SANDIA REPORT

SAND2011-2488
Unlimited Release
Printed April, 2011

*r*MPI: Increasing Fault Resiliency in a Message-Passing Environment

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

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energys National Nuclear Security Administration under contract DE-AC04-94AL85000.
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
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
Online ordering: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2011-2488
Unlimited Release
Printed April, 2011

*r*MPI: Increasing Fault Resiliency in a Message-Passing Environment

Kurt Ferreira (Org. 01423) kbferre@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) rbbrigh@sandia.gov
Sandia National Laboratories
P. O. Box 5800
Albuquerque, NM 87185-1319

Rolf Riesen rolf.reisen@ie.ibm.com
IBM Research, Ireland

Abstract

As High-End Computing machines continue to grow in size, issues such as fault tolerance and reliability limit application scalability. Current techniques to ensure progress across faults, like checkpoint-restart, are unsuitable at these scale due to excessive overheads predicted to more than double an applications time to solution. Redundant computation, long used in distributed and mission critical systems, has been suggested as an alternative to checkpoint-restart on its own. In this paper we describe the rMPI library which enables portable and transparent redundant computation for MPI applications. We detail the design of the library as well as two replica consistency protocols, outline the overheads of this library at scale on a number of real-world applications, and finally outline the significant increase in an applications time to solution at extreme scale as well as show the scenarios in which redundant computation makes sense.

1 Introduction

Today's large-scale parallel computers experience outages from a number of sources including 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 [26, 14, 27].

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 [26]. 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 will spend 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.

The cost of checkpoint/restart is dependent on a number of system parameters including the time to write a checkpoint (δ), the time to perform a restart (R), and the MTBF of the application (Θ). Daly [10] presents an equation for calculating the total wall clock time ($T_w(\tau)$)) an application requires to complete a fixed amount of work (T_s). In Equation 1, τ is the optimal checkpoint interval. The MTBF of the application is dependent on the number of nodes in a system and their individual MTBFs. We must consider all types of failure that can interrupt an application: node hardware, external I/O systems, machine support; e.g., fans and power supplies, environmental; e.g., operator error and power and colling failures, as well as software failures.

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

Figure 1 shows the time to solution using Daly's equation for a million node application with a 168 hour solve time. In this figure green (on the right) corresponds to a wall clock time of 168 hours, white to 252 hours (50% increase in wall time), and brown (on the left) to 336 hours (100% increase) or greater. There are a number of routes that can be taken to make checkpointing more tractable at scale. For example, in our million node application example, to keep checkpoint overhead to less than 50% (brown area in Figure 1) of our application we need a node MTBF greater than 200 years and a checkpoint time less than 10 minutes. To keep overhead less than 1/3 more demanding advances are needed. This situation worsens when node counts exceed a million. Though there is ongoing active research in this area to increase node MTBF and decrease checkpoint time, there is concern whether technological advances can deliver this required performance.

As an alternative, we present a method to increase resilience through redundant computation that places no demands on advances in system reliability or reduced checkpoint time. Previously, this approach was dismissed as being too expensive for the meager benefits seen at smaller machine scales. This approach effectively increases the time between faults which results in fewer restarts, and less rework. All of these lead to better system throughput. Additionally, this technique can be used in conjunction with other techniques to reduce checkpoint overhead.

In redundant computing, 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.

Another critical issue is the overhead of redundant computing in a message-passing environment. Keeping replica state consistent requires additional messaging 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 prefer the two nodes to be physically as far apart as possible from each other. This introduces

