{rep}(P)$, one would expect these to initially be increasing functions of $P$. However, as $P$ grows and the failure frequency increases, the expected speedups will eventually reach their respective peaks and then start to decline as the growing overhead of failures starts to dominate. The optimal (or equivalently,

the upper bound on) number of processors is the value of $P$ that maximizes the expected speedup (this value will be different for replication and no replication). This is also equivalent to finding $P$ that minimizes $\mathbb{H}_{norep}(P) = 1/\mathbb{S}_{norep}(P)$ (and similarly, with replication), where $\mathbb{H}_{norep}(P)$ (and $\mathbb{H}_{rep}(P)$) is the expected time to finish a unit of work on $P$ processors, without and with replication, respectively.

We can further write $\mathbb{H}_{norep}(P) = \mathbb{E}_{norep}(P)/S_{norep}(P)$ (and similarly for replication), where $\mathbb{E}_{norep}(P)$ (and $\mathbb{E}_{rep}(P)$) represent the time it would take, under failures, to complete $S_{norep}(P)$ (and $S_{rep}(P)$) units of work. Note that, without failures, $S_{norep}(P)$ (and $S_{rep}(P)$) units of work can be completed on $P$ processors in one unit of *time*. Thus $\mathbb{E}_{norep}(P)$ (and $\mathbb{E}_{rep}(P)$) represent the expected time under failures on $P$ processors normalized by the failure-free time on the same number of processors. $\mathbb{E}_{norep}(P)$ and $\mathbb{E}_{rep}(P)$ will be modeled using different approaches, which we discuss below.

### 3.2.1 Expected Time without Replication

For individual processor failures that follow the exponential failure distribution with rate $\lambda$, the resulting failure distribution on $P$ processors without replication is also exponential with rate $\lambda_P = \lambda P$. Thus, one can use the memoryless property to simplify the derivation of expected completion time. The memoryless property ensures that the likelihood of failure within an interval does not depend on when the previous failure happened. Thus, it suffices to estimate $E_{norep,\tau}(P)$, the expected time to finish a single checkpoint interval, which is given by[20]

$$E_{norep,\tau}(P) = (\frac{1}{\lambda P} + D)e^{\lambda PR}(e^{\lambda P(\tau+C)} - 1) \tag{3.1}$$

$\mathbb{E}_{norep}(P)$ can then be estimated as $\mathbb{E}_{norep}(P) = E_{norep,\tau}(P)/\tau$.

### 3.2.2 Expected Time with Replication

When replication is employed, even if the individual failure distributions are exponential, the distribution of failures that interrupt a replicated job execution is not. For any distribution other than the exponential, the memoryless property does not hold. Therefore, the expected time cannot be modeled by estimating the expected time to finish one interval in

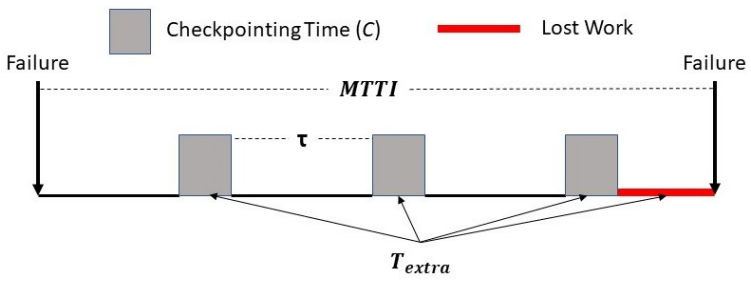



Figure 12: Average behavior between consecutive failures.

isolation. While Equation 3.1 can still be used as an approximation to estimate the expected completion time of a distribution with mean $MTTI_P = 1/\lambda_P$, for replication we will use the generic approximation approach used in [53] which considers each failure (defined as the failure of a processor and its replica) as a renewal process and computes the average time spent performing useful work between such consecutive failures, which is the difference of the duration between these successive failures and the time spent in performing extra tasks, $T_{extra}$, as shown in Figure 12.

$T_{extra}$ consists of two components: the time spent writing checkpoints, and the time spent doing work that was wasted due to failure in the interval in which the failure struck. The average number of checkpoints within two successive failures is given by $MTTI_P/\tau$. Thus, the average time spent writing checkpoints will be given by $C \times MTTI_P/\tau$. The second component, which is the work wasted due to failure, is equal to the expected time of failure within an interval of length $\tau$ given that a failure happens within the interval. This value can be written as $k\tau$, where $0 < k < 1$ represents the expected proportion of an interval that is lost due to a failure. We use the first order approximation[76][68] for the value of $k$ which assumes that failure strikes in the middle of the interval on average, i.e. $k = 0.5$. Putting all of this together, we get that $T_{extra} = C \times MTTI_P/\tau + \tau/2$. We can then write $\mathbb{E}_{rep}(P)$ as

$$\mathbb{E}_{rep}(P) = \frac{MTTI_P}{MTTI_P - \frac{C \times MTTI_P}{\tau} - \frac{\tau}{2}} \tag{3.2}$$

We will thus use the above equation to estimate the expected completion time for dual replication, where $MTTI_P \approx \lambda^{-1}\sqrt{\pi/2P}$ from [7]. It should be noted, however, that Equation

3.2 becomes a less accurate model for the expected completion time as $T_{extra}$ gets closer to $MTTI_P$. However, as long as $T_{extra}$ is less than and not too close to $MTTI_P$, Equation 3.2 provides a close approximation of the expected performance.

## 3.3 Optimal Processor Count

I first investigate the optimal number of processors that maximize the expected speedup under failures. This can be done by setting $\frac{\partial \mathbb{H}_{(no)rep}(P)}{\partial P} = 0$. In order to validate the results that I derive based on this analysis, I also built a simulator to measure the reliability-aware speedups. The basic purpose of simulator is to compute the time to finish a given checkpoint interval under failures generated by a given distribution. The simulator starts at $t = 0$ (where $t$ represents the time to finish one interval) and randomly generates a failure time, say $x$, from the distribution. If $x$ is greater than the interval duration, the run completes, and the interval duration is added to $t$. Otherwise, $x$ is added to $t$, then another draw is made and the process repeated. Upon run completion, I calculate the useful and extra work as shown in Figure 12. I take the average of 50,000 such runs for a given set of system parameters to obtain $\mathbb{E}_{(no)rep}(P)$. When no replication is employed, the failure times are generated according to an exponential distribution with rate $\lambda P$. In the case of replication, since the interrupt distribution is not exponential, I generate failure times according to the actual probability distribution for replicated failures as follows: Let $R(t)$ be the probability that no interrupt (defined as the failure of a processor and its replica) happens until time $t$. Then, with exponential processor failures, $R(t) = (1 - (1 - e^{-\lambda t})^2)^{P/2} = (2e^{-\lambda t} - e^{-2\lambda t})^{P/2}$. The cumulative distribution function (CDF) from which I generate interrupts for replication is then given by $1 - R(t)$.

### 3.3.1 Without Replication

The problem of finding the optimal number of processors with checkpoint-restart alone has been studied to some extent in the literature. Jin et al.[46] and Zheng et al.[82] provided

procedures to numerically evaluate the optimal number of processors given other system and application level parameters. Cavelan et al.[12] derived closed form expressions for optimal processor counts by taking the Taylor series expansion of the exponential term in Equation 3.1 and simplifying using first order approximation for the failure rate ($\lambda_P$). This procedure yields the optimal number of processors as

$$P^*_{norep} = (\frac{1-\alpha}{\alpha})^{2/3}(\frac{2}{\lambda C})^{1/3} \tag{3.3}$$

For perfectly parallel applications, i.e. $\alpha = 0$, this value goes to infinity. However, the optimal number of processors for perfectly parallel applications is a finite value as we will soon see. The discrepancy occurs because the first order approximation holds only when the application workload consists of a non-negligible sequential fraction, $0 < \alpha < 1$.

Since one of the design goals of HPC applications is that they be highly parallelizable, scaling to thousands of cores, the desired value of $\alpha$ for exascale jobs is small in order to fully utilize the system. As $\alpha$ becomes small, however, the accuracy of the first order approximation worsens, as can be seen in Figure 13. In the figure, the optimal number of processors for $\alpha = 0$ is extrapolated and plotted as a horizontal line in order to show the range of $\alpha$ over which this horizontal line (representing the optimal count at $\alpha = 0$) is closer to the actual optimal counts than the counts given by the first order approximation. We can see from the figure that, for values of $\alpha$ below $5 * 10^{-6}$, the actual optimal processor counts are closer to the optimal for $\alpha = 0$ (a perfectly parallel workload) than to the values given by the first order approximation. We carried out similar analyses with other values of processor MTBFs which also revealed that the threshold of $\alpha$, below which the first order approximation become inferior, is of the order of $10^{-6}$ for the range of realistic values of processor MTBF (1-50 years). While this value of $\alpha$ seems quite small by itself, an exascale job with this fraction of sequential workload would not be efficient in utilizing all of its allocated processors. As an example, consider a system with $10^5$ nodes, which, as [1] projects, will be typical of the order of node counts at exascale. On such a system, a job running with $\alpha = 5 * 10^{-6}$ will spend a third of its time in sequential execution, wasting its allocated resources.

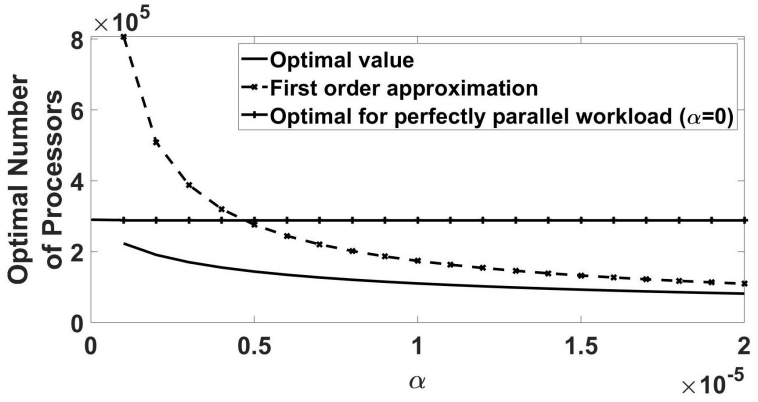



Figure 13: Optimal number of processors when no replication is employed. The actual optimal value is calculated by writing $\mathbb{H}_{norep}(P)$ using Equation 3.1 and numerically locating its minimum. Individual processor MTBF = 10 years while $C = R = D = 300$ seconds.

Based on the discussion above, it is pertinent to also investigate the optimal processor counts of no replication for perfectly parallel jobs ($\alpha = 0$), because the results of such an analysis would serve as better approximations to the performance models of exascale jobs than the first order approximations that assume a relatively larger value of $\alpha$. While [12] observed, based on empirical results, that the optimal number of processors for perfectly parallel jobs is of the order $\lambda^{-1}$, no analytical results to that effect exist. In the theorem below we prove that the optimal processor count is indeed of the order $\lambda^{-1}$ when $\alpha = 0$.

**Theorem 3.** *When using checkpoint-restart without replication, and assuming a perfectly parallel job, the optimal number of processors that maximize the expected speedup is equal to $K/\lambda$, where $K$ is a constant that does not depend on $\lambda$. In other words, the optimal number of processors is directly proportional to the individual node MTBF.*

*Proof.* Without replication, for a system comprising of $P$ processors, each having an exponential failure distribution with rate $\lambda$, the system failure distribution is also exponential with rate $\lambda P$. Hence, one can write $\mathbb{H}_{norep}(P)$ exactly using Equation 3.1 (which applies to exponential failure distributions). Taking the checkpointing interval to be $\tau = \sqrt{2C/\lambda P}$ (using Young's first order approximation[76]) and assuming a perfectly parallel job (i.e

$S_{norep}(P) = 1/P$), we get from Equation 3.1 the following expression

$$\begin{aligned}\mathbb{H}_{norep}(P) &= \frac{1}{P}(\frac{1}{\lambda P} + D)e^{\lambda_P R}\frac{(e^{\lambda P(\sqrt{\frac{2C}{\lambda P}}+C)} - 1)}{\sqrt{\frac{2C}{\lambda P}}} \\ &= \sqrt{\frac{\lambda}{2C}}(\frac{1}{\lambda P^{3/2}} + \frac{D}{\sqrt{P}})(e^{\lambda P(\sqrt{\frac{2C}{\lambda P}}+C+R)} - e^{\lambda P R})\end{aligned} \tag{3.4}$$

Let $H(P) = (\frac{1}{\lambda P^{3/2}} + \frac{D}{\sqrt{P}})(e^{\lambda P(\sqrt{\frac{2C}{\lambda P}}+C+R)} - e^{\lambda P R})$ (i.e. ignoring the constant $\sqrt{\lambda/2C}$ above). Differentiating wrt $P$, we get

$$\begin{aligned}\frac{\partial H}{\partial P} = (\frac{1}{\lambda P^{\frac{3}{2}}} + \frac{D}{\sqrt{P}})e^{\lambda P R}((\sqrt{\frac{\lambda C}{2P}} + \lambda(C+R))e^{\lambda P(\sqrt{\frac{2C}{\lambda P}}+C)} \\ - \lambda R) - (\frac{3}{2\lambda P^{\frac{5}{2}}} + \frac{D}{2P^{\frac{3}{2}}})(e^{\lambda P(\sqrt{\frac{2C}{\lambda P}}+C+R)} - e^{\lambda P R})\end{aligned} \tag{3.5}$$

Setting $\frac{\partial H}{\partial P} = 0$ and simplifying, we obtain

$$2\lambda P((\sqrt{\frac{C}{2\lambda P}} + C + R)e^{\lambda P(\sqrt{\frac{2C}{\lambda P}}+C)} - R) = (\frac{3 + \lambda P D}{1 + \lambda P D})(e^{\lambda P(\sqrt{\frac{2C}{\lambda P}}+C)} - 1) \tag{3.6}$$

Setting $P = K/\lambda$, Equation 3.6 becomes

$$2K((\sqrt{\frac{C}{2K}} + C + R)e^{K(\sqrt{\frac{2C}{K}}+C)} - R) = (\frac{3 + KD}{1 + KD})(e^{K(\sqrt{\frac{2C}{K}}+C)} - 1) \tag{3.7}$$

We note that $\lambda$ has been eliminated from the equation above. This means that the value of $K$ that satisfies this equation does not depend on $\lambda$, thus concluding the proof. □

I use the simulator to validate the result in Theorem 3 about the order of optimal processor count without replication for perfectly parallel jobs. Figure 14 shows the optimal number of processors as a function of the individual processor MTBF($= 1/\lambda$). Along with the simulation results, we also plot a best fit curve to the simulation results. The form of the curve is assumed to be $K/\lambda$ for the case of $\alpha = 0$ and $K/\lambda^{1/3}$ (based on Equation 3.3) when $\alpha > 0$. We see that the best fit curve for perfectly parallel workload, using the form given by Theorem 1, matches closely with simulation results. When $\alpha > 0$, the form given by the first order approximation gets closer to the simulation results as $\alpha$ gets larger. Note that, in this analysis, the value of $K$ has been estimated using simulation, since the purpose was to assess the growth of $P^*$ with $\lambda$. For $\alpha > 0$, the first order approximation provides a formula

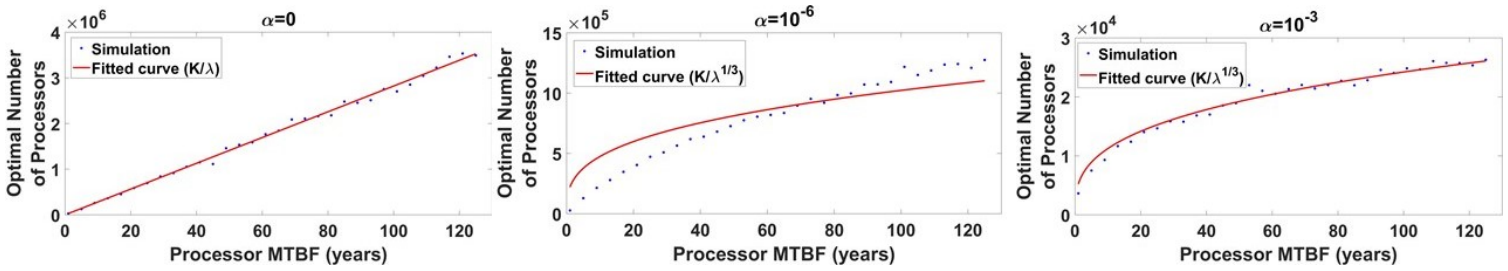



Figure 14: Optimal number of processors when no replication is employed. $C = R = D = 300$ seconds. The scale of y-axis is different for the three plots.

for $K$ as can be seen in Equation 3.3. As for perfectly parallel jobs, we see that Equation 3.7 cannot be solved analytically for $K$. In such a case, one can resort to numerical methods to solve that equation and compute $K$ given $C$, $R$ and $D$. An alternative would be the method we used here, which is to estimate $K$ using simulation, as shown in Figure 14.

### 3.3.2 Replication

For replication, I study in this dissertation the smallest degree of replication, i.e. dual replication. As mentioned in Section 3.2, we will use the result from [7] to approximate the platform failure rate, $\lambda_P$. Thus, for a platform with a total of $P$ processors where half of them are replicas of the other half, we take the failure rate to be $\lambda_P \approx \lambda\sqrt{2P/\pi}$. We will therefore take the checkpointing interval to be $\tau = \sqrt{\frac{2C}{\lambda}\sqrt{\frac{\pi}{2P}}}$. Using Equation 3.2 and the expression of $S_{rep}(P)$, we get the following approximation for $\mathbb{H}_{rep}(P)$

$$\mathbb{H}_{rep}(P) \approx (\alpha + \frac{2(1-\alpha)}{P})\frac{\frac{1}{\lambda}\sqrt{\frac{\pi}{2P}}}{\frac{1}{\lambda}\sqrt{\frac{\pi}{2P}} - \frac{\frac{C}{\lambda}\sqrt{\frac{\pi}{2P}}}{\tau} - \frac{\tau}{2}} = (\alpha + \frac{2(1-\alpha)}{P})\frac{1}{(1 - \sqrt{2\lambda C\sqrt{\frac{2P}{\pi}}})} \quad (3.8)$$

I will now discuss the optimal processor counts for replication for the two cases, $\alpha > 0$ and $\alpha = 0$, separately:

*Non-negligible Sequential part ($\alpha > 0$)*: From Equation 3.8, we can expand the $(1 - \sqrt{2\lambda C\sqrt{\frac{2P}{\pi}}})^{-1}$ term, using Taylor expansion, as $\sum_{j=0}^{\infty}(2\lambda C\sqrt{\frac{2P}{\pi}})^{\frac{j}{2}}$. We now make the first

order approximation by taking the first couple of terms of this series to obtain

$$\mathbb{H}_{rep}(P) \approx (\alpha + \frac{2(1-\alpha)}{P})(1 + \sqrt{2\lambda C\sqrt{\frac{2P}{\pi}}}) \tag{3.9}$$

In order for the approximation above to be valid, the term $(2\lambda C\sqrt{\frac{2P}{\pi}})^{\frac{j}{2}}$ should get smaller as $j$ increases. Thus, by making the above approximation, we are assuming that the term $2\lambda C\sqrt{\frac{2P}{\pi}}$ is of the order $\lambda^{\epsilon}$ where $\epsilon > 0$, as this would make the earlier terms of the series dominant (since $\lambda$ is small in practice). This translates to the condition that if $P$ is of the order $\lambda^{-x}$, where $x < 2$, the first order approximation will hold, which is the assumption we make here. Differentiating the approximate expression in Equation 3.9 with respect to $P$, we get

$$\frac{\partial \mathbb{H}_{rep}}{\partial P} = \frac{\alpha}{4}\sqrt{\frac{2\lambda C}{P}\sqrt{\frac{2}{\pi P}}} - \frac{2(1-\alpha)}{P^2} + \Theta(\lambda^{\frac{1}{2}}P^{\frac{-7}{4}}) \tag{3.10}$$

With our assumption on the order of $P$ (i.e. $P = \Theta(\lambda^{-x})$ where $x < 2$), the last term in the equation above is negligible compared to the first two terms. Thus, by setting $\frac{\partial \mathbb{H}_{rep}}{\partial P} = 0$ using the most dominant terms, we obtain the optimal processor count as

$$P^*_{rep} = (\frac{8(1-\alpha)}{\alpha\sqrt{2\lambda C}}(\frac{\pi}{2})^{\frac{1}{4}})^{\frac{4}{5}} \tag{3.11}$$

Note that, according to this expression, when $\alpha > 0$, $P^*_{rep}$ is of the order $\lambda^{-\frac{2}{5}}$, which indeed satisfies the condition on the order of $P$ mentioned above (i.e. $P = \Theta(\lambda^{-x})$, $x < 2$) and thus justifies the first order approximation made in obtaining this optimal value. When $\alpha$ approaches 0, the expression above goes to infinity. However, the optimal number of processors in that case is still finite as we will see shortly. The reason for this discrepancy, same as with no replication, is that the optimal processor count for perfectly parallel workloads ($\alpha = 0$) is larger, which means that the condition for the first-order approximation to be valid, i.e. $P = \Theta(\lambda^{-x})$ where $x < 2$, does not hold in that case.

Figure 15 plots the first order approximation derived above along with optimal processor counts obtained using simulation results, when $\alpha = 10^{-6}$. We see that the values of $R$ and $D$ do not have a significant impact on the actual value of optimal processor counts, and also that the first order approximation is quite accurate in determining those counts.

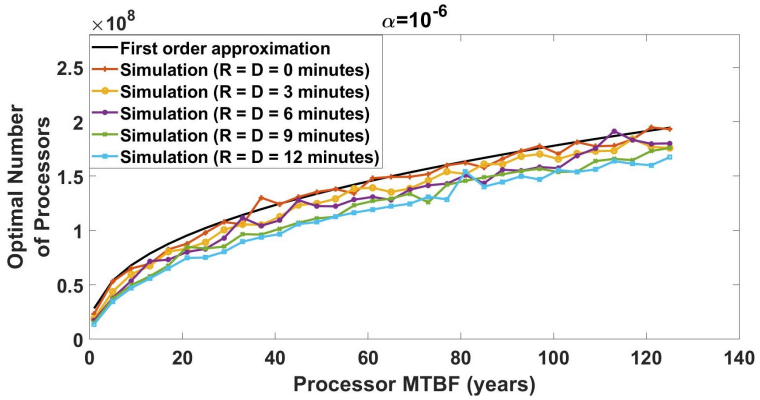



Figure 15: Optimal number of processors with dual replication. Checkpointing cost = 300 seconds, same as in Figure 14.

Similar to the case of no replication, I also analyse the values of $\alpha$ for which the first order approximation is a closer match to the actual optimal counts for replication. Figure 16 shows that, for values of $\alpha$ greater or equal to $10^{-10}$, the first order approximation is closer to the actual counts than the perfectly parallel approximation. Recall from Figure 13 that this threshold for $\alpha$ without replication, assuming a system with the same parameters, was much higher (around $5 * 10^{-6}$). This means that with replication, the first order approximation will serve as a good fit for a much larger set of parallel workloads, even if they are highly scalable, as long as they are not embarrassingly parallel.

*Perfectly Parallel workload ($\alpha = 0$)*: Although we cannot make the simplifying first-order approximation as above, setting $\alpha = 0$ simplifies Equation 3.8, yielding $\mathbb{H}_{rep}(P) \approx 2/P(1 - \sqrt{2\lambda C\sqrt{\frac{2P}{\pi}}})$. Taking the derivative, we get

$$\frac{\partial \mathbb{H}_{rep}}{\partial P} = \frac{-2(1 - 5\sqrt{2\lambda C\sqrt{2P/\pi}}/4)}{P^2(1 - \sqrt{2\lambda C\sqrt{2P/\pi}})^2} \tag{3.12}$$

Setting the derivative equal to 0 yields the optimal value as

$$P^*_{rep} \approx \frac{32\pi}{625\lambda^2 C^2} \tag{3.13}$$

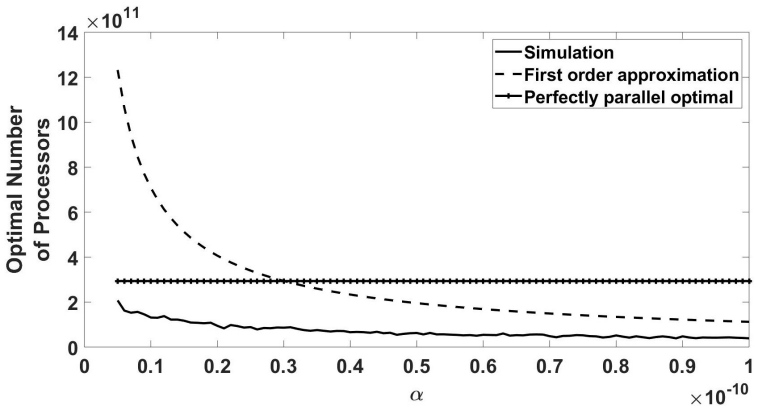



Figure 16: Optimal number of processors with dual replication, as obtained by the simulation as well as the first order approximation (Equation 3.11). X-axis range is from $\alpha = 0$ to $\alpha = 10^{-10}$. Individual processor MTBF = 10 years while $C = R = D = 300$ seconds.

