

# **SANDIA REPORT**

SAND2009-6753

Unlimited Release

Printed October, 2009

## **Increasing Fault Resiliency in a Message-Passing Environment**

Kurt Ferreira, Rolf Riesen, Ron Oldfield, Jon Stearley, James Laros, Kevin Pedretti,  
Todd Kordenbrock, Ron Brightwell

Prepared by

Sandia National Laboratories

Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,  
a Lockheed Martin Company, for the United States Department of Energy's  
National Nuclear Security Administration under Contract DE-AC04-94-AL85000.

Approved for public release; further dissemination unlimited.



**Sandia National Laboratories**

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

**NOTICE:** This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government, nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, make any warranty, express or implied, or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represent that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from  
U.S. Department of Energy  
Office of Scientific and Technical Information  
P.O. Box 62  
Oak Ridge, TN 37831

Telephone: (865) 576-8401  
Facsimile: (865) 576-5728  
E-Mail: [reports@adonis.osti.gov](mailto:reports@adonis.osti.gov)  
Online ordering: <http://www.osti.gov/bridge>

Available to the public from  
U.S. Department of Commerce  
National Technical Information Service  
5285 Port Royal Rd  
Springfield, VA 22161

Telephone: (800) 553-6847  
Facsimile: (703) 605-6900  
E-Mail: [orders@ntis.fedworld.gov](mailto:orders@ntis.fedworld.gov)  
Online ordering: <http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online>



# Increasing Fault Resiliency in a Message-Passing Environment

Kurt Ferreira (Org. 01423) kbferre@sandia.gov  
Rolf Riesen (Org. 01423) rolf@sandia.gov  
Ron Oldfield (Org. 01423) raoldfi@sandia.gov  
Jon Stearley (Org. 01422) jrstear@sandia.gov  
James Laros (Org. 01422) jhlaros@sandia.gov  
Kevin Pedretti (Org. 01423) ktpedre@sandia.gov  
Ron Brightwell (Org. 01423) rbbrih@sandia.gov  
Sandia National Laboratories  
P. O. Box 5800  
Albuquerque, NM 87185-1319

Todd Kordenbrock todd.kordenbrock@hp.com  
Hewlett-Packard Company

## Abstract

Petaflops systems will have tens to hundreds of thousands of compute nodes which increases the likelihood of faults. Applications use checkpoint/restart to recover from these faults, but even under ideal conditions, applications running on more than 30,000 nodes will likely spend more than half of their total run time saving checkpoints, restarting, and redoing work that was lost.

We created a library that performs redundant computations on additional nodes allocated to the application. An active node and its redundant partner form a node bundle which will only fail, and cause an application restart, when both nodes in the bundle fail. The goal of this library is to learn whether this can be done entirely at the user level, what requirements

this library places on a Reliability, Availability, and Serviceability (RAS) system, and what its impact on performance and run time is.

We find that our redundant MPI layer library imposes a relatively modest performance penalty for applications, but that it greatly reduces the number of applications interrupts. This reduction in interrupts leads to huge savings in restart and rework time. For large-scale applications the savings compensate for the performance loss and the additional nodes required for redundant computations.

# Contents

<b>1</b>	<b>Introduction</b>	<b>9</b>
<b>2</b>	<b>Design</b>	<b>11</b>
2.1	Basic operation . . . . .	12
2.2	Non-blocking receives: Preserving message order . . . . .	14
2.3	Probe, wait and test functions . . . . .	15
2.4	Other functions plus groups . . . . .	15
2.5	Any-tag receives . . . . .	16
2.6	Dependence on RAS system . . . . .	16
<b>3</b>	<b>Implementation</b>	<b>19</b>
<b>4</b>	<b>Evaluation</b>	<b>21</b>
4.1	Benchmarks . . . . .	21
4.2	Collectives . . . . .	24
4.3	Applications . . . . .	24
4.4	Evaluation summary . . . . .	31
<b>5</b>	<b>Analysis</b>	<b>35</b>
5.1	The lifetime of an application . . . . .	35
5.2	Behavior of <i>rMPI</i> . . . . .	36
5.3	Simulating an application . . . . .	39
5.4	Application behavior . . . . .	42
5.5	Validating the simulation . . . . .	46

<b>6</b>	<b>Implications and trade-offs</b>	<b>51</b>
6.1	System throughput . . . . .	51
6.2	Level of redundancy . . . . .	53
6.3	Spare nodes . . . . .	53
<b>7</b>	<b>Related work</b>	<b>55</b>
<b>8</b>	<b>Summary and future work</b>	<b>57</b>
	<b>References</b>	<b>58</b>

# List of Figures

2.1	Redundant nodes, if enabled, continue computation of active nodes that become disabled. . . . .	11
2.2	Node A transmits a message to B in the presence of redundant nodes. . . . .	13
2.3	Active and redundant messages with the same tag must maintain the same order. . . . .	13
4.1	Bandwidth comparison. Native is benchmark without the <i>r</i> MPI library. Base is with <i>r</i> MPI, but no redundant nodes. Forward is fully redundant. . . . .	22
4.2	Latency comparison. . . . .	23
4.3	Latency using <code>MPI_ANY_SOURCE</code> . . . . .	23
4.4	Broadcast performance on 2,048 nodes. . . . .	24
4.5	Reduce performance on 2,048 nodes. . . . .	25
4.6	Allreduce performance on 2,048 nodes. . . . .	25
4.7	Alltoall performance on 512 nodes. . . . .	26
4.8	Barrier performance. . . . .	26
4.9	Performance slowdown of redundant <code>MPI_Bcast()</code> versus native for forward, reverse and shuffle mappings . . . . .	27
4.10	Performance slowdown of redundant <code>MPI_Reduce()</code> versus native for forward, reverse and shuffle mappings . . . . .	28
4.11	Performance slowdown of redundant <code>MPI_Allreduce()</code> versus native for forward, reverse and shuffle mappings . . . . .	29
4.12	CTH performance. . . . .	30
4.13	SAGE performance. . . . .	31
4.14	LAMMPS performance. . . . .	32
4.15	HPCCG performance. . . . .	32

5.1	Lifetime of an application. . . . .	35
5.2	Simulating the impact of faults on the number of application interrupts. . . . .	37
5.3	Ratio of application interrupts to node faults. . . . .	38
5.4	Number of interrupts seen by an application using various levels of redundancy. . . . .	40
5.5	State diagram of application simulator. . . . .	40
5.6	Block diagram of the application simulator. . . . .	41
5.7	An application that completes 168 hours of work on a system with a five-year node MTBF. . . . .	43
5.8	Percentage of time spent in each phase of a long-running application. This graph is for 700-hours of work with a node MTBF of five years. . . . .	44
5.9	An application that completes 5,000 hours of work on a system with a one-year node MTBF. . . . .	44
5.10	An application that completes 168 hours of work on a system with a five-year node MTBF. . . . .	45
5.11	An application that completes 5,000 hours of work on a system with a one-year node MTBF. . . . .	45
5.12	An application that completes 168 hours of work on a system with a five-year node MTBF. No and full redundancy shown. . . . .	46
5.13	An application that completes 5,000 hours of work on a system with a one-year node MTBF and full redundancy. . . . .	47
5.14	Comparing Equation ?? with our application simulator. . . . .	49
6.1	Levels of redundancy versus number of interrupts. . . . .	53

# List of Tables

5.1	Number of interrupts seen by an application with various levels of redundancy.	39
5.2	Number of interrupts seen by a 5,000-hour application and a one-year MTBF.	47
6.1	Comparing total execution times. . . . .	52



# Chapter 1

## Introduction

Today's large-scale machines experience outages from failed components, software bugs, and power disruptions. A common method to allow an application to compute longer than the interval between faults is to checkpoint the application state at regular intervals and restart the application from the most recent successful checkpoint after a fault occurs. Checkpoint/restart works but is predicted to be inefficient in future machines [?, ?, ?].

Million-core machines for petascale computing will have so many parts that faults will be frequent. The system-wide Mean Time Between Failures (MTBF) will become so small that more than 50% of an application's total execution time will be spent writing checkpoints and recovering from failures [?]. The more failures occur during the execution time of an application, the longer it will take to finish its work. At large node counts the application spends more time writing checkpoints, restarting, and redoing work than the actual work. This decreases the throughput of the machine: fewer applications finish in a unit of time.

We present a method to increase resilience through redundant computation. This approach effectively increases the time between faults which results in fewer restarts, and less rework. All of these lead to better system throughput. Each redundant node is coupled to an active node and continues the computation should the active node fail, and vice versa. Since the application can continue to work in the presence of some faults, it is now possible to increase the checkpoint interval and allow the application to make uninterrupted progress for a longer slice of time. The cost is a small performance degradation and the overhead of using more nodes than the application and problem would need otherwise.

Redundant computing has been employed in real-time and high-reliability systems for several decades [?]. With this project we wanted to answer a few specific questions. Is it feasible to create a replicating infrastructure for a message-passing environment at user level? If so, what are the exact requirements on the system software and the Reliability, Availability, and Serviceability (RAS) system? The latter is of interest because, in light of the looming large-machine-fault crisis, several research groups and manufacturers are reevaluating and redesigning RAS systems for future machines.

Another critical issue is overhead of redundant computing in a message-passing environment. Additional messages are needed to enable redundant computing and that negatively impacts performance. To avoid sharing a single point of failure between an active and a redundant node, such as a fan or power-supply in the chassis, we would like the two nodes

to be physically as far away as possible from each other. This introduces additional delays and network congestion that slows down synchronizations and application performance.

This paper makes four contributions:

1. We show how a user-level library can be designed and implemented that allows MPI applications to use redundant nodes for computation (Sections 2 and 3).
2. We evaluate the message-passing overhead introduced by our library and find that it is not significant for most applications (Chapter 4).
3. We created a tool that simulates an application's work, checkpoint, restart, and rework cycles and allows the modeling of various combinations of node count, MTBF, and work to be performed (Chapter 5).
4. The results of these simulations let us determine when it is beneficial to use twice the number of nodes to run in redundant mode (Chapter 6).

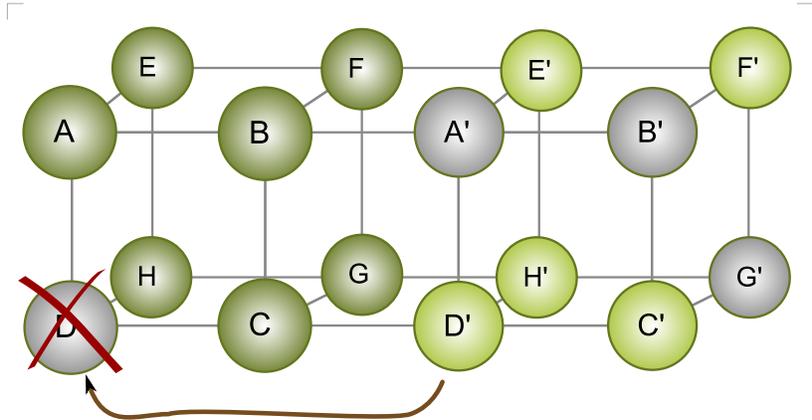
The paper closes with a related work section (Chapter 7), and a summary and future work (Chapter 8).

# Chapter 2

## Design

The basic idea for the *r*MPI library is simple: mirror each active node in an application and let the redundant nodes continue when an active node fails. This does not completely eliminate application interruptions since both nodes in a bundle could fail. However, the application now requires fewer restarts and therefore finishes more quickly. As is common with checkpoint/restart, we assume that the application will restart on the same number of nodes. That means spare nodes must be available or nodes must be repaired before a restart.

Figure 2.1 shows an example. An eight-node application is started on up to sixteen nodes. Eight nodes are designated active nodes (dark green on the left in the figure) and carry out the original computation. Additional nodes allocated are redundant nodes. Each redundant node is paired with one specific active node and carries out the same computation as the active node. Not every active node needs to have a redundant node assigned to it. Some algorithms, such as work-stealing, may survive the failure of a worker node. In this case it makes sense to replicate only the master coordinator node. For most applications where any node failure terminates the application, all *n* nodes should be replicated.



**Figure 2.1.** Redundant nodes, if enabled, continue computation of active nodes that become disabled.

When a node fails, it is removed from the mapping and no further messages are sent to it, or posted for it. If this happens during an operation in progress; e.g., while *r*MPI is waiting

for a message from a failed node, *r*MPI cancels the in-progress operation and updates its internal state so it will no longer look for messages from that node.

Nodes A', B', and G' in Figure 2.1 represent redundant nodes that did not get allocated at start time, or failed during the run of the application. If an active node fails and it has a redundant node allocated to it, the redundant node will take over the role of the active node. Redundant node D' in Figure 2.1 continues the work of active node D after it has failed.

Because of the message-passing guarantees MPI makes, an active node and its redundant node cannot be completely symmetrical. They need to coordinate. For example, the order in which unexpected messages arrive at C must match the order in which they arrive at C'. Since some computations are message-order dependent, different arrival orders would lead to different computations on the two nodes and therefore an inconsistent state.

The *r*MPI library is implemented at the profiling layer of an MPI implementation. The design of *r*MPI is agnostic of the underlying MPI library and requires only a standard conforming profiling layer.

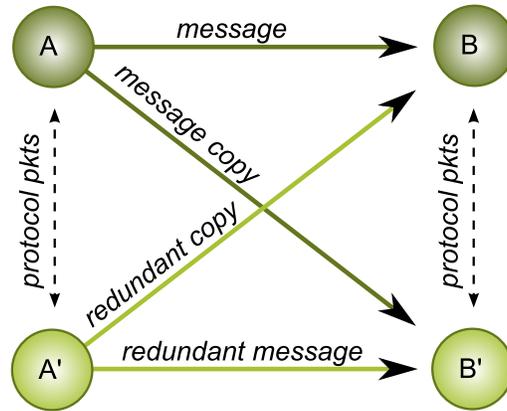
## 2.1 Basic operation

*r*MPI must interact with the RAS system and check the status of pending operations. Therefore, *r*MPI uses non-blocking operations to transfers data. We begin the description of the *r*MPI design with MPI's blocking send and receive operations.

The sending side is simple: the active and the redundant node perform the non-blocking version of the send to the destination node. Node A sends to B, and redundant node A' sends to redundant node B'. These sends occur after consultation with a RAS system maintained table to make sure the respective destinations are available and have not faulted yet. Then these sends are repeated to the alternate destination: A to B' and A' to B. The process is illustrated in Figure 2.2.