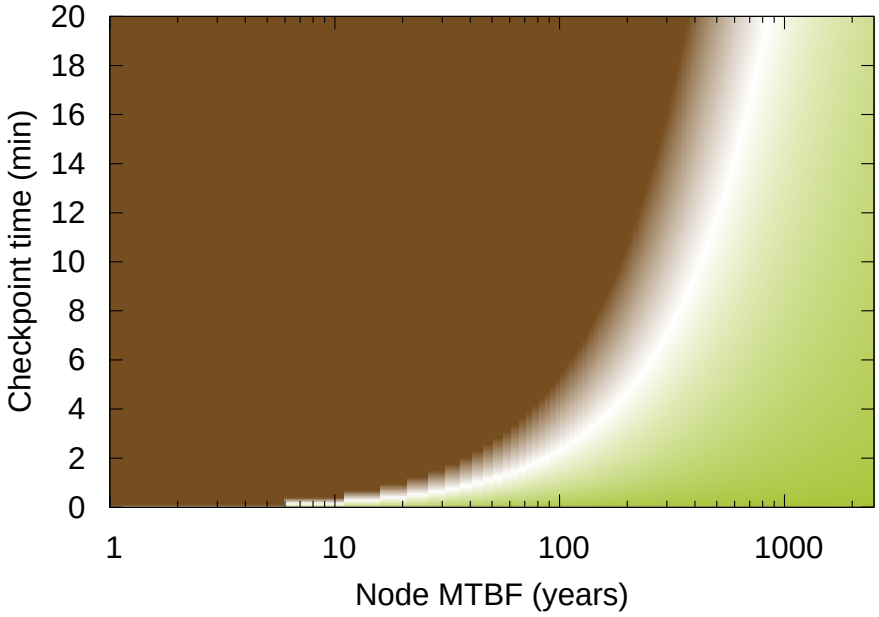


Figure 1. Wall clock time evaluation using Daly's equation for a 168 hour, one million node application given a node MTBF and checkpoint time. Green (right) corresponds to a wall clock time equal to 168 hours, white to 252 hours (50% increase in wall time), and brown (left) to 336 hours (100% increase) or greater.

additional delays and network congestion that slows down synchronizations and application performance.

Note that redundant computing is not a substitute for checkpoint/restart. It reduces the number of interruptions an application experiences and therefore reduces the overhead of checkpointing but does not eliminate it. Redundant computation also enables the detection of silent errors. These are errors that occur in registers, caches, and data paths that are not protected by error detection or correction codes. These errors are often silent because they produce a corrupted result but do not further disrupt the computation. Most applications assume that calculations done by the hardware are correct and do not or cannot check for invalid results. Double redundancy lets us flag silent errors, and triple redundancy would let us correct these errors.

In this study, we directly quantify the cost of our redundant MPI library on a range of microbenchmarks and real-world, large-scale applications. We have implemented this transparent user-level rank replication library to hook in at the MPI profiling layer and have tested it with application running on the Sandia/Cray Red Storm system [6]. The main contributions of this paper are:

- presentation of *r*MPI a portable, transparent user-level redundant MPI profiling library
- evaluation of the performance overhead of two consistency protocols each of which places different demands on a RAS system
- given this overhead, a presentation of when redundant computation pays off in terms of the extra resources used

To out knowledge, this is the first such empirical study on the cost of redundant computation for HPC.

2 Design

To evaluate the benefits and costs of redundant computing, we set out to design a portable user-level MPI library that does redundant computation transparent to the user. The basic idea for the *r*MPI library is simple: replicate each MPI rank in an application and let the replicas continue when an original rank fails. As is common with checkpoint/restart, we assume that the application will restart on the same number of nodes. This means spare nodes must be available or nodes must be repaired before a restart.

To ensure duplicate state across replicas, the library implements a protocol that enforces consistent state. In MPI, state consists of both local memory state and the messages received by a node. The library must balance keeping replica state consistent while keeping the overhead due to the protocol low. In this study we focus on two protocols that *r*MPI contains. These protocols differ in the messages they send and the demands they place of the RAS system. The two protocols, denoted *mirror* and *parallel*, are described in the following sections.

2.1 Mirror Protocol

Figure 2 shows the message exchanges necessary to ensure consistency in the mirror protocol. In this figure A and B represent distinct MPI ranks and A' and B' are A's and B's replicas respectively. In this protocol, each sender transmits duplicate messages to each of the destinations. Similarly, receivers must post multiple receives for the duplicate messages, but only require one of those messages to arrive in order to progress. Special care and additional replica coordination is required for `MPI_ANY_SOURCE` operations. We describe these operations in more detail below.