While this procedure yielded an exact expression for the optimal number of processors, it should be noted that this value is optimal for the approximate expression used in Equation 3.8. To assess the accuracy of this approximation, Figure 17, similar to Figure 15, plots the optimal processor counts, this time for a perfectly parallel workload, along with our derived formula. Comparing with Figure 15, we first note that the counts are significantly lower with a non-zero value of $\alpha$, which is to be expected. We also see that, while the parameters $R$ and $D$ did not have much impact on the optimal processor count for non-zero $\alpha$, their values have a non-negligible impact on the optimal processor counts for perfectly parallel workloads. This trend is similar to the case of no replication, where $R$ and $D$ do not have much impact when $\alpha > 0$ since they vanish from the first-order approximation[12], but have a greater effect on the optimal counts for perfectly parallel workloads. We thus conclude from Figure 17 that the actual optimal number of processors for perfectly parallel workloads also depends on the recovery cost $R$ and the downtime $D$, which the model used to write Equation 3.8 does not take into account. Nevertheless, the derivation based on this model yields a simple and handy expression which, as seen in the plot, is close to the optimal processor counts in practice.

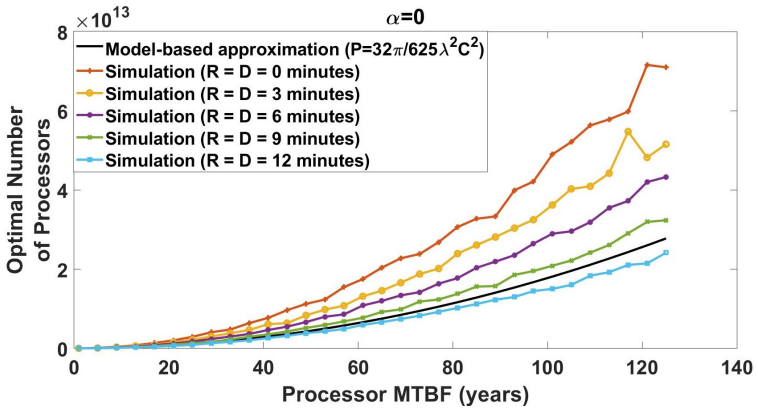



Figure 17: Optimal number of processors with dual replication for a perfectly parallel workload. Checkpointing cost = 300 seconds.

An additional benefit of the approximation we just derived is that it leads us to the observation that the optimal number of processors using dual replication is of the order $\lambda^{-2}$. Thus, similar to Figure 14, where we investigate the dependence of the optimal counts on $\lambda$ for no replication, we plot, in Figure 18, best-fit curves over simulation based optimal processor counts of replication using the forms indicated by our derivations (i.e. $P^*_{rep} = K/\lambda^2$ when $\alpha = 0$ and $P^*_{rep} = K/\lambda^{2/5}$ when $\alpha > 0$). We see that the best-fit curve for $\alpha = 0$ matches well with the simulation results. For $\alpha > 0$, the fitted curve gets closer in shape to the actual counts as $\alpha$ gets larger, similar to the observations made in Figure 14 for no replication. This means that the optimal processor counts in both cases agree with our derived formulae on their order in terms of $\lambda$. Thus, our formulae serve as reasonable approximations to the optimal processor counts for dual replication.

Based on the results so far, I summarize my findings about the optimal processor counts in this section in Table 2, which lists the form of the optimal counts in terms of $\lambda$ as well as closed form approximations where available. We see that, in both cases of parallelism, the optimal counts of replication are of a higher order in terms of the processor MTBF ($= 1/\lambda$) when compared with their counterparts for no replication. Thus, in each case we can say that the range of system scales over which one can continue to improve performance by enrolling

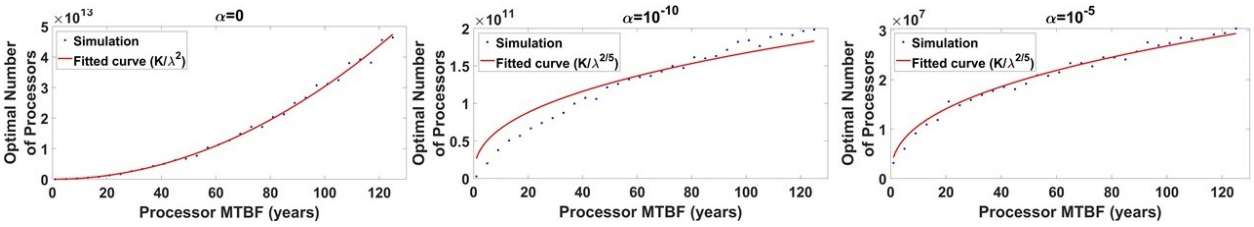



Figure 18: Optimal number of processors with replication. $C = R = D = 300$ seconds. Scale of y-axis is different for each plot.

more processors is much larger with replication than without it.

### 3.4 Performance comparison of Replication with No Replication

In the previous section we saw that the optimal processor counts for replication are much higher than those possible without replication. However, that does not necessarily mean that the reliability-aware speedup of replication will actually be better than what is possible without replication at higher processor counts. This is because the optimal processor counts were optimal for their specific fault tolerance mechanisms, i.e. replication and no-replication respectively, and so none of our earlier results say how the speedup of replication compares

Table 2: Optimal Processors Counts

| | C/R without Replication | C/R with Replication |
|---|---|---|
| $\alpha > 0$ | $\Theta(\lambda^{-1/3})$<br>$P^* \approx (\frac{1-\alpha}{\alpha})^{2/3}(\frac{2}{\lambda C})^{1/3}$[12] | $\Theta(\lambda^{-2/5})$<br>$P^* \approx (\frac{8(1-\alpha)}{\alpha\sqrt{2\lambda C}}(\frac{\pi}{2})^{\frac{1}{4}})^{\frac{4}{5}}$ |
| $\alpha = 0$ | $\Theta(\lambda^{-1})$ | $\Theta(\lambda^{-2})$<br>$P^* \approx \frac{32\pi}{625\lambda^2 C^2}$ |

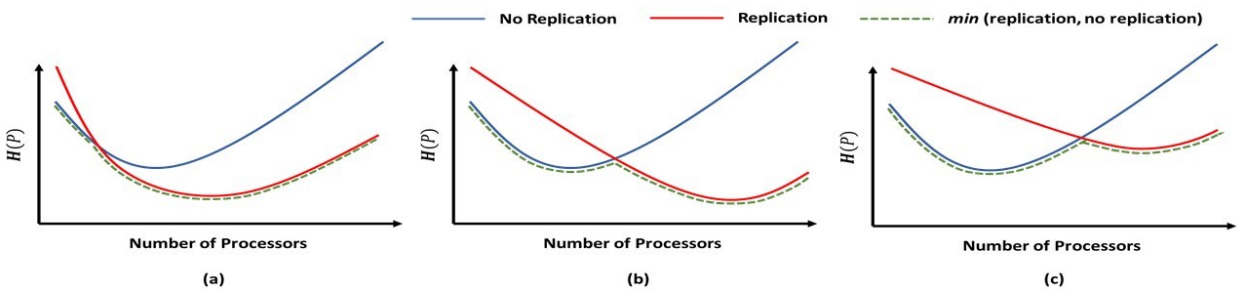



Figure 19: Different possibilities of how the performance of replication and no replication compare to each other as the number of processors increases.

with no replication.

When comparing the performance of no-replication with replication with respect to the scale of the system, we know that replication always starts off worse than no-replication at lower processor counts, since the system failure rate is low, in which case redundancy is an overkill. We also know from [29] and [11] that at some point replication starts outperforming no-replication and this trend subsequently holds for increasing system sizes. Depending on when that crossover happens, though, we get the different possibilities depicted in Figure 19. Note that the figure plots the normalized expected completion time which is minimum at the optimal number of processors, since it is the inverse of reliability-aware speedup.

If we define the global speedup at any processor count as the best speedup possible at that scale (i.e. the better of replication or no replication at that processor count), then it is interesting to explore the form and behavior of this global speedup function in terms of the performance of replication and no-replication. Figure 19 shows normalized expected completion time of such a global speedup in each case as a dashed green curve. Finding out which of those three scenarios is true in practice would have implications on the feasibility of replication. For example, if either of subfigure (a) or (b) represents the true form of the global speedup function, then the optimal global speedup will be achieved at the optimal of replication. However, only in scenario (a) does the global speedup continuously improve until that optimal is reached. If, on the other hand, scenario (b) is how the speedup behaves, it

would be surprising since that would mean that the speedup improves until some system scale (i.e. the optimal of no-replication), then is worse for larger system scales until some point, but after that point it starts improving once again before hitting the optimal of replication. Finally, subfigure (c), if true, would mean that the outlook for replication is bleak since it would never reach the optimal performance of no-replication and that the global speedup reaches its optimal with C/R alone without replication. The findings in this dissertation suggest that, of these three scenarios, scenario (a) in Figure 19 is what seems to be true in practice, which is what I will demonstrate through theoretical and simulation-based analyses in the rest of this section.

### 3.4.1 Theoretical Analysis

My approach in this section will be to theoretically compare the reliability aware speedups of replication and no-replication at the optimal processor counts for no-replication. The rationale for this analysis is that, if replication outperforms no-replication at the optimal processor counts of no-replication, it will mean that the global speedup behavior will be according to scenario (a) shown in Figure 19. Below we discuss our analysis for the two cases of parallelism, based on the value of $\alpha$.

*Non-negligible Sequential part ($\alpha > 0$)*: When $\alpha > 0$, the authors in [12] showed that for no-replication at its optimal processor count, $P^*_{norep}$, the expected time can be written as $\mathbb{H}^*_{norep} = \mathbb{H}_{norep}(P^*_{norep}) \approx \alpha + 3(\alpha^2(1-\alpha)\lambda/2)^{1/3}$. We can also plug their expression for $P^*_{norep}$ (Equation 3.3) in the first-order approximation for the expected time of replication, as given by Equation 3.9. The resulting expression, ignoring higher order terms, is

$$\mathbb{H}_{rep}(P^*_{norep}) \approx \alpha + (4\alpha^2(1-\alpha)\lambda/2)^{1/3} \tag{3.14}$$

Comparing the two expressions, we can clearly see that $\mathbb{H}_{norep}(P^*_{norep}) > \mathbb{H}_{rep}(P^*_{norep})$ which means that the performance of replication at $P^*_{norep}$ is better than the performance without replication. Thus, for workloads with non-negligible sequential part, we can expect that replication will start outperforming no replication before the optimal processor counts without replication are reached.

*Perfectly Parallel workload ($\alpha = 0$)*: When $\alpha = 0$, we do not have an explicit formula for the optimal number of processors without replication. We will, therefore, perform a simplified analysis by assuming that the recovery cost and downtime are both zero. Taking $R = D = 0$ in Equation 3.4 for no replication, we get

$$\mathbb{H}_{norep}(P) = \frac{(e^{\lambda P(\sqrt{\frac{2C}{\lambda P}}+C)} - 1)}{\sqrt{2\lambda C P^3}} \tag{3.15}$$

Taking the derivative with respect to $P$, setting it equal to 0 and simplifying, we obtain the following equation

$$(\lambda PC + \sqrt{\frac{\lambda PC}{2}})e^{\lambda PC+\sqrt{2\lambda PC}} - \frac{3(e^{\lambda PC+\sqrt{2\lambda PC}} - 1)}{2} = 0 \tag{3.16}$$

Let $x = \lambda PC$, then the equation above reduces to

$$(x + \sqrt{x/2})e^{x+\sqrt{2x}} - 3(e^{x+\sqrt{2x}} - 1)/2 = 0 \tag{3.17}$$

We can solve this equation numerically to obtain $x \approx 0.68015$. Note that this value of $x$ is determined solely from the equation above and is independent of the values of $\lambda$ and $C$. This means that $\lambda PC$ is an invariant with respect to $\lambda$ and $C$ when $P$ represents the optimal processor counts and this invariant can be determined from the equation above. We therefore obtain that the optimal processor count in this case is $P^*_{norep} = x/\lambda C \approx 0.68015/\lambda C$. Plugging this value into Equation 3.15, we get the normalized expected time for no replication at its optimal processor count as

$$\mathbb{H}_{norep}(P^*_{norep}) = \mathbb{H}_{norep}(\frac{x}{\lambda C}) = \lambda C(\frac{e^{x+\sqrt{2x}} - 1}{x\sqrt{2x}}) \approx 6.7283\lambda C \tag{3.18}$$

Using Equation 3.8, we can write the performance of replication at this value of $P$ as

$$\mathbb{H}_{rep}(P^*_{norep}) = \mathbb{H}_{rep}(\frac{x}{\lambda C}) = \frac{2\lambda C}{x(1 - \sqrt{2\lambda C\sqrt{\frac{2x}{\pi\lambda C}}})} \tag{3.19}$$

By comparing the two equations above, we can derive that $\mathbb{H}_{norep}(P^*_{norep}) > \mathbb{H}_{rep}(P^*_{norep})$, i.e. replication outperforms no replication at $P^*_{norep}$, whenever

$$\lambda C < \frac{\pi}{8x}(1 - \frac{2\sqrt{2x}}{e^{x+\sqrt{2x}} - 1})^4 \approx 0.058 \tag{3.20}$$

This means that, whenever $\mu = 1/\lambda$ is greater than $C/0.058 \approx 17.25C$, the expected performance of replication as given by the approximation in Equation 3.8 will be better than the performance of no replication. This bound is satisfied by all realistic values of node MTBFs and checkpointing costs, since individual node MTBFs usually are much higher than the checkpointing cost. As an example, let's say that the checkpointing cost for a platform is 1 hour (a conservative estimate, given the state of the art). Even with such a high value of C, the bound above says that if individual node MTBF is higher than 17.25 hours (which usually is the case since the node MTBF usually is of the order of years), then the global speedup behavior will be according to scenario (a) in Figure 19. With lower values of $C$, this threshold will be even lower, which again should be satisfied by all practical node MTBF values. This means that, for all realistic platform values of $\lambda$ and $C$, replication would have already outperformed no-replication by the time we reach the optimal processor counts for no-replication.

### 3.4.2 Empirical Evaluation

The theoretical results in the previous section give a very strong indication that replication outperforms no-replication at the optimal processor counts of no-replication, which in turn means that the global normalized expected completion time has the form depicted in subfigure (a) in Figure 19. However, the two limiting factors in the theoretical analysis were i) the contributions from restart time, $R$, and the downtime, $D$, both of which are non-zero in practice, were ignored for the case of perfectly parallel workloads, and ii) the expression for $\mathbb{H}_{rep}(P)$ is an approximation since it uses the approach in Figure 12, unlike $\mathbb{H}_{norep}(P)$, for which we have the exact expression using Equation 3.1. Therefore, in this subsection I use simulation to investigate if the results of the theoretical analysis hold in practice, despite the above mentioned limitations. Therefore, I obtain $\mathbb{H}_{rep}(P^*_{norep})$ as follows:

1. For a given set of parameters ($\lambda, C, R$ and $D$), we first numerically find the optimal $P^*_{norep}$ of $\mathbb{H}_{norep}(P)$, using its exact formula.
2. For the $P^*_{norep}$ found in step 1, we use our simulator to estimate $\mathbb{H}_{rep}(P^*_{norep})$.

Figure 20 compares $\mathbb{H}_{rep}(P^*_{norep})$ obtained using the steps above versus the expression for

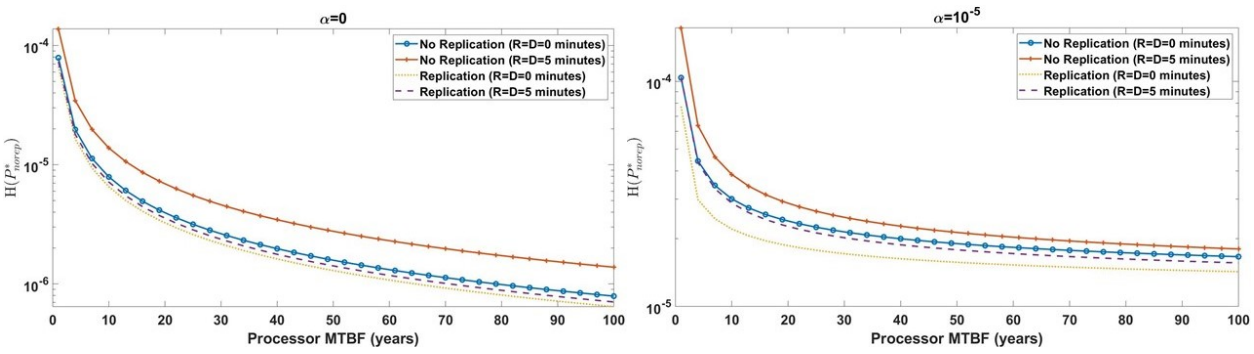



Figure 20: The normalized expected completion time, $\mathbb{H}(P^*_{norep})$ of no replication and replication. For both figures, $C = 300$ seconds. Note: Y-axis scale is different in the two plots.

$\mathbb{H}_{norep}(P^*_{norep})$, for the two cases of parallelism (i.e. perfectly parallel and $\alpha > 0$). It can be seen that, with other parameters being the same, replication always has lower normalized expected completion time than no-replication. Remember that for perfectly parallel jobs ($\alpha = 0$), our theoretical analysis was carried out with the assumption that $R = D = 0$. While that analysis is validated by the figure, we also see that the difference in the performance of replication and no-replication is even greater for the $\alpha = 0$ case when $R$ and $D$ are non-zero. Hence, the results from our empirical analysis further strengthen our conclusion that replication outperforms no-replication at the optimal processor counts for no replication.

Based on the theoretical and empirical results in this section, we can conclude that, for all practical values of the platform parameters, by the time the optimal number of processors for no-replication is reached, replication already offers better performance. Thus, the global speedup ($= \min(\mathbb{H}_{norep}(P), \mathbb{H}_{rep}(P))$) is monotonic with respect to the number of processors up until the optimal count of processors for replication is reached. Furthermore, the global optimum is achieved at processor counts that are optimal for replication.

### 3.5 Overhead of Replication

The assumption so far in this chapter has been that the only hit to replication is the doubling of failure free execution time of the parallel part of a job because of duplicated work. However, as several prior studies on replication[29][27][26][43] have noted, replication also induces an additional overhead to message passing applications because of several factors, such as the additional communication required between replicas in order maintain consistency among them, increase in memory utilization and network congestion. In this section I assess how this additional overhead affects the reliability aware speedup of replication, in contrast to no-replication which does not suffer from any such overheads.

For the cost of replication, I use the model of [29]. They analyzed the overhead of replication on several message passing applications and used curve fitting to infer that the growth of the overhead of replication is proportional to the logarithm of the number of processors, $P$. Thus, I assume that the extra overhead induced by replication, as a fraction of the original time, is equal to $\delta \log P$, where $\delta$ is an application specific constant. To obtain a formula for the speedup, assume the original work to be completed takes $W$ units of time on a single processor without failures and fault tolerance. Without the additional overhead of replication, the same time, using $P$ total processors, would become $W(\alpha + 2(1-\alpha)/P)$ in the absence of failures and without C/R. With the additional overhead of replication, this time would become $W(\alpha + 2(1-\alpha)/P)(1 + \delta \log P)$. The failure free speedup can then be obtained as $S_{rep}(P) = \frac{W}{W(\alpha+2(1-\alpha)/P)(1+\delta \log P)} = \frac{1}{(\alpha+2(1-\alpha)/P)(1+\delta \log P)}$. Thus, we update $\mathbb{H}_{rep}(P)$ to use this expression for $S_{rep}(P)$.

Figure 21 shows the impact of overhead with different values of $\delta$. For no-replication, the curve is generated through the exact expression for $\mathbb{H}_{norep}(P)$. For replication, I evaluate $\mathbb{H}_{rep}(P)$ through simulation with the updated expression for $S_{rep}(P)$ as described above. We can see from the figure that, for values of $\delta = 10^{-2}$ or lower, replication still outperforms no replication before no replication reaches its optimal. It should also be mentioned that the value of $\delta$ determined in [29] for the application with the highest overhead of replication was of the order $10^{-3}$. Moreover, this value could be even lowered by leveraging application specific properties of most message passing applications[52]. Thus we can say that the

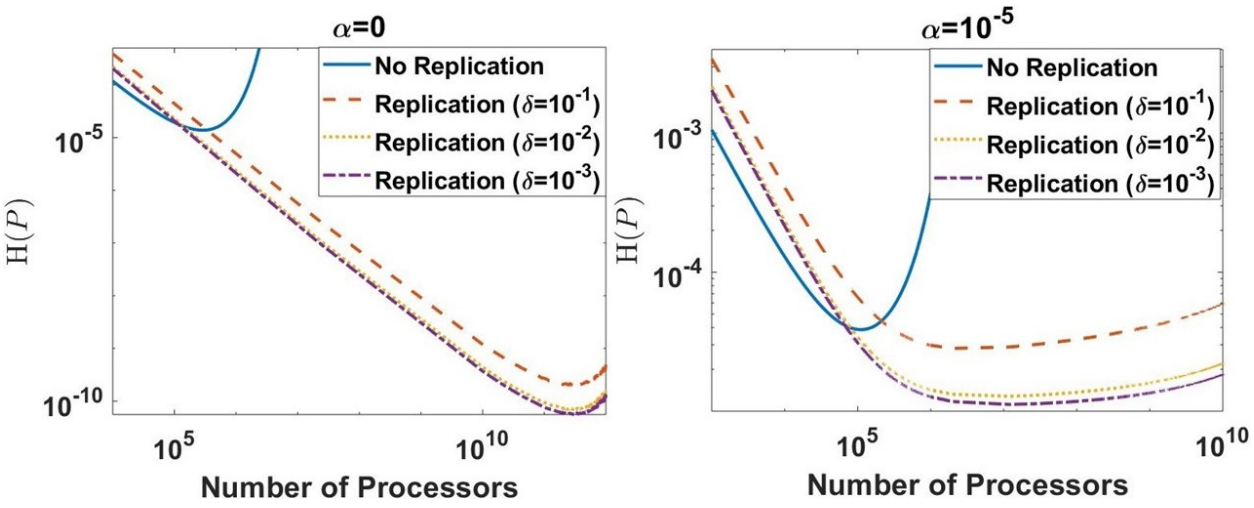



Figure 21: Normalized expected completion time versus the number of processors. Node MTBF = 10 years, while $C = R = D = 300$ seconds. Note: Y-axis scale is different in the two plots.

overhead of replication, as long as it is not unreasonably high, does not change the form of the global speedup function, which would still behave according to scenario (a) in Figure 19.

We can also see from Figure 21 that the performance of replication at its optimal is superior to the optimal performance of no replication for all values of $\delta$. This means that even if the replication overhead grows large enough, the form of the global speedup function shifts from that in scenario (a) to that in scenario (b) in Figure 19. We did observe in our investigations that further increasing $\delta$ does cause the speedup to behave as in scenario (c) in Figure 19, which would render replication infeasible at any processor count. However, this behavior is only exhibited when unreasonably high values for the parameter $\delta$ are assumed. Finally, we note from Figure 21 (right) that, when $\alpha > 0$, the crossover between no-replication and replication happens much closer to the optimal processor count of no-replication. However, all the observations made above regarding the form of the overall global speedup hold for both cases, i.e. $\alpha = 0$ and $\alpha > 0$.

We can thus conclude from this section that, while impact of the overhead of replication is to diminish the reliability-aware speedup of replication, it does not change any of the

conclusions in this work about how replication fares against no-replication with scale. Only if the overhead is impractically high, replication may be rendered suboptimal at any processor count in comparison with the optimal performance of no-replication. However, for overhead numbers observed for replication in practice, the reliability-aware speedup still outperforms no-replication at the optimal processor counts of no-replication, and can significantly improve on the optimal performance that is possible without using replication.

## 3.6 Related Work

The most closely related body of work to this chapter is the study of reliability-aware speedups. In that domain, [46] and [82] formulated reliability aware speedups for checkpoint-restart (C/R) and [82] numerically computed the optimal number of processors. [12] provided theoretical results on the optimal number of processors for non-perfectly parallel jobs. I follow their approach in formulating the optimal processor count problem, and also extend their results to cover the case of perfectly parallel jobs. All of these works, however, consider C/R only without replication.

There have been several studies in the HPC domain that have studied the idea of combining replication with C/R. [29] suggested replication as a viable fault tolerance scheme for exascale HPC systems by showing that, at sufficiently large scales, C/R alone will be less efficient that C/R with replication. [11] theoretically studied replication and derived a summation based formula for the mean-time-to-interrupt (MTTI) of a replicated execution. The authors in [7] recently derived a closed form expression for the MTTI in the case of dual replication. There have been several other works investigating both the implementation[27][52] and theoretical[5] issues surrounding replication. These works do not consider the problem of finding the optimal number of processors using replication, with the exception of [5]. However, [5] focuses primarily on silent errors because of which their model considers the failure of even one processor (silent or fail-stop) in a replica-pair as a failure for the entire job. Thus, their results are not applicable to the fail-stop model in which only the failure of both processors in the replica-pair causes a failure to the job, which is what I study in

this dissertation. Additionally, none of the above works on replication assess how replication can impact the overall form of reliability-aware speedups, which I do in this chapter by showing that replication doesn't simply start outperforming no-replication after some system scales, but rather outperforms the optimal achievable by no-replication and that the crossover happens before or close to the optimal system scale for no replication.

## 3.7 Summary

In this chapter, I studied the reliability-aware speedup of a replicated execution, and contrasted it with the reliability-aware speedup without replication. I derived novel results on how the optimal processor counts of replication and no-replication relate to the individual node failure rate $\lambda$. I further showed that replication generally starts outperforming no-replication before or close to the point where no-replication reaches its optimal processor counts. Taken collectively, the results in this chapter indicate that replication significantly enhances reliability-aware speedup beyond what is possible without replication.

## 4.0 Partial Replication under Heterogeneous Failure Likelihoods

The previous chapter strengthened the case for considering dual replication, paired with checkpoint-restart (C/R), as a superior fault tolerance scheme to C/R alone without replication when system scale is large. In this chapter, I investigate how replication can utilize, and be affected by, the heterogeneity in failure likelihoods. The main contribution of this chapter is to demonstrate that such heterogeneity is the key to making partial replication feasible[1]. I first start with a discussion on the motivations behind considering partial replication instead of full replication.