When performing a blocking receive the sequence of events depends on whether the request is for a specific source or `MPI_ANY_SOURCE`. If the application has specified a specific source, then both the active (B) and redundant (B') nodes post up to two non-blocking receives. The exact number depends on whether senders, A and A', exist. Again, *r*MPI uses a RAS system maintained table to determine the status of nodes.

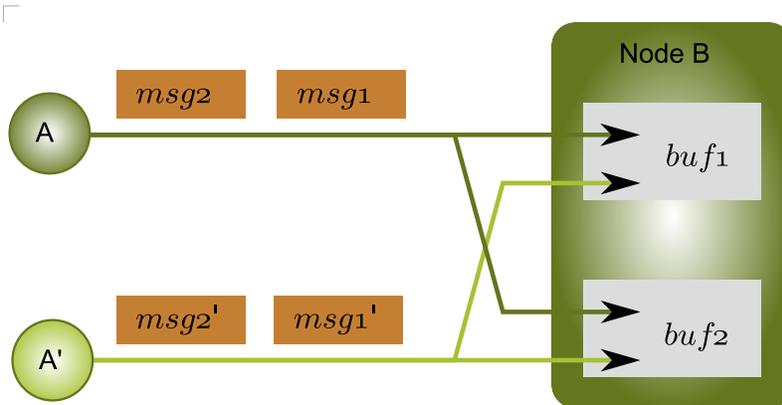
The two non-blocking receives posted on each node, one for each node in the source bundle, differ in their tag bits used. *r*MPI uses one high order bit in the tag to distinguish messages from active and redundant nodes. *r*MPI receives both the active and the redundant message into the buffer provided by the user. Since the data in the two arriving messages is identical, no danger of corrupting the buffer exists. If multiple messages with the same tag arrive, *r*MPI must make sure that the first active and first redundant arrive in the first buffer, and the second active and second redundant in the second buffer. *r*MPI achieves this



**Figure 2.2.** Node A transmits a message to B in the presence of redundant nodes.

by setting an unused tag bit in all outgoing redundant messages and setting the same bit for all receives of redundant messages.

The situation is illustrated in Figure 2.3. Node A sends messages *msg1* and *msg2* with the same tag to node B. MPI message ordering semantics demand that *msg1* arrives in *buf1* and *msg2* arrives in *buf2*. If the redundant messages *msg1'* and *msg2'* had the same tags as the active messages, then it would be possible for *msg1* and *msg2* to both arrive in *buf1* or *buf2*, since *r*MPI posts two receives for each buffer. Using an unused tag bit to mark redundant messages avoids the possible mix-up.



**Figure 2.3.** Active and redundant messages with the same tag must maintain the same order.

When the application uses `MPI_ANY_SOURCE` to receive messages, the situation gets more complicated. Messages *msg1* and *msg2*, if they come from different nodes, can each end up in

*buf1* or *buf2*. Whatever that order, it must be preserved on the redundant node. To ensure this order, *rMPI* performs the following steps: On the active receive node B, only one receive with tag `MPI_ANY_SOURCE` is posted. When a message arrives for *buf1*, node B sends the MPI envelope information to node B' (if it exists). Node B' uses the envelope information to post a specific receive with the extra tag bit set to receive the redundant message from the node that sent the first message to node B.

Node B in the meantime posts a fully specified receive for the redundant message. Node B' does the same to receive the redundant message from the sender's redundant partner. Depending on which redundant nodes are currently active, for the two messages from different senders to node B, up to four messages arrive at B. Node B', if it exists, ensures that it receives the same four messages in the same order as B. In addition, there are short protocol messages between B and B' to coordinate the receives.

When the receive of a message and the redundant message is complete, a blocking receive returns to the application. The status information about the receive on node B and B' must be updated such that both nodes report the same message source and tag, without the extra bit set, to the user.

In general, *rMPI* must carefully keep track of node rank information and always let redundant nodes return to the user the rank of their active partner. For example, `MPI_Comm_rank()` must return the same value on an active node and its redundant partner. Message destinations and sources must be treated the same way.

Other point-to-point transport functions are implemented using the basic operations described in this section. For non-blocking send operations, *rMPI* issues non-blocking sends to the active and redundant destination nodes and completes them during wait and test operations. Non-blocking receive operations are more complicated and described in the next section.

## 2.2 Non-blocking receives: Preserving message order

We mentioned that preserving message order is important and that `MPI_ANY_SOURCE` is a problem. It is an even bigger problem for non-blocking receives. As soon as *rMPI* posts the first `MPI_ANY_SOURCE` receive it must wait to post the corresponding receive for the redundant message until it has received the original message which will provide enough information to filter incoming messages for the relevant redundant message.

Since the user requested a non-blocking receive, both nodes must return at this point. The redundant node cannot post the receive yet, since it does not have the envelope information of the original message to receive its messages. That means the redundant node must maintain a queue of receives the user has posted but that *rMPI* has not been able to submit to the underlying MPI library.

During test and wait operations, the active node may complete receives and send the envelope information to the redundant node. The redundant node matches these envelopes with the receives in its queue and posts the corresponding ones to complete those operations.

MPI guarantees message ordering between node pairs. In addition, *r*MPI needs to ensure that all message are received in the same order on an active and its redundant node. `MPI_ANY_SOURCE` makes this especially difficult and introduces additional overhead. When the queue of posted receives on the redundant node is empty, and while no further `MPI_ANY_SOURCE` receives are posted, new receive requests can be submitted to the MPI library right away.

*r*MPI uses its own request handles to return to the user because many receives will not have been submitted to the MPI library at the time *r*MPI needs to return a request handles to the user. This means *r*MPI needs to maintain data structures that map its request handles to the ones used by the underlying MPI implementation.

## 2.3 Probe, wait and test functions

Redundant nodes must return the same information as their active nodes for probe, wait, and test functions. Since receives on a redundant node may not be posted with the MPI library yet, the implementation of these functions requires coordination between the active and the redundant node.

An active node must test for both the original message and the redundant message before it can report positively to a user request. It then sends that information to the redundant node which waits for its message with a specific tag and source. Wait operations are implemented by looping over the corresponding test operations.

## 2.4 Other functions plus groups

Some applications and benchmarks make decisions based on elapsed time. Therefore, `MPI_Wtime()` needs to return the same value on active and redundant nodes. The active node sends its `MPI_Wtime()` value to the redundant node. Collective operations in *r*MPI call the point-to-point operations internal to *r*MPI.

*r*MPI also needs to implement its own groups. Because *r*MPI re-maps ranks between the user level and the underlying MPI implementation, *r*MPI needs to carefully track which nodes and redundant nodes belong to which groups. This is necessary so that message transfer functions and function calls like `MPI_Group_rank()` work properly.

The same is true for communicators and functions like `MPI_Comm_dup()`. Implementing collectives, request handles, groups, and communicators inside *r*MPI reduces the underlying

MPI implementation to a simple transport mechanism and increases the complexity of *r*MPI greatly.

## 2.5 Any-tag receives

We already describes the difficulty in handling `MPI_ANY_SOURCE` receives. It requires that redundant nodes wait until the active node has completed a receive and can tell the redundant node to post a receive for that specific source and tag. It also requires posted receive queues on redundant nodes to keep track of receives that cannot be submitted to the MPI library yet.

Receives using `MPI_ANY_TAG` are also problematic. Redundant messages use an extra tag bit to identify them so that messages with the same tag will not mix with the corresponding redundant messages. Using an extra tag bit assumes the underlying MPI implementation provides a larger tag space than the  $2^{15}$  range mandated by the MPI standard. Most MPI implementations provide a much larger tag space and *r*MPI can easily reserve one of those bits.

The problem with this approach is that of course we cannot post a receive with a tag of `MPI_ANY_TAG` and set the bit, which *r*MPI needs, at the same time. A similar scheme is needed where redundant nodes wait to post `MPI_ANY_TAG` receives until such messages have arrived at the active node. Due to this complexity, *r*MPI currently does not support receives that use `MPI_ANY_TAG` and `MPI_ANY_SOURCE` simultaneously.

## 2.6 Dependence on RAS system

*r*MPI's requirements of the RAS system are modest. We expect that there is a method to learn whether a given node is available or has failed. This could be a table which *r*MPI consults and the RAS system updates when a node's status changes. Or an event mechanism that informs *r*MPI whenever the RAS system detects a failed node.

*r*MPI also requires that messages to failed nodes will be consumed and do not deadlock the network or cause other resources, such as status in the underlying MPI implementation, to be consumed. Furthermore, failing nodes must not corrupt state on other nodes. I.e., corrupted or truncated messages in flight must be discarded. Most systems already do this using CRC or other mechanisms to detect corrupt messages. The RAS system is responsible that the machine stops the retry of messages from and to failed nodes.

In order not to increase buffer space requirements and limit memory copies, *r*MPI receives both the original and the redundant message into the same buffer. We assume that two identical messages arriving in same buffer will not "collide" and that, once both messages have been received, the buffer memory will be in the same state it would have been had it

received only one or the other message. Again, we are not aware of any system today which does not fulfill this requirement.



# Chapter 3

## Implementation

We implemented the design described in Chapter 2 and list in this section some things that are specific to our current implementation. *r*MPI is implemented as a library that inserts at the MPI profiling layer between the the application and the MPI library. It is activated during `MPI_Init()` at which time it partitions `MPI_COMM_WORLD` into a set of active and redundant nodes. We performed this work on a Cray XT3 Red Storm system which uses a version of MPICH [?, ?] for message transport. Although the design described in the previous section does not depend on a specific version of MPI, our current implementation of *r*MPI does. To accelerate prototyping we used several functions from MPICH, such as the collectives, and adapted them to work inside *r*MPI. While doing this we left several low-level, MPICH internal, function calls in place. Examples include functions to determine the size and extent of data-types, figuring out whether a user-provided reduction function was declared non-commutative, checking for thread-safety, and dealing with heterogeneous systems (data type alignment and padding). That means *r*MPI will currently only work running on top of the specific MPICH version we used.

Since few machines actually provide a RAS system that gives us the minimal set of functions we listed in Section 2.6, we had to improvise. *r*MPI maintains a table of all nodes in the application and whether they have failed. We use signals to alert *r*MPI of failed nodes and can thus simulate the failure of nodes for testing purposes. However, since all nodes still are part of a complete MPI application and due to the way MPICH interacts with the XT3, simulated failed nodes cannot simply exit. They enter `MPI_Finalize()` and wait for all other nodes to finish. This also means that if we failed a node during an *r*MPI operation that involves several MPI messages, we could get MPICH into an inconsistent state. Proper integration of *r*MPI, a RAS system, and MPI would solve this problem.

When users start an application linked with *r*MPI they selects how many redundant nodes to allocate and how to map them to the active nodes. An environment variable specifies this mapping. The *r*MPI implementation imposes some restrictions on these mappings. The redundant nodes must always be at end of the `MPI_COMM_WORLD` rank list. Not every active node needs to be assigned a redundant partner. If nodes A, B, C, and D are active nodes, then `ABCD|A'B'C'D'`, `ABCD|A'B'`, `ABCD|D'C'B'A'`, and `ABCD|D'C'` are some of the many valid mappings. At most one redundant node can be assigned to an active node.



# Chapter 4

## Evaluation

From the discussion in the previous sections it should be clear that using *r*MPI will add overhead and lengthen the execution time of an application. To measure this overhead we ran multiple tests with benchmarks and applications on a Cray XT3 Red Storm systems at Sandia National Laboratories. Some of these systems have two or four CPUs per node that share a single Seastar NIC. To make sure active and redundant nodes were on separate nodes, and to avoid memory and bandwidth bottlenecks on the nodes themselves, we only used one CPU on each node.

Redundant nodes should be physically as far away from their active node as possible. The goal is to share as few hardware resources between these nodes as possible. Co-locating an active and its redundant node on two cores of the same CPU makes sense from a performance perspective, but not for reliability. Ideally, no power-supplies, fans, communication channels to other nodes, boards, or chips are shared. However, that is difficult to achieve in today's machines. Furthermore, it is often impossible or difficult to assign MPI ranks to specific nodes in the system.

Because of this and because of the impact a given allocation may have on the performance of an application, we ran our tests in three different modes. The first mode, called *forward*, assigns rank  $n/2$  as a redundant node to rank 0, rank  $n/2+1$  to rank 1, and so on resulting in a mapping like this: ABCD|A'B'C'D'. *Reverse* mode is ABCD|D'C'B'A', and *shuffle* mode (Fisher/Yates) is a random order such as ABCD|C'B'D'A'.

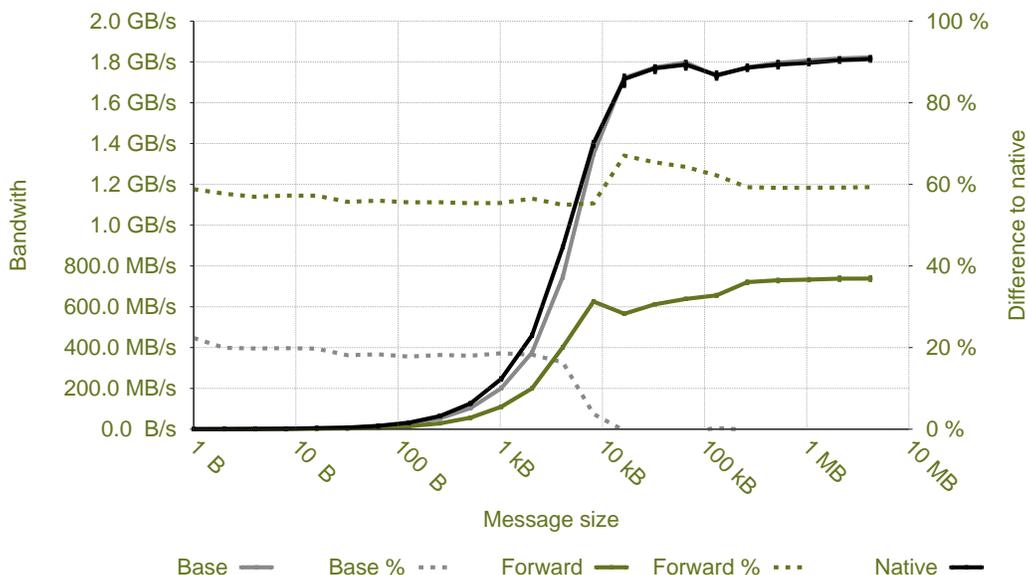
We expect that the *r*MPI library adds some overhead, even if no redundant nodes are used, due to the checks whether there are redundant nodes available and the way we implemented the collective operations. We compare this *baseline* overhead to the *native* performance when the *r*MPI library is not linked in at all. We then run in a fully redundant configuration using the forward, reverse, and shuffle mappings.