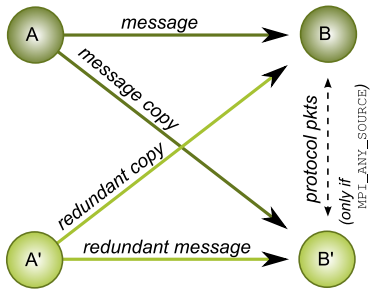


Figure 2. Each sender transmits the user messages twice and additional protocol exchanges are needed in the case of `MPI_ANY_SOURCE`

At job launch, a user-specified map which associates replicas with MPI ranks is distributed to each node. When a node is believed to have failed, 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.

Due to MPI message-passing semantics and the possibility of wildcard source receives, rank replicas cannot be completely symmetrical. They need to coordinate on certain operations. For example, the order in which unexpected messages arrive at rank B must match the order in which they arrive at rank B'. Since some computations are message-order dependent, different arrival orders would lead to different computations on the two nodes and therefore an inconsistent state across replicas. To handle this, the protocol uses the notion of a leader (or active) node in an MPI rank bundle. Non-leader nodes are referred to as replicas or redundant nodes. When a leader drops out of computation the protocol chooses a replica to take over as leader. *r*MPI uses one high order bit in the tag to distinguish messages from leader and replica nodes.

Blocking operations The sequence of events depends on whether the receive request is for a specific source or `MPI_ANY_SOURCE`. If the application has requested a specific source, let say for rank A, then the protocol posts receives for both A and replica A', assuming of course replica A' exists. *r*MPI receives the message from both A and A' into the buffer provided by the user. Since the data in the two arriving messages are identical, there is no danger in corrupting the user buffer. If multiple messages from the bundle B arrive with the same tag, *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 by using one high-order tag bit, setting it on all outgoing redundant messages and setting the same bit for all receives of redundant messages.

The situation is illustrated in Figure 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 original 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.

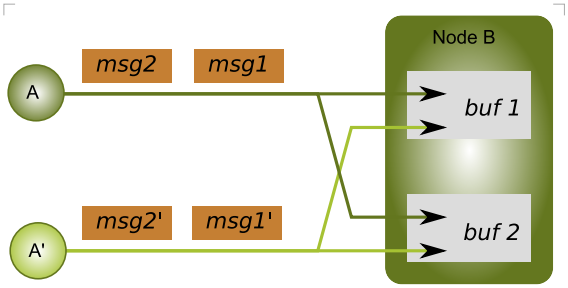


Figure 3. Original and redundant messages with the same tag must maintain the same order.

The situation gets more complicated when an application uses `MPI_ANY_SOURCE` to receive messages. For example, if messages *msg1* and *msg2* come from different nodes, they can each end up in *buf1* or *buf2*. Whatever that order, it must be preserved on all rank within a replica bundle. To enforce this order, *r*MPI performs the following steps: On the leader 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 from the node that sent the first message to node B. On all nodes in the receive replica bundle, if the received message has a redundant receive that receive must also be consumed.

Non-blocking operations The situation described above is further complicated if the `MPI_ANY_SOURCE` is a non-blocking receive operation. As soon as *r*MPI posts the first `MPI_ANY_SOURCE` non-blocking receive it must wait to post any further receives (`MPI_ANY_SOURCE` or otherwise) on replica nodes until all outstanding `MPI_ANY_SOURCE` receives have been satisfied or the application waits on the receive operation. This is again to enforce the same message ordering on all replicas This implies that applications with `MPI_ANY_SOURCE` receive operations are likely to see many more unexpected messages then their leader node counterparts.

Since the user requested a non-blocking receive, both nodes must return at this point. The replica node cannot post the receive until it has the envelope information from the leader. This means the replica node must maintain a queue of receives the user has posted but that *r*MPI 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 compares these envelopes with the receives in its queue and posts the matching ones to complete those operations.

MPI guarantees message ordering between node pairs. In addition, rMPI 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. However, 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.

rMPI 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 rMPI needs to return a request handles to the user. This means rMPI needs to maintain data structures that map its request handles to the ones used by the underlying MPI implementation.

When the receive of a message is complete, 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 replica nodes return to the user the rank of the leader node in a bundle. For example, `MPI_Comm_rank()` must return the same value on an leader node and its replica, similarly `MPI_Comm_size()` must return the number of the unique ranks in the application. Message destinations and sources must be treated similarly.