## 4.1 Motivation Behind Partial Replication

In pure replication with dual redundancy, all work is duplicated on separate hardware resources. This allows the application to continue execution even in the presence of failures as long as both processes (or nodes) that are replicas of each other have not failed. This significantly improves the mean time to interrupt (MTTI) of the system[29][11], requiring fewer checkpoints compared to the case without replication. However, it comes at the cost of system efficiency, which is capped at 50%. Hence, the argument for pure replication as a fault tolerance mechanism holds weight only at system scales at which the efficiency of checkpointing alone drops below 50%.

To break the 50% efficiency barrier of pure replication, [26] studied partial replication where only a subset of application visible nodes are replicated. However, neither [26] nor any other works since then have established any range of node counts for which it makes sense to only partially replicate an execution. In this chapter I revisit partial replication under the assumption that node failure likelihoods in the system are not necessarily identical and show that this change in the assumption makes partial replication superior to full and no replication for a range of system scales.

[1]This work appeared in Supercomputing (SC) 2018.

## 4.2 Replica Selection and Pairing

I start with the question of how, knowing the number of nodes to replicate, should the replicated nodes be selected and paired. Consider a system with $N$ nodes with individual node failure density functions given by $h_i(t), 1 \leq i \leq N$, where $t > 0$ is the time. These functions are typically taken to be exponential or Weibull, and characterized by failure rate $\lambda_i$, where $\lambda_i$ is the inverse of node $i$'s MTBF. We assume without loss of generality that the nodes are ordered by their failure rates, such that $\lambda_i(t) \leq \lambda_{i+1}(t)$ for all $1 \leq i \leq N-1$. In order to answer the question of optimal selection and pairing of replicas, it is simpler to work with the nodes' probability of survival until time $t$ (or *reliability*) given by $g_i(t) = 1 - \int_0^t h_i(x)dx$, $1 \leq i \leq N$. With the nodes sorted by increasing failure rates, we see that $g_i(t) \geq g_{i+1}(t)$ for all $1 \leq i \leq N-1$.

Assume, for now, that a particular job requires $n$ nodes to execute in parallel, where $n \leq N$. Moreover, assume that the remaining $N-n$ nodes are to be used as replicas of some of the $n$ nodes, in order to provide better protection from failures. We will relax these assumptions in subsequent sections to make $n$ variable in order to explore if partial replication is beneficial at all. For now, however, I try to answer the first question: Which of the $n$ nodes should have replicas, and how should they be paired with the other $N-n$ nodes to form node-replica pairs? We restrict ourselves to maximum dual node replication only, so $N/2 \leq n \leq N$. In such a configuration, let $a = n - (N-n) = 2n - N$ be the number of non replicated nodes, and $b = n - a = N - n$ be the number of node replica pairs, such that $a + 2b = N$ and $a + b = n$. The partial replication factor, $r$, will thus be given by $r = (a+2b)/(a+b)$, and $1 \leq r \leq 2$. The original question can thus be reformulated as: Given values of $a$ and $b$ and reliability $g_i(t), 1 \leq i \leq N$, which $2b$ out of the $N$ nodes should be replicated and how should the replicated nodes be paired so that overall system reliability is maximized? The answer is to pick the least reliable $2b$ nodes for replication. Among those $2b$ nodes, the least reliable node should be paired with the most reliable node, and so on. This is shown in Fig 22, and formally stated in the following theorem:

**Theorem 4.** *Given $a$, $b$ and an $N$ node system ($a + 2b = N$) with node reliability given by $g_i(t)$ and $g_i(t) \geq g_{i+1}(t)$ for $1 \leq i \leq N-1$, let $A \subseteq \{1, 2, \ldots, N\}$, $|A| = a$, be the set of*

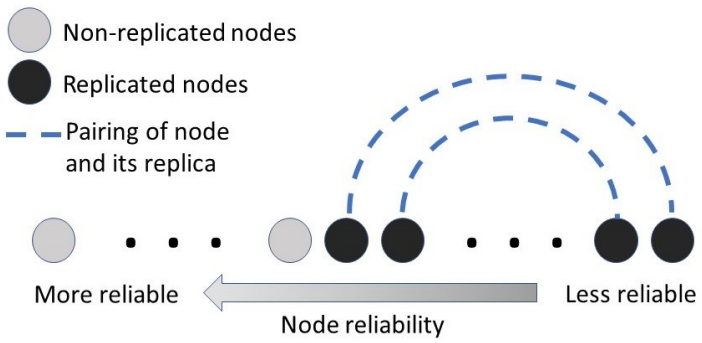



Figure 22: Selection and pairing of replicas to maximize reliability.

*non-replicated nodes and* $B = \{(j,k) \mid j,k \in \{1,2,\ldots N\} - A \text{ and } j \neq k\}$, $|B| = b$, *be the set of node-replica pairs. Maximum overall system reliability is achieved when* $A = \{1,2,\ldots,a\}$ *and* $B = \{(j, 2(a+b)+1-j) \mid j \in \{a+1, a+2, \ldots a+b\}\}$.

To determine the overall reliability for a given partial replication configuration, we observe that, for a node-replica pair $(j,k)$, application failure occurs when both nodes in the pair fail. Hence, the reliability of pair $(j,k)$ is given by $1-(1-g_j(t))(1-g_k(t))$. For sets $A$ and $B$ as defined above, the overall system reliability $R(t)$ can thus be written as

$$R(t) = \prod_{i \in A} g_i(t) \prod_{(j,k) \in B} (1 - (1 - g_j(t))(1 - g_k(t))) \tag{4.1}$$

For simplicity, we remove variable $t$ and obtain

$$R = \prod_{i \in A} g_i \prod_{(j,k) \in B} (1 - (1 - g_j)(1 - g_k)) \tag{4.2}$$

I prove the above theorem in two lemmas. First I will prove that maximum reliability is achieved when the set of non-replicated nodes consists of the most reliable nodes.

**Lemma 3.** *$R$ is maximized when* $A = \{1,2,\ldots,a\}$.

*Proof.* Assume by contradiction that we have a configuration in which $A \neq \{1, 2, \ldots, a\}$. This means there is a node with higher reliability in the replicated set and a node with lower reliability that is not replicated. In other words, $\exists g_i$ where $i \in A$ and $i > a$ and $\exists$ a pair $(j, k) \in B$ such that at least one of $j$ or $k$ is in $\{1, 2, \ldots, a\}$. Assume without loss of generality that $j \in \{1, 2, \ldots, a\}$. This means that $j < i$, and we know, from the ordering of node reliability, that $g_j \geq g_i$. The contribution of nodes $i, j, k$ in this configuration to system reliability, $R$, is given by $g_i(1 - (1 - g_j)(1 - g_k)) = g_i(g_j + g_k - g_j g_k)$. We have

$$g_i(g_j + g_k - g_j g_k) = g_i g_j + g_i g_k - g_i g_j g_k \leq g_i g_j + g_j g_k - g_i g_j g_k = g_j(1 - (1 - g_i)(1 - g_k)) \tag{4.3}$$

Since $g_i(1-(1-g_j)(1-g_k)) \leq g_j(1-(1-g_i)(1-g_k))$ with equality iff $g_j = g_i$, we observe that if we exchange nodes $i$ and $j$ between sets $A$ and $B$, while keeping everything else the same, we obtain a system with reliability $R'$ such that $R' \geq R$. We can keep performing these exchanges as long as $A \neq \{1, 2, \ldots, a\}$. Each exchange step will either improve the system reliability, $R$, or keep it the same. Hence, $R$ will be maximized when $A = \{1, 2, \ldots, a\}$. $\square$

We now move to the second part of the theorem regarding the pairing of replicas. Rewriting $R = R_A R_B$ where $R_A = \prod_{i \in A} g_i$ and $R_B = \prod_{(j,k) \in B}(1 - (1 - g_j)(1 - g_k))$, we focus solely on $R_B$ since $R_A$ is determined from lemma 3. Our job, then, is to show that, given $2b$ numbers $g_1 \geq g_2 \geq \cdots \geq g_{2b}$, $R_B$ is maximized when $B = \{(j, 2b + 1 - j) \mid j \in \{1, 2, \ldots b\}\}$. To simplify the expressions, we will rewrite $R_B$ in terms of the node failure probabilities, $p_i = 1 - g_i$, $1 \leq i \leq 2b$ as $R_B = \prod_{(j,k) \in B}(1 - p_j p_k)$. The ordering of the failure probabilities then becomes $p_1 \leq p_2 \leq \cdots \leq p_{2b}$.

**Lemma 4.** *$R_B$ is maximum when $B = \{(j, 2b + 1 - j) \mid j \in \{1, 2, \ldots, b\}\}$.*

*Proof.* I prove this through induction on $b$. When $b = 1$, there are only 2 nodes, and only one possible pairing, so $B = \{(1, 2)\}$ trivially.

For the inductive hypothesis, assume that the lemma is true for $b = k$. For $b = k + 1$, we first prove that, for $R_B$ to be maximum, $(1, 2k + 2) \in B$. Assume by contradiction that $(1, 2k+2) \notin B$. This means that $\exists (1, i), (j, 2k+2) \in B$ where $i, j \in \{2, \ldots, 2b-1 = 2k+1\}$. Similar to lemma 3, we will show that swapping the nodes in the two pairs to get $B'$,

where $(1, 2k+2), (i, j) \in B'$, will improve system reliability. The contribution of pairs $(1, i), (j, 2k+2)$ to $R_B$ is given by $(1 - p_1 p_i)(1 - p_j p_{2k+2})$. We have

$$\begin{aligned}(1 - p_1 p_i)(1 - p_j p_{2k+2}) &= 1 - p_1 p_i - p_j p_{2k+2} + p_1 p_i p_j p_{2k+2} \\ &\leq 1 - p_1 p_{2k+2} - p_i p_j + p_1 p_i p_j p_{2k+2} = (1 - p_1 p_{2k+2})(1 - p_i p_j)\end{aligned} \tag{4.4}$$

The inequality is obtained by noting that $p_1 \leq p_j$ and $p_i \leq p_{2k+2}$. By rearrangement inequality[73], we know that $p_1 p_i + p_j p_{2b} \geq p_1 p_{2k+2} + p_i p_j$ which leads to the inequality obtained above. This means that for any $B$ such that $(1, i), (j, 2k+2) \in B$, we can get $R_{B'} \geq R_B$ where $B' = (B - \{(1, i), (j, 2k+2)\}) \cup \{(1, 2b), (i, j)\}$. Using the same argument as in lemma 3, we conclude that $R_B$ is maximum when $(1, 2k+2) \in B$. We can thus write the maximum $R_B$ as $R_B = (1 - p_1 p_{2k+2}) R'_B$ where $R'_B$ is the combined reliability of all node-replica pairs other than $(1, 2k+2)$. $R'_B$ can also be considered as the reliability of $2k$ nodes making $k$ pairs which, according to the inductive assumption, is maximum when the $2k$ nodes are paired as stated in the lemma. The overall reliability, $R_B$, is therefore maximized when $B = \{(j, 2(k+1) + 1 - j) \mid j \in \{1, 2, \ldots k+1\}\}$ which concludes the proof. □

Lemma 3 and lemma 4 combined complete the proof of the theorem.

At this point, one may also wonder if a similar result can be obtained for replication degrees greater than 2, for example if triple replication is also allowed. In that case, the only result I could obtain is the following

**Lemma 5.** *If $B$ contains replica groups with degrees $\geq 2$, i.e. $x \in B \rightarrow |x| \geq 2$, $R$ is still maximized when $A = \{1, 2, \ldots, a\}$.*

*Proof.* The proof proceeds by contradiction in the same way as lemma 3 by taking a tuple in $B$ which has an element $i$ where $i \leq a$, and similarly a $j \in A$ where $j > a$. It can then be shown that swapping $i$ and $j$ between the two sets causes $R$ to increase. We omit the detailed steps since they are identical to that of lemma 3. □

The same result, however, does not extend to the case of deciding, for example, which nodes should be doubly replicated and which should be triply replicated.

It should be noted that, although the proof in this section is formulated in terms of node reliabilities, the result holds for any time interval in which the relative ordering of the individual nodes' likelihoods of failure is known. This means that if, at different time intervals, the ordering of nodes based on their likelihoods of failure changes, the optimal configuration, while still determined based on the result in this section, will be different during different time intervals. Handling such configuration changes in practical settings may be possible through an adaptive method to switch replicas on the fly, as in [31]. A theoretical analysis to determine when to change the configuration, taking into consideration the cost of reconfiguring the system during execution, is left for future work. In this dissertation, I will only consider cases where the nodes failure densities are exponential, or Weibull with the same shape parameter. In both of these cases, the relative ordering of node reliabilities remains the same throughout the lifetime and is determined from the individual node MTBFs.

## 4.3 Expected Completion Time

In the previous section, we looked at how the nodes should be grouped into replicas when the number of nodes to be replicated is fixed. In other words, the number of application visible nodes $n$ was already decided, and the goal was to intelligently pick nodes to be placed in replicated and non-replicated sets based on their individual reliability. Now we attempt to answer the more general question: Given an $N$ node system with node reliability $g_1(t) \geq g_2(t) \geq \cdots \geq g_N(t)$, how many of the $N$ nodes should be used and how many should be replicated? This question cannot be answered by considering system reliability alone. Although a higher value of $n$ will reduce the work per node due to parallelism, system reliability will go down making failures more likely. On the other hand, higher replication factors are likely to add more runtime overhead to the application, although they lead to a more resilient configuration. These trade offs can only be captured by computing the expected completion time for given number of nodes and replica pairs, and then picking the values of these variables that yield the minimum completion time.

### 4.3.1 Job Model

The first thing to determine, as $n$ becomes variable, is the amount of work that will be distributed over each node and executed in parallel. Similar to the previous chapter, we use Amdahl's law to determine $W_n$, the time required to execute the job on $n$ parallel nodes without failures:

$$W_n = (1 - \alpha)W/n + \alpha W \tag{4.5}$$

where $0 \leq \alpha \leq 1$ represents the sequential part of the job. Since the focus of this dissertation is on HPC applications that usually have a high level of parallelism, most of the analysis we perform and the results we report will be for values of $\alpha$ equal to, or close to, 0.

### 4.3.2 Overhead Model for Partial Replication

As mentioned in the previous chapter, in addition to reducing the nodes over which work is parallelized, replication also induces additional overhead to message passing applications because of the communication required between replicas in order maintain consistency among them. Naturally this overhead increases when the replication degree increases, since more replicas mean more messages being duplicated. An approach to model the overhead versus the degree of replication was proposed in [26] using $\gamma$, the ratio of application time spent in communication under no replication. We use their idea but update the model so that it agrees with the experimental results reported subsequently in [27]. According to this model, for an application executing under partial replication factor $r$, the time, $W_r$, that includes the overhead of partial replication, is given by

$$W_r = W_n + \sqrt{r - 1}\gamma W_n \tag{4.6}$$

This estimate provides a more pessimistic overhead for partial replication compared to the original model of [26]. Moreover, it matches with the experimental result of [27] on real systems since, for $r = 1.5$, the overhead will be $1/\sqrt{2} \approx 71\%$ of the overhead of full replication. We, therefore, use Eq. 4.6 to compute and add the overhead of partial replication.

### 4.3.3 Combining with Checkpointing

Having figured out the failure-free execution time, $W_r$, of a partially replicated application, we now proceed to compute the expected completion time of such an application under failures. For that, however, we first need to determine the checkpointing interval, which in turn depends on the mean time to interrupt (MTTI). The MTTI, $M$, can be computed using the reliability as:

$$M = \int_0^\infty R(t)dt \tag{4.7}$$

where $R(t)$ is the overall reliability for a a given partial replication configuration. Although we mentioned in the previous chapter the closed form formula for MTTI of dual replication as derived in [7], in general, with partial replication, it is not always possible to evaluate the integral in Eq. 4.7 analytically. We, therefore, resort to numerical integration to obtain the MTTI for our results. For the results we compute numerically, we will be using the more accurate higher order approximation by Daly[20] to calculate the checkpointing interval, $\tau$.

In order to determine the expected completion time, we employ the same approach used in the previous chapter for non-exponential distributions, which is based on the works of [53] and [68]. As a brief recap, this involves computing the extra work, which consists of the time spent writing checkpoints and the lost work due to failures. By considering each failure as a renewal process, the average fraction of extra time during an execution can be taken to be the same as the fraction of extra time between two consecutive failures. Let $F(t) = 1 - R(t)$ be the cumulative failure distribution and $f(t) = F'(t)$ be the failure density function. The extra time between consecutive failures will then be given by

$$E(extra) = \int_0^\infty \frac{Ct}{\tau} f(t)dt + k\tau = \frac{C * M}{\tau} + k\tau \tag{4.8}$$

where $k$ is the average fraction of work lost during a checkpointing interval due to failure and can be evaluated numerically[53].

Once we obtain $E(extra)$, the expected time to finish work $W_r$ will be given by

$$E(W_r) = W_r \frac{M}{M - E(extra)} \tag{4.9}$$

where $W_r$ is determined from Eq. 4.6. This is the equation that we use to compute and compare the expected completion time of a system under different partial replication factors.

### 4.3.4 Optimization Problem

The purpose of computing the expected completion time was to find the replication factor $r$ that minimizes it for a given system. We thus formulate the search for $r$ as an optimization problem as follows:

$$\begin{aligned} \underset{a,b}{\text{minimize}} \quad & E(W_r) \\ \text{subject to} \quad & a + 2b \leq N \\ & n = a + b \geq 1 \end{aligned}$$

where $r = (a+2b)/(a+b)$ and $a$ and $b$ can only take nonnegative integer values. The inputs include work $W$, total number of system nodes $N$, individual node reliability functions $1 > g_1(t) \geq \cdots \geq g_N(t) > 0$, checkpointing cost $C$, the parameter $\alpha$, and communication ratio, $\gamma$. In the next section, we will discuss our findings and results about the optimal $r$ for different kinds of systems. In our results, we report the expected completion time normalized by $W_N$, which is calculated from Eq. 4.5, and represents the time it takes to execute the job on all $N$ nodes without replication or checkpointing and under no failures.

## 4.4 Results

This section discusses the results from the analysis and the optimization problem on specific types of large scale systems, starting with a system where all nodes have the same failure rate and then moving on to systems with multiple classes of nodes, where nodes in the same class have the same failure rate, different from the other classes. We only study exponential and Weibull node failure distributions. Our goal is to explore whether there are cases in which partial replication, where $r$ is strictly between 1 and 2, results in the lowest expected completion time.

### 4.4.1 System with IID Nodes

I start off with the simplest possible scenario, a system where the individual node failure distributions are identical. This has been the traditional assumption when analyzing fault

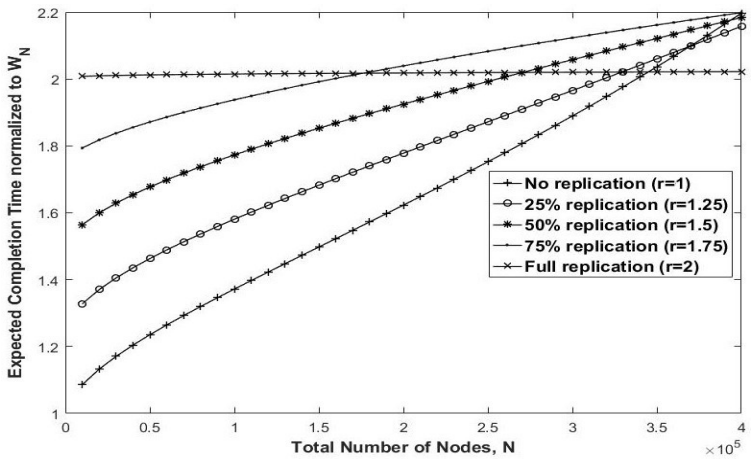



Figure 23: Normalized Expected Completion Time for different values of $r$. Node MTBF = 5 years. Checkpointing cost is taken to be 60 seconds. $\alpha = 0$ and also $\gamma = 0$.

tolerance techniques in HPC systems. Our goal is to explore whether there are cases in which partial replication, where $r$ is strictly between 1 and 2, results in the lowest expected completion time. It should be noted that, since all nodes are identical, it does not matter which individual nodes are picked for replication or how they are paired together.

**4.4.1.1 Exponential Distribution** I first consider a system where node failure probabilities are exponentially distributed. When taking both $\gamma$ and $\alpha$ as 0, our optimization never yielded an optimal value of $r$ strictly between 1 and 2 for any scenario we tested. This can also be seen in figure 23 where the expected completion time according to Eq. 4.9 (normalized by the time it takes to run the same job on $N$ nodes without any fault tolerance and without failures) with different partial replication degrees is plotted against the total number of nodes in the system. We see that the minimum time is always attained either when $r = 1$ or $r = 2$. The trend was the same for other node MTBF values, with the crossover between full and no replication occurring at higher node counts as the node MTBF increases.

I further investigate this scenario analytically with the goal of determining if $1 < r < 2$ is ever optimal for uniformly exponential node distributions when $\gamma$ and $\alpha$ are both 0. Assuming that the configuration uses all of the system nodes $N$, so that $a + 2b = N$, and

individual node failure rate is $\lambda$, we can write the MTTI, $M$, as:

$$M = \int_0^\infty e^{-a\lambda t}(1-(1-e^{-\lambda t})^2)^b dt = 2^N \int_0^\infty (e^{-\lambda t}/2)^{N-b}(1-e^{-\lambda t}/2)^b dt \tag{4.10}$$

Since obtaining a closed form expression for the above integral is not possible, we try to provide a closed form approximation for $M$. Setting $x = e^{-\lambda t}/2 \Leftrightarrow t = -ln(2x)/\lambda$ in the above expression, we get

$$M = \frac{2^N}{\lambda} \int_0^{1/2} x^{(N-b-1)}(1-x)^b dx \tag{4.11}$$

I employ Laplace's method of approximating integrals[74] to derive an approximation of the above expression. We can rewrite the function inside the integral as $x^{(N-b-1)}(1-x)^b = e^{(N-b-1)f(x)}$ where $f(x) = ln(x) + bln(1-x)/(k-b-1)$. Assuming $2b < N$, within the interval of integration $f(x)$ is maximum at $x = 1/2$ which is the endpoint of the integration, so the integral can be approximated as

$$\int_0^{1/2} x^{(N-b-1)}(1-x)^b dx \approx \frac{e^{(N-b-1)f(1/2)}}{(N-b-1)f'(1/2)} = \frac{(1/2)^N}{N-2b-1} \tag{4.12}$$

Plugging this into the expression for MTTI above we obtain $M \approx 1/\lambda(N-2b-1)$. This reasonably approximates the MTTI as long as $2b$ is not close to $N$, which corresponds to the full replication case. To the best of our knowledge, this is the first closed form approximation of the MTTI of a partially replicated system with exponential node failure distributions with rate $\lambda$.

Having obtained a closed form approximation for $M$ in terms of $N$ and $b$, we will infer the behavior of the expected completion time. Using Young's[76] expression for the expected completion time we get

$$E(W) = W(1 + \frac{C}{\tau} + \frac{(\tau + C)^2}{2\tau M}) \tag{4.13}$$

where we take $\tau = \sqrt{2CM}$ which is also Young's approximation for the optimum checkpoint interval. Assuming that a perfectly parallel job takes unit time on $N$ nodes without checkpoints and failures, the work per node will be $r$ units when the system is partially replicated, since $r = (a+2b)/(a+b) = N/n$. This means that $W_r = r$, and $E(W_r)$ then is given by

$$E(W_r) = r(1 + \sqrt{\frac{2C}{M}} + \frac{C}{M} + \frac{C^2}{2M\sqrt{2CM}}) \tag{4.14}$$

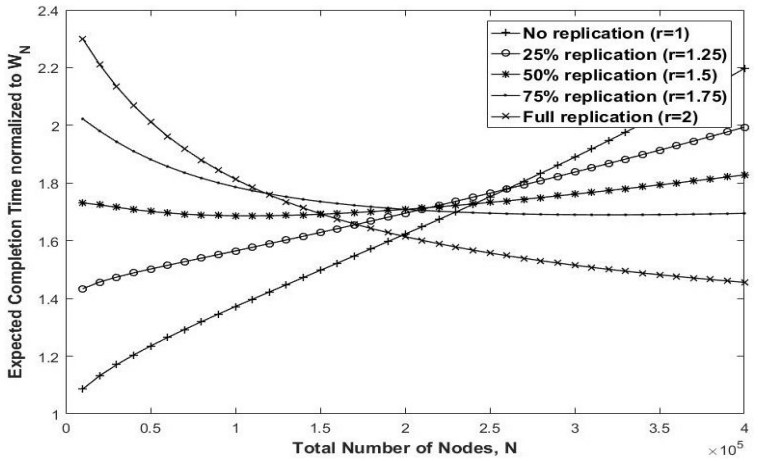



Figure 24: Expected Completion Time for different values of $r$ for exponential node distribution. Node MTBF = 5 years, $\alpha = 0.00001$, Checkpointing cost = 60 seconds, $\gamma = 0.2$

Since both $r$ and $M$ can be defined in terms of $b$ and $N$, and since $N$ is fixed, $E(W_r)$ is a function of $b$. By taking the first and second derivative of this expression *wrt* $b$, we find that this expression has no local minimum over the range $0 \leq b < N/2$ as long as $M > C$. For conciseness, we omit the calculations. This means, though, that the minimum of $E(W_r)$ occurs only at one of the endpoints of $r$ which correspond to either no replication or full replication. While Equation 4.14 is an approximation for the expected completion time, this analysis supports our numerical results that partial replication never yields optimal performance for jobs with $\alpha = \gamma = 0$ and when individual node failures are iid with exponential distributions. This is also consistent with the findings of [67] where it was observed that in cases where replication is better than no replication, full replication offers the best performance.