### 4.1 Benchmarks

The micro-benchmarks we expect to see most impacted by the overhead of *r*MPI are bandwidth and latency. Bandwidth because we send twice as much data, or four times as much data when traffic among the additional redundant nodes is also counted. Latency is affected

because of the logic overhead inside *r*MPI and the additional messages.

Our bandwidth experiment is shown in Figure 4.1. The first observation is that baseline, when no redundant messages are sent, does not lower bandwidth appreciably compared to native operation. When redundant messages are sent, the bandwidth measured by the benchmark drops by about 60%. This halving of bandwidth is expected since we are sending twice as much data through a given NIC. Since bandwidth and latency tests are between two nodes, changing the mappings does not have much effect outside the error-bars of each run.



**Figure 4.1.** Bandwidth comparison. Native is benchmark without the *r*MPI library. Base is with *r*MPI, but no redundant nodes. Forward is fully redundant.

Figure 4.2 shows the results of our latency tests. Again, baseline shows some *r*MPI library overhead for smaller messages but it becomes negligible as message size increases. When we add redundant nodes, the impact is much more significant. Just as *r*MPI halves the bandwidth achievable, it also doubles the latency since, basically, we send two messages for each one the application sends. (Two more messages are injected into the network by the redundant node.) In addition, there is overhead for the coordination protocol.

This coordination overhead between active and redundant nodes becomes more severe when `MPI_ANY_SOURCE` is used. Remember from the discussion in Chapter 2, `MPI_ANY_SOURCE` causes redundant nodes to delay positing of receives until the active node has received its message and informed the redundant node. In Figure 4.3 we see the result of this. Latency increases by a factor of 1.5.

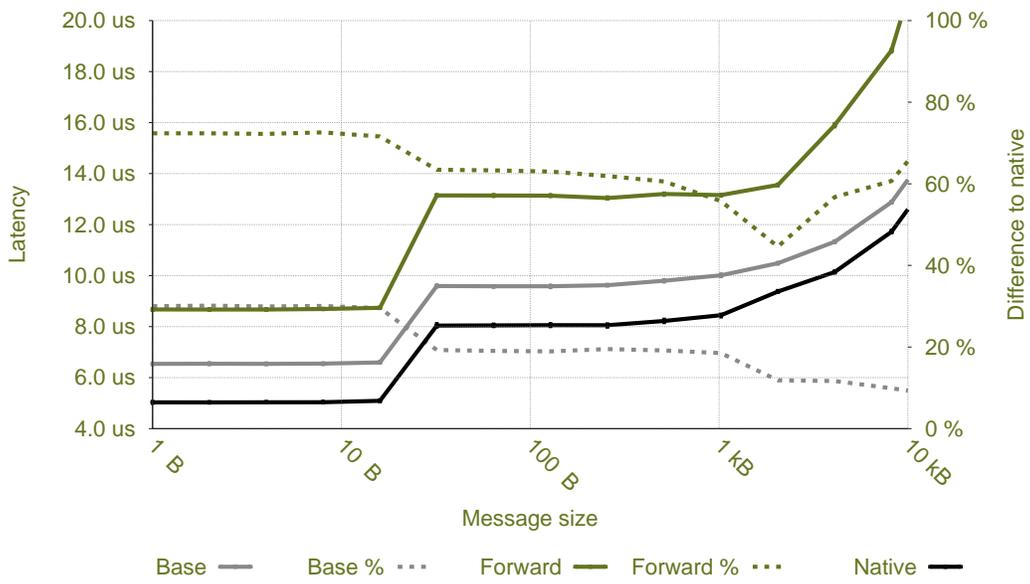


Figure 4.2. Latency comparison.

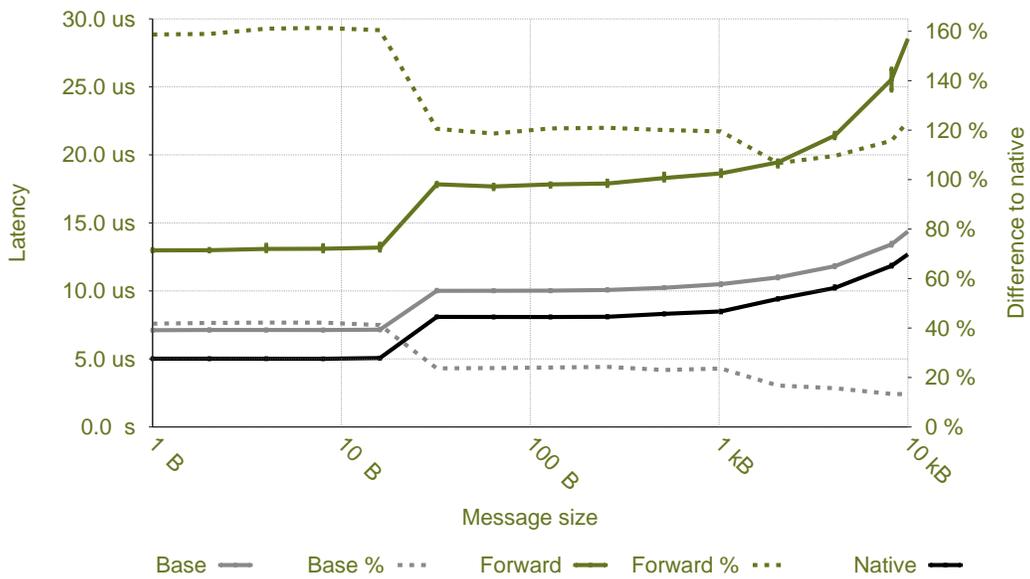


Figure 4.3. Latency using MPI\_ANY\_SOURCE.

## 4.2 Collectives

In addition to the point-to-point benchmarks described in Section 4.1, we also measured the performance of four collective operations benchmarks. We ran each test ten times, except native and baseline, which we ran five times each. Because of the current method of implementing collectives and the performance overhead of *r*MPI, we expect collective operations to perform poorly. Figure 4.4, Figure 4.5, Figure 4.6, and Figure 4.7, Figure 4.8 shows the performance graphs for broadcast, reduce, allreduce, alltoall, and barrier for the largest node counts we ran. Figure 4.9, Figure 4.10, Figure 4.11, and Figure ?? shows the performance of a redundant broadcast, reduce, allreduce, and all-to-all varying both the number of nodes the operation ran on as well as the size in bytes of the operation.

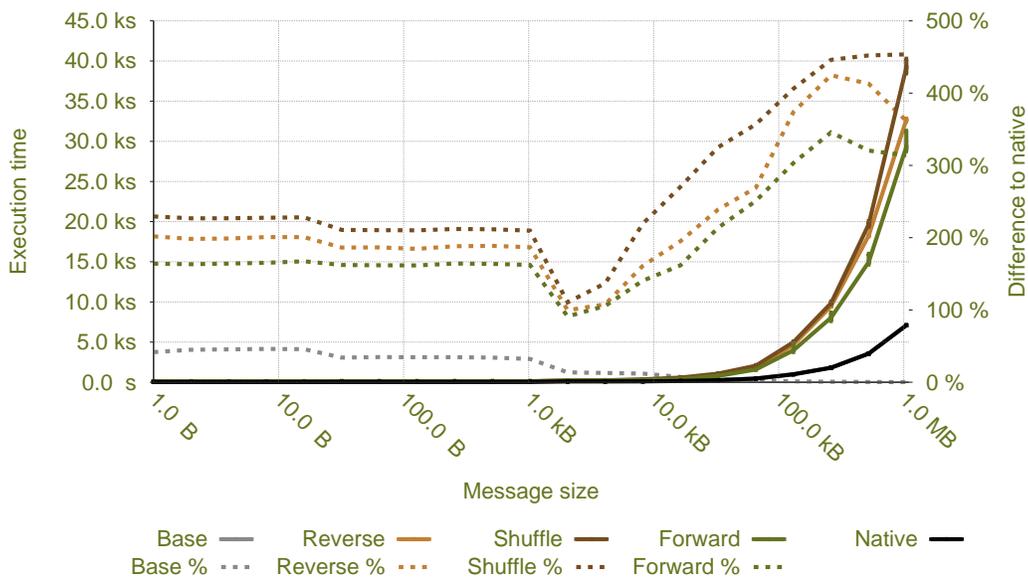


Figure 4.4. Broadcast performance on 2,048 nodes.

## 4.3 Applications

Performance loss introduced by the *r*MPI library is large for simple point-to-point benchmarks. In this section we investigate how this impacts applications. We ran tests using four different applications.

CTH [?] is a multi-material, large deformation, strong shock wave, solid mechanics code developed by Sandia National Laboratories with models for multi-phase, elastic viscoplastic, porous, and explosive materials. CTH supports three-dimensional rectangular meshes; two-dimensional rectangular, and cylindrical meshes; and one-dimensional rectilinear, cylindrical,

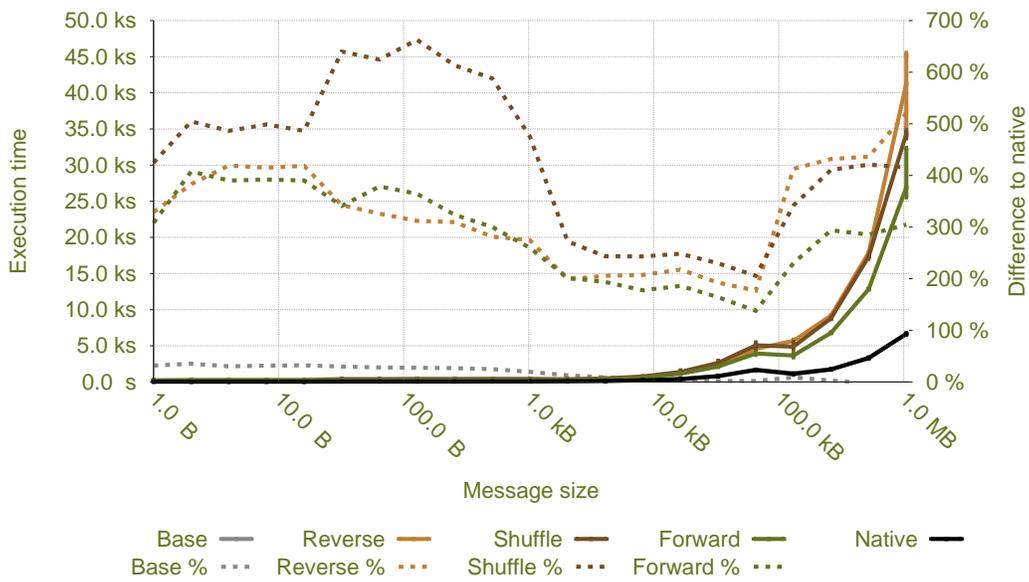


Figure 4.5. Reduce performance on 2,048 nodes.

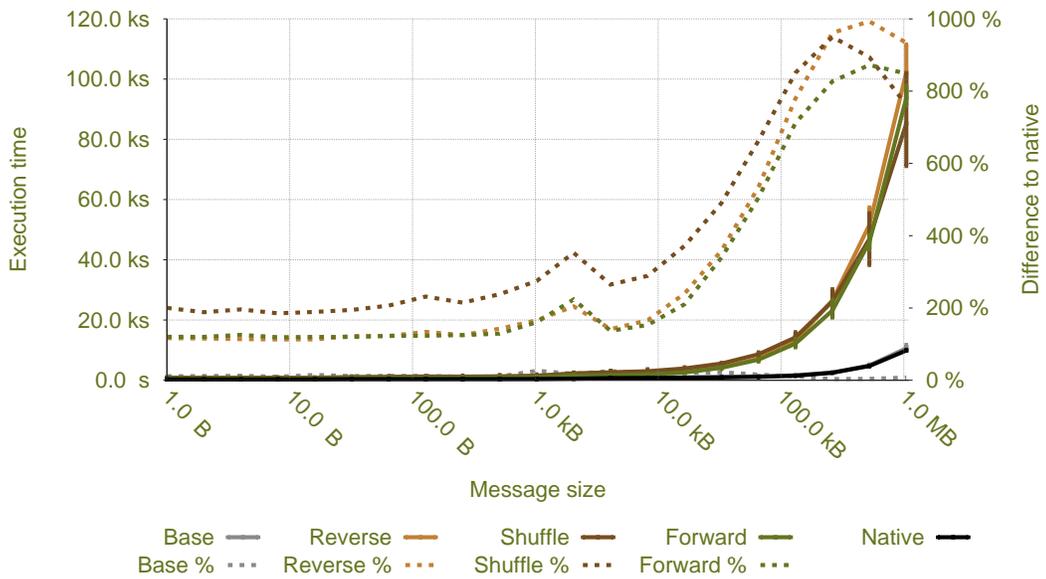


Figure 4.6. Allreduce performance on 2,048 nodes.

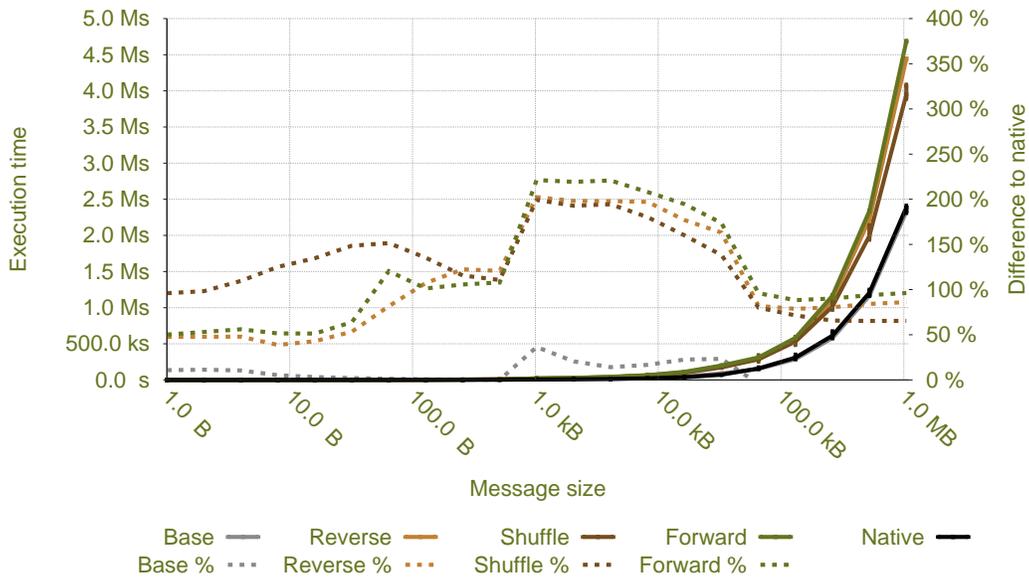


Figure 4.7. Alltoall performance on 512 nodes.