Probe and test Similar to the operations described above, replica nodes must return the same information as their leader nodes for probe 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 leader 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.

These function are once again further complicated when `MPI_ANY_SOURCE` is used. Similar to receive operations, `MPI_ANY_SOURCE` requires replica nodes to coordinate with their leader nodes on the function result, ensuring all replica nodes return the same information as their leaders.

Other functions and 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. As an option, rMPI can synchronize the `MPI_Wtime()` clocks across the nodes [23]. Collective operations in rMPI call the point-to-point operations internal to rMPI.

rMPI also needs to implement its own groups. Because rMPI re-maps ranks between the user level and the underlying MPI implementation, rMPI 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 rMPI reduces the underlying MPI implementation to a simple transport mechanism and increases the complexity of rMPI greatly.

`MPI_ANY_TAG` receives Above we describe the difficulty in handling `MPI_ANY_SOURCE` receives. It requires that replica nodes wait until the leader 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.

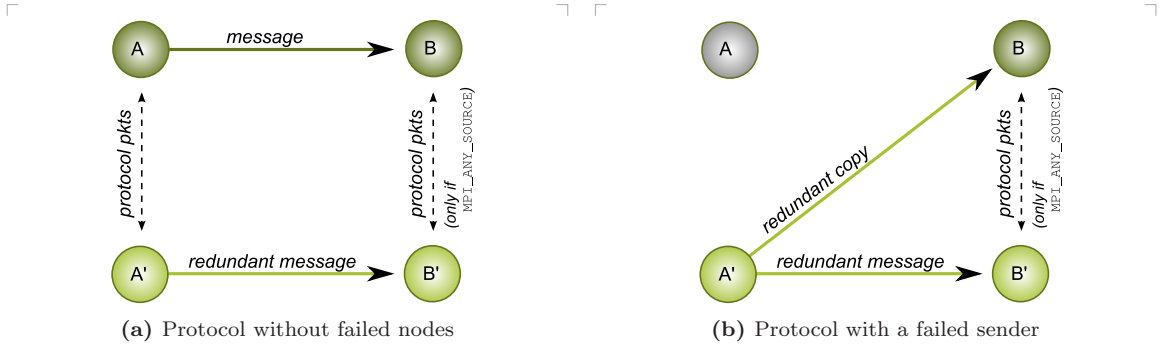
Receives using `MPI_ANY_TAG` are also problematic. Redundant messages use an extra tag bit to identify them so that messages with the same tag will not mix with the corresponding redundant messages. Using an extra tag bit assumes the underlying MPI implementation provides a larger tag space than the 2^{15} range

mandated by the MPI standard. Most MPI implementations provide a much larger tag space and rMPI 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 rMPI 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, the mirror protocol currently does not support receives that use `MPI_ANY_TAG` and `MPI_ANY_SOURCE` simultaneously.

2.2 Parallel Protocol

From the discussion in the previous section, we see that while the mirror protocol requires very little from a RAS system the duplicate messages can severely degrade application performance in scenarios where frequent large messages are sent. With this in mind we set out to design a second protocol such that a node will usually only send each application message once. Additional protocol messages are needed among the senders to coordinate in case of failures. This protocol, which we refer to as parallel, is diagramed in Figure 2.2. In the remainder of this section we outline the similarities and differences from the mirror protocol.



(a) Protocol without failed nodes

(b) Protocol with a failed sender

Figure 4. Parallel protocol both with and without a failed node

The main difference to mirror is the decreased need for network bandwidth. Excluding protocol messages, if no nodes have failed and the user has specified full redundancy, all leader nodes communicate as one copy of the application and all the replica nodes communicate as another copy. In this optimal scenario only protocol packets cross our two parallel versions of the application. A second difference is that on send operations, the nodes in a replica bundle must synchronize to ensure that both the original and the redundant message are sent; even if one of the senders has failed: Figure 4(b). In contrast to mirror, this protocol requires querying the RAS system to determine if the source or destination rank has failed.

This means that, compared to the mirror protocol, the parallel protocol will initiate approximately double the number of messages for each send operation. These extra messages contain MPI envelope information and are small. Therefore, the parallel protocol trades bandwidth for an increased number of short messages. It is also important to note that as nodes begin to fail with this protocol and replicas must take over the active sender's message responsibilities, this protocol essentially degenerates to mirror.