When $\gamma > 0$, it may theoretically be possible to have cases where the optimal $r$ is strictly between 1 and 2. This is because there can be cases in which the expected completion time with $\gamma = 0$ (i.e. no additional overhead of full or partial replication) is minimized when $r = 2$, but that minimum may shift to $r < 2$ if $\gamma > 0$. That being said, I did not observe this for any values of parameters that I tried. As for when $\alpha > 0$, although it may be possible for $1 < r < 2$ to be optimal, I again did not observe any such case. Figure 24

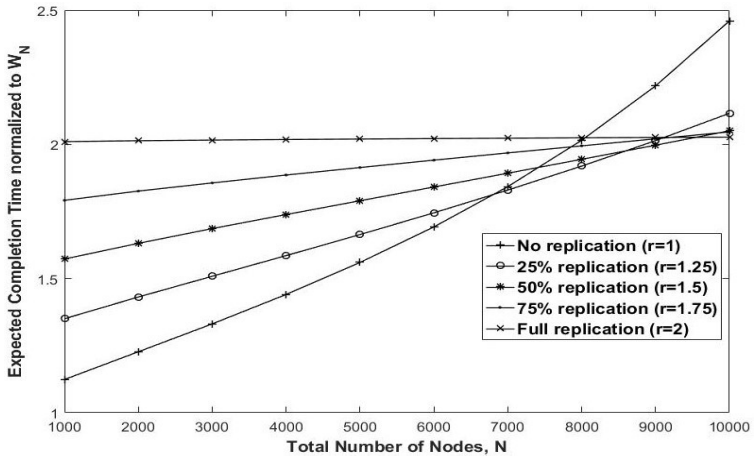



Figure 25: Expected Completion Time for different values of $r$ with Weibull node failures. For the distribution, shape parameter = 0.7 and MTBF = 5 years. Checkpointing cost = 60 seconds and $\alpha = \gamma = 0$.

shows one example with both $\alpha > 0$ and $\gamma > 0$. We observe that, although the crossover between full and no replication happens earlier compared to Fig. 23, partial replication again does not win against the two extremes. Hence, the conclusion from this subsection is that partial replication is almost always never optimal for systems with iid exponential node distributions.

**4.4.1.2 Weibull Distribution** The Weibull failure distribution consists of a shape parameter as well as the rate parameter $\lambda$. In practice, values of the shape parameter between 0 and 1 are used for real world failures. In this chapter, I show results with the parameter value of 0.7. I also considered several examples with a value of 0.5 and observed similar trends as those with shape parameter set 0.7.

Fig. 25 shows one example with Weibull node failures where the completion times are plotted against system scale. When comparing with Fig. 23 (which used similar parameter values but with exponential failures), several differences stand out. Firstly, notice that the crossover between full and no replication happens at much smaller scales. Since Weibull distribution is generally considered to be a closer fit to actual system failure distributions,

this would mean that, in practice, as we move to larger scales, the need for some type of replication would arise earlier than the estimates generated using exponential distributions. Another difference from exponential failures is that there are node counts where partial replication (for example, $r = 1.25$ and $r = 1.5$ in Figure 25) has the lowest completion times. The range of node counts for which this happens is still quite small, however.

I observed similar behavior as in Figure 25 with non-zero values of the parameters $\alpha$ and $\gamma$, except that the crossover point is shifted. The effect of increasing $\gamma$ is to shift the crossover points to the right. For example, with $\gamma = 0.2$, I found that the crossover between no replication and $r = 1.25$ happens around 9000 nodes instead of 7000 nodes. Increasing $\alpha$, on the other hand, brings the crossover points to the left towards smaller node counts. For example, $\alpha = 10^{-5}$ caused the crossover between $r = 1$ and $r = 1.25$ to happen at 6500 nodes instead of 7000 nodes. Moreover, just like in Figure 25, when $\alpha$ and/or $\gamma$ are non-zero, there is only a very small range of node counts for which partial replication provides the lowest completion time.

The main takeaway from this section is that when the nodes in the system have identical failure distributions, which has been the traditional assumption in fault tolerance research for HPC, partial replication rarely provides any gains in performance against full and no replication. Depending on the number of nodes in the system, the choice should then only be between running an application under full replication or running it with no replication at all.

### 4.4.2 System with Two Types of Nodes

I now move one step further by considering a system where nodes are of two kinds: i) Good, which have a low probability of failure, and ii) Bad, which have a higher probability of failure. We assume that all the Good nodes have the same failure distribution and all the Bad nodes have the same failure distribution. This can be a scenario in a system where individual system nodes can be approximately divided into two categories: those which are more prone to failures and those which are less prone to failures.

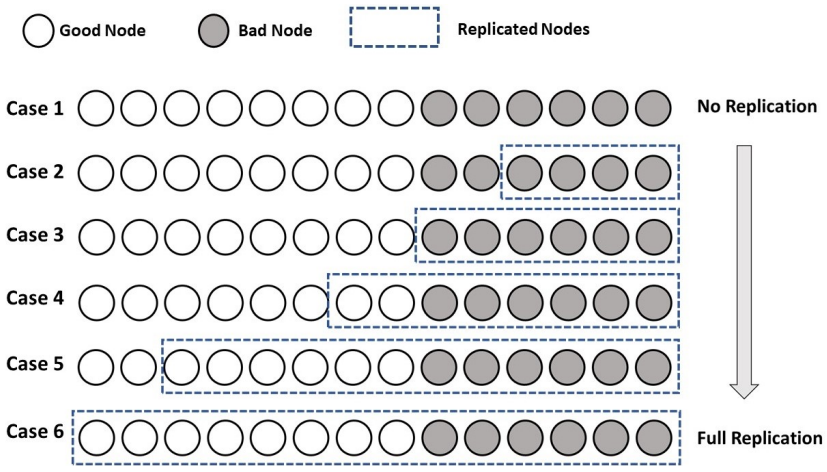



Figure 26: Possible cases of partial replication for system with Good and Bad nodes. Nodes within the replicated set are paired according to the arrangement depicted in Figure 22.

Let $N_G$ be the number of Good nodes and $N_B$ be the number of bad nodes, such that $N_G + N_B = N$. Thanks to the main result of section 4.2, we know that if partial replication is to be employed, we should start replicating from the lower end. Moreover, within the nodes to be replicated, pairing should be done as indicated by Figure 22. Using this knowledge, we can enumerate all possible cases for different partial replication degrees of a Good-Bad node system. This enumeration is depicted in Fig. 26. Starting from the no replication case, increasing replication degree would mean initially replicating the Bad nodes among themselves. Case 3 is the boundary of case 2, when all of the Bad nodes have been replicated. As the replication degree is further increased, some of the Good nodes enter the replicated set as well. Case 4 thus contains two kinds of replica pairs: a Good node paired with a Bad node, and a Bad node paired with a Bad node. Case 5 is again a boundary of case 4 where all replica pairs consist of a Good and a Bad node each. The full replication case contains additional node pairs depending on the difference between the number of Good and Bad nodes. I will explore below how the average completion times of these different cases fare against each other in different settings.

**4.4.2.1 Exponential Distribution** Assuming all the nodes in the system have exponential failure distribution, we can take the failure rate of Good nodes as $\lambda_g$ and the failure rate of the Bad nodes as $\lambda_b$, where $\lambda_g \leq \lambda_b$. Since case 2 in Fig. 26 is quite similar to the partially replicated iid system in section 4.4.1, we first attempt to approximate its MTTI. For this case, we can write the reliability of the system as

$$R(t) = e^{-N_G\lambda_g t}e^{-(N_B-2b)\lambda_b t}(2e^{-\lambda_b t} - e^{-2\lambda_b t})^b \tag{4.15}$$

where $2b$ is the number of Bad nodes that are replicated. To obtain the MTTI of such a system, we can follow the same approach as in section 4.4.1 to approximate the integral of $R(t)$. This yields the following approximation for the MTTI, $M$, of the system in case 2

$$M \approx \frac{1}{N_G\lambda_g + (N_B - 2b - 1)\lambda_b} \tag{4.16}$$

This expression again reasonably approximates the MTTI as long as $2b$ is not close to $N_B$.

Similar to section 4.4.1, I use Equation 4.16 to understand the behavior of the expected completion time of the application *wrt* $b$ when $\alpha = \gamma = 0$. We plug $M$ into eq. 4.14 along with $r$ for this case, which is equal to $(N_G + N_B)/(N_G + N_B - b)$. Taking the first and second derivatives of the resulting expression *wrt* $b$, we again conclude that the function has no local minima and thus the minimum occurs only at the extremes, i.e. $b = 0$ (no replication), or $b = N_B/2$ (all Bad nodes replicated among themselves). This indicates that, between cases 1, 2 and 3, the minimum expected time can only be achieved by cases 1 and 3 for exponential node failures with $\alpha = \gamma = 0$. We again mention that while this derivation holds only for the approximations of $M$ and expected completion time, our numerical search also never yielded any scenarios in which case 2 resulted in lower average time than both cases 1 and 3.

While we were unable to obtain an approximation of MTTI for case 4, our numerical search indicates that the minimum average completion time occurs again at the boundary cases, i.e. 3 or 5. This means that, in general, we need only consider the boundaries of partial replication in a Good-Bad node system. As an example, Figure 27 shows the expected completion time of full and no replication along with cases 3 and 5 from Figure 26. From the plot on the left in Figure 27, we see that replicating the Bad nodes among themselves (Case 3) yields the lowest completion time. Case 5, which replicates each Bad

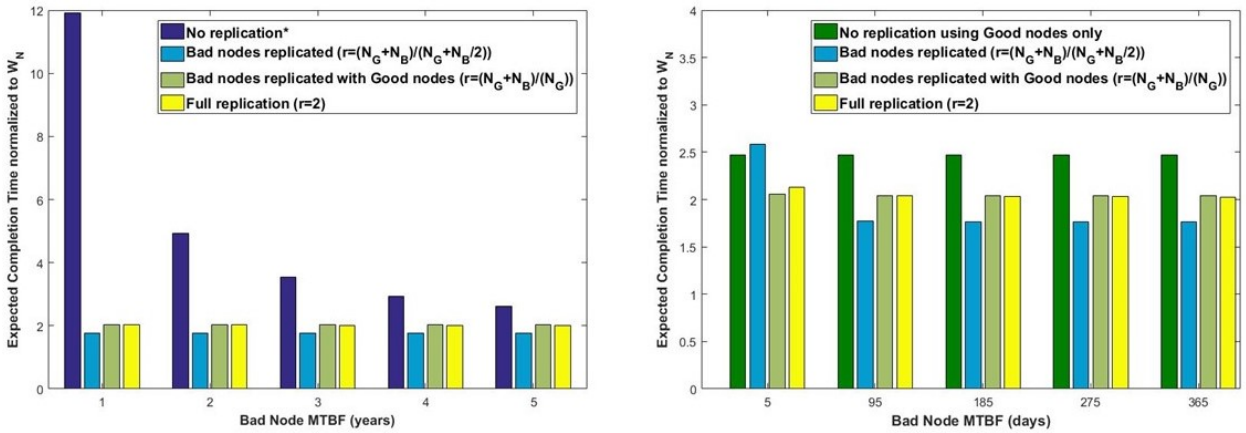



Figure 27: Execution time of different partially replicated executions. $N_G = 10^6$, $N_B = 8 \times 10^5$, $\lambda_g = 1/50$ years, $C = 60$ seconds and $\alpha = \gamma = 0$. Y-axis scale is different for each of the two figures.

node with a Good node, offered almost the same performance as full replication. While we do not show the results with higher Bad node MTBF, we saw that no-replication started outperforming Case 3 when Bad node MTBF went above 20 years, with the same parameters as in Figure 27.

In order to find out if there can be a scenario where Bad node MTBF is so low that not using the Bad nodes, replicated or not, at all is the best performing scheme, we reduced the Bad node MTBF to the order of days and also compare with a no replicated configuration using the Good nodes only ($a = N_G$, $b = 0$). The plot on the right in figure 27 depicts the results. We see that only in the unrealistic case of individual Bad node MTBF dropping to the order of a few day does using Good nodes only outperform Case 3. We deduce from this that, as long as Bad node MTBF is larger than a few days, utilizing the Bad nodes results in lower completion time on average instead of not using them at all. Whenever Bad node MTBF is so low that using them without replication hurts application runtime, the lowest expected time can be achieved by replicating the Bad nodes among themselves and still utilizing them along with the non-replicated Good nodes.

Figure 28 shows the behavior of the schemes with varying percentage of Bad nodes in

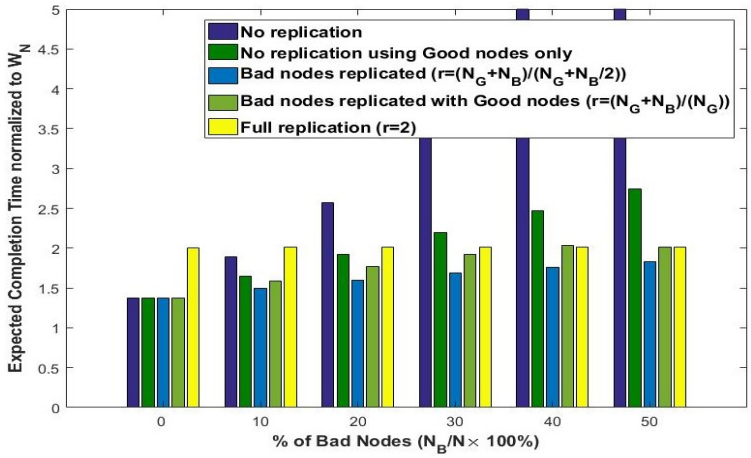



Figure 28: Expected time vs % of Bad Nodes in the system. $N = 2 \times 10^6$. Bad Node MTBF = 5 years. Other parameters are the same as in Fig 27.

the system, while the total number of nodes, $N$, is kept constant. When all nodes are Good, no replication is the best choice. However, as further nodes are added, no replication has a much higher normalized time. The normalized time for the no replication scheme which uses the Good nodes only also increases as % of $N_B$ in the system increases. This is because the time is normalized by $W_N$ which is the failure free time of running the job on all $N$ system nodes. In all cases, however, we see that Case 3 offers the best expected completion time.

Figure 29 shows the behavior of the different partial replication schemes for different values of $\gamma$. The time for all the partial replication schemes increases with increasing $\gamma$. However, since Case 3 has smaller replication factor than Cases 5 and 6, the impact of $\gamma$ is much smaller. Only when $\gamma \geq 0.8$ does partial replication of Case 3 start losing to no replication using Good nodes only. Hence, we can say that for most practical values of $\gamma$, using the Bad nodes with full replication amongst themselves is still better than not using them at all. Although we do not present similar plots for the parameter $\alpha$, the impact of increasing $\alpha$ is to favor more the cases with higher replication factor, $r$. Hence, as $\alpha$ increases, the lowest completion time shifts from case 3 towards full replication ($r = 2$).

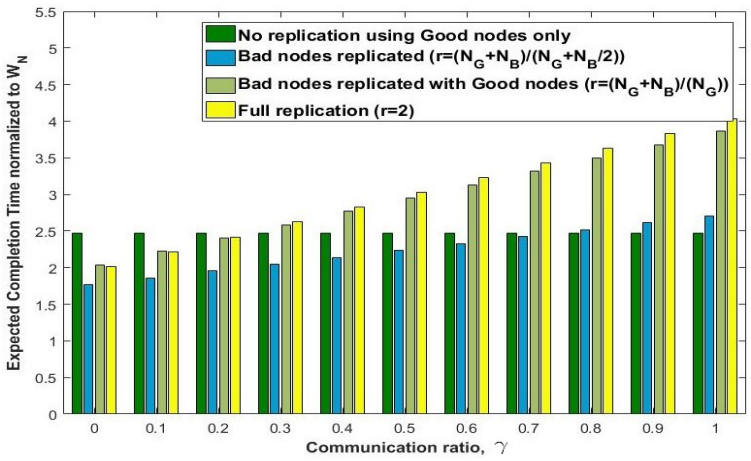



Figure 29: Expected time for different values of $\gamma$ when Bad Node MTBF = 5 years. Other parameters are the same as in Figure 27. The expected time for no replication using all system nodes is much higher than all other schemes so it is omitted from the plot.

**4.4.2.2 Weibull Distribution** For node failures given by the Weibull distribution, we assume that all nodes' distribution have the same shape parameter. Only the rate parameter, $\lambda$, is different for the Good and Bad nodes. With this assumption, and again taking $\lambda_g \leq \lambda_b$, the Good node will always be more reliable than the Bad node throughout its lifetime. Hence, this assumption allows us to apply the theorem of section 4.2 when deciding the pairing of nodes and so the possible partial replication schemes will still be given by Fig. 26.

Figure 30 shows the normalized runtimes of different partial replication cases similar to the exponential distribution subsection, but over a larger range of Bad node MTBFs. We again see that with lower Bad node MTBF, replicating Bad nodes among themselves yields the lowest expected completion time. Moreover, this happens at system scales much smaller than the ones for exponential distribution. We omit the plots for the cases when $\gamma > 0$. The trends, however, were the same as the ones observed for exponential distribution.

Based on the results from both exponential and Weibull distributions, we conclude this section with the following insight: If an HPC system has some nodes that are more likely to fail than others, those nodes can still be utilized to achieve performance gains. When the likelihood of failures in such Bad nodes is not too high, those nodes can simply be used

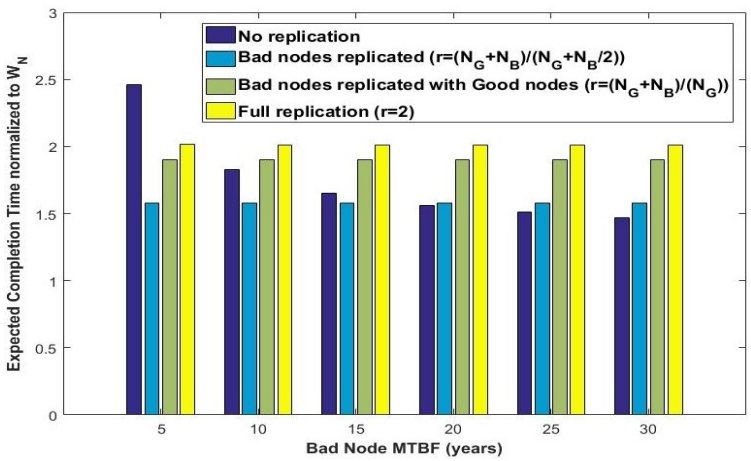



Figure 30: Execution time of different replication schemes with Weibull node failures. $N_G = 10^4$, $N_B = 8 \times 10^3$ and Good node MTBF = 50 years. The other parameters are the same as in Fig. 27.

alongside the rest of the system nodes to execute a job in parallel, without replication. If, however, the likelihood of failures in those nodes increases, they can be replicated among themselves and still be used along with the other system nodes to provide better performance compared to the case of not using such nodes at all.

### 4.5 Systems beyond two categories of nodes

The optimization problem formulated in section 4.3 is capable of finding the optimal $r$ for a system with any set of non-uniform node reliability values $g_i(t)$, as long as they maintain the ordering $g_1(t) \geq g_2(t) \geq \cdots \geq g_N(t)$. This is useful when all the individual node reliability functions are known. However, we do not present any results for such a generic system because they don't provide any interesting insights about $r$ or the behavior of the expected completion time. We, therefore, only present one example of a system with 5 categories of nodes. The MTBFs of the five categories range from 1 to 5 years in increments of 1 year, with each category having the same number of nodes. Figure 31 shows

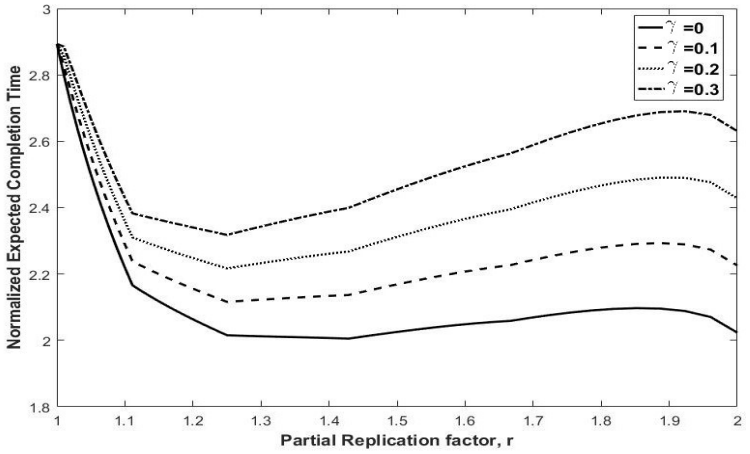



Figure 31: Expected completion time versus $r$ for different values of $\gamma$. The values of other parameters are: $\alpha = 0$, $C = 30$ seconds and each category contains 100k nodes, for a total of 500k system nodes.

the normalized expected completion time, using Eq 4.9, versus the partial replication factor, $r$, for different values of $\gamma$. We kept $a + 2b = N$ instead of the inequality $a + 2b \leq N$. This is because, similar to the conclusions for the Good-Bad node system, we usually found that using all the nodes in the system is beneficial, as long as the lowest MTBF nodes do not have unrealistically low values of the MTBF.

We can make several observations from Fig. 31. For all values of $\gamma$, the optimal $r$ is less than 2. For $\gamma = 0$, optimal value of $r \approx 1.42$, but for other values of $\gamma$, the optimal value of $r = 1.25$. These results highlight the importance of having and utilizing a deeper understanding of the failure characteristics of the underlying system. If, for example, instead of considering the 5 categories of nodes, one were to assume uniform node failure likelihoods and take the average value of the node MTBF over all classes, and then use that to decide the replication degree, the answer would be to fully replicate the execution. However, as we can see in the figure, partially replicating the right nodes can result in lower expected completion time than full replication. In fact, if the decision to fully replicate is made without the knowledge of the different categories of nodes, the replica pairing may not be the same as that described in section 4.2, and may lead to even higher expected completion time.

We make one final remark about the behavior versus $r$. We see in Fig. 31 that the curve is piecewise smooth in segments. The values of $r$ at the boundary points of these segments correspond to the boundary cases of different partial replication configurations. So, for example, if only the nodes in the lowest MTBF category are all replicated among themselves, we get $r = 1.1$. We see in Fig. 31 that for $1 \leq r \leq 1.1$, the curve is smooth. Similarly, the next smooth segment finishes at $r = 1.25$, which is the boundary case achieved when the lowest MTBF category is fully replicated with the next lowest category. Although we do not have any analytical results about this, our investigations of multiple scenarios always yielded the optimal $r$ on one of these boundary cases. This indicates that, in cases where node MTBFs take a small set of discrete values, rather than doing a full search for the best $r$, it may be a reasonable heuristic to only consider boundary cases and pick $r$ with the lowest completion time.

## 4.6 Related Work

Full[29] and partial[26] replication were both proposed for large scale systems when failures become frequent. A deeper analysis of pure replication and its comparison with simple checkpoint/restart was carried out in [11]. For partial replication, [67] provides a limited analysis and comparison with full and no replication. Even though my focus in this dissertation is on systems with non-uniform failure distributions of individual nodes, section 4.4.1 provides a more detailed analysis of partial replication with iid node failures. I provide theoretical results for the MTTI and evidence that partial replication is never optimal on such systems.

All of the above have assumed systems with identical nodes in their analyses. We are only aware of two works that distinguish between different failure likelihoods in the underlying hardware. [6] considers two instances of an application running on two different platforms, that execute at different speeds and are subject to different failure rates. Our work differs from it in several aspects. Firstly, the paper considers *group replication*, where a complete instance of the parallel application is executed redundantly, rather than replicating individual

processes. This avoids communication between instances but a single failure causes the whole instance to fail. Secondly, the framework does not allow for partial replication. Thirdly, their work assumes a single platform failure distribution, without considering the underlying nodes in the system. [61] is closer to our work since it considers individual node failure rates. However, it only performs a post hoc analysis based on failure logs to determine which nodes have the most failures and how many of those failures could be eliminated by duplicating those nodes with spare nodes. Moreover, this work only considers the improvement in MTTI without looking at the impact on completion time. Our work provides a comprehensive theoretical framework which not only determines how the nodes should be duplicated, but also when it pays off to duplicate some nodes in the system.

While our work looks at partial redundancy in the presence of non-identical node failures, there are papers that consider the problem of selectively replicating tasks based on criticality[69][70][71]. These works replicate tasks from an application task dependence graph by measuring the criticality of an individual task. The idea of criticality is orthogonal to the task of selectively replicating *nodes* based on their individual reliability. Our work, additionally, is application agnostic since it only considers the failure distributions of individual nodes.

## 4.7 Summary

In this chapter, I explored partial replication for HPC systems where individual nodes have non-identical failure distributions. I obtained theoretical results on the optimal way to divide the nodes into replicated and non replicated sets and to pair the nodes in the replicated sets. By computing the MTTI and expected completion time of a job executed in a partially replicated configuration, I also investigated the optimal fraction of replication. The takeaway from this chapter is that, while rarely optimal for IID node failure platforms, partial replication can yield the best performance for systems comprising of nodes with heterogeneous failure rates.

## 5.0 Co-located Shadows for Fault Tolerance

This chapter discusses the design and implementation details of a fault tolerance scheme based on co-located shadows, which are essentially replica processes that execute at slower speeds than the original application processes, or *mains*[16][57]. The motivation for co-locating shadows with mains stems from their potential to cut down on the re-execution time in case of failures, for MPI applications that spend a significant percentage of their execution time waiting for communication to complete. The primary source of that idle time is the imbalance among the different processes in a parallel computation, which can be considered another form of heterogeneity within the context of HPC. In principle, therefore, the goal of this chapter is to explore the possibility of utilizing such idle times for fault tolerance.