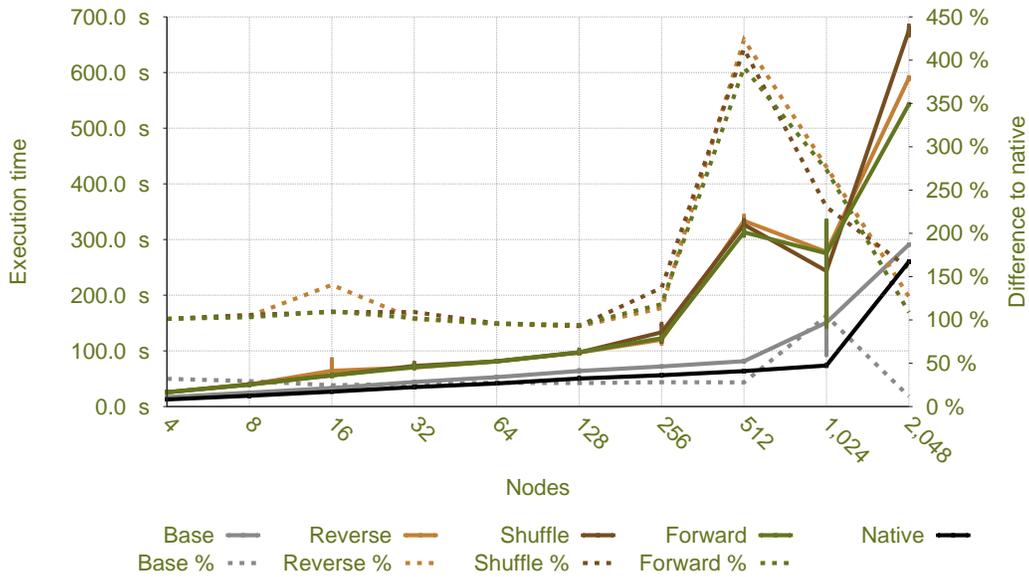
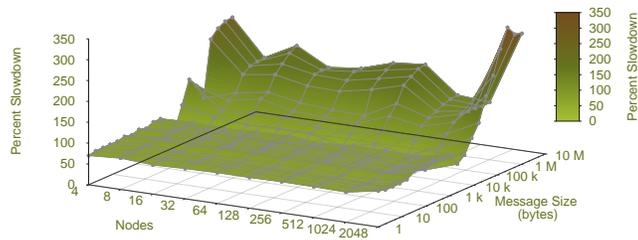
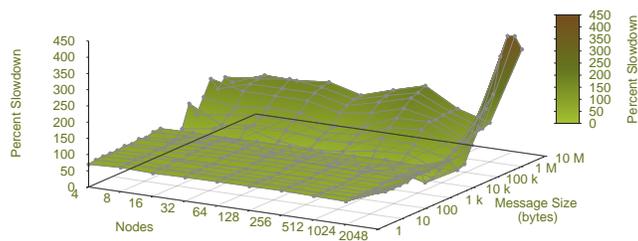


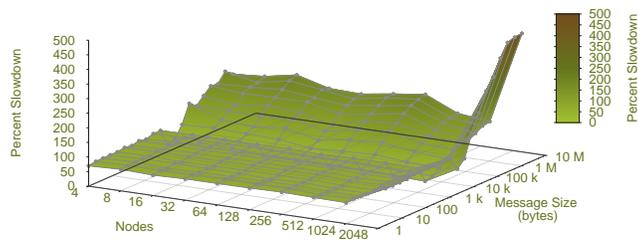
Figure 4.8. Barrier performance.



(a) Forward

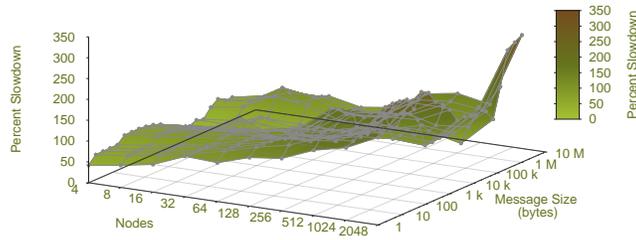


(b) Reverse

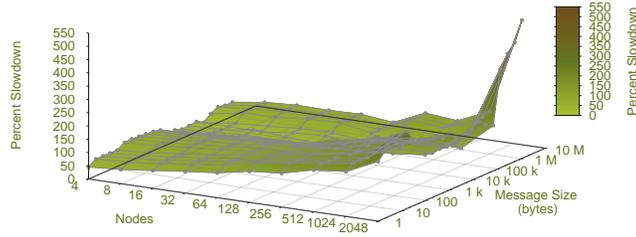


(c) Shuffle

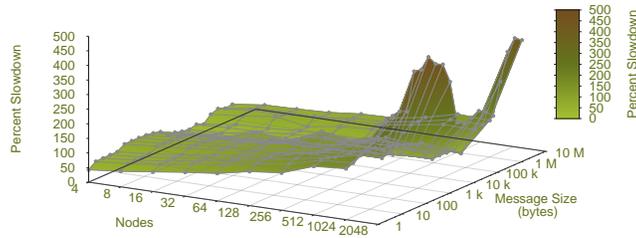
**Figure 4.9.** Performance slowdown of redundant MPI\_Bcast() versus native for forward, reverse and shuffle mappings



(a) Forward

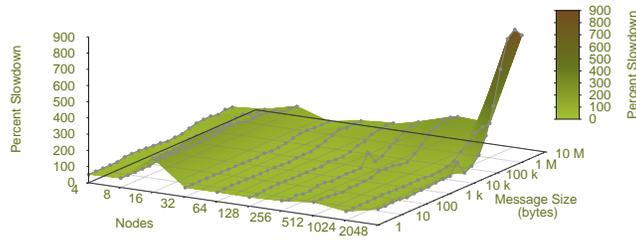


(b) Reverse

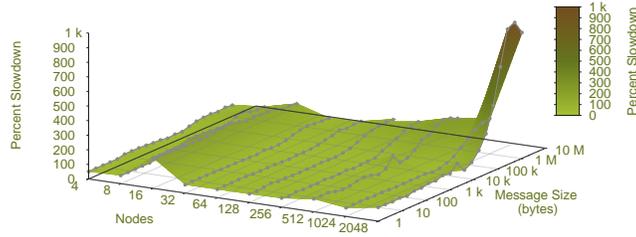


(c) Shuffle

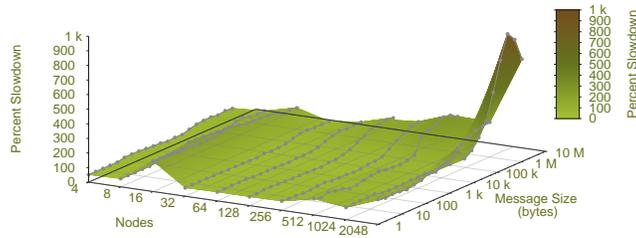
**Figure 4.10.** Performance slowdown of redundant MPI\_Reduce() versus native for forward, reverse and shuffle mappings



(a) Forward



(b) Reverse



(c) Shuffle

**Figure 4.11.** Performance slowdown of redundant MPI\_-Allreduce() versus native for forward, reverse and shuffle mappings

and spherical meshes, and uses second-order accurate numerical methods to reduce dispersion and dissipation and to produce accurate, efficient results. It is used for studying armor/anti-armor interactions, warhead design, high explosive initiation physics, and weapons safety issues.

Figure 4.12 shows the performance of CTH on node counts up to 2,048. That means CTH uses 4,096 nodes when running in fully redundant mode. As we observed with the benchmarks, simply adding the *r*MPI library without using redundant nodes does not greatly impact the performance.

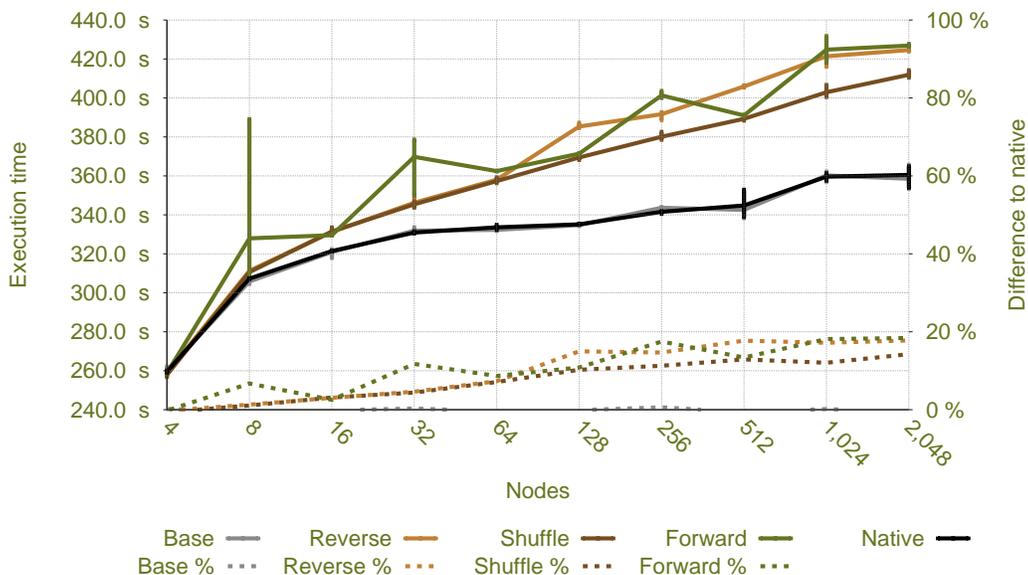
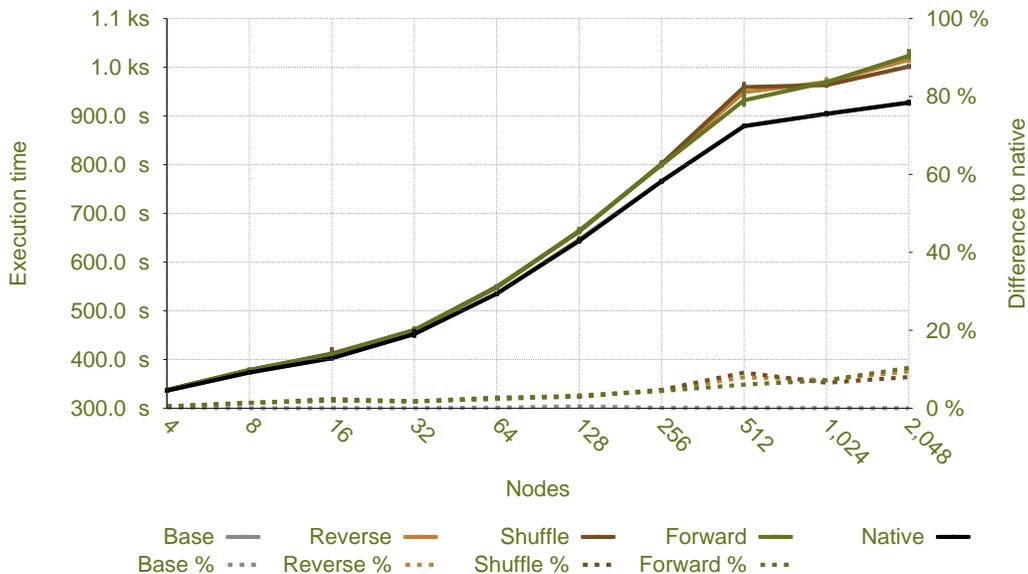


Figure 4.12. CTH performance.

Each curve in Figure 4.12 and all other graphs in this section, represents ten runs of each benchmark and application (five runs each for native and baseline). We use error-bars to show the variations between runs. The performance impact of using redundant nodes with CTH, which is communication intensive, is less than 20%. A forward mapping of redundant nodes to active nodes is not significantly different from a reverse or shuffle mapping. Although the shuffle mapping seems to produce more repeatable and slightly better results.

SAGE, SAIC’s Adaptive Grid Eulerian hydro-code, is a multi-dimensional, multi-material, Eulerian hydrodynamics code with adaptive mesh refinement that uses second-order accurate numerical techniques [?]. It represents a large class of production applications at Los Alamos National Laboratory. It is a large-scale parallel code written in Fortran 90 and uses MPI for inter-processor communications. It routinely runs on thousands of processors for months at a time. The SAGE performance is shown in Figure 4.13. When we enable full redundancy, we lose about 10% in performance on large node counts.

LAMMPS [?] is a classical molecular dynamics code developed at Sandia. For our ex-



**Figure 4.13.** SAGE performance.

periments we use the embedded atom method (EAM) metallic solid input script which is used by the Sequoia benchmark suite. The LAMMPS code and input scripts are provided on the LAMMPS web site [?]. For this experiment we ran LAMMPS in weak-scaling mode. The performance impact of *r*MPI on LAMMPS is shown in Figure 4.14. It is less than 3% independent of the number of nodes used.

The HPCCG mini-application, part of the Mantevo project [?], is a simple sparse conjugate gradient solver designed to capture an important component of Sandia’s production workload. The majority of its runtime is spent performing sparse matrix-vector multiplies, where the sparse matrix is encoded in compressed row storage format. The interprocessor communication is minimal, requiring exchange of nearest neighbor boundary information, in addition to global `MPI_Allreduce()` operations required for the scalar computations in the CG algorithm. The performance impact on HPCCG when using *r*MPI and redundant nodes is minimal. The results are shown in Figure 4.15.

## 4.4 Evaluation summary

Results on different systems will vary with the capacity and performance of the network, application and data sets used, as well as the parallelism and architecture of the individual nodes. Nevertheless, the numbers presented here are representative for a large-scale machine and should be comparable to similar configurations on other machines.