Another key difference to mirror is what happens on a send. In mirror, as long as the application has enough nodes to continue, there is always a destination node to send to; not so in parallel. For example, in parallel it is possible for a destination node B to fail but computation to continue if at least one replica of B still exists. In that case, node A sending to B does not actually perform the send operation. If send timing consistency is necessary, parallel can delay send functions by a typical send delay or the average amount the replicas spent inside the function.

Similar to the mirror protocol, special care must be taken to enforce message order on all replicas in the

case of MPI_ANY_SOURCE. On the other hand, the parallel protocol does not require the special tag bit handling we describe in the previous section. It is not needed since a receiver only receives from one source node and the underlying MPI message ordering guarantees are enough to main consistency.

2.3 Dependence on RAS system

*r*MPI's requirements of the RAS system are modest. Both mirror and parallel protocols require that messages from 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.

For the parallel protocol 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 it could be an event mechanism that informs *r*MPI whenever the RAS system detects a failed node.

3 Implementation

The *r*MPI library is implemented as a library at the MPI profiling layer between an application and an MPI implementation. In this section we list some things that are specific to our current implementation.

The *r*MPI library 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 XT4 Red Storm system which uses MPICH [18, 17] for message transport. Although the design described in the previous section is agnostic of the underlying MPI implementation, our current implementation of *r*MPI is not. To accelerate prototyping we used several functions from MPICH, such as the MPI collective functions which call our protocol aware point-to-point functions. While doing this we left several low-level, MPICH internal, function calls in place. Examples include MPICH error handling and reporting functions, checking for thread-safety, and dealing with heterogeneous systems. This means *r*MPI will currently only work running on top of our specific MPICH version. We are in the process of removing this MPICH specific dependency.

Since few machines actually provide a RAS system that gives us the minimal set of functions we need, we had to improvise. *r*MPI maintains a table of all nodes in the application and their status. We use signals and messaging to update this table 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 Red Storm, 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 select 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. For this testing at most one replica node is assigned to a leader node.

Lastly, to avoid using additional buffer space and to 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 the 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. We are not aware of any system today which does not fulfill this requirement.

4 Evaluation

From the discussion in the previous sections it should be clear that using rMPI will add overhead and lengthen the execution time of an application. To measure this overhead we ran multiple tests with benchmarks and applications on the Cray Red Storm system at Sandia National Laboratories. Red Storm is a XT3/4 series machine consisting of over 13,000 nodes, with each compute node containing a 2.2 GHz quad-core AMD Opteron processor and 8 GB of main memory. Additionally, each node contains a Cray SeaStar [4] network interface and high-speed router. The SeaStar is connected to the Opteron via a HyperTransport link. The current generation SeaStar is capable of sustaining a peak unidirectional injection bandwidth of more than 2 GB/s and a peak unidirectional link bandwidth of more than 3 GB/s. To ensure leader and replica are on separate physical 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 is a random shuffle (Fisher/Yates) such as ABCD|C'B'D'A'.

We expect that the rMPI 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 rMPI library is not linked in at all. To get a worst case bound on the cost of rank level replication a fully redundant configuration is used for the forward, reverse, and shuffle mappings.

To evaluate the performance of the two rMPI protocols we will present results for MPI latency and bandwidth micro-benchmarks, and four applications; CTH [12], SAGE [22], LAMMPS [31, 34], and HPCCG [35]. These application represent a range of computational techniques, are frequently run at very large scales, and are key simulation workloads to both the US DOD and DOE. These four applications represent different communication characteristics and compute to communication ratios. Therefore, the overhead of rMPI affects them in different ways.

4.1 Micro-benchmark Performance

Because these benchmarks do nothing but transmit messages, we expect them to show greater overhead than the full applications we tested. For the MPI latency tests we show the performance overhead for both specific as well as `MPI_ANY_SOURCE` receives as each scenario has different performance characteristics.