## 5.1 Nature of HPC Workloads

I provide justification for this approach by first discussing the characteristics of message passing applications and their underlying implementation in the Message Passing Interface (MPI). The MPI standard defines several routines for communication between processes, which encompass both point to point communications (such as `send` and `receive`) and collectives (such as `broadcast` and `reduce`). Each of the communication routines has a blocking and a nonblocking version. Typically, the blocking function calls are internally implemented as a busy-wait until the requested operation completes. The rationale for this is the assumption that there is only one MPI process running on a processor. Since no other process is sharing the processor with that MPI process, it makes sense from a performance point of view to just perform a busy wait rather than relinquish the processor for any amount of time. Moreover, since MPI already provides the nonblocking versions of most communication operations, the programmer can reduce the times spent in these busy-waits by overlapping computation and communication using nonblocking communication.

Although MPI applications can reduce idle times during communication by using nonblocking operations, this still allows only a coarse-grained overlap of computations and communications. The time for a communication operation to finish is variable and depends on a lot of factors independent of the application, such as the state of the interconnect, system buffers, and so on. Hence it is difficult to predict this time beforehand. Even if the application makes a nonblocking communication call, performs some other operations and then calls `MPI_Wait` (the function that tests and waits until the nonblocking communication has finished), there is no guarantee that the communication operation would have finished by this time. Thus, the program may again be forced to do a busy-wait until the requested communication operation completes.

To test how much of the application's execution time is composed of these busy-waits, I performed an experiment where I used OpenMPI's (a popular implementation of the MPI standard) profiling interface PMPI to trap all of the blocking OpenMPI communication calls, both point to point and collectives, made by an application and replaced them by a call to their nonblocking counterpart followed by a loop which continuously tests or probes to see if the operation has completed. For example, a call to `MPI_Reduce` would be replaced by a call to `MPI_Ireduce` (the nonblocking version of `reduce` operation) followed by a loop which repeatedly calls `MPI_Test` on the request handle returned by `MPI_Ireduce` until it sets the flag to true, indicating the operation has finished. Similarly, a call to `MPI_Wait` would also be translated to a loop of `MPI_Test` that exits when the test sets the flag to true. I timed the loops only and not the calls to the nonblocking operations, so that I only capture the idle time spent waiting for the operation to finish and exclude any time spent setting up the buffers or doing other bookkeeping. I performed this experiment on a number of MPI benchmarks that represent HPC applications, the results of which are presented in Table 3 below.

It can be seen from Table 3 that the average idle time for most of the benchmarks is in the 10-20% range, except for PENNANT, which spends most of its time ($\approx 70\%$) waiting for communication to finish. This indicates that there is the potential, especially for some applications, to utilize the processors that are idle during the communication routine for fault tolerance.

Table 3: Average idle time as a percentage of total execution time

| Benchmark | Minimium | Maximum | Average |
|---|---|---|---|
| HPCCG[39] | 7.7 | 17.9 | 13.0 |
| CoMD[40] | 2.4 | 20.8 | 6.9 |
| LULESH[48] | 6.8 | 32.3 | 17.6 |
| PENNANT[28] | 49.8 | 69.9 | 65.1 |
| miniFE[38] | 7.6 | 15.3 | 10.1 |

## 5.2 Co-Located Shadows Model

Since the primary purpose of this scheme is to utilize the idle processor time for fault tolerance, at its heart, the design consists of shadow processes co-located with main processes. However, in order to provide effective fault tolerance for message passing applications, this setup needs additional ingredients. In this section, I will describe those ingredients needed to make the co-located shadows capable of providing fault tolerance, as well as a theoretical analysis of the basic setup. Specific implementation details, as well as a description of how the shadows are used to capitalize on the idle times, are presented in the next two sections.

### 5.2.1 Basic Setup

This model consists of one shadow process for each main process, resulting in twice the number of original application processes. Each shadow is placed on the same core as one other main. Depending on whether failures can strike individual cores or individual nodes, which comprise multiple cores, the core/node on which the shadow is located should be different from the main it is responsible for rescuing. An example of this is shown in Figure 32. This will allow the shadow to be used for recovery when its main on another processor fails. Conceptually, therefore, this approach can recover even if a failure strikes multiple processes, as long as no main and its shadow simultaneously fail. The main purpose of the

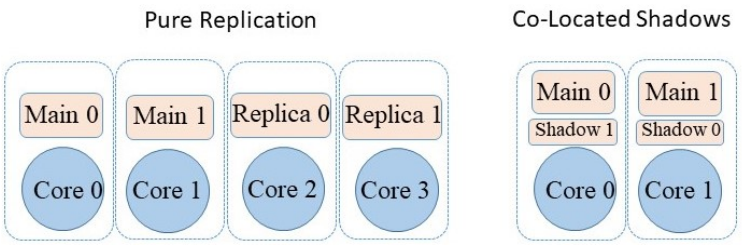



Figure 32: Difference between Pure replication, which requires twice the original number of processors, and co-located shadows, which do not require extra processors.

shadow is to do the work of its main if the main fails. The shadow therefore executes the same code as the main, and can be thought of as a lazy replica.

### 5.2.2 Failure Free Execution

During normal execution, the shadows do not execute at the same speed as the mains. In fact, they should ideally execute only when the co-located main is idle waiting for a communication to finish. Since the amount of time that each shadow may get to execute is variable, we need a mechanism that allows each shadow to make progress independently. This necessitates the need for message logging. I employ receiver based message logging in which each main forwards the messages it receives to its shadow. These messages are buffered at the shadow and consumed by the shadow at its own pace. The forwarding of messages and their reception and buffering at the shadows may add an overhead to the failure free execution of the application. I model this overhead by the parameter $\beta \geq 0$ which represents the ratio between the extra time taken and the original time without overhead.

### 5.2.3 Recovery from Failures

Due to the tightly coupled nature of message passing applications, when a main fails, all other mains are also blocked and so suspend their execution until recovery. The shadow of the failed main is used for recovery by executing it till the point of failure. The shadow,

while executing, suppresses all send message operations, and consumes all the messages to be received from its local buffer. While the recovering shadow is executing, either the resource on which the failed main resided is rebooted or a new spare is brought in its place. In either case, the fresh resource hosts a new process which can take over as the main. Here we see one advantage of this scheme: recovery can happen in parallel with the reboot, since they both involve different resources.

In order to bring the new process to the same stage in execution as before the failure, we use the idea of leaping[18] to transfer the shadow's state to the new process, which replaces the failed main. The idea is similar to application level checkpointing, and involves a state transfer between a process at an earlier stage in the computation and a process that is performing the same computation but is at a more advanced stage in execution. Section 5.3.4 provides details on how leaping is implemented. The result of this mechanism is that total recovery time will be given by the sum of: i) maximum of restart and reexecution time, and ii) the time it takes to *leap* the restarted main to the state of its shadow. We can, therefore, model it as

$$T_{rec}(w_f) = max(R, w_f) + L_c \tag{5.1}$$

where $R$ is the restart time, $w_f$ is the time it takes the shadow to execute the lost work, and $L_c$ is the time it takes to leap the new process to the shadow's state. Both $R$ and $L_C$ are assumed to be constants.

All the surviving mains whose executions are suspended during recovery can also be used to leap their shadows to their current state while recovery is happening. Therefore, once the recovery is complete, all the shadows are at the same stage as their mains, giving the effect of *Checkpoint on Failure*[9]. This gives us the advantage that the recovery time for a subsequent failure will be bounded by the time between current and the previous failure. Moreover, this also results in a clearing of the message logs accumulated up until that point.

### 5.2.4 Periodic Leaping

Similar to periodic checkpointing, I also perform a periodic updating of all the shadows by their mains. This is accomplished by using the idea of leaping[14], where the current state

of the main, captured by the values of its local variables, is transferred to its shadow. This has the same effect as checkpointing, since the earliest point a shadow will start to execute from, if a failure happens, will be the point of the latest periodic leap. For the analysis in the next subsection, I will assume the interval for the periodic leap to be the same as the optimum checkpointing interval.

### 5.2.5 Analysis

We now have the basic ingredients to model the performance of co-located shadows (without utilizing the idle time) and contrast it with coordinated C/R. I derive the expected completion time of co-located shadows within a single periodic leaping interval. The total expected completion time can then be obtained by multiplying the expected time for one interval by the number of intervals, similar to checkpointing. I assume individual failure probabilities to be independent and identically distributed (iid) and driven by an exponential distribution with rate $\lambda$. Depending on what the unit of failure is, for a system with $n$ such independent failure units, the system failure rate becomes $n\lambda$.

Let $W_i$ be the time to complete the work within a leaping interval without failures and without the overheads involved with co-located shadows. Thus, if no failure strikes within the interval, the completion time will be $W_i(1+\beta)$, $\beta$ being the failure-free overhead of co-locating shadows. If, however, a failure strikes at time $t < W_i(1+\beta)$, recovery time given by $T_{rec}(t/(1+\beta))$ will be added to the completion time. Note that I add $T_{rec}(t/(1+\beta))$ instead of $T_{rec}(t)$ since the shadow simply consumes messages from its buffer and so none of the overheads related to message logging and processor sharing are present during recovery. The time to finish work $W_i$ can then be written as a recursive equation:

$$T(W_i) = \begin{cases} W_i(1+\beta) & t_f \geq W_i(1+\beta) \\ t_f + T_{rec}(\frac{t_f}{1+\beta}) + T(W_i - \frac{t_f}{1+\beta}) & t_f < W_i(1+\beta) \end{cases} \tag{5.2}$$

where $t_f$ represents the time at which the failure strikes. Note that the remaining time after recovery is given by $T(W_i - t_f/(1+\beta)$ since all the other shadows not participating in recovery are leaped to the state of their mains, and so a subsequent failure will not require the shadow to execute the work done before the previous failure. Note also that, since only

one shadow is involved in the recovery, the probability of failure of that shadow can safely be ignored in comparison with the overall system failure probability. This assumption cannot be made for coordinated C/R, since all processes are rolled back and start reexecuting from the last checkpoint. The expected value of $T(W_i)$ can then be computed as

$$\begin{aligned} E(T(W_i)) &= W_i(1+\beta)e^{-n\lambda W_i(1+\beta)} + \\ &\int_0^{W_i(1+\beta)} (t_f + T_{rec}(\frac{t_f}{1+\beta}) + E(T(W_i - \frac{t_f}{1+\beta})))n\lambda e^{-n\lambda t_f} dt_f \end{aligned} \tag{5.3}$$

Since $T_{rec}$ as defined in Eq 5.1 consists of a term that is the maximum of two values, I compute an upper bound for Eq 5.3 as follows: I break the integral interval over $T_{rec}$ into two intervals, one from 0 to $R(1+\beta)$, in which $T_{rec} = R + L_c$, and the other from $R(1+\beta)$ to $W_i(1+\beta)$, in which $T_{rec} = t_f/(1+\beta) + L_c$. This is an upper bound since it overestimates $E(T(W_i))$ when $R > W_i$. Solving, for $E(T(W_i))$ using this approximation yields:

$$\begin{aligned} E(T(W_i)) = W_i(1+\beta) + \frac{e^{-\lambda W_i(1+\beta)} + e^{-\lambda R(1+\beta)} - 2}{\lambda(1+\beta)} \\ + R + (L_c + R + \frac{e^{-\lambda R(1+\beta)}}{\lambda(1+\beta)})\lambda W_i \end{aligned} \tag{5.4}$$

Finally, if $W$ is the total work to be done, the number of periodic intervals will be given by $W/W_i$. Hence, the expected time to finish work $W$ can be written as

$$E(T(W)) = \frac{W}{W_i}(E(T(W_i)) + L_c) \tag{5.5}$$

where the additional cost $L_c$ is due to the periodic leaping that happens after every interval.

Figure 33 shows a plot of the normalized expected completion time of co-located shadows (without utilizing idle time) for different values of the overhead $\beta$. We see form the figure that, when co-located shadows have no overhead ($\beta = 0$), the completion time is always lower than traditional C/R. When the overhead is non-zero, however, it starts off worse than traditional C/R and starts outperforming C/R at higher system sizes. For example, with 20% overhead ($\beta = 0.2$), co-located shadows outperform traditional C/R when system size increases beyond 30,000 nodes. With even higher overheads, though, the crossover may happen too late. For example, when $\beta \geq 0.4$, at system scales at which the shadow based scheme outperforms C/R, replication has lower completion time than both schemes.

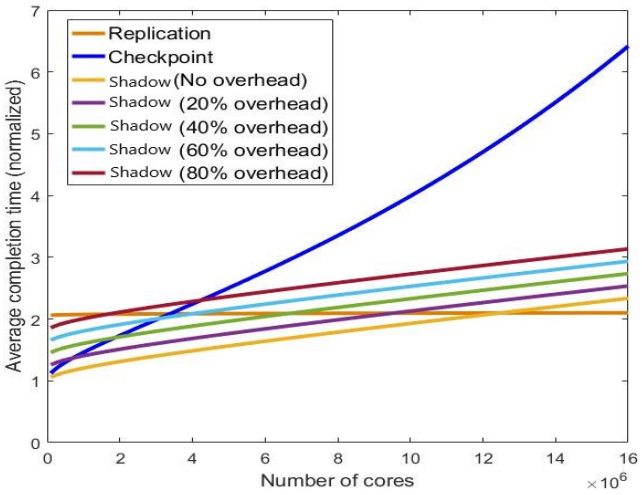



Figure 33: Model based performance of co-located shadows vs traditional C/R and replication. Both the checkpointing and leaping cost are taken to be 100 seconds each. Reboot time is taken as 300 seconds.

We can therefore conclude that, provided the overhead of co-locating shadows and message forwarding is not too high, there is a range of system scales at which this scheme can outperform both traditional C/R and pure replication.

### 5.3 Implementation Background

In order to realize the fault tolerance model based on shadows discussed in the previous section, I use the implementation of co-located shadows[19] in OpenMPI. This section will describe the different components of the implementation, most of which are retained from the original work in [19], with the notable exception that my implementation places shadows on the same cores as the mains. The implementation utilizes MPI's profiling interface, PMPI, to wrap MPI calls with the shadow library code and thus sits between the application and the MPI library. The implementation requires no modification to the original application

code, except for the specification of the variables that constitute the process state, whose values would be transferred to the other process in case of leaping.

### 5.3.1 Process Management

An MPI application that originally runs $N$ processes will start $2N$ processes in this implementation from which $N$ will be shadows. By specifying the rankfile used by *mpirun*, I place a main and its shadow on different processor cores. The results in this chapter are generated from experiments that used a simple round robin placement strategy, where main $i$'s shadows is placed with main $i + 1$. This spawning of shadows and their placement is transparent to the application. When main $i$ and its shadow, which has the original rank $i + N$ in the underlying MPI runtime, call `MPI_Rank`, they both see their rank as $i$. The implementation also transforms all collective operations on the `MPI_COMM_WORLD` communicator, into operations on a duplicated communicator that spans the mains only. I will describe in Section 5.3.2 how communication for the shadows is handled.

### 5.3.2 Message Passing and Consistency

During normal execution, receiver based message logging is employed to log messages at the shadows, as shown in Figure 34 (left). Hence, for a main, any MPI routine that results in the caller receiving any data from one or multiple processes is followed by a forwarding of the received data to the main's shadow. The shadow, therefore, receives data only from its main, which also ensures consistency between the main and its shadow.

An issue that arises with this forwarding protocol is the handling of forwarded messages at the shadow, since they are not consumed immediately. For this reason, another thread, called the *helper thread*, is created in the shadow process and its sole purpose is to receive the message from the main and log it into the buffer, as shown in figure 34 (right). This thread remains suspended and wakes up only when there is a message from the main, which it receives, copies into the buffer, and goes back to sleep. The compute thread, which executes the application code in the shadow and consumes messages from the buffer, is the one that is used for recovery.

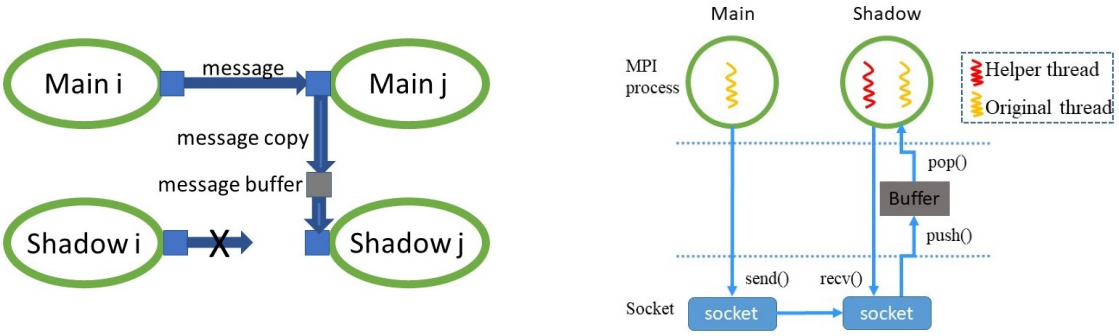



Figure 34: (a) Message transfer when Main i sends a message to Main j. (b) Message forwarding from a main to its shadow. The shadow's Helper thread receives the forwarded message and places it immediately into its local buffer (*push()* operation. The slower original thread at the shadow reads the data when it reaches the point where it needs that message (*pop()* operation).

### 5.3.3 Failure Recovery

When a main fails, the other mains also pause execution. The shadow of the failed main thus gets to execute at full speed and consumes messages from its buffer. Once the shadow has executed up to the point of failure, its state is transferred to the main through application level leaping[19], where the values of all the variables that capture the current state of the process are transferred to the rebooted main which uses those values to update its state. the remaining shadows that are not involved in the recovery process are simply *leaped* to the state of their mains. Since this leaping can happen while recovery is taking place, these shadows will also be up-to-date with their mains' state once recovery is complete.

### 5.3.4 Buffer Overflow

When a shadow's buffer is going to overflow, its main can simply transfer its state to its shadow via leaping and the shadow can then flush its buffer. Every time this happens, it adds an overhead to the main that is equal to the cost of leaping. However, this also results

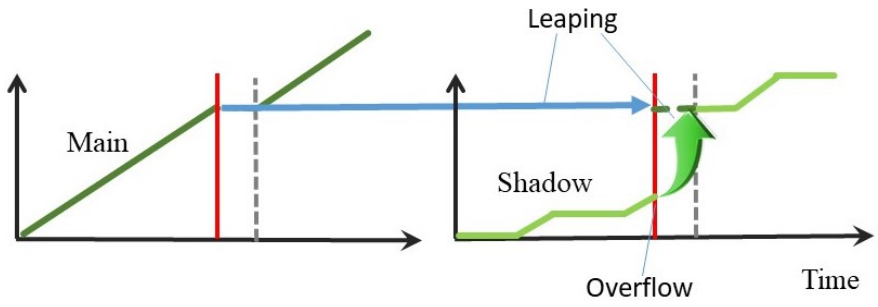



Figure 35: Leaping in case of buffer overflow

in the shadow being up-to-date with its main, as shown in figure 35, which can result in a lower recovery time in case of future failure. To perform leaping and buffer handling before the shadow's buffer overflows, a threshold, usually around 95% of the buffer capacity, is set within the library. When the threshold is reached, the *helper* thread at the shadow sets a flag and sends a message to the main notifying it that the buffer has reached its threshold. The main continues execution until a designated point in the loop from where leaping can take place, and then leaps the shadow. Since the main and shadow are now at the same stage in their execution, the shadow simply flushes the buffer and the main resumes normal execution.

## 5.4 Processor Sharing to Utilize Idle Times

I have so far described the design and implementation of co-located shadows with the exception of one key detail: how to execute the shadow during the times in which its co-located main is idle? The ideal realization of this goal would allow the shadow to execute only when the main is in a code section where it would usually perform a busy wait. In reality, however, scheduler characteristics and message logging requirements make this a much more challenging task. In this section I will describe the mechanisms by which I attempt to achieve this goal.

### 5.4.1 Processor Yielding

The first step in utilizing the idle time is to make the main relinquish the the processor rather than do a busy wait, for which I use the `sleep()` call. Hence, I transform every blocking MPI call by the main into a nonblocking call to the same operation followed by a loop that does the following in every iteration: i) test and, if the operation has completed, exit the loop, otherwise ii) relinquish the processor by calling `sleep()` for a small amount of time $t_s$. The sleep time $t_s$ is a lower bound on the time it will take the calling process to get the processor back. This is a lower bound because the `sleep()` function works by taking the calling process off the ready queue and setting a timer to expire when the amount of time elapsed equals the input sleep time. When the timer does expire, however, the process is put back on the ready queue, which means it may have to wait an arbitrary amount of time before being scheduled back onto the processor. I observed from my tests on the `sleep()` call that the lowest amount of *average* sleep duration would be on the order of a few microseconds, which means that the range of meaningful values for $t_s$ starts at a few microseconds. I then ran experiments with different values of $t_s$, starting with 1 $\mu s$, on HPC benchmarks where the MPI calls were transformed as described above, and found that values between 2 and 50 microseconds often yielded the best performance.

### 5.4.2 Behavior of Shadow Process

Note from the previous section that the shadow process consists of two threads: i) helper thread which buffers the forwarded messages from the main, and ii) the compute thread which executes the original application code and is used for recovery. Since the helper thread only wakes up to log a message, it only competes with the co-located main when it has a message pending, at which point it should quickly receive it so that the forwarding main at the other end is not kept waiting for long. Thus the helper thread is kept at either the same or a higher priority than the co-located main.

The compute thread, on the other hand, is the thread that we want to execute only when the co-located main is idle. By putting the main to sleep and relinquishing the processor, I allow the OS to schedule the shadow's compute thread to run while the main sleeps. In order

to prevent the compute thread from being scheduled at other times when the co-located main is not sleeping, I resort to Linux's implementation of priorities using the *niceness* concept. I *nice* the compute thread by giving it the highest possible nice value, thus reducing its priority. Linux's CFS tries to divide processor time between threads in the same ratio as their priorities[54]. By making this ratio as small as possible, I minimize the possibility of the compute thread competing with the co-located main for processor time when the main is not idle. Also, since the scheduler is work conserving in nature, which means it would not leave the core idle when there are runnable tasks, the expectation is that the compute thread would run when the main goes to sleep.

There is another parameter in the Linux kernel scheduler that affects the behavior of the implementation described above. This parameter, called `sched_wakeup_granularity_ns`, controls how quickly a task that wakes up from a sleep can preempt a low priority task. If this parameter is set to a higher value, the co-located main will have to wait longer after waking up to be rescheduled onto the core. This disrupts the entire mechanism because the main is unnecessarily delayed at the expense of the compute thread and the whole application might actually end up behaving as if the main and shadow are sharing the processor equally among themselves. To avoid this, I changed the default value of this parameter to 250 microseconds. This does not mean that the main must wait 250 microseconds after waking up to get the core back from the shadow. This value is scaled up or down inside the kernel scheduler based on the relative priorities of the preempting and preempted tasks, so the main actually starts executing much earlier than 250 microseconds after waking up.

## 5.5 Evaluation

To compare co-located shadows with traditional C/R, I also implemented a simple version of in-memory checkpointing where each process stores its initial state, i.e. the values of its local variables, in the memory of another process, known as the *buddy* process[80]. In case of failure, all processes fetch their last stored state from their buddies and re-execute. The crux of the model presented in section 5.2 deals with the behavior of CoLoR within an

interval, which is multiplied by the number of intervals just like in traditional checkpointing. Hence, all the experiments were done without periodic leaping and the results represent the behavior that should be expected within one periodic interval. Similarly, to make the results representative of the behavior on one checkpointing interval, the checkpoint/restart (C/R) implementation takes only one checkpoint at the beginning of the execution, with the end of the application execution representing the end of an interval.

### 5.5.1 Experimental Setup

I tested the implementation on the University of Pittsburgh Center for Research Computing's cluster which contains 100 nodes, each consisting of dual socket 28 core Intel Xeon E5-2690 2.60 GHz (Broadwell) processors and 64 GB of RAM. The nodes are connected via 100 GB OmniPath interconnect. The system runs Linux kernel version 3.10.0 and I used OpenMPI library 2.0.2. For all of the experiments, I placed shadows with the mains as follows: main $i$'s shadow is co-located with main $i + 1$, except for the main with the highest rank whose shadow is simply co-located with main 0. This mapping assumes an individual core as the unit of failure. If, however, the failure model assumes node failures (i.e. all cores on a node fail simultaneously), the mapping should be done so that all shadows belonging to mains on a single node are placed on a different node. The buffer size was usually set at 1 GB. All the results reported in this section are generated after taking the average of 3 runs.

### 5.5.2 Failure Injection

Since failure detection and notification is beyond the scope of this work, I follow the approach of [79] to simulate a failure by inserting a `inject_failure()` function at an arbitrary point in the code. The function randomly selects a process rank and an iteration number at which it simply suspends itself if called from the process chosen for failure to strike. A limitation imposed by the MPI runtime is that it kills the entire job if a process dies. Therefore, I simulate failure by waking up the same failed process after a certain amount of *downtime* after which I leap it to the state of its shadow, once the shadow reaches the point of failure, to completely mimic the recovery process.

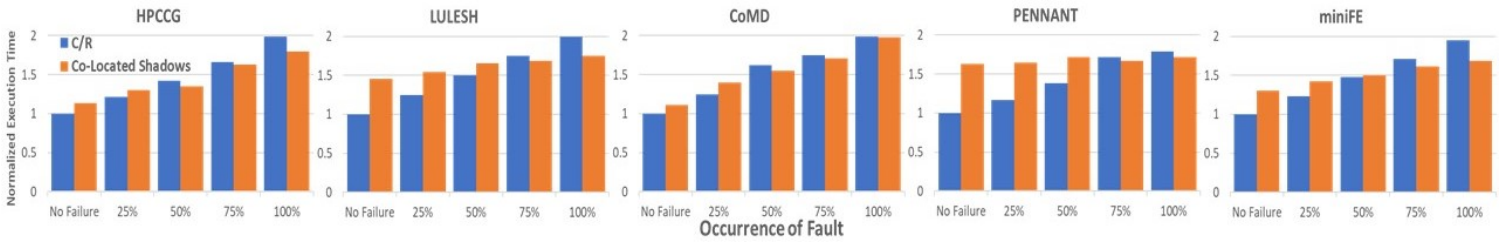



Figure 36: Performance with no and single failure injected at different point of execution, normalized by completion time of original application under no failures.