While micro-benchmarks clearly show the overhead introduced by *r*MPI, the impact on

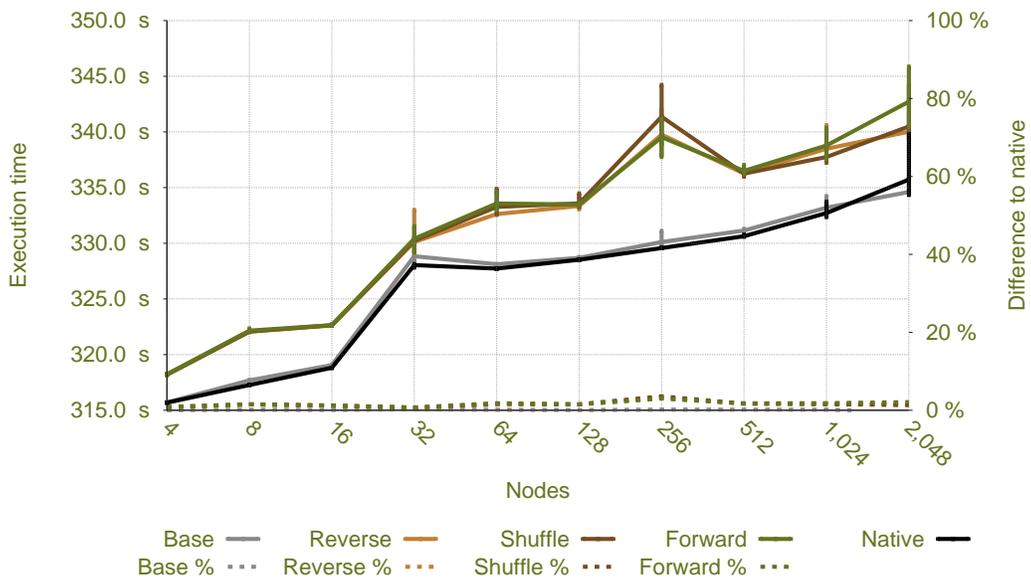


Figure 4.14. LAMMPS performance.

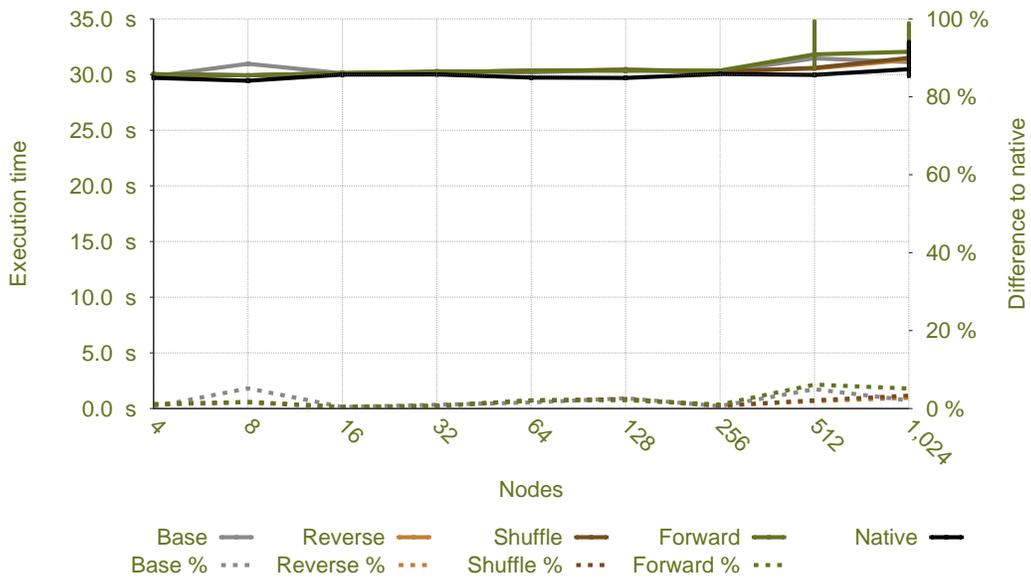


Figure 4.15. HPCCG performance.

application is much less severe. It ranges from 20% overhead for CTH, a communication intensive application, to almost no overhead for HPCCG and LAMMPS. In the next section we analyze whether it is worth it to pay the price of this performance overhead and use twice as many nodes to reduce the number of application interruptions.



# Chapter 5

## Analysis

Since *r*MPI introduces some message-passing overhead and requires up to twice the number of nodes to perform the same amount of work, it is only beneficial to use it if it can improve the throughput of a system. The throughput of a system is determined by how much work each job does, how many nodes it needs, and how much overhead it has due to interrupts, restarts, checkpoints, and rework.

### 5.1 The lifetime of an application

Figure 5.1 illustrates the phases an application goes through in order to complete a certain amount of work. After it starts running, the application does some of its work and then pauses to write a checkpoint to stable storage. Once it has written a checkpoint, the application continues with its work.

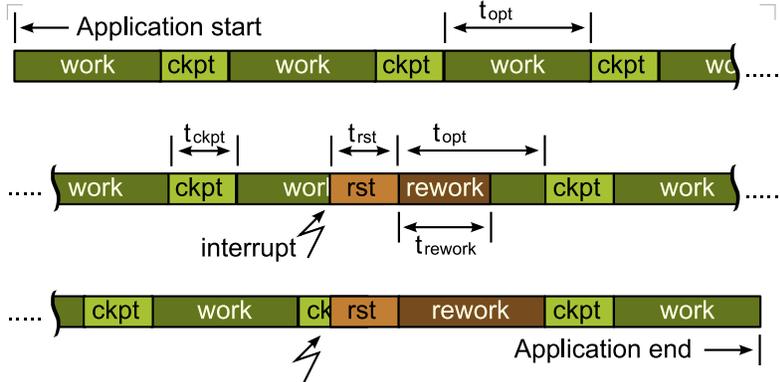


Figure 5.1. Lifetime of an application.

When an interrupt (application failure) occurs, the application restarts and has to redo the work that was lost since the last successful checkpoint. Making progress on the work load continues after that until it is time again to write another checkpoint, or a new interruption occurs.

Interrupts can occur during any phase of the application. If an interrupt occurs during a work phase, the work since the last successful checkpoint is wasted and needs to be redone after the application restarts. An interrupt during the writing of a checkpoint is in some sense the worst case. Even though a whole work phase has been completed, all of it needs to be repeated, since saving was unsuccessful.

It is very common for applications to schedule checkpoints at regular, fixed-duration, intervals. Given the knowledge of how long it takes to write a checkpoint  $\delta$  and the application’s Mean Time Between Failures (MTBF)  $\Theta$  (based on the number of nodes), it is possible to calculate an optimal checkpoint interval  $\tau_{\text{opt}}$  [?]:

$$\tilde{\tau}_{\text{opt}} = \begin{cases} \sqrt{2\delta\Theta} \left[ 1 + \frac{1}{3}\sqrt{\frac{\delta}{2\Theta}} + \frac{\delta}{18\Theta} \right] - \delta & \text{for } \delta < 2\Theta \\ \Theta & \text{for } \delta \geq 2\Theta \end{cases} \quad (5.1)$$

For all of our work, we assume that the checkpoint interval  $\tau_{\text{opt}}$  is calculated using Equation 5.1. The checkpoint interval applies to work and rework phases. If the rework phase does not consume the entire interval, then the remaining time until the next checkpoint is used to continue regular work. When all the successfully completed work phases add up to the total work time an application needs to perform, then the application will end.

From Figure 5.1 it is clear that an application, even if it gets interrupted rarely, will need more time to complete than the actual amount of work suggests. If interrupts are frequent, then a considerable amount of the total application time may be used for checkpoints, restarts, and rework.

## 5.2 Behavior of *r*MPI

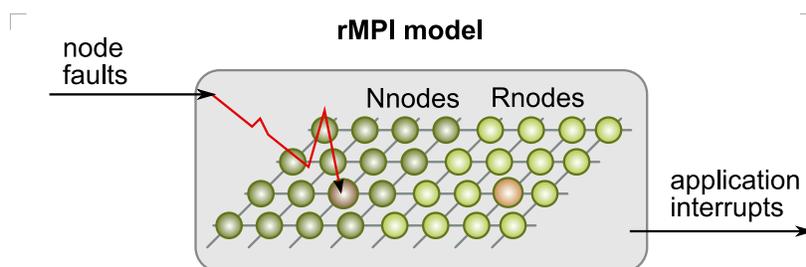
With the help of the *r*MPI library we hope to reduce the number of interrupts an application experiences and, therefore, the time required to complete the work. By reducing the number of interrupts we are improving the MTBF the application experiences. This in turn has an impact on  $\tau_{\text{opt}}$  from Equation 5.1. The resulting, longer checkpoint interval further improves the efficiency of the application.

Using *r*MPI reduces the number of application interruptions because the application can continue as long as at least one node in each bundle of nodes is still working. We created a simulation of *r*MPI to learn how many fewer application interruptions we might expect when compared to running an application without redundant. In the latter case, the application will have to restart after each node fault.

The probability that a fault causes an application interrupt when the fault affects a node without a redundant partner is 1. Using a redundant nodes decreases that probability. Since we are assuming, just as in the non-redundant case, that the application will be restarted on the same number of nodes, the calculation of the probability that a failure will cause an

application interrupt is not easy. For this reason we we wrote a model of *r*MPI to help us determine the fault to interrupt ratio.

Figure 5.2 shows the model we created. The model is configured with the number of active and redundant nodes as parameters. We then feed it with a certain number of faults. For each fault, the *r*MPI model pseudo randomly picks a node that is killed by that fault. If that node has a redundant partner that is still alive, the *r*MPI model continues killing a node for each additional fault. When a fault kills the last node in a bundle, the *r*MPI model records that occurrence and “restarts” the application; i.e., all active and redundant nodes are reset to an “alive” state. This continues until the *r*MPI model has processed all faults. The model prints the number of faults and resulting application interrupts at the end.



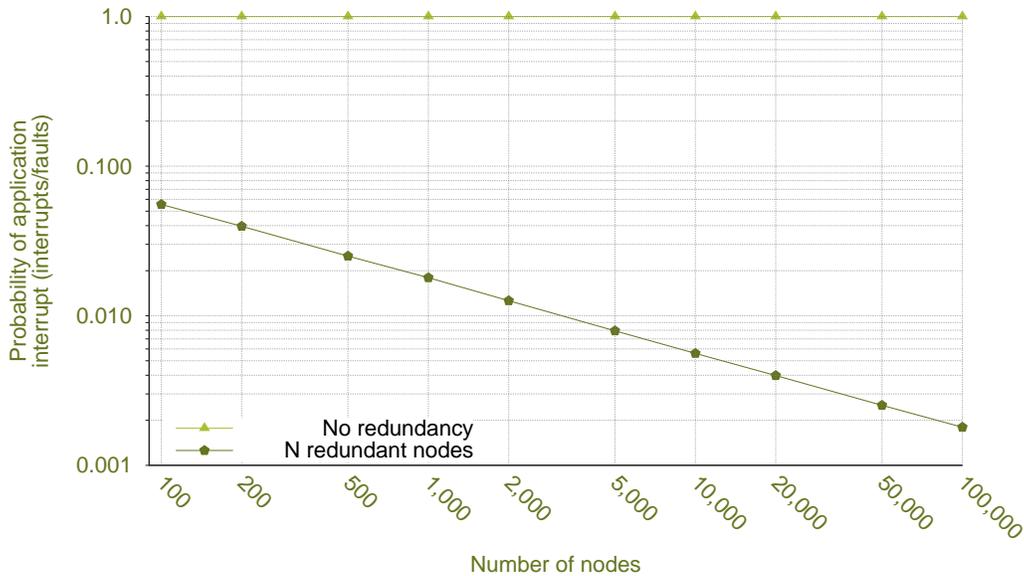
**Figure 5.2.** Simulating the impact of faults on the number of application interrupts.

For our initial evaluation of *r*MPI behavior we generate 100 faults for each node on average. When simulating a 10,000-node application, we generate  $100 * 10,000$  faults to feed into the *r*MPI model. When simulating a fully redundant application we generate twice that many faults, since application uses twice as many nodes.

If there are no redundant nodes, each fault causes an application interrupt. The ratio of interrupts to faults; i.e., the probability that a fault causes an application interrupt, will always be 1 (top line in Figure 5.3). With redundant nodes, some faults will not cause an application interrupt and the interrupt/fault ratio will be less than 1 (sloping bottom line in Figure 5.3).

When running in fully redundant mode, the probability of an application experiencing a fault decreases if it is run on a higher number of nodes. This is surprising at first, but a thought experiment can help with understanding this behavior.

The very first fault will have no effect on the application, since the redundant node that is in the same bundle as the faulted one will continue the work. When a second fault occurs, one of the remaining  $n - 1$  nodes will fail. The probability that the second fault affects the second node in the bundle where the first fault occurred is  $1/(n - 1)$ . As we increase  $n$ , it is less and less likely that the second fault falls within the same bundle and causes an application interrupt.



**Figure 5.3.** Ratio of application interrupts to node faults.

For the experiment shown in Figure 5.3 we generated 100 faults per node in the simulation. This is somewhat unrealistic, but gave us enough data points to calculate a interrupts to faults ratio.

In the next section we describe the simulation we used to to obtain the numbers in Table 5.1. For now let us explain what they mean. The *rMPI* library allows us to allocate up to  $N$  additional nodes for redundant computations. We chose to look at no redundant nodes, 1/4, 1/2, 3/4, and  $N$  redundant nodes. The latter being fully redundant and the other cases are when only some of the active nodes have a redundant partner.

The numbers in Table 5.1 are for an application that needs to complete 168 hours of work (a week). We simulated a system that has an MTBF of 43,800 hours (about five years) for each node. The table shows that as more nodes are used, the number of application interrupts also increases. The first column shows how many nodes the application sees. That number is higher when the redundant nodes are included. For a fully redundant run on 50,000 nodes, 100,000 nodes are used, all of which can fail and contribute to the overall application MTBF.

Table 5.1 shows that an application running without redundancy on 50,000 nodes, experiences 550 interrupts. The same application experiences zero interrupts when full redundancy is enabled in *rMPI* (last column of the table). Different runs of the simulation and the distribution properties of faults make it so that that number is not always zero. But, it is always very low when compared to the non-redundant case. Note the big increase in the number of interrupts between 100,000 and 500,000 nodes.