### 5.5.3 Results

Fig 36 shows the performance of co-located shadows under 0 and 1 failures. To assess the impact of 1 failure, a failure was injected in randomly selected processes at 25%, 50%, 75% and 100% of their execution respectively. All applications were run over 1024 cores, except LULESH, which ran on 512 since it runs on cubic number of processes. The reported time is normalized to the original execution time of the application in each case.

We can see that the failure free overhead of co-located shadows ranges from 11% (for HPCCG) to 63% (PENNANT benchmark). In case of a failure, however, we see the potential of co-located shadows to cut down on the recovery time. For all of the benchmarks, the completion time when a failure occurs at the end of the interval/execution is lower with co-located shadows than with simple reexecution under C/R. For the statistically representative failure point (50%), co-located shadows perform better than (or almost the same as) C/R for three of the benchmarks, namely, HPCCG, CoMD and miniFE. For LULESH and PENNANT, however, the failure free overhead is too high for the shadows to make it up during recovery when failure happens at the midpoint (50%).

The two primary sources of the overhead of co-located shadows are: i) message forwarding from main to shadow, and ii) the overhead due to the shadow's compute thread executing at times other than the idle periods of the co-located main. In order to distinguish between the two overheads, I ran an experiment where I basically deactivated the compute thread in the shadow by making it sleep until the mains finished their execution. Since only the help

Table 4: Impact of Shadow Compute Thread on Normalized Execution Time

| Benchmark | Inactive Shadow Compute Thread | Active Shadow Compute Thread |
|---|---|---|
| HPCCG | 1.03 | 1.14 |
| LULESH | 1.10 | 1.46 |
| CoMD | 1.04 | 1.11 |
| PENNANT | 1.73 | 1.62 |
| miniFE | 1.04 | 1.30 |

thread remained active, this allows us to capture the overhead due to the first source, i.e. message forwarding. The results of this experiment are presented in Table 4. It can be seen that the impact of the compute thread on the overhead is quite significant for miniFE and LULESH benchmarks.

One surprising result from the above experiment is that the execution time of PENNANT increases when the compute thread is deactivated. This suggests the possibility of there being wasted time due to the forwarding of messages by the mains and their reception by the co-located shadows. A distinguishing feature of the PENNANT benchmark is that it is characterized by a large number of small messages. I postulate that this feature makes the following scenario likely: Main $A$ initiates the forwarding of a received message but is preempted by its co-located shadow's help thread before it can complete the operation. The help thread belonging to the shadow of main $A$ will be occupying the processor on the other end but will simply be waiting on $A$ to transfer the message, wasting processor cycles. The possibility of this scenario cascading and playing out over multiple mains and shadows further exacerbates the situation. The likelihood of such a scenario playing out increases when the compute thread is out of the picture. To validate this hypothesis, I ran the same experiment with PENNANT but set the scheduling policy of the help threads as real time[30], which would mean that the help threads acquire the processor as soon as they wake up, rather than the wait enforced on them by the default Linux scheduler. This setup should further increase

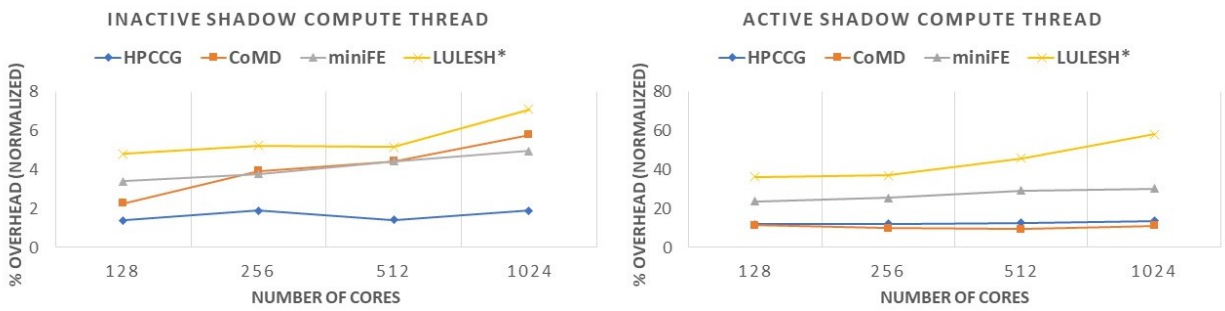



Figure 37: Weak scaling (*LULESH was tested on 125, 216, 512 and 1000 cores)

the likelihood of wasted time, especially when there are a lot of messages to be forwarded, and is in fact reflected by the jump in normalized completion time from 1.73 to 6.14 when real time help threads are used.

Finally, Figure 37 presents the weak scaling overhead of co-located shadows, when the compute thread is active as well as when it is deactivated. We see that when the compute thread is inactive, the overhead generally increases with scale. Although the overhead with an active compute thread is much higher, the increase with scale is not as steep, with the exception of LULESH.

## 5.6 Related Work

There have been several prior works that looked at the phases in which an HPC application is waiting on communication to finish, with the goal of either minimizing such periods or leveraging them to optimize other objectives. A study of the impact of imbalance on the cost of synchronisation in MPI programs is done in [63] whereas [64] studies how idle periods in HPC applications propagate from one process to another. The idea of using these idle periods to reduce energy consumption has been explored in [66] and [32]. Similarly, [78] uses these idle phases for improving the performance of in-situ workloads. In this work, I try to capitalize on those idle times during communication to run the shadow processes and thus

aim to reduce the recovery time in case of a fault.

The work in this chapter builds heavily on the idea and implementation of shadow computing[57][16]. The basic idea of shadow computing is to have redundant processes that can execute at a slower speed than the original application processes, or *mains*. The model can be tailored to different platforms [17] or failure models [15]. An implementation of shadows for MPI applications was presented in [19], which I also discuss in Section 5.3, since I build the co-located shadows using this implementation. The main difference between the two implementations is that the original work in [19] requires extra resources since shadows are placed on separate cores from the mains. In this work, however, I place the shadows on the same resources as the original application, avoiding the need for additional resources.

## 5.7 Summary

This chapter explored the utility of idle times, due to imbalance in HPC applications, in providing efficient fault tolerance. To that end, I designed and implemented a fault tolerance scheme using co-located shadows. I conducted experimental results on real benchmarks to investigate the overhead of co-located shadows as well as their performance under failures. The results indicate that while there is the potential to utilize idle times in cutting down the recovery time, further support from the system scheduler and its integration with MPI runtime may be needed in order to fully realize the potential of co-located shadows.

# 6.0 Failure-Aware Resource Allocation under Heterogeneous Failure Likelihoods

So far this dissertation has focused on fault tolerance under heterogeneity for individual jobs. This chapter takes a wider view over the system, spanning multiple jobs, and focuses on one particular area where heterogeneity leads to new avenues of exploration. Specifically, I look at resource allocation in a cluster with non-identical failure rates, with the aim of reducing waste due to failures. I formulate and study two objective functions that capture the reliability of jobs to be scheduled onto the resources, which lead to two heuristics for reliable resource allocation. Using simulation as well as analysis of job and failure traces, I show that these heuristics can contribute towards significant reduction in system waste.

## 6.1 Introduction

One of the roles of job schedulers in large scale clusters is to assign the available computational resources to the set of jobs ready to be scheduled. Traditionally, such assignments are done without considering their impact on the reliability of jobs, because the default assumption is that the compute nodes have identical failure distributions, rendering all the assignment candidates at a given time equivalent in terms of their reliability. However, with non-identical failure distributions, allocating resources to jobs while disregarding the differences in reliabilities of those resources ends up increasing the waste incurred by the system. This represents a missed opportunity since utilizing a finer understanding of system reliability can lead to significant savings in processing power wasted on failed jobs. Consider, for example, that, after a job completes, there are enough free nodes in the system to start the next two jobs in the queue. The question that the scheduler must answer is, which nodes should be assigned to each job, when node reliabilities are non-identical? To answer this question, the scheduler would need a metric that quantifies the system waste as a result of failed jobs, as well as an allocation strategy that minimizes said waste. A scheduler that is

equipped with such capabilities is well positioned to minimize system waste due to failures, since it has access to the global state of the system, both in terms of the jobs present and in terms of the current reliability of system resources, which it can gauge through different monitoring tools and system logs that individual jobs may not have access to.

In this chapter, I specifically tackle the problem of how best to assign compute resources with heterogeneous reliabilities whenever there are enough resources to start more than one waiting job. I formulate two objective functions that incorporate node reliabilities to assess the goodness of an allocation relative to others, and use those objectives to devise resource allocation heuristics that maximize their respective objective function. A novel contribution of this work is to study the resource allocation problem in the presence of replicated jobs.

## 6.2 Making Resource Allocation Failure-Aware

I start by describing the problem setting, which is typical of a computing cluster that serves multiple users. Individual users submit job requests containing, among other things, information on the number of nodes that the job will run on as well as the wall clock time limit. A submitted job is placed on one of the available queues depending on the scheduling policy in the system and starts once its scheduling priority is met and its required number of nodes become available. Unlike a cloud environment, a scheduled job is given exclusive access to its allocated nodes for its entire runtime, or its wall clock time, whichever comes first.

I assume that individual compute node failures are independent and follow an exponential distribution, albeit with different failure rates. The impact of a failure of a node is to terminate the entire job using that node as part of its allocation, wasting the work done from the start of the job (or the last checkpoint) to the time of failure. A failed job may be resubmitted at a later point in time, either by the user or automatically by the system, at which point it will be treated as any other job vying for resources. Thus, in my analysis, a failed job's contribution is added to the wasted work at the time of failure, after which it is considered to have exited the system.

### 6.2.1 Problem Statement

Once the scheduler reaches the point where it needs to decide which available nodes to allocate, if there is only one job ready to execute, the failure-aware allocation strategy simply boils down to assigning the most reliable nodes to the ready job. However, if the number of available nodes is enough to start more than one jobs, finding the best failure-aware allocation for the set of *ready* jobs is not so straightforward. In the rest of this section we formulate the objective functions and heuristics for finding good failure-aware allocation strategies when there are more than one jobs ready to be executed.

Based on the above discussion, we can now formally define the problem we are trying to solve. Let $K$ be the number of jobs that are ready to be executed at a given instant of time. Each job is characterized by the number of nodes it is requesting, $n_j$, and its specified wall clock time, $t_j$, where $j \in \{1, ..., K\}$. The number of available nodes from which the $K$ jobs have to be assigned is $N$, where $N \geq \sum_{j=1}^{K} n_j$. Each of the $N$ nodes follows an exponential failure distribution with failure rate $\lambda_i$ ($i \in \{1, ..., N\}$). We denote by $S_j$ the set of nodes allocated to a job, such that $|S_j| = n_j$. Our problem, then, is to determine the sets $S_j$, $j \in \{1, ..., K\}$, for the $K$ ready jobs.

### 6.2.2 Maximizing Reliability

An intuitive choice for a failure-aware allocation scheme would be to simply assign the most reliable nodes to the longest job. This allocation strategy was actually proposed in [34], though without any theoretical justification. I briefly discuss below how this heuristic can be motivated theoretically for nodes with exponential failure rates. This is useful not just as a theoretical exercise but also in understanding whether this intuition carries over to the case where replicated jobs are also thrown into the mix, as we will discuss in the next section.

The heuristic of assigning the best nodes to the longest job actually comes about if we take the probability of success, or reliability of the $K$ jobs, as our objective function to maximize. To see this, let $R_j$ denote the reliability of job $j$ and let $R$ denote the reliability of the $K$ jobs combined. Since individual node failure distributions are independent and since

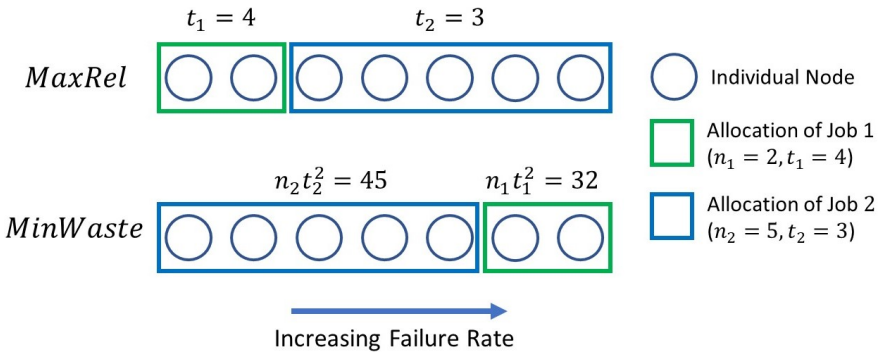



Figure 38: Example demonstrating the difference between *MaxRel* and *MinWaste* heuristics over two non-replicated jobs.

jobs do not share nodes, the reliability of the $K$ jobs is simply a product of the reliability of the individual jobs. A job succeeds if all of its nodes remain failure-free during its execution time. Thus, with exponential node failures, we can write $R_j = \prod_{i \in S_j} e^{-\lambda_i t_j} = e^{-t_j \sum_{i \in S_j} \lambda_i}$. The overall reliability is then given by

$$R = \prod_{j=1}^{K} R_j = \prod_{j=1}^{K} e^{-t_j \sum_{i \in S_j} \lambda_i} = e^{-\sum_{j=1}^{K} t_j \sum_{i \in S_j} \lambda_i} \tag{6.1}$$

We can see from the above equation that maximizing $R$ is equivalent to minimizing the quantity $\sum_{j=1}^{K} t_j \sum_{i \in S_j} \lambda_i$. Using rearrangement inequality[73], we obtain that this quantity is minimized when the largest $t_j$ is paired with the smallest $\lambda_i$'s, and so on. This leads to the simple heuristic of allocating the most reliable nodes to the longest job, and proceeding in this manner for the rest of the jobs. Thus, the discussion in this subsection serves as a theoretical justification for allocating the best nodes to the longest jobs. In the rest of this paper, we will refer to this allocation strategy as the *MaxRel* heuristic (short for Maximize Reliability).

### 6.2.3 Minimizing Waste

One aspect that the *MaxRel* heuristic ignores is the scale of the jobs. For example, if Job 1 requests 2 nodes for 4 hours and Job 2 requests 5 nodes for 3 hours, the 2 most reliable

nodes will be allocated to Job 1 under the MaxRel heuristic, as shown in Figure 38. If we go by the first order assumption[76] that a failure, if it occurs, strikes during the middle of an execution, then, in case of a failure in one of the jobs, Job 1 is expected to lose the equivalent of 2 nodes' work of done in 2 hours, while Job 2 stands to lose 5 nodes' worth of work done in one and a half hour. From the system's perspective, any work wasted due to failure translates into wasted energy expended in doing that work. Thus, the waste metric that we will use in this paper to quantify the impact of failed jobs is the product of the number of nodes and the time for which they were used to produce the work lost due to failure. Going back to our two job example, the expected waste in case of a failure would be 4 *node-hours* for job 1 and 7.5 *node-hours* for job 2, yet the *MaxRel* heuristic favors job 1 when allocating the nodes since it runs longer.

With this drawback of *MaxRel* in mind, we also propose to use the expected waste of the $K$ jobs as an alternate objective function to decide on a failure-aware resource allocation. The expected waste of a job is given by the probability of job failure multiplied by the average time of failure and the number of nodes. We follow the simplifying assumption of [76] that a failure strikes during the middle of an execution. Thus, the expected waste of a job can be written as $W_j = n_j(1 - e^{-t_j \sum_{i \in S_j} \lambda_i})t_j/2$. The expected waste of all $K$ jobs is simply the sum of the expected waste of each job, yielding

$$W = \sum_{j=1}^{K} W_j = \sum_{j=1}^{K} n_j(1 - e^{-t_j \sum_{i \in S_j} \lambda_i})\frac{t_j}{2} \tag{6.2}$$

In an attempt to understand the allocation that minimizes the waste, we further simplify the above expression using the first order approximation for the exponential term. This yields an approximation for the waste as $W \approx \sum_{j=1}^{K} n_j t_j^2 \sum_{i \in S_j} \lambda_i/2$. This is similar to the expression in the previous subsection except that we have $n_j t_j^2$ instead of $t_j$. This leads us to the following result:

***The allocation heuristic to minimize expected waste of the ready jobs is to assign the most reliable nodes to the job with the highest value of*** $n_j t_j^2$ ***and so on.***

Note that this heuristic should be applicable as long as the first order approximation is valid, i.e the quantity $t_j \sum_{i \in S_j} \lambda_i$ is small, which would be case whenever the average time to

failure of a job, given by $1/\sum_{i \in S_j} \lambda_i$, is much larger than the job duration $t_j$. In the rest of the paper, we will refer to this allocation strategy as the *MinWaste* heuristic. An example contrasting this heuristic with *MaxRel* is shown in Figure 38.

### 6.2.4 Discussion

One desirable aspect of both the heuristics discussed in this section is that they do not require actual values of the failure rates of the nodes. All that is required is the ordering of the nodes based on their reliabilities, and the nodes can then be assigned in order to the jobs based on one of the two heuristics.

I would also like to point out the rationale for our use of the term 'heuristic' for the allocation strategies discussed here, despite their being derived from theoretical objective functions (especially the *MaxRel* heuristic which exactly maximizes the reliability in Equation 6.1). A truly optimal failure-aware allocation would be based not just on the current batch of ready jobs but also on the future batches. This is because any allocation for the current batch of ready jobs will impact the set of nodes that become available to the next batches of ready jobs, thus affecting their reliability as well. This means that both the *MaxRel* and *MinWaste* allocation schemes are of the *greedy* variety, since they try to optimize their respective objective functions over the present set of jobs, regardless of the impacts their choices will have on subsequent jobs, which is why we refer to both of the allocation strategies as heuristics.

At this point one may also wonder how the discussion so far would change with jobs that take checkpoints. I assume that the decision of whether and when to checkpoint is taken by the user before submitting their job. With this assumption, all that the scheduler needs to be made aware of by the user, upon job submission, is the checkpointing interval of a job that will be taking checkpoints. The specified interval for job $j$ can then be treated as $t_j$ instead of its wall clock time. This is because the wasted work in case of failure will be the work done since the last checkpoint till the time of failure, which means the expected waste of a job will be a function of its checkpointing interval and not total wall clock time.

## 6.3 Handling Job with Replication

In Chapter 4, I explored how nodes with different reliabilities should be replicated and paired within a job. This solves the problem of deciding how best to pair nodes once they have been assigned to a job. However, to the best of our knowledge, no results exist when it comes to determining a reliable resource allocation to multiple jobs when some of the jobs use replication. In this section, I investigate how to augment the *MaxRel* and *MinWaste* heuristics to handle some cases of a job with replication, by theoretically studying the properties of the two objective functions.

Note that, similar to the assumption regarding checkpointed jobs that I mentioned at the end of last section, the assumption in this chapter is that the decision to replicate is made by the user, and the job scheduler is notified of that decision upon job submission. The scheduler's responsibility is then to decide how to do the failure-aware allocation of available nodes to the jobs, knowing which of the jobs will be employing replication.

### 6.3.1 Results on Optimizing Reliability

It may be expected that, even with replicated jobs in the mix, maximum reliability should be attained by simply ordering the jobs based on their duration and assigning the nodes in decreasing order of reliability to this ordered list of jobs. However, by simply computing a few simple numerical examples, I found that this is actually not true when replicated jobs are present. Roughly speaking, this happens because a pair of replicas only fails when both of the nodes in the pair fail, which generally makes a replica-pair more reliable. Thus, depending on the values of the failure rates of the individual nodes as well as the durations of the jobs, it is quite possible that a weak pair of nodes replicated together may be less likely to fail over a longer time period compared to the reliability of a stronger node taken by itself even over a shorter period of time.

I also found in my investigations that, in the optimally reliable node allocation, a replicated job does not always get a consecutive set of nodes, when nodes are ordered by their reliabilities. Both of these observations suggest that the allocation problem becomes much

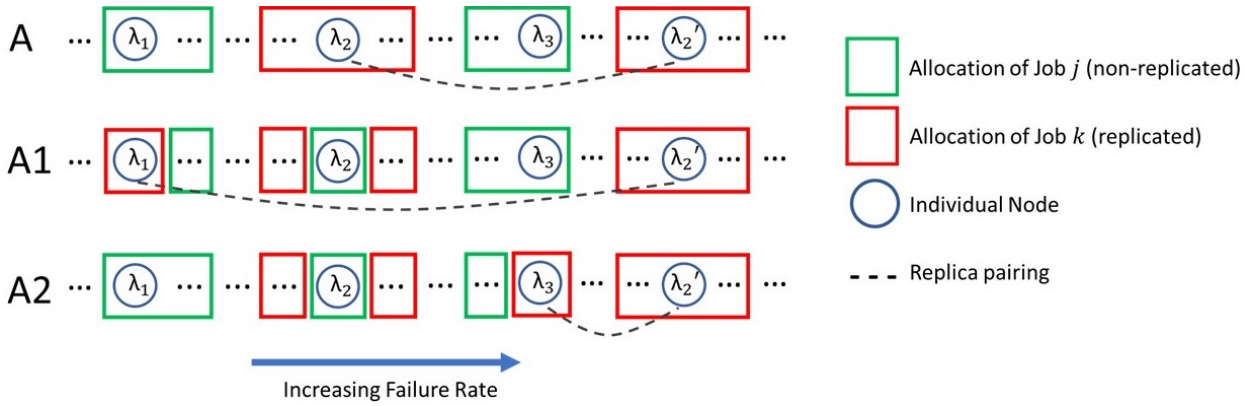



Figure 39: Some possible allocations of nodes to a replicated and a non-replicated job.

harder with replicated jobs. In the rest of this subsection, I present my theoretical result that can cut down on the search space for finding the optimal allocation for the reliability objective function.

**Theorem 5.** *In the allocation that achieves maximum reliability over the ready set of jobs, each non-replicated job $j$ satisfies the following property:*

$$i \notin S_j \rightarrow \lambda_i \leq \lambda_{min^j} \text{ or } \lambda_i \geq \lambda_{max^j} \tag{6.3}$$

*where $min^j$ is the node with the lowest failure rate in $S_j$ and $max^j$ is the node with the highest failure rate in $S_j$.*

*Proof.* The statement above basically says that, even when the ready set of jobs also contains some jobs with replication, the optimal allocation (i.e. one that maximises reliability) always assigns a set of consecutive nodes, from the list of nodes ordered by reliability, to each non-replicated job. Note that, when no jobs with replication are present, the result from the previous sections already tells us that each jobs gets a contiguous sets of nodes from the ordered list of nodes. Thus, we need to prove that in the presence of replicated jobs, every non-replicated job still gets a contiguous set of nodes. I will prove this by contradiction by showing that in an optimal node allocation, if a non-replicated job is not assigned a

contiguous set of nodes, we can always get a better allocation by swapping nodes such that the non-replicated job moves closer to having a contiguous set of nodes.

Formally, consider a non-replicated job $j$ that is not assigned a consecutive set of ordered nodes in an optimal node allocation, $A$, as shown in Figure 39. Let $\lambda_1$ be the failure rate of the most reliable node assigned to $j$ and $\lambda_3$ be the rate of the least reliable node in $j$. Based on the non-contiguous assumption, there must be a node with rate $\lambda_2$, such that $\lambda_1 < \lambda_2 < \lambda_3$ and that node is assigned to some replicated job $k$. Let $\lambda_2^{'}$ be the failure rate of the node paired with the node with rate $\lambda_2$ (the example in Figure 39 shows $\lambda_2^{'}$ as being higher than $\lambda_3$, but the proof below holds for all possibile values of $\lambda_2^{'}$). We can write the reliability, $R_A$, of this node allocation as

$$R_A = e^{-(\lambda_1+\lambda_3)t_j}(e^{-\lambda_2 t_k} + e^{-\lambda_2^{'} t_k} - e^{-(\lambda_2+\lambda_2^{'})t_k})R_{rem} \tag{6.4}$$

where $R_{rem}$ is the reliability of the remaining nodes in the allocation. Let $A1$ be the modification of $A$ obtained by swapping the assignment of nodes with rates $\lambda_1$ and $\lambda_2$ and similarly let $A2$ be the modification of $A$ obtained by swapping the assignment of nodes with rates $\lambda_2$ and $\lambda_3$. Both of these allocations are also depicted in Figure 39. We will now prove that one of $A1$ or $A2$ is more reliable than $A$.

For simplicity, let $z = e^{-\lambda_2^{'} t_k}$. Now, if $R_{A1} \geq R_A$, we are done. Hence, let us assume that $R_{A1} < R_A$, which, after simplifying the expressions, yields the following inequality

$$e^{-\lambda_1 t_j}(z+(1-z)e^{-\lambda_2 t_k}) > e^{-\lambda_2 t_j}(z+(1-z)e^{-\lambda_1 t_k}) \rightarrow \frac{e^{-\lambda_1 t_j}}{(z+(1-z)e^{-\lambda_1 t_k})} > \frac{e^{-\lambda_2 t_j}}{(z+(1-z)e^{-\lambda_2 t_k})} \tag{6.5}$$

We now need to show that $R_{A2} \geq R_A$, which means showing that

$$\begin{aligned} & e^{-\lambda_2 t_j}(z+(1-z)e^{-\lambda_3 t_k}) \geq e^{-\lambda_3 t_j}(z+(1-z)e^{-\lambda_2 t_k}) \\ \rightarrow & \frac{e^{-\lambda_2 t_j}}{(z+(1-z)e^{-\lambda_2 t_k})} \geq \frac{e^{-\lambda_3 t_j}}{(z+(1-z)e^{-\lambda_3 t_k})} \end{aligned} \tag{6.6}$$

To show that the inequality 6.6 is true if the inequality 6.5 holds, let $f(\lambda) = e^{-\lambda t_j}/(z + (1-z)e^{-\lambda t_k})$. The inequality 6.5 then becomes $f(\lambda_1) > f(\lambda_2)$ and inequality 6.6 becomes $f(\lambda_2) > f(\lambda_3)$. By taking the partial derivative of $f$ with respect to $\lambda$, it can be shown that $f$ is either i) strictly decreasing for all $\lambda > 0$ or ii) it initially increases with $\lambda$ upto

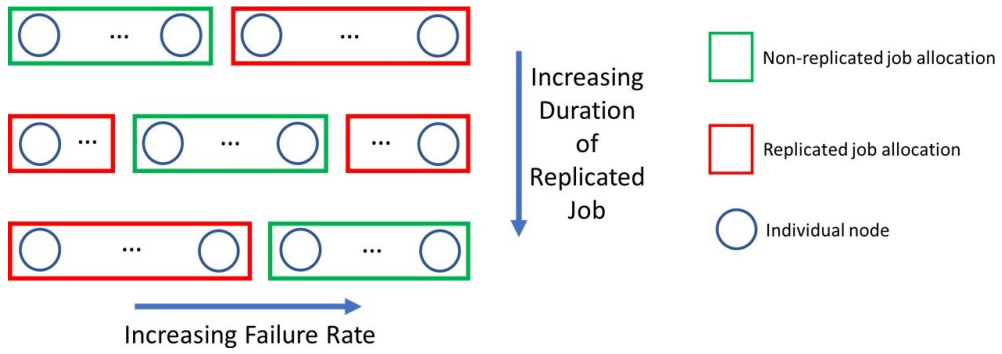



Figure 40: Possible node allocations among a pair of replicated and non-replicated jobs, when maximizing reliability. The non-replicated job always gets a contiguous set of nodes.

some value (say $\lambda_{threshold}$) and then is strictly decreasing for $\lambda > \lambda_{threshold}$. In either case, if $f(\lambda_1) > f(\lambda_2)$ as per inequality 6.5, then, since $\lambda_1 < \lambda_2$, we know that $\lambda_2$ is in the region where $f$ is strictly decreasing. Now, since $\lambda_2 < \lambda_3$, this also means that $f(\lambda_2) > f(\lambda_3)$ which concludes the proof. □

Figure 40 shows the possible ways that nodes would be allocated among a replicated and a non-replicated job based on the above result. To see how theorem 5 reduces the search space, let's say we have $K$ ready jobs requesting a total of $N$ nodes. Assume also that there is one job $r$ $(1 \leq r \leq K)$ that will employ replication. Without the above result, the number of possible allocations to check would be $N!/\prod_{i=1}^{K} n_i!$. Using the above result and the fact that we know how to order the non-replicated jobs, the number of possible candidate allocations drops down to $\binom{N+K}{K}$. While this is a substantial improvement, this number of possibilities can still be quite large in practice. For example, with $N = 1000$ and $K = 20$, this number is upwards of $4.15 \times 10^{41}$, making it practically impossible to search for the optimal allocation.

In order to make finding the optimal allocation computationally tractable I make the simplifying assumption that the nodes' failure rates can be discretized into a small number of failure classes[43]. The case of nodes belonging to two failure classes actually has precedent in the example of the Titan supercomputer[84]. With two failure classes and with our example of $K$ ready jobs with one being replicated, finding the optimal allocation simply means

determining the number of nodes of each class in the replicated job, since we know how to do the assignment for the rest of the jobs. Thus, the number of possible allocations to check is simply $n_r + 1$, which is quite feasible. Therefore, in our experiments involving replicated jobs, we will only simulate cases in which we allow only one job to be replicated from a set of ready jobs and the simulated platform will only have two types of nodes in terms of their failure rates.

### 6.3.2 Results on Minimizing Expected Waste

Switching to *expected waste* as our objective function to minimize, we know from the previous section that, as long as the first order approximation is valid, the optimal allocation will order non-replicated jobs by $n_j t_j^2$ and assign nodes, ordered by their failure rates, to this ordered list of jobs. However, with the presence of a replicated job, I again found that the replicated job does not always get a consecutive set of nodes, making it unclear how the optimal allocation would look like. Surprisingly, though, we were able to prove the same result as Theorem 5 here as well, despite the objective functions being completely different. The formal statement of the theorem and its proof is presented below:

**Theorem 6.** *In the allocation that achieves minimum expected waste over the ready set of jobs, each non-replicated job $j$ satisfies the following property:*

$$i \notin S_j \rightarrow \lambda_i \leq \lambda_{min^j} \text{ or } \lambda_i \geq \lambda_{max^j} \tag{6.7}$$

*where $min^j$ is the node with the lowest failure rate in $S_j$ and $max^j$ is the node with the highest failure rate in $S_j$.*

*Proof.* Similar to the proof for theorem 5, I will show that if a non-replicated job has non-consecutive nodes in an allocation, we can move one of its edge nodes inside by swapping with a node in between that belongs to another replicated job. Using the same notations as those in the proof of theorem 5, the expected waste for the non-replicated job $j$ in this allocation, $A$, can be written as

$$W_j^A = n_j(1 - p_j e^{-t_j(\lambda_1+\lambda_3)})\frac{t_j}{2} \tag{6.8}$$

where we denote by $p_j$ the probability of success of the rest of the nodes assigned to $j$. Similarly, for the replicated job $r$ which has a node with rate $\lambda_2$ ($\lambda_1 < \lambda_2 < \lambda_3$), the expected waste can be written as

$$W_r^A = n_r(1 - p_r(e^{-t_r\lambda_2} + e^{-t_r\lambda_2^{'}} - e^{-t_r(\lambda_2+\lambda_2^{'})}))\frac{t_r}{2} \tag{6.9}$$

where $p_r$ is the reliability of the rest of the replica pairs in job $r$.

Again, just as in Theorem 5 and as shown in Figure 39, let $A1$ be the allocation which is the same as $A$ except that the assignments of nodes with rates $\lambda_1$ and $\lambda_2$ are flipped, and $A2$ be the allocation in which the assignments of nodes with rates $\lambda_2$ and $\lambda_3$ from $A$ are flipped. We will now prove that either $W_j^A + W_r^A \geq W_j^{A1} + W_r^{A1}$ or $W_j^A + W_r^A \geq W_j^{A2} + W_r^{A2}$, which means that one of $A1$ or $A2$ has smaller expected waste than $A$.

Assume by contradiction that both $W_j^A + W_r^A < W_j^{A1} + W_r^{A1}$ and $W_j^A + W_r^A < W_j^{A2} + W_r^{A2}$. By writing out the actual expressions, we obtain that if $W_j^A + W_r^A < W_j^{A1} + W_r^{A1}$, the following has to hold

$$W_j^A + W_r^A < W_j^{A1} + W_r^{A1} \rightarrow \frac{n_j p_j t_j}{n_r p_r t_r} > \frac{(e^{-t_r\lambda_1} - e^{-t_r\lambda_2})(1 - e^{-t_r\lambda_2^{'}})}{e^{-t_j\lambda_3}(e^{-t_j\lambda_1} - e^{-t_j\lambda_2})} \tag{6.10}$$

Similarly, $W_j^A + W_r^A < W_j^{A2} + W_r^{A2}$ leads to the following

$$W_j^A + W_r^A < W_j^{A2} + W_r^{A2} \rightarrow \frac{n_j p_j t_j}{n_r p_r t_r} < \frac{(e^{-t_r\lambda_2} - e^{-t_r\lambda_3})(1 - e^{-t_r\lambda_2^{'}})}{e^{-t_j\lambda_1}(e^{-t_j\lambda_2} - e^{-t_j\lambda_3})} \tag{6.11}$$

By eliminating the quantity $n_j p_j t_j / n_r p_r t_r (1 - e^{-t_r\lambda_2^{'}})$, we can combine the above two inequalities to obtain

$$\begin{gathered} W_j^A + W_r^A < W_j^{A1} + W_r^{A1} \text{ and } W_j^A + W_r^A < W_j^{A2} + W_r^{A2} \\ \rightarrow \frac{e^{-t_r\lambda_1} - e^{-t_r\lambda_2}}{1 - e^{-t_j(\lambda_2 - \lambda_1)}} < \frac{e^{-t_r\lambda_2} - e^{-t_r\lambda_3}}{e^{-t_j(\lambda_2 - \lambda_3)} - 1} \end{gathered} \tag{6.12}$$

However, it can be shown, using the fact that $\lambda_1 < \lambda_2 < \lambda_3$, that the above inequality is not true. Rather the converse of this inequality can be proved to be true (details omitted for brevity). This contradicts our assumption that both $W_j^A + W_r^A < W_j^{A1} + W_r^{A1}$ and $W_j^A + W_r^A < W_j^{A2} + W_r^{A2}$, which means that either $A1$ or $A2$ is a better allocation than $A$ in terms of expected waste, thus concluding our proof. □

The above theorem makes the size of the search space for the allocation with the optimal waste the same as that for the optimal allocation for the reliability objective function, which means that the restrictions we will impose on the simulations involving a replicated job will also apply for the *MinWaste* heuristic.

### 6.3.3 Allocation in presence of replicated job

Based on the theoretical results in the previous subsections, we can now describe how to perform the reliability-aware assignment based on the *MaxRel* and *MinWaste* heuristics, if a replicated job is present in the set of jobs *ready* to be scheduled. Note that this *ready* set can have at most 1 replicated job and any number of non-replicated jobs. Both heuristics will test $n_r + 1$ allocations, where $n_r$ is the number of nodes in the replicated job. We can denote the possible allocations with $i$ $(0 \leq i \leq n_r)$, where $i$ will be the number of nodes of Class 1 in the replicated job. The nodes for the rest of the (non-replicated) jobs in allocation $i$ will be assigned by sorting the jobs by $t_j$ for the *MaxRel* heuristic and by $n_j t_j^2$ for the *MinWaste* heuristic, just as depicted in Figure 38. Thus, although the total number of allocations that each heuristic will consider is the same (i.e. $n_r + 1$), the allocations themselves may be different for the two heuristics. Each heuristic will then pick, from the ones it will consider, the allocation that optimises its respective objective function. In calculating the reliability of the replicated job, I apply the optimal pairing as determined in Chapter 4. Note that if the number of available nodes of a class is less than $n_r$, the number of possibilities to check will simply be equal to the number of nodes in the smaller class.

## 6.4 Empirical Results

In this section, I conduct several types of experiments to assess the utility of the failure-aware heuristics. Table 5 lists the different systems along with their individual class MTBFs that I use to generate failure traces in our experiments involving non-replicated jobs. I consider a small system of 600 nodes for the validation subsection and a large system with

Table 5: Node and System Level MTBFs ($h$: hours, $d$: days, $y$: years)

| | Small System (600 nodes) | | | Large System (49152 nodes) | | |
|---|---|---|---|---|---|---|
| | A | B | C | A | B | C |
| Class 1 | $22d$ | $45d$ | $67d$ | $5y$ | $10y$ | $15y$ |
| Class 2 | $111d$ | $134d$ | $156d$ | $25y$ | $30y$ | $35y$ |
| Class 3 | $201d$ | $223d$ | $245d$ | $45y$ | $50y$ | $55y$ |
| Class 4 | $290d$ | $312d$ | $334d$ | $65y$ | $70y$ | $75y$ |
| Class 5 | $379d$ | $401d$ | $423d$ | $85y$ | $90y$ | $95y$ |
| Class 6 | $468d$ | $490d$ | $512d$ | $105y$ | $110y$ | $115y$ |
| System-level | $3.6h$ | $5.7h$ | $7.3h$ | $3.6h$ | $5.7h$ | $7.3h$ |

49152 nodes to match with the system whose job traces I use in the subsequent subsections. For both those systems, I consider 3 sets of values for the node MTBF classes. The MTBF values for the smaller system are simply scaled down from those of the (realistic) large system's classes, so that the overall system MTBF of the two systems match. All the systems consist of 6 classes of nodes with each class having equal number of nodes.

### 6.4.1 Validation

In this subsection, I evaluate the quality of the choices made by the *MaxRel* and *Min-Waste* heuristics by computing their waste on a single *ready* set of jobs whose allocations are determined by the two heuristics. We will conduct this validation using the smaller system from Table 5. To obtain the average waste of an allocation for a given system, we conduct 100,000 trials where, in each trial, I generate failure times for each of the 600 nodes and pick the node with the first failure. If the failure time of that node falls within the duration of the job it is assigned to in a particular allocation, this counts towards the waste of that allocation. I consider three allocations: i) random, which represents the allocation which is done without taking into consideration the failure rate of the nodes, ii) the allocation

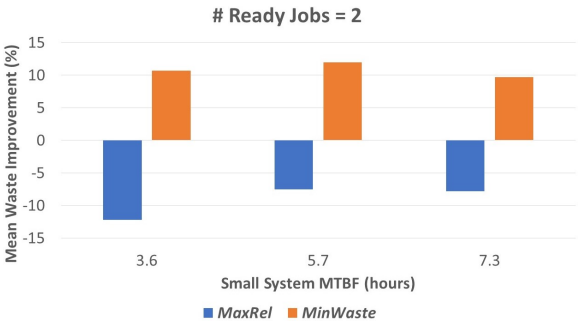
Table 6: % Waste Improvement over Random Allocation for Small System B

| # of Job 1 nodes | 60 | 120 | 180 | 240 | 300 |
|---|---|---|---|---|---|
| *MaxRel* | -5.93 | -10.51 | -11.206 | -8.02 | 0.01 |
| *MinWaste* | 20.49 | 24.40 | 20.78 | 11.43 | 0.01 |

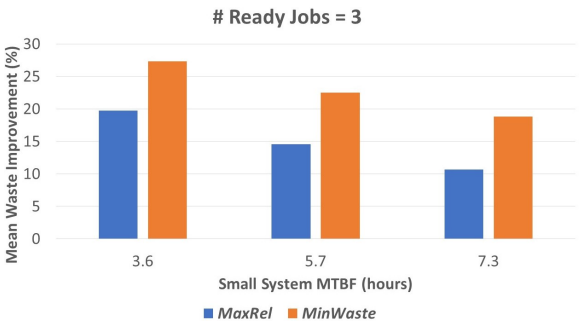
according to the *MaxRel* heuristic and iii) the allocation as per the *MinWaste* heuristic.

We start by considering a simple example of two non-replicated jobs that are ready to be scheduled. The duration of Job 1 is set to 3 hours and the duration of Job 2 is set to 2.5 hours. We investigate different values for the number of nodes requested by each job while keeping the total number of nodes for the two jobs fixed at 600 (i.e. total nodes in the system). Table 6 lists the average waste improvement of the allocation heuristics over random assignment for the system with 5.7 hours MTBF. We can see that the *MaxRel* heuristic does worse than random for all sizes of Job 1 except for when it uses 300 nodes. This is because the *MaxRel* heuristic always assigns the best nodes to Job 1 because of its longer duration. However, when the number of nodes in Job 1 is less than Job 2, it's expected waste is lower than Job 2, which means Job 1 should be assigned the least reliable nodes. When both Job 1 and Job 2 use 300 nodes each, both heuristics will assign the best nodes to Job 1 because of its longer duration, yielding identical allocations. Figure 41a shows how the allocations perform for the different system MTBFs when we fix the size of Job 1 to 240 nodes and Job 2 to 360 nodes. These results demonstrate that *MinWaste* actually performs a better allocation over the current set of jobs than *MaxRel*.

The example above demonstrates the superiority of *MinWaste* over the *MaxRel* heuristic, because the example results in different orderings using the two heuristics, and we see that *MinWaste* does the correct ordering to minimize waste. In general, it may not always be the case that the job duration and job size (in terms of the number of nodes it requests) are negatively correlated as in the example just considered. Thus, we consider another example where we have 3 jobs: Job 1 which requests 200 nodes with wall clock time of 3 hours, Job




(a) Job 1: Node counts = 240, Duration = 3 hours, Job 2: Node count = 360, Duration = 2 hours.




(b) Job 1: Node count = 200, Duration = 3 hours, Job 2: Node count = 300, Duration = 2.5 hours, Job 3: Node count = 100, Duration = 2 hours.

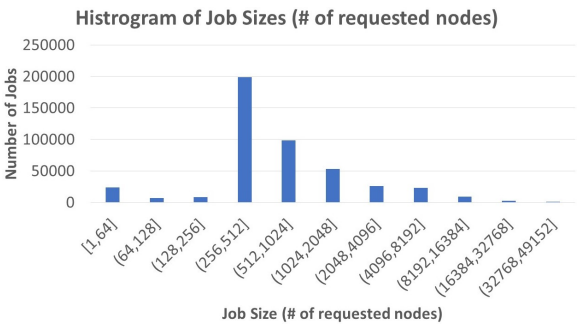
Figure 41: Relative improvement/degradation in waste of allocations made by the failure-aware heuristics. Note the difference in the scale of y-axis in the two plots.

2 which requests 300 nodes with wall clock time of 2.5 hours and Job 3 which requests 100 nodes with wall clock time of 2 hours. The ordering based on *MaxRel* will be: Job 1, Job 2 and then Job 3, whereas the *MinWaste* heuristic will use the ordering: Job 2, Job 1 and then Job 3. We see that both the heuristics agree on the position of Job 3, which results in both heuristics yielding net improvement compared to random allocation, as seen in Figure 41b. Still, *MinWaste* has the highest improvement because of its overall correct ordering.

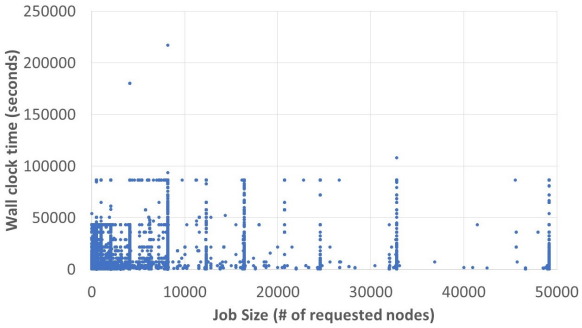
### 6.4.2 Job Trace Description

The previous section served as validation for the *MaxRel* and *MinWaste* heuristics by analyzing the impact of their choices in case a failure strikes in the same *ready* set over which those choices are made. However, as mentioned in Section 6.2, the greedy nature of these heuristics means that, even if they make the correct choice for the current batch of jobs, the choice may not be optimal for the upcoming batches of jobs. Hence, I also evaluate the impact of these heuristics over job traces using actual arrival times of the jobs.

I use the data of jobs submitted to the Mira supercomputer at Argonne National Lab.




(a) Counts of jobs with respect to the number of nodes requested.




(b) Plot of wall clock time of jobs against the number of nodes requested.

Figure 42: Job statistics from the Mira trace[50].

Mira is a 49,152 node supercomputer where the nodes are divided into 48 racks of 1024 nodes each. Each rack consists of 2 midplanes consisting of 512 nodes. The nodes are connected by a 5D torus interconnect. The dataset that I use is publicly available at [50]. The job logs, at the time of this writing, span more than 5 years of job traces starting from April 2013 to February 2020, yielding a total of 451,324 job entries, where each entry contains, among other information, the time the job entered the system, its stated wall clock time and the number of nodes requested by the job.

Figure 42a shows a histogram of number of nodes requested by the individual jobs. We can see that more than half of the jobs requested 512 nodes or less. In fact, more than two-thirds of the total jobs requested either 512 or 1024 nodes. This is because the lowest granularity of resource allocation in Mira is a midplane. Thus a job that requests 1 node only will also be assigned a midplane consisting of 512 nodes. We also see from the scatter plot in Figure 42b that the range of requested wall clock times is visually quite similar for the different job sizes. In fact, the correlation coefficient between the jobs' requested node counts and wall clock times is 0.191, which indicates there is a positive correlation between these two attributes of a job. Both of the observations suggest that that there are likely to be many jobs in any given *ready* set over which the two heuristics will agree upon the order

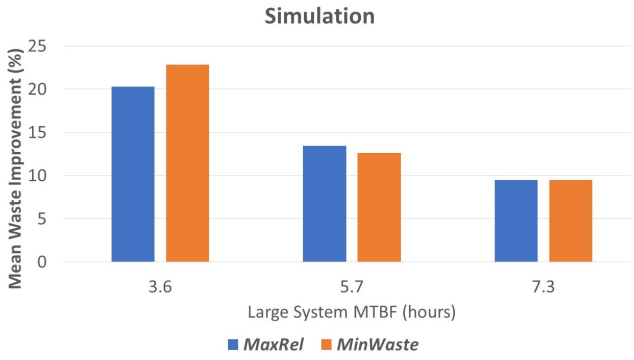



Figure 43: Relative waste improvement using simulation over the Mira job trace.

in which the nodes should be assigned.

Although the job trace also contains the actual execution time of each job, in simulations I use the wall clock time requested by the job as the measure of its duration. I do this primarily because the actual execution time is not something that the job scheduler knows *apriori* when scheduling the job, thus any heuristic that is to be deployed in practice will work on the information available at the time of making that decision. However, this is not an inherent limitation of the heuristics. There is a body of work around run-time estimation of jobs for scheduling [3][60][72]. Hence, if an accurate estimator is present, the heuristics can use those estimates in calculating the job ordering. Note that I use the wall clock times in simulations only. When doing the analysis with real failures in subsection 6.4.4, the computation of waste will be done using the actual execution times of the jobs.

### 6.4.3 Simulation

I simulate a FIFO scheduling strategy with a single queue, using the arrival times of the jobs as the time they enter the queue. I generate failure traces using the MTBFs for the larger system in Table 5 and feed them, along with the job trace, as input to the simulator. For each system MTBF, I generated three failure traces. The waste of each allocation over a system was thus calculated as the average of the wastes obtained using simulation over the three failure traces. The results from the simulation, using generated failure traces, are

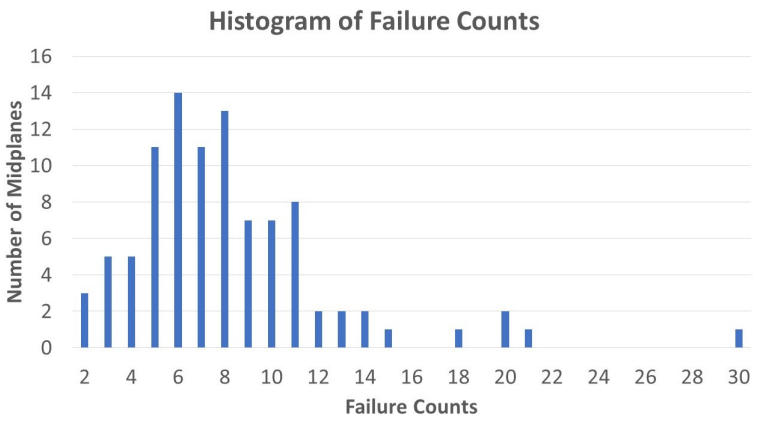



Figure 44: Distribution of failures over the midplanes.

shown in Figure 43. We see that the relative improvement increases as the system failure rate increases (i.e. decreasing system MTBF). This is consistent with the validation results in subsection 6.4.1. We also observe that both heuristics yield similar improvement to each other, because of the characteristics of the trace that we discussed in the previous subsection, such as the correlation between job duration and number of nodes.

### 6.4.4 Analysis with Actual Failure Data

In addition to the job traces, other system logs from the Mira supercomputer are also available at https://reports.alcf.anl.gov/data/mira.html. I used the *task history* logs to determine jobs that failed due to system events. Such jobs would have one or more of their tasks in the *task history* log report an error of the form: "abnormal termination by signal 35 from rank 10664 due to RAS event with record ID ..." The RAS events refer to potential failure events in the Reliability, Availability and Serviceability (RAS) logs which contain, among other information, the location of the event. Using these logs, I found that 892 jobs failed due these RAS events. I also determined from the logs the locations of the midplanes struck by these failures. Figure 44 shows a histogram of failure counts experienced by the midplanes. We see that there is sufficient spread starting with some midplanes that experienced 2 failures only, all the way to a midplane that experienced 30 failures. This

indicates that there is significant heterogeneity in the failure likelihoods of the individual midplanes.

I use the failure counts obtained above to come up with a ranking of the midplanes in terms of their failure likelihoods. I then went back to the jobs logs to determine how many of the 892 failed jobs started with a *ready* set that contained more than 1 job. To get this information, I first sorted the jobs by their ending time in the log. Between any two jobs with consecutive ending times, we looked for the jobs that started execution between that interval of consecutive ending times to construct the *ready* sets. We found that 479 of the 892 jobs started with other jobs in their *ready* set. For each such ready set, we compiled a list of available midplanes based on the actual allocations of the jobs in that set. Next, we determined allocations of those available midplanes to the jobs in the set as per the two heuristics. If the allocations differed from the original allocation in the logs, we would compute the difference in waste between the originally failed job and the job that was assigned the failing midplane in a particular heuristic. We considered the time of failure as the ending time of the originally failed job. Note that, if the job that was assigned a failing midplane, based on a heuristic, finished before failure, we would allocate its resources to the next *ready* set that came after the job that finished. With this analysis, we found that, compared to the waste of the original 892 failed jobs, the *MaxRel* heuristic resulted in a saving of 16.97% while the *MinWaste* heuristic yielded a saving of 17.07%. This indicates the potential of failure-aware heuristics to save on computational resources and processing time that are wasted on failed jobs.