**Table 5.1.** Number of interrupts seen by an application with various levels of redundancy.

Num. nodes	Level of redundancy				
	none	1/4	1/2	3/4	full
100	2	0	0	0	0
200	3	2	1	0	0
500	3	2	2	1	0
1,000	2	2	1	1	0
2,000	9	5	3	2	0
5,000	35	27	12	7	0
10,000	53	49	24	18	1
20,000	117	84	52	29	0
50,000	550	339	185	87	0
100,000	1,658	1,103	497	177	1
200,000	7,269	3,713	1,693	531	2
500,000	135,905	47,550	13,819	2,882	7
1,000,000		992,103	152,113	17,115	13

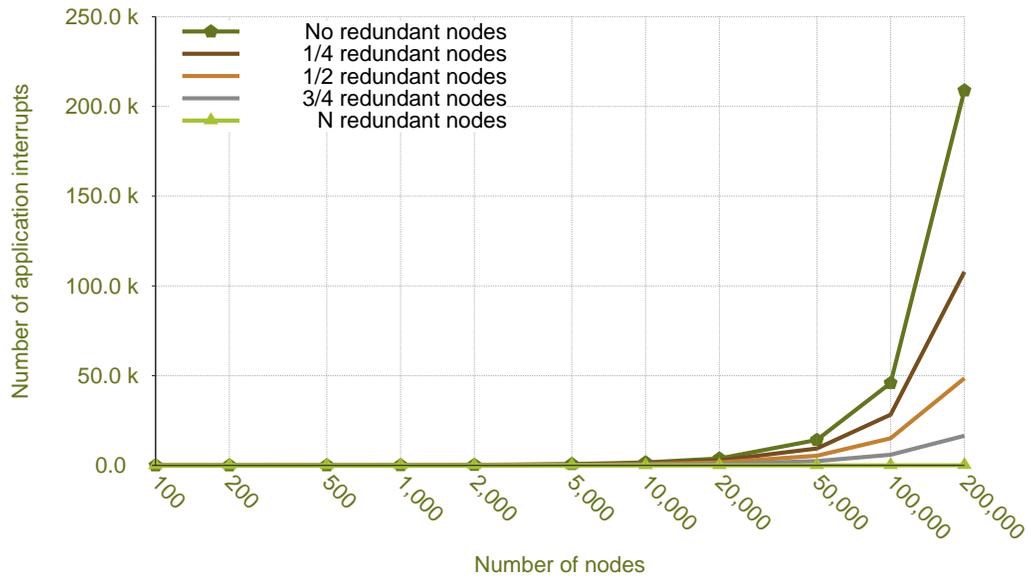
We ran a simulation of an application that requires 5,000 hours (about seven months) to complete with the same node MTBF of five years as the system shown in Table 5.1. The result is shown in Figure 5.4. Without redundancy on a 50,000-node system, the application will have to restart 14,215 times. By dedicating another 50,000 nodes to the application run, that number drops to 39.

We will analyze how such a drastic reduction in the number of interrupts an application experiences impacts its execution time after we describe our simulator in the next section.

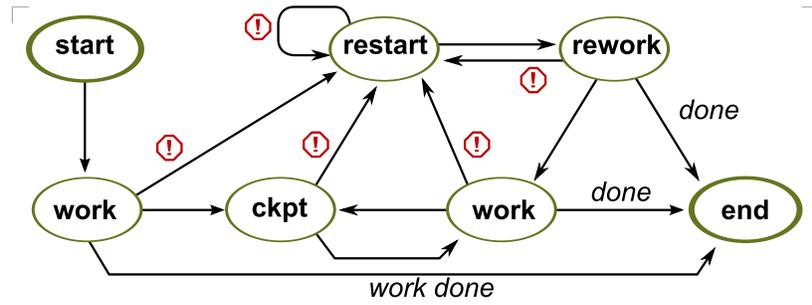
### 5.3 Simulating an application

In Section 5.1 we described the lifetime of an application. Using Figure 5.1 as a guide we created a simulator to help us understand the effect of using redundant nodes on total execution time of an application. The application simulator mimics the interaction of an application with the system it runs on. The state diagram in Figure 5.5 shows that the simulator, just like a real application, transitions from working on a problem, to check-pointing to stable storage, and back to working again.

When an interrupt occurs, either during a work or checkpoint phase, a restart from the last successful checkpoint is initiated. The work that was lost since the last checkpoint has to be redone in the rework phase. After that, the regular cycle of work and check-pointing continues.



**Figure 5.4.** Number of interrupts seen by an application using various levels of redundancy.

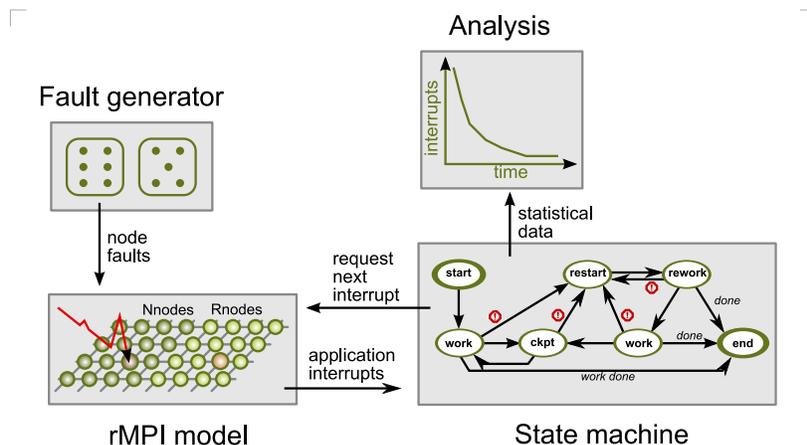


**Figure 5.5.** State diagram of application simulator.

The transitions to the checkpoint state occur whenever the checkpoint interval timer expires. That timer is reset in the checkpoint state. Our simulator uses Equation 5.1 to calculate the optimal checkpoint interval. When the amount of successfully completed work reaches the simulated workload, the simulator enters the end state and stops. At that point it outputs the amount of time spent in each state and various other statistics it accumulated during the run.

One of the parameters to control the simulator is the MTBF of the simulated system. The simulator generates random events that are exponentially distributed around the MTBF. Each event is a fault that we feed into an *r*MPI model. The model determines which node has failed and whether the application receives an interrupt or can continue doing its work. If the model determines that an application interrupt should occur, it forces a transition to the restart state in the state diagram in Figure 5.5. These transitions are indicated by the exclamation point signs in the diagram. We are assuming that for a restart, the same number of nodes will be available again. This is the same assumption that is made for applications using checkpoint/restart running without redundant nodes.

The MTBF parameter for the application simulator is the MTBF of a single node. The fault generator within the simulator generates faults for the individual nodes. Therefore, we expect the overall MTBF, the system MTBF, to be much smaller than the one for an individual node. The simulator has the option to output the times of each individual interrupt and we can calculate the mean time between these interrupts. This is shown in Figure 5.6. The state machine requests the next time an application interrupt will occur from the *r*MPI model. The fault generator generates exponentially distributed faults for each node. The *r*MPI model then determines at what time to cause an application interrupt. It will be the earliest time both nodes in a bundle have failed. The application interrupt times are fed into an analysis module which computes the system MTBF and prints various statistics about the run.



**Figure 5.6.** Block diagram of the application simulator.

When there are no redundant nodes, each node fault causes an application interrupt.

The system MTBF for  $n$  components in series (all depending on all others) is [?]

$$\Theta_{\text{sys}} = \frac{1}{\frac{1}{\Theta_1} + \frac{1}{\Theta_2} + \dots + \frac{1}{\Theta_n}} \quad (5.2)$$

If we assume that the MTBF for all the nodes is the same ( $\Theta_1 = \Theta_2 = \dots = \Theta_n$ ), then the system MTBF is

$$\Theta_{\text{sys}} = \frac{1}{\frac{1}{\Theta_1} + \frac{1}{\Theta_2} + \dots + \frac{1}{\Theta_n}} = \frac{1}{n \frac{1}{\Theta}} = \frac{\Theta}{n} \quad (5.3)$$

When calculating the mean of the times output by the application simulator for a given number of nodes without redundant partners, we get the result mandated by Equation 5.3. This indicates that our simulator which uses the node MTBF, correctly simulates the MTBF of the complete system.

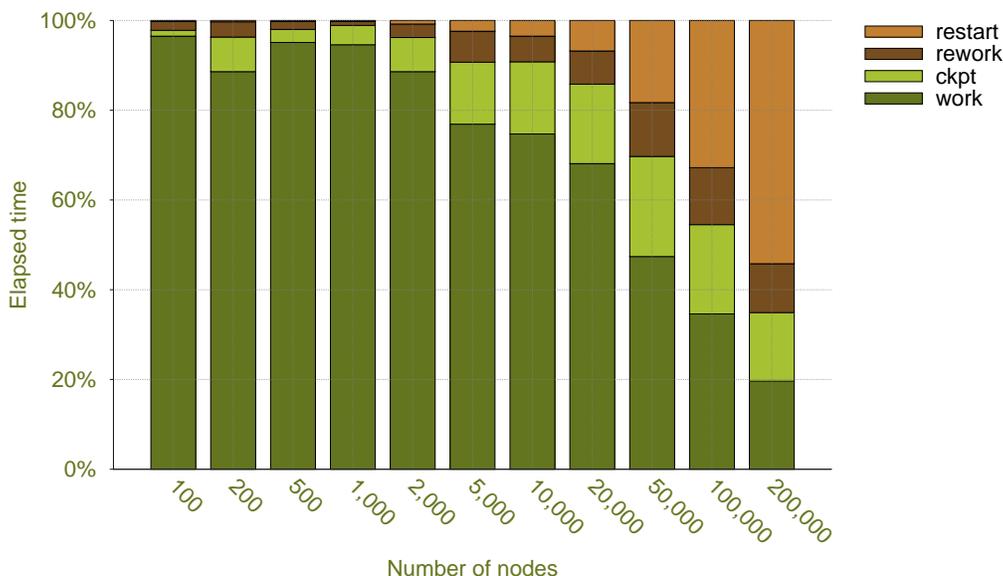
## 5.4 Application behavior

We will now use the simulator described in the previous section to investigate the behavior of applications on large number of nodes. For our first experiment we chose an application that needs to get 168 hours of work done. That amount of work is not uncommon for large-scale simulations. We assume the amount of work per node remains constant, independent of the number of nodes used. Such weak-scaling applications are common, but our assumed perfect efficiency is not. Each application exhibits its own scaling behavior. Since we are not simulating a specific application, we use the simplifying assumption of perfect scaling. For real applications with less than perfect scaling the situation described below gets even worse and the use of redundant nodes becomes even more beneficial.

For the node MTBF we chose 43,800 hours; about five years. Manufacturers often claim a higher MTBF for their products. However, [?] found an MTBF of about four years more realistic for a large high-performance-computing site. The MTBF we are considering is not purely due to hardware faults. Any interruptions that causes an application restart adds to the overhead an application experiences. Even scheduled maintenance is not handled properly by many applications and causes work to be lost. For comparisons we also include some results assuming an MTBF of one year. While probably not common, it has been used in the literature [?], and provides a lower bound on what to expect.

Figure 5.7 shows how much time, as a percentage, a 168-hour work application spends working, writing checkpoints, restarting, and redoing work that was lost. We see that, on large numbers of nodes, the amount of time spent doing the actual work drops below 50%. The dark green bars represent the amount of work plus work that was subsequently lost due

to an application interrupt. For our example that means the application performed more than 168 hours of work to complete its task. Figure 5.8 shows an example for a 700-hour, five-year MTBF application.



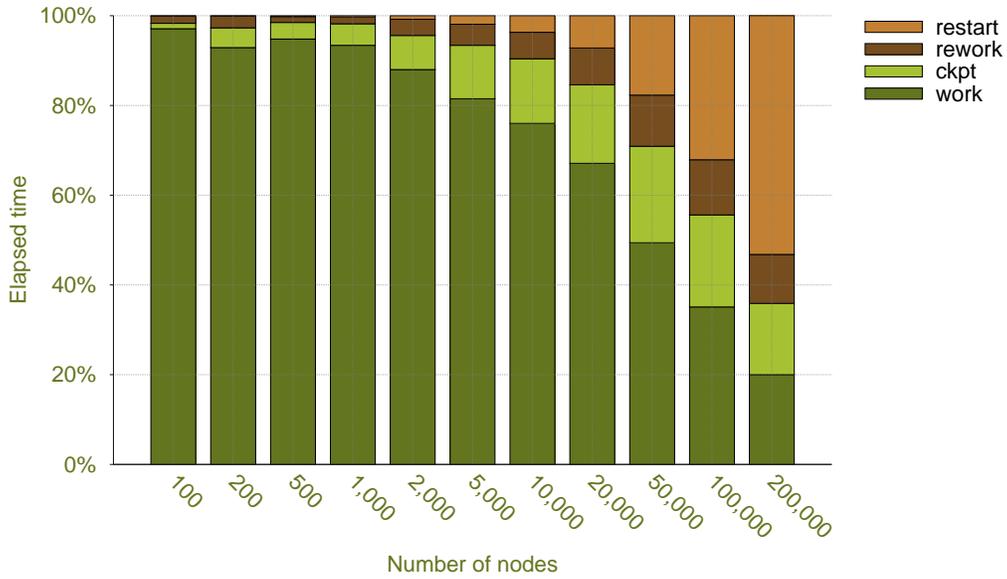
**Figure 5.7.** An application that completes 168 hours of work on a system with a five-year node MTBF.

The amount of work does not seem to influence these percentages greatly. Lowering the MTBF, however, has a large impact. A 5,000-hour (about six months) application on a system with a one-year node MTBF spends less than 20% of its total execution time making progress on 50,000 nodes. Most of the time is spent in restarts. Figure 5.9 shows this graphically.

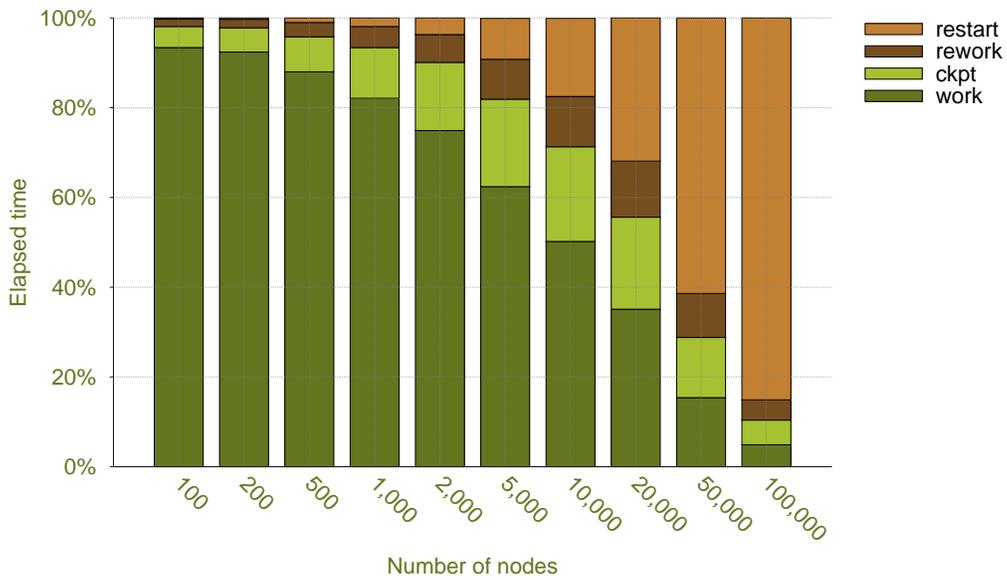
The graph in Figure 5.10 shows the 168-hour application again but using the total elapsed hours for the  $y$ -axis. It requires more than twice that time to complete its task. The situation for the 5,000-hour application with an MTBF of one year per node is even worse. On 50,000 nodes it runs twelve times longer than the amount of work it is assigned to complete (Figure 5.11).

In Section 5.2 we have seen that using redundant nodes results in a drastic reduction in application interrupts. Given that, on large number of nodes, an application runs many times as long as the amount of work it is completing due to frequent interrupts and restarts, using redundant nodes should be very beneficial for large scale applications.

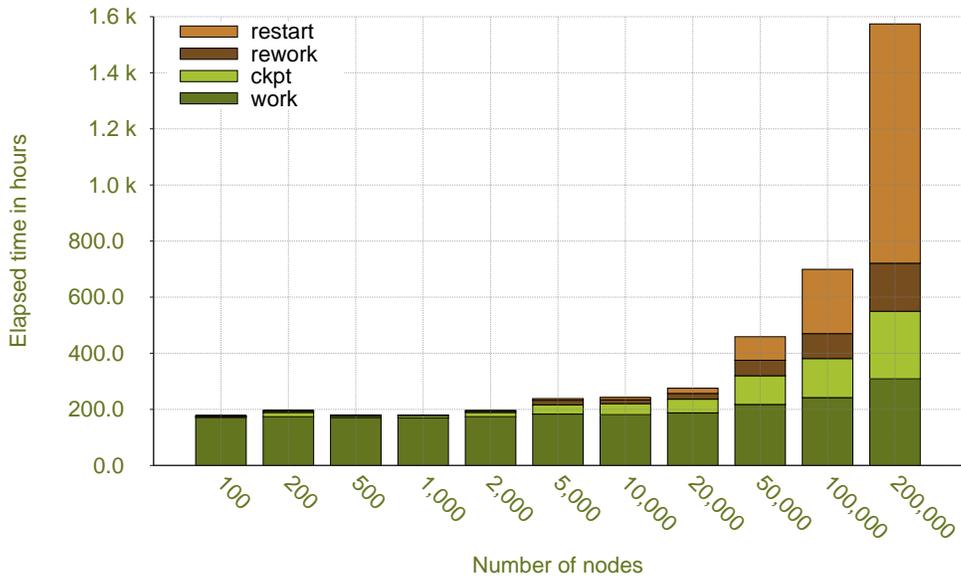
Figure 5.12 shows the drastic reduction in execution time side by side with the data from Figure 5.10. When running with redundant nodes there are very few restarts which almost eliminates restart and rework time. Note that in the data we present in this section, we are not considering the slowdown our  $r$ MPI library introduces, since it is application specific



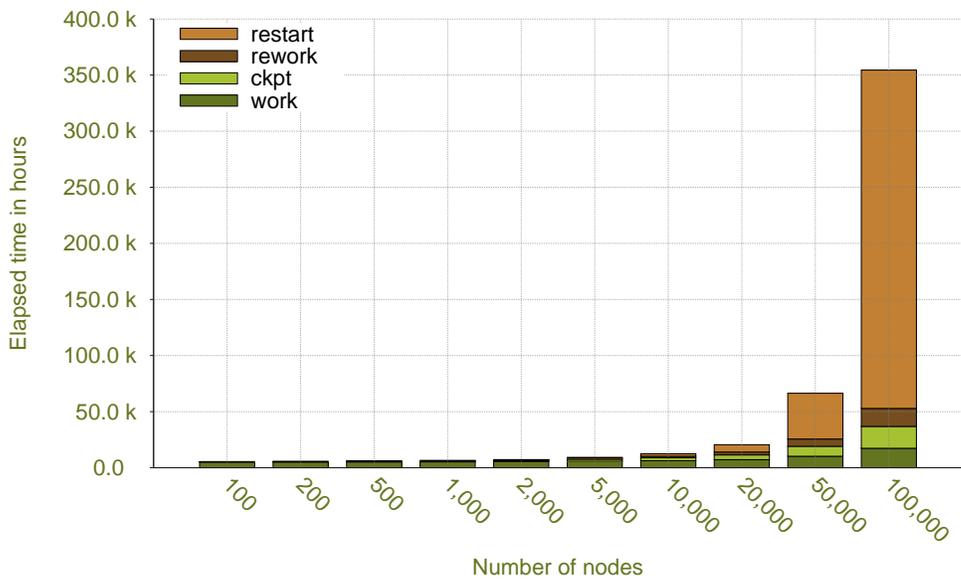
**Figure 5.8.** Percentage of time spent in each phase of a long-running application. This graph is for 700-hours of work with a node MTBF of five years.



**Figure 5.9.** An application that completes 5,000 hours of work on a system with a one-year node MTBF.

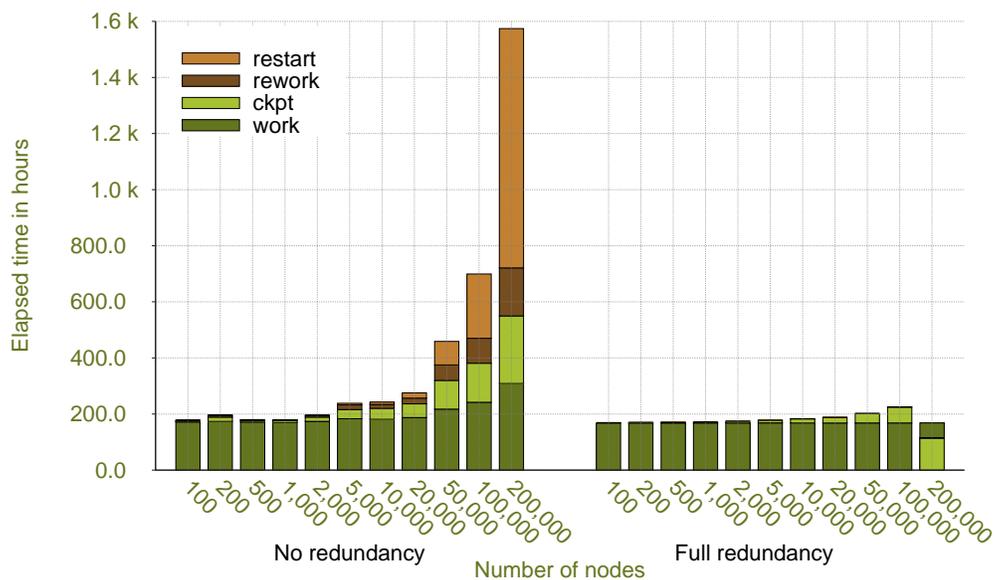


**Figure 5.10.** An application that completes 168 hours of work on a system with a five-year node MTBF.



**Figure 5.11.** An application that completes 5,000 hours of work on a system with a one-year node MTBF.

and dwarfed by the time savings when running on large numbers of nodes.



**Figure 5.12.** An application that completes 168 hours of work on a system with a five-year node MTBF. No and full redundancy shown.

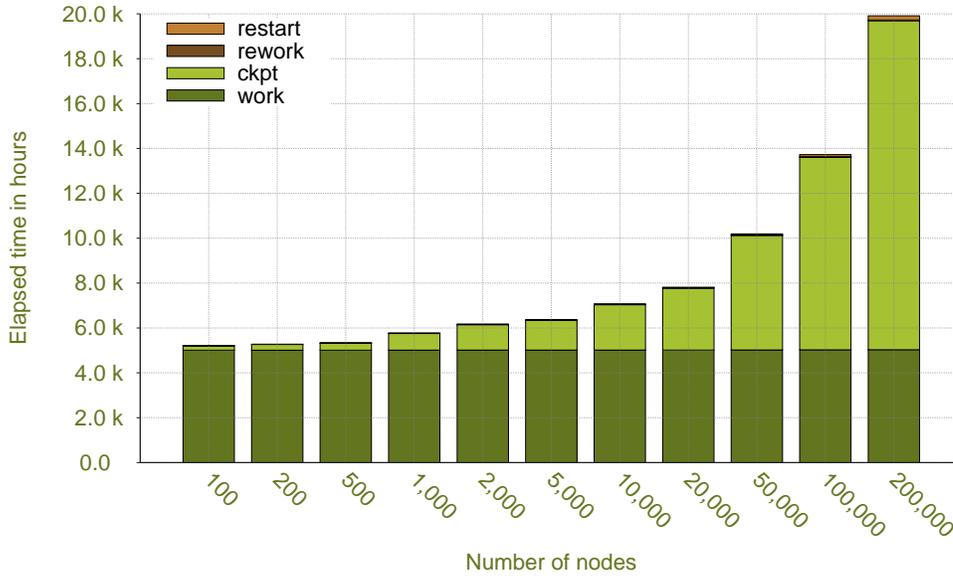
The number of nodes on the  $x$ -axis of Figure 5.12 is what the application sees. In fully redundant mode, it uses twice that many. Figure 5.13 shows the totals for our 5,000 hour application and a one-year node MTBF. Compare to Figure 5.11

Table 5.2 gives the detailed numbers for the 5,000-hour and one-year MTBF case with different level of redundancy.

## 5.5 Validating the simulation

With any simulation, validation is important. Because of the time scales involved in running such large applications and the scarcity of measured data, it is impractical for us to empirically evaluate the  $r$ MPI library. However, we can scrutinize our application simulator from various angles and arrive at the conclusion that it accurately reflects the behavior of an application on a massively parallel system,

In Section 5.3 we explained that the application simulator takes the node MTBF as one of its input parameters. It simulates node failures over the time an application needs to complete its task. We can output the time of each application interruption and calculate the mean. When we do that for simulations without redundant nodes we get the application



**Figure 5.13.** An application that completes 5,000 hours of work on a system with a one-year node MTBF and full redundancy.

**Table 5.2.** Number of interrupts seen by a 5,000-hour application and a one-year MTBF.

Num. nodes	Level of redundancy				
	none	1/4	1/2	3/4	full
100	72	58	37	23	6
200	122	95	61	36	7
500	345	239	160	89	16
1,000	771	539	363	184	22
2,000	1,658	1,166	794	358	35
5,000	5,292	3,692	2,240	998	61
10,000	14,313	9,169	5,541	2,459	92
20,000	47,235	28,622	14,997	5,913	143
50,000	379,640	181,632	76,535	23,959	302
100,000	4,047,758	1,376,461	406,890	89,265	548
200,000			4,439,041	503,426	1,127
500,000					3,337
1,000,000					9,035

MTBF as calculated by Equation 5.3. This is a good indication that the *r*MPI model within the simulator is doing its job correctly.

The *r*MPI model within the application is only 135 lines of code including white space and comments. There are 52 lines with a semicolon, which is often used to count statements. Therefore, a code inspection of the *r*MPI model is not difficult and gives us confidence in that portion of the code.

We use the default random number generator from the GNU Scientific Library (GSL) to generate the expected time for the next fault a node will experience. We use `gsl_ran_exponential()` to retrieve exponentially distributed values with a mean of the node MTBF from the random number generator.

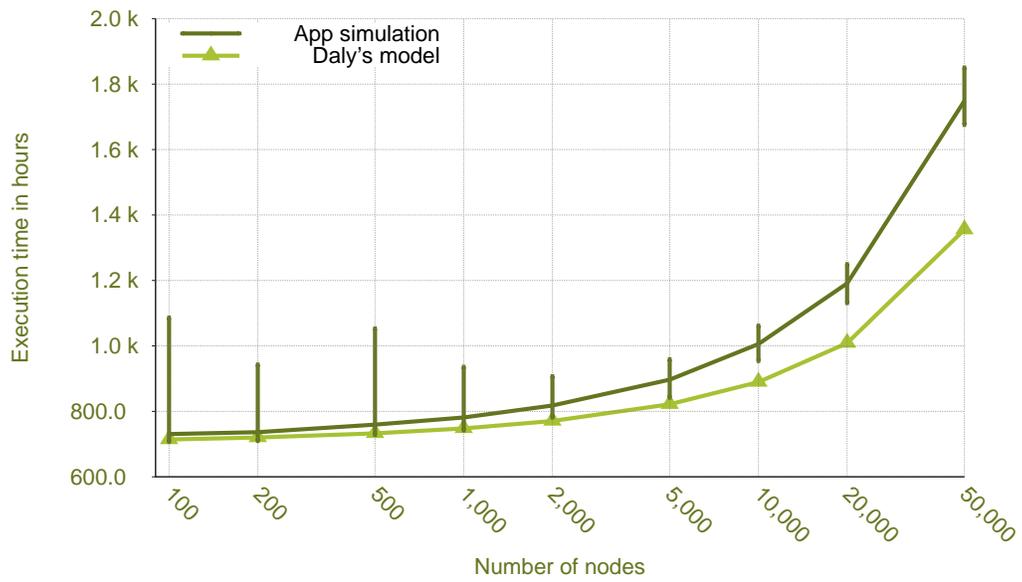
We already mentioned Jon Daly’s paper [?] where he calculates an optimal checkpoint interval (Equation 5.1). His Equation 20, which we repeat below, assumes the same checkpoint/restart behavior as we do in Section 5.1 and calculates the estimated run time of an application.

$$T_w(\tau) = \Theta e^{\frac{R}{\Theta}} (e^{\frac{\tau+\delta}{\Theta}} - 1) \frac{T_s}{\tau} \quad \text{for } \delta \ll T_s \quad (5.4)$$

Where  $T_w(\tau)$  is the total wall clock time for the checkpoint interval  $\tau$ .  $\Theta$  is the MTBF for the application and  $T_s$  is the solve time, the amount of time required to complete the assigned work.