### 6.4.5 Replication

Since prior works[29][7] have shown that replication only becomes viable at large scales where the system MTBF is high, I simulated a projected exascale system by scaling the number of nodes in Mira by 4x. Thus, the number of nodes requested by the jobs in the trace were also scaled by a factor of 4. I then designated all the jobs that requested more than half of this new system's nodes as jobs that would be using replication. This also ensures that, in any set of *ready* jobs, there would only be one replicated job. We cut down

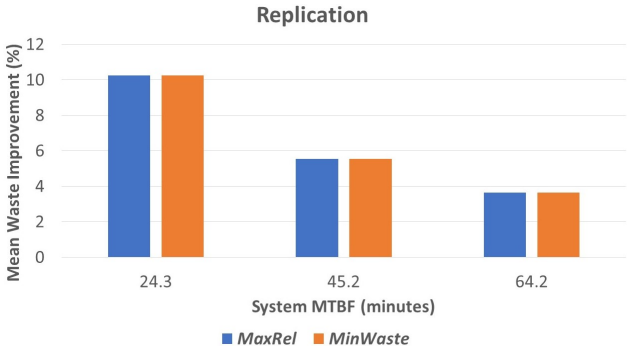



Figure 45: Relative waste improvement for systems with two equal-sized classes of nodes. For the lowest system MTBF, node MTBFs were 5 and 50 years. The system in the middle had nodes with MTBFs of 10 and 55 years, while the system on the right had nodes with 15 and 60 year MTBFs. The waste of each allocation was averaged over three runs.

the job trace to the first 100,000 jobs, based on their arrival time, out of the total 451,324, in order for the simulation on the large scale system to finish in a reasonable amount of time. The results for different values of system MTBF are depicted in Figure 45. We see again that both heuristics yield similar improvements in waste, which decrease as system MTBF increases, consistent with earlier results involving non-replicated jobs.

### 6.4.6 Discussion

Based on the results in this section, I conclude that the failure aware resource allocation heuristics do end up saving a significant portion of the energy that would normally be wasted with an allocation scheme that is oblivious to the differences in the failure likelihoods of the individual nodes. Of the two heuristics studied, *MinWaste* is closer to the optimal resource allocation over the current set of jobs. In simulation on real job traces over a longer time duration, both the *MaxRel* and *MinWaste* heuristics yield similar improvements.

It is also worth pointing out that the failure-aware allocation heuristics in this chapter solely focused on the reliability characteristics of the resources, ignoring other factors that may be considered when allocating resources. In fact, the default consideration when as-

signing nodes to jobs in large scale supercomputers is network performance, which is why the usual allocation mechanism is to devise an ordered list of nodes in the system based on network topology, where nodes closer to each other in the list generally have higher communication performance, and to assign nodes to jobs from the list in that order. However, due to the unpredictable nature of the workloads and scheduling optimizations such as backfilling[60], fragmentation can still occur in practice[83]. Thus, the performance impact of reliability aware allocation due to potential degradation in network performance may not necessarily be significant when measured at the application level. An interesting example in this regard is the work in [84], where the authors take the list of nodes (in their case GPUs) ordered by network topology and then do another pass over the list, moving reliable nodes to the front of the list, essentially reordering the list with respect to reliability. They found that, while this did increase the average hop count between nodes in a job in their simulation runs, the performance impact to applications in the production run was negligible. This indicates that in a system with few classes of nodes this 2 step ordering can be utilized to get a list of nodes ordered by reliability such that the list maintains the network topological ordering among nodes of the same class.

Yet another approach to preserving some of the network induced performance while doing a failure-aware allocation could be based on our analysis in subsection 6.4.4. We took a midplane consisting of 512 nodes as the granularity for failure-aware allocation, which ensures that the nodes with the greatest connectivity, i.e. those within a midplane, are allocated to the same job. Thus the general approach based on this concept would be to use a coarser granularity for failure-aware allocation so that the network based allocation is respected for the nodes with the highest connectivity, i.e. those within the allocation block. This, however, may lead to lower utilization in systems where the original allocation is done over a finer granularity.

## 6.5 Related Work

The most closely related work to ours is [34], where the authors modeled the reliability of different nodes in an HPC system based on the time of last failure of each node. Using that reliability function they proposed the heuristic of assigning the most reliable nodes to jobs longer than a threshold, similar to the *MaxRel* heuristic. My work differs from theirs in several aspects. For example, their waste metric was simply the time spent by a job before failure, whereas I take into account the scale of the job as well as the duration, using which I derived the *MinWaste* allocation heuristic. Another difference is that I provide novel theoretical results for the heuristics proposed.

Yet another closely related work that is also more recent is [84] where the authors' goal was to improve the reliability of the higher priority *leadership* jobs in a system where they essentially had two types of nodes in terms of their likelihood of failure. Their approach was tailor made for their specific scenario, i.e. system with two types of nodes and jobs with two priority classes. In contrast, our allocation heuristics are more general and can be applied to any system with arbitrary heterogeneity in the nodes' failure likelihoods without requiring that they specifically belong to one of two classes. Secondly, I do not enforce a fixed, binary priority criteria for the jobs but rather order the ready jobs based on an impartial, global criteria such as overall reliability or expected system waste.

Although several prior works have demonstrated the effectiveness of replication at large scale[29][7][45] as well as our prior work [43] on reliable node pairing under heterogeneous failure likelihoods, to the best of our knowledge, this is the first work that looks at the reliability-aware resource allocation problem in the presence of replicated jobs.

## 6.6 Summary

In this chapter, I tackled the problem of how to find a reliable resource allocation to jobs in clusters where the individual compute resources may have different likelihoods of failure. I derived two heuristics for carrying out such a failure-aware allocation by theo-

retically analysing two objective functions that capture the reliability of the jobs that are being scheduled. I further studied how those objective functions provide relevant theoretical insights that can be used to augment the proposed heuristics in situations where replicated jobs are also possible. Finally, I used simulation and analysis of real workload traces to demonstrate the significant savings in energy that can be achieved by using the proposed failure-aware heuristics.

# 7.0 Conclusion and Future Directions

One major theme in the research and development of Exascale systems is resilience. This is because systems at such massive scales will be realized by a proportional increase in the number of underlying components, thus increasing the likelihood of failures in one or more of those individual components. Thus, there is a pressing need to explore all possible avenues that could lead towards highly efficient fault tolerance techniques for large scale computing systems. I explored in this dissertation one such avenue, namely heterogeneity, and showed that it holds significant potential in making fault tolerance techniques more efficient.

In the first part of this dissertation, I explored the consequences of heterogeneity in failure likelihoods in large scale systems for the two most popular resilience techniques: checkpointing and replication. I showed that changing the assumption of identical failure likelihoods to a non-uniform characterization leads to novel problems regarding checkpoint placement and replica pairing. I provided solutions to these problems and also demonstrated that heterogeneity can be exploited by both the fault tolerance techniques to significantly improve their efficiency.

Another type of heterogeneity that I considered in this dissertation is the imbalance in HPC workloads. I discussed how this imbalance can be used to provide savings in recovery time after failures and thus lead to lower overhead of fault tolerance. I designed and implemented a fault tolerance scheme using co-located shadows to capitalize on the idle times resulting from the imbalance and demonstrated the potential of using those idle times in reducing the impact of failures.

In the final part of this dissertation, I explored how the idea of heterogeneity in failure likelihoods could be incorporated at the system level, in addition to its usefulness when applied at the level of individual jobs. I studied the problem of resource allocation to jobs in an HPC cluster and showed that techniques that take into account the heterogeneity in failure likelihoods of the individual resources and perform resource allocations accordingly can significantly reduce the failure induced waste in the system. This makes the case for equipping runtime systems and job schedulers in large scale system with an awareness of the

heterogeneity, if it exists, in the failure likelihoods of resources within the system.

The research in this dissertation opens up further avenues of exploration with heterogeneity in failure likelihoods. For current fault tolerance techniques, a future direction would be to explore even more ways in which heterogeneity can help to reduce the overheads of resilience. For example, with checkpointing, more sophisticated encoding approaches for diskless checkpointing could be evaluated for their feasibility in the presence of heterogeneity. Similarly, with replication, the investigation of reliability-aware speedups for partial and higher degrees of replication is called for in the context of heterogeneous failure likelihoods, based on the results in the first part of this dissertation. At the system level as well, there are several research questions pertaining to job schedulers and resource managers that can arise in the presence of resources with non-identical fault rates. One such research problem would be to determine the level of responsibility assigned to the resource manager, since a resource manager may be better suited to decide the level of fault tolerance for individual jobs. Such research questions could radically overhaul the next generation of job schedulers for large scale systems with heterogeneous failure likelihoods.

All in all, this dissertation demonstrates the significant potential of using heterogeneity-awareness in making fault tolerance for current and future large scale systems more efficient and cost effective.

## Bibliography


[1] Abhinav Agrawal, Gabriel H Loh, and James Tuck. Leveraging near data processing for high-performance checkpoint/restart. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 60. ACM, 2017.

[2] 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*, pages 483–485. ACM, 1967.

[3] Guillaume Aupy, Ana Gainaru, Valentin Honoré, Padma Raghavan, Yves Robert, and Hongyang Sun. Reservation strategies for stochastic jobs. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 166–175. IEEE, 2019.

[4] Leonardo Bautista-Gomez, Seiji Tsuboi, Dimitri Komatitsch, Franck Cappello, Naoya Maruyama, and Satoshi Matsuoka. Fti: high performance fault tolerance interface for hybrid systems. In *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, pages 1–12. IEEE, 2011.

[5] Anne Benoit, Aurélien Cavelan, Franck Cappello, Padma Raghavan, Yves Robert, and Hongyang Sun. Coping with silent and fail-stop errors at scale by combining replication and checkpointing. *Journal of Parallel and Distributed Computing*, 122:209–225, 2018.

[6] Anne Benoit, Aurélien Cavelan, Valentin Le Fèvre, and Yves Robert. *Optimal checkpointing period with replicated execution on heterogeneous platforms.* PhD thesis, INRIA, 2017.

[7] Anne Benoit, Thomas Hérault, Valentin Le Fèvre, and Yves Robert. Replication is more efficient than you think. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 89. ACM, 2019.

[8] Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Kerry Hill, Jon Hiller, et al. Exascale computing study: Technology challenges in achieving exascale systems. *Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep*, 15, 2008.

[9] Wesley Bland, Peng Du, Aurelien Bouteiller, Thomas Herault, George Bosilca, and Jack J Dongarra. Extending the scope of the checkpoint-on-failure protocol for forward recovery in standard mpi. *Concurrency and computation: Practice and experience*, 25(17):2381–2393, 2013.

[10] Franck Cappello, Al Geist, Bill Gropp, Laxmikant Kale, Bill Kramer, and Marc Snir. Toward exascale resilience. *The International Journal of High Performance Computing Applications*, 23(4):374–388, 2009.

[11] Henri Casanova, Yves Robert, Frédéric Vivien, and Dounia Zaidouni. *Combining process replication and checkpointing for resilience on exascale systems*. PhD thesis, INRIA, 2012.

[12] Aurélien Cavelan, Jiafan Li, Yves Robert, and Hongyang Sun. When amdahl meets young/daly. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 203–212. IEEE, 2016.

[13] Zizhong Chen, Graham E Fagg, Edgar Gabriel, Julien Langou, Thara Angskun, George Bosilca, and Jack Dongarra. Fault tolerant high performance computing by a coding approach. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 213–223. ACM, 2005.

[14] Xiaolong Cui. *Adaptive and Power-aware Fault Tolerance for Future Extreme-scale Computing*. PhD thesis, University of Pittsburgh, 2018.

[15] Xiaolong Cui, Zaeem Hussain, Taieb Znati, and Rami Melhem. A systematic fault-tolerant computational model for both crash failures and silent data corruption. In *2018 21st Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*, pages 1–8. IEEE, 2018.

[16] Xiaolong Cui, Bryan Mills, Taieb Znati, and Rami Melhem. Shadow replication: An energy-aware, fault-tolerant computational model for green cloud computing. *Energies*, 7(8):5151–5176, 2014.

[17] Xiaolong Cui, Bryan N. Mills, Taieb Znati, and Rami G. Melhem. Shadows on the cloud: An energy-aware, profit maximizing resilience framework for cloud computing. In *CLOSER*, 2014.

[18] Xiaolong Cui, Taieb Znati, and Rami Melhem. Adaptive and power-aware resilience for extreme-scale computing. In *Scalable Computing and Communications (ScalCom), 2016 Intl IEEE Conferences*, pages 671–679. IEEE, 2016.

[19] Xiaolong Cui, Taieb Znati, and Rami Melhem. Rejuvenating shadows: Fault tolerance with forward recovery. In *High Performance Computing and Communications(HPCC), 2017 IEEE 19th International Conference on*. IEEE, 2017.

[20] John T Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future generation computer systems*, 22(3):303–312, 2006.

[21] Sheng Di. Logaider. Avaliable at `https://github.com/disheng222/LogAider`.

[22] Sheng Di, Hanqi Guo, Rinku Gupta, Eric R Pershey, Marc Snir, and Franck Cappello. Exploring properties and correlations of fatal events in a large-scale hpc system. *IEEE Transactions on Parallel and Distributed Systems*, 2018.

[23] Sheng Di, Rinku Gupta, Marc Snir, Eric Pershey, and Franck Cappello. Logaider: A tool for mining potential correlations of hpc log events. In *Cluster, Cloud and Grid Computing (CCGRID), 2017 17th IEEE/ACM International Symposium on*, pages 442–451. IEEE, 2017.

[24] Sheng Di, Yves Robert, Frédéric Vivien, and Franck Cappello. Toward an optimal online checkpoint solution under a two-level hpc checkpoint model. *IEEE Transactions on Parallel and Distributed Systems*, 28(1):244–259, 2017.

[25] Nosayba El-Sayed and Bianca Schroeder. Reading between the lines of failure logs: Understanding how hpc systems fail. In *Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on*, pages 1–12. IEEE, 2013.

[26] James Elliott, Kishor Kharbas, David Fiala, Frank Mueller, Kurt Ferreira, and Christian Engelmann. Combining partial redundancy and checkpointing for hpc. In *2012 IEEE 32nd International Conference on Distributed Computing Systems*, pages 615–626. IEEE, 2012.

[27] Christian Engelmann and Swen Böhm. Redundant execution of hpc applications with mr-mpi. In *Proceedings of the 10th IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN)*, pages 15–17, 2011.

[28] Charles R Ferenbaugh. Pennant: an unstructured mesh mini-app for advanced architecture research. *Concurrency and Computation: Practice and Experience*, 27(17):4555–4572, 2015.

[29] Kurt Ferreira, Jon Stearley, James H Laros, Ron Oldfield, Kevin Pedretti, Ron Brightwell, Rolf Riesen, Patrick G Bridges, and Dorian Arnold. Evaluating the viability of process replication reliability for exascale systems. In *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, pages 1–12. IEEE, 2011.

[30] Ankita Garg. Real-time linux kernel scheduler. *Linux Journal*, 2009(184):2, 2009.

[31] Cijo George and Sathish Vadhiyar. Fault tolerance on large scale systems using adaptive process replication. *IEEE Transactions on Computers*, 64(8):2213–2225, 2015.

[32] Neha Gholkar, Frank Mueller, Barry Rountree, and Aniruddha Marathe. Pshifter: Feedback-based dynamic power shifting within hpc jobs for performance. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, pages 106–117, 2018.

[33] Leonardo Arturo Bautista Gomez, Naoya Maruyama, Franck Cappello, and Satoshi Matsuoka. Distributed diskless checkpoint for large scale systems. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 63–72. IEEE Computer Society, 2010.

[34] Narasimha Raju Gottumukkala, Chokchai Box Leangsuksun, Narate Taerat, Raja Nassar, and Stephen L Scott. Reliability-aware resource allocation in hpc systems. In *2007 IEEE International Conference on Cluster Computing*, pages 312–321. IEEE, 2007.

[35] 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*, page 44. ACM, 2017.

[36] Saurabh Gupta, Devesh Tiwari, Christopher Jantzi, James Rogers, and Don Maxwell. Understanding and exploiting spatial properties of system failures on extreme-scale hpc systems. In *Dependable Systems and Networks (DSN), 2015 45th Annual IEEE/IFIP International Conference on*, pages 37–44. IEEE, 2015.

[37] Thomas Herault and Yves Robert. *Fault-tolerance techniques for high-performance computing*. Springer, 2015.

[38] M Heroux and S Hammond. Minife: Finite element solver, 2019.

[39] Michael A Heroux. Hpccg solver package. Technical report, Sandia National Laboratories, 2007.

[40] Michael Allen Heroux. Mantevo 3.0 overview. Technical report, Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), 2015.

[41] Zaeem Hussain, Xiaolong Cui, Taieb Znati, and Rami Melhem. Color: Co-located rescuers for fault tolerance in hpc systems. In *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 569–576. IEEE, 2018.

[42] Zaeem Hussain, Taieb Znati, and Rami Melhem. Partial redundancy in hpc systems with non-uniform node reliabilities. In *High Performance Computing, Networking, Storage and Analysis (SC), 2018 International Conference for*. IEEE, 2010.

[43] Zaeem Hussain, Taieb Znati, and Rami Melhem. Partial redundancy in hpc systems with non-uniform node reliabilities. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 566–576. IEEE, 2018.

[44] Zaeem Hussain, Taieb Znati, and Rami Melhem. Optimal placement of in-memory checkpoints under heterogeneous failure likelihoods. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 900–910. IEEE, 2019.

[45] Zaeem Hussain, Taieb Znati, and Rami Melhem. Enhancing reliability-aware speedup modeling via replication. In *2020 IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2020.

[46] Hui Jin, Yong Chen, Huaiyu Zhu, and Xian-He Sun. Optimizing hpc fault-tolerant environment: An analytical approach. In *2010 39th International Conference on Parallel Processing*, pages 525–534. IEEE, 2010.

[47] Saurabh Kadekodi, KV Rashmi, and Gregory R Ganger. Cluster storage systems gotta have heart: improving storage efficiency by exploiting disk-reliability heterogeneity. In *17th {USENIX} Conference on File and Storage Technologies ({FAST} 19)*, pages 345–358, 2019.

[48] Ian Karlin, Jeff Keasler, and JR Neely. Lulesh 2.0 updates and changes. Technical report, Lawrence Livermore National Laboratory (LLNL), Livermore, CA, 2013.

[49] Richard E Korf. Multi-way number partitioning. In *IJCAI*, pages 538–543, 2009.

[50] Argonne National Laboratory. Argonne mira logs. Avaliable at `https://reports.alcf.anl.gov/data/mira.html`.

[51] Gary Lakner, Brant Knudson, et al. *IBM system blue gene solution: blue gene/Q system administration*. IBM Redbooks, 2013.

[52] Arnaud Lefray, Thomas Ropars, and André Schiper. Replication for send-deterministic mpi hpc applications. In *Proceedings of the 3rd Workshop on Fault-tolerance for HPC at extreme scale*, pages 33–40. ACM, 2013.

[53] Yudan Liu, Raja Nassar, Chokchai Leangsuksun, Nichamon Naksinehaboon, Mihaela Paun, and Stephen L Scott. An optimal checkpoint/restart model for a large scale high performance computing system. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–9. IEEE, 2008.

[54] Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. The linux scheduler: a decade of wasted cores. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 1. ACM, 2016.

[55] Esteban Meneses, Xiang Ni, and Laxmikant V Kalé. Design and analysis of a message logging protocol for fault tolerant multicore systems. *Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign, Tech. Rep*, pages 11–30, 2011.

[56] Wil Michiels, Jan Korst, Emile Aarts, and Jan Van Leeuwen. Performance ratios for the differencing method applied to the balanced number partitioning problem. In *Annual Symposium on Theoretical Aspects of Computer Science*, pages 583–595. Springer, 2003.

[57] Bryan Mills, Taieb Znati, and Rami Melhem. Shadow computing: An energy-aware fault tolerant computing model. In *2014 International Conference on Computing, Networking and Communications (ICNC)*, pages 73–77. IEEE, 2014.

[58] Adam Moody. Overview of the scalable checkpoint/restart (scr) library. *S&T Principal Directorate—Computation Directorate*, 2009.

[59] Adam Moody, Greg Bronevetsky, Kathryn Mohror, and Bronis R De Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, pages 1–11. IEEE, 2010.

[60] Ahuva W. Mu'alem and Dror G. Feitelson. Utilization, predictability, workloads, and user runtime estimates in scheduling the ibm sp2 with backfilling. *IEEE transactions on parallel and distributed systems*, 12(6):529–543, 2001.

[61] Nithin Nakka and Alok Choudhary. Failure data-driven selective node-level duplication to improve mttf in high performance computing systems. In *High Performance Computing Systems and Applications*, pages 304–322. Springer, 2010.

[62] DOE Office of Science. Exascale computing project update, 2017. Avaliable at `https://science.energy.gov/~/media/ascr/ascac/pdf/meetings/201712/ECP_Update_ASCAC_20171220.pdf`.

[63] Ivy Bo Peng, Stefano Markidis, and Erwin Laure. The cost of synchronizing imbalanced processes in message passing systems. In *Cluster Computing (CLUSTER), 2015 IEEE International Conference on*, pages 408–417. IEEE, 2015.

[64] Ivy Bo Peng, Stefano Markidis, Erwin Laure, Gokcen Kestor, and Roberto Gioiosa. Idle period propagation in message-passing applications. In *High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS), 2016 IEEE 18th International Conference on*, pages 937–944. IEEE, 2016.

[65] James S Plank, Kai Li, and Michael A Puening. Diskless checkpointing. *IEEE Transactions on Parallel and Distributed Systems*, 9(10):972–986, 1998.

[66] Barry Rountree, David K Lownenthal, Bronis R De Supinski, Martin Schulz, Vincent W Freeh, and Tyler Bletsch. Adagio: making dvs practical for complex hpc applications. In *Proceedings of the 23rd international conference on Supercomputing*, pages 460–469. ACM, 2009.

[67] Jon Stearley, Kurt Ferreira, David Robinson, Jim Laros, Kevin Pedretti, Dorian Arnold, Patrick Bridges, and Rolf Riesen. Does partial replication pay off? In *Dependable Systems and Networks Workshops (DSN-W), 2012 IEEE/IFIP 42nd International Conference on*, pages 1–6. IEEE, 2012.

[68] Omer Subasi, Gokcen Kestor, and Sriram Krishnamoorthy. Toward a general theory of optimal checkpoint placement. In *Cluster Computing (CLUSTER), 2017 IEEE International Conference on*, pages 464–474. IEEE, 2017.

[69] Omer Subasi, Osman Unsal, and Sriram Krishnamoorthy. Automatic risk-based selective redundancy for fault-tolerant task-parallel hpc applications. In *Proceedings of the Third International Workshop on Extreme Scale Programming Models and Middleware*, page 2. ACM, 2017.

[70] Omer Subasi, Gulay Yalcin, Ferad Zyulkyarov, Osman Unsal, and Jesus Labarta. A runtime heuristic to selectively replicate tasks for application-specific reliability targets. In *Cluster Computing (CLUSTER), 2016 IEEE International Conference on*, pages 498–505. IEEE, 2016.

[71] Omer Subasi, Gulay Yalcin, Ferad Zyulkyarov, Osman Unsal, and Jesus Labarta. Designing and modelling selective replication for fault-tolerant hpc applications. In *Cluster, Cloud and Grid Computing (CCGRID), 2017 17th IEEE/ACM International Symposium on*, pages 452–457. IEEE, 2017.

[72] Dan Tsafrir, Yoav Etsion, and Dror G Feitelson. Backfilling using system-generated predictions rather than user runtime estimates. *IEEE Transactions on Parallel and Distributed Systems*, 18(6):789–803, 2007.

[73] Wikipedia. Inequality of arithmetic and geometric means.

[74] Wikipedia. Laplace's method.

[75] Benjamin Yakir. The differencing algorithm ldm for partitioning: a proof of a conjecture of karmarkar and karp. *Mathematics of Operations Research*, 21(1):85–99, 1996.

[76] John W Young. A first order approximation to the optimum checkpoint interval. *Communications of the ACM*, 17(9):530–531, 1974.

[77] Jilian Zhang, Kyriakos Mouratidis, and HweeHwa Pang. Heuristic algorithms for balanced multi-way number partitioning. In *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*, volume 22, page 693, 2011.

[78] Fang Zheng, Hongfeng Yu, Can Hantas, Matthew Wolf, Greg Eisenhauer, Karsten Schwan, Hasan Abbasi, and Scott Klasky. Goldrush: resource efficient in situ scientific

data analytics using fine-grained interference aware execution. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2013.

[79] Gengbin Zheng, Xiang Ni, and Laxmikant V Kalé. A scalable double in-memory checkpoint and restart scheme towards exascale. In *Dependable Systems and Networks Workshops (DSN-W), 2012 IEEE/IFIP 42nd International Conference on*, pages 1–6. IEEE, 2012.

[80] Gengbin Zheng, Lixia Shi, and Laxmikant V Kalé. Ftc-charm++: an in-memory checkpoint-based fault tolerant runtime for charm++ and mpi. In *Cluster Computing, 2004 IEEE International Conference on*, pages 93–103. IEEE, 2004.

[81] Ziming Zheng and Zhiling Lan. Reliability-aware scalability models for high performance computing. In *2009 IEEE International Conference on Cluster Computing and Workshops*, pages 1–9. IEEE, 2009.

[82] Ziming Zheng, Li Yu, and Zhiling Lan. Reliability-aware speedup models for parallel applications with coordinated checkpointing/restart. *IEEE Transactions on Computers*, 64(5):1402–1415, 2015.

[83] Christopher Zimmer, Saurabh Gupta, Scott Atchley, Sudharshan S Vazhkudai, and Carl Albing. A multi-faceted approach to job placement for improved performance on extreme-scale systems. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1015–1025. IEEE, 2016.

[84] Christopher Zimmer, Don Maxwell, Stephen McNally, Scott Atchley, and Sudharshan S Vazhkudai. Gpu age-aware scheduling to improve the reliability of leadership jobs on titan. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, page 7. IEEE Press, 2018.