We run our simulator 200 times, without redundant nodes, for a 700-hour application and a five-year node MTBF for various node numbers. The result is shown in Figure 5.14. We also plot the result of Daly’s equation for the same configuration. Out to about 5,000 nodes, Equation 5.4 matches the minimum values our simulation produces.

At higher node counts the simulation and the calculation begin to diverge. Multiple experiments show that the divergence gets larger if the MTBF and runtime becomes lower. A lower MTBF means more application interruptions and even a small difference in the assumptions underlying Equation 5.4 and our simulator will be amplified. We are currently investigating the differences.



**Figure 5.14.** Comparing Equation 5.4 with our application simulator.



# Chapter 6

## Implications and trade-offs

In this section we discuss what we have learned from our experiments and analysis, and discuss some of the implications and the trade-offs that need to be considered. Initially, we thought a library such as *r*MPI would require substantial support from the underlying RAS system. Our experience implementing *r*MPI has taught us that the requirements are relatively few (see Section 2.6). Most existing RAS systems and current frameworks such as [?] provide the features necessary for *r*MPI.

It would be beneficial, if failed nodes could be repaired and reintegrated into the running application. The latter would require changes to *r*MPI and the MPI library, and a mechanism for the RAS system to inform the libraries about the availability of repaired nodes.

Given the extensive portion of MPI functionality inside the *r*MPI library, it would make sense to integrate it into an existing MPI implementation. This would reduce overhead and improve the performance of collective operations.

Chapter 5 showed that using redundant nodes can improve system throughput by a factor of more than two for large scale systems. The decision to use twice as many nodes to run redundant computations has several trade-offs which we consider next.

### 6.1 System throughput

The amount of work  $T_w$  a system delivers per unit of time is its throughput. All activities that are not work; i.e., restarts, checkpoints, and rework, are overhead. If we denote this overhead  $T_{o(\text{none})}$  when no redundant nodes are in use, and  $T_{o(\text{full})}$  for the fully redundant case, then we can calculate the following ratio:

$$R = \frac{T_w + T_{o(\text{none})}}{T_w + T_{o(\text{full})} + T_{r\text{MPI}}} \quad (6.1)$$

$T_{r\text{MPI}}$  is the overhead the *r*MPI library adds. Only when  $R$  exceeds two does it make sense to use additional nodes for redundancy.  $R$  is dependent on the MTBF of a system, the number of nodes in use, and how much work ( $T_w$ ) a job has to do. In Table 6.1 we list some examples and calculate  $R$ .

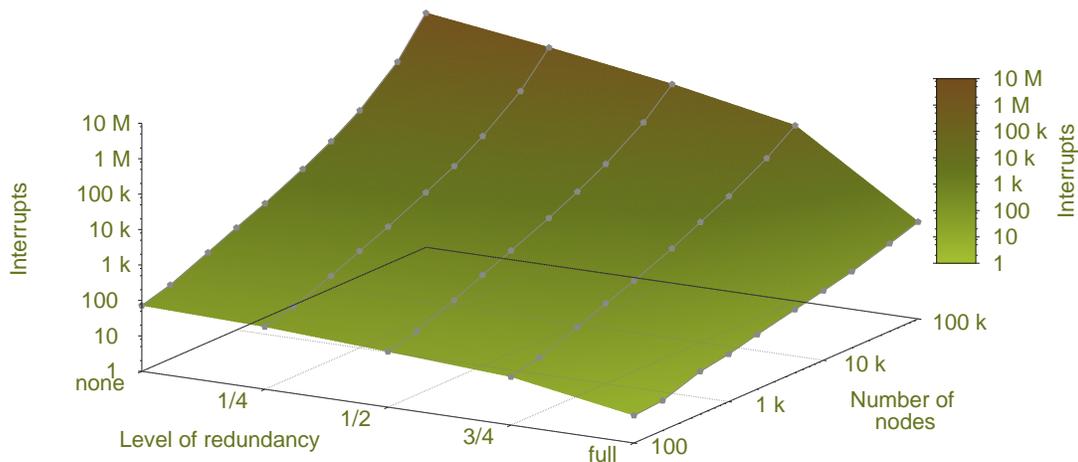
**Table 6.1.** Comparing total execution times.

Job	Redun.	Node count		
		10,000	50,000	100,000
$T_w = 8$ h	full	8.67	9.58	10.33
5-year MTBF	none	8.83	20.81	30.73
	$R =$	1.02	<b>2.17</b>	<b>2.97</b>
$T_w = 24$ h	full	26.00	28.75	31.17
5-year MTBF	none	43.74	58.68	94.38
	$R =$	1.68	<b>2.04</b>	<b>3.03</b>
$T_w = 168$ h	full	184.17	201.75	225.20
5-year MTBF	none	242.63	458.98	699.36
	$R =$	1.32	<b>2.27</b>	<b>3.11</b>
$T_w = 700$ h	full	833.00	945.75	1044.46
5-year MTBF	none	994.02	1785.14	2853.09
	$R =$	1.19	1.89	<b>2.73</b>
$T_w = 5,000$ h	full	5778.95	6985.25	7526.20
5-year MTBF	none	7200.97	12396.50	20208.84
	$R =$	1.25	1.77	<b>2.69</b>
$T_w = 24$ h	full	37.67	36.67	73.75
1-year MTBF	none	61.29	284.27	1728.48
	$R =$	1.63	<b>7.75</b>	<b>23.44</b>
$T_w = 168$ h	full	232.77	351.20	498.79
1-year MTBF	none	422.00	2229.56	11960.61
	$R =$	1.81	<b>6.35</b>	<b>23.98</b>
$T_w = 5,000$ h	full	7067.40	10194.01	13724.78
1-year MTBF	none	12456.18	66389.87	354470.25
	$R =$	1.76	<b>6.51</b>	<b>25.83</b>

Even on relatively small systems, if  $T_w$  is small enough,  $R$  becomes larger than two (Highlighted in the table using a bold green font). The ratios in the table do not include  $T_{r\text{MPI}}$  which is dependent on the actual application and problem set being solved.

## 6.2 Level of redundancy

When using redundant nodes, it does not make sense to run on anything but full redundancy. Figure 6.1 shows data from a 5,000-hour, one-year MTBF simulation. We show five levels of redundancy along the  $x$ -axis: none, 1/4, 1/2, 3/4, and full. Only full redundancy delivers the dramatic reduction in application interrupts we seek. This effect can be seen at 100 nodes, but does not really pay off until several thousand nodes ( $y$ -axis). Note that the  $z$ -axis (number of interrupts) and the  $y$ -axis (number of nodes) use a logarithmic scale.



**Figure 6.1.** Levels of redundancy versus number of interrupts.

## 6.3 Spare nodes

Most applications doing checkpoint/restart assume that the same number of nodes will be available at the begin of a restart. In this paper we do not consider repair or reboot time between an application interrupt and when it can restart again. How many spare nodes do we need to restart? Without redundancy, each node failure causes a restart and an application will consume spares equal to the number of failures it encounters throughout its

execution time. As we have seen, this number is enormous for long running applications on large numbers of nodes.

In the redundant case, many nodes may have failed before the application is interrupted. All of those nodes need to be replaced with spares. Clearly, for both the redundant and the non-redundant case, nodes need to be repaired once in a while to replenish the set of spares for applications that run for weeks and months. While the redundant approach requires more spares after a given fault, the total number of spares for a complete system does not change. If a given system needs  $s$  spare nodes to carry it to the next maintenance period, whether we are running in redundant or non-redundant mode does not change  $s$ .

# Chapter 7

## Related work

Peta-scale systems will require measures beyond checkpoint/restart to be used effectively [?, ?]. Several approaches are currently under investigation such as the work done in [?] which uses overlay nodes to improve checkpoint writing. Other approaches include optimistic checkpointing [?] and message logging [?].

The requirements for RAS systems have been studied before [?] and newer work more directly aimed at large-scale systems is under way [?]. In addition, systems that are specifically built to reduce the number of faults, and use redundancy to do so, have been available [?].



# Chapter 8

## Summary and future work

In this paper we introduced the *rMPI* library which inserts between an application and the MPI library. *rMPI* allows to allocate additional compute nodes for redundant computation. In the description of the design and implementation of *rMPI* we detailed the techniques that are necessary to maintain MPI semantics, especially message ordering on an active node and its redundant partner. We then ran a series of experiments and determined that *rMPI*'s overhead is less than 20%, usually much less, for our applications. We have shown that implementing node redundancy at the user level is possible, but a higher integration of *rMPI* with the MPI library would reduce *rMPI*'s overhead.

Redundant nodes reduce the number of interrupts an application experiences. That means less time performing checkpoint, restart, and repeated work that was lost at the last interrupt. Reducing this overhead allows more jobs to move through a system and increases the system throughput. We found that for large-scale systems with redundancy enabled, it is often possible to move more than two jobs through a system in the time it takes a job without redundant nodes to finish. In those cases the overhead of using twice as many nodes is justified. *rMPI* places very few demands on the RAS system. The main requirement is a method for the RAS system to inform *rMPI* of failed nodes.

In the future we plan to complete our current *rMPI* implementation, remove the current dependencies on MPICH and make it open source. The results of our simulation work show that applications running on redundant nodes spend too much time writing checkpoints. It may be possible to calculate a better value for the checkpoint interval.

An alternate way to enable redundant computation is to use transactions between an active node and its redundant partner. Only the active node would send messages to the active and redundant destinations. Transactions would make sure both messages have been sent, or let the redundant sender perform the transmission, if the active node fails to complete the operation. We are planing an implementation and comparison to the current design.



# References

- [1] Michael Barborak, Anton Dahbura, and Mirosław Malek. The problem in fault-tolerant computing. *ACM Comput. Surv.*, 25(2):171–220, 1993.
- [2] Joel F. Bartlett. A nonstop kernel. In *SOSP '81: Proceedings of the eighth ACM symposium on Operating systems principles*, pages 22–29, 1981.
- [3] J. T. Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Gener. Comput. Syst.*, 22(3):303–312, 2006.
- [4] Jr. E. S. Hertel, R. L. Bell, M. G. Elrick, A. V. Farnsworth, G. I. Kerley, J. M. McGlaun, S. V. Petney, S. A. Silling, P. A. Taylor, and L. Yarrington. CTH: A software family for multi-dimensional shock physics analysis. In *Proceedings of the 19th International Symposium on Shock Waves*, pages 377–382, July 1993.
- [5] E.N. Elnozahy and J.S. Plank. Checkpointing for peta-scale systems: a look into the future of practical rollback-recovery. *Dependable and Secure Computing, IEEE Transactions on*, 1(2):97–108, April 2004.
- [6] William Gropp. MPICH2: A new start for MPI implementations. In Dieter Kranzlmüller, Peter Kacsuk, Jack Dongarra, and Jens Volkert, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 9th European PVM/MPI Users' Group Meeting*, volume 2474 of *Lecture Notes in Computer Science*, September/October 2002.
- [7] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.
- [8] R. Gupta, P. Beckman, B. H. Park, E. Lusk, P. Hargrove, A. Geist, D. K. Panda, A. Lumsdaine, and J. Dongarra. Cifts: A coordinated infrastructure for fault-tolerant systems. In **To appear** in the *Proceedings of the 38th International Conference on Parallel Processing*, 2009.
- [9] Qiangfeng Jiang, Yi Luo, and D. Manivannan. An optimistic checkpointing and message logging approach for consistent global checkpoint collection in distributed systems. *J. Parallel Distrib. Comput.*, 68(12):1575–1589, 2008.
- [10] David B. Johnson and Willy Zwaenepoel. Recovery in distributed systems using asynchronous and checkpointing. In *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*, pages 171–181, 1988.

- [11] Dimitri B. Kececioglu. *Reliability Engineering Handbook*, volume 2. DEStech Publications, Inc, May 2002.
- [12] D. J. Kerbyson, H. J. Alme, Adolfo Hoisie, Fabrizio Petrini, H. J. Wasserman, and M. Gittings. Predictive performance and scalability modeling of a large-scale application. In *Proceedings of the ACM/IEEE conference on Supercomputing*, pages 37–48, 2001.
- [13] Ron A. Oldfield, Sarala Arunagiri, Patricia J. Teller, Seetharami Seelam, Maria Ruiz Varela, Rolf Riesen, and Philip C. Roth. Modeling the impact of checkpoints on next-generation systems. In *24th IEEE Conference on Mass Storage Systems and Technologies*, pages 30–46, September 2007.
- [14] A. J. Oliner, R. K. Sahoo, J. E. Moreira, and M. Gupta. Performance implications of periodic checkpointing on large-scale cluster systems. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 18*, page 299.2, 2005.
- [15] Steve J. Plimpton. Fast parallel algorithms for short-range molecular dynamics. *J Comp Phys*, 117(1):1–19, 1995.
- [16] Sandia National Laboratory. LAMMPS molecular dynamics simulator. <http://lammps.sandia.gov>, Nov. 6 2008.
- [17] Sandia National Laboratory. Mantevo project home page. <https://software.sandia.gov/mantevo>, Nov. 6 2008.
- [18] Bianca Schroeder and Garth A. Gibson. A large-scale study of failures in high-performance computing systems. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN2006)*, June 2006.

## DISTRIBUTION:

1	MS 1319	Kurt Ferreira , 1423
1	MS 1319	Rolf Riesen , 1423
1	MS 1319	Ron Oldfield , 1423
1	MS 1319	Kevin Pedretti , 1423
1	MS 1319	Todd Kordenbrock , 1423
1	MS 1319	Ron Brightwell , 1423
1	MS 1319	Jon Stearley , 1422
1	MS 1319	James Laros , 1422
1	MS 0899	Technical Library, 9536 (electronic)
1	MS 0123	D. Chavez, LDRD Office, 1011