Our bandwidth experiment in Figure 5 shows that baseline (rMPI linked in, but no redundant nodes used) for both protocols does not lower bandwidth appreciably compared to native; especially at larger message sizes. The parallel protocol redundant runs, on the other hand, shows considerable overhead, especially at smaller message sizes, showing 60% to 70% slowdown in comparison to native. This slowdown is identical for each of the three tested mappings (shuffle, forward, and reverse). The overhead for the parallel runs is due to the overhead of the increased number of messages required for bundle synchronization. As message size increases the performance of parallel approaches that of native. The mirror protocol redundant run performance is also identical among the three mappings, but its 60% slowdown over native remains nearly constant through the range tested. This halving of bandwidth is expected and consistent with the fact that

we are sending twice as much data through a given NIC.

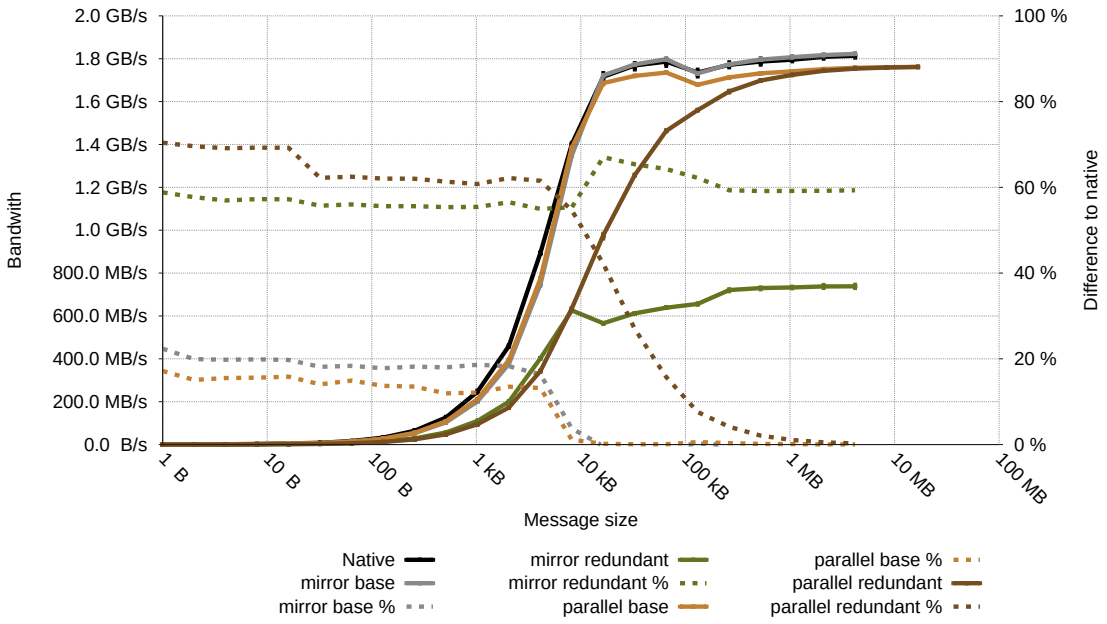


Figure 5. Bandwidth comparison. Native is benchmark without the rMPI library. Base is with rMPI for each protocol, but no redundant nodes. For this test the performance of forward, reverse, and shuffle fully redundant runs are equivalent.

Figure 6 illustrates the overhead of our MPI latency tests without `MPI_ANY_SOURCE`. The baseline mapping for the two protocols shows some overhead over native which is due to the accounting done in rMPI and becomes negligible as message size increases. The latency overhead for the redundant runs is a factor of 1.5 over native for smaller messages. For parallel this latency increase is one third that of native and is due to the extra messages used for synchronization on sends and decreases with message size. The reason the latency is less than n extra message latencies is that, assuming no nodes have failed, a sender node first performs the send operation and then performs the synchronization with replicas in its rank bundle to ensure it does not need to fulfill another send. If a node has failed, the performance of parallel closely matches that of mirror. For mirror, the slowdown with full redundancy is 3/4 that of native. The reason for this increased slowdown is as follows. A receive in mirror can return once at least one of the two possible receives has completed. Before the receive can return we must wait for the other receive or cancel it. `MPI_Cancel()` in our MPICH implementation is an expensive and non-local operation. The current implementation waits twice a measured round trip time for the other send to arrive. If it has not been received in that time, the library cancels the other receive and then returns. Similar to our bandwidth tests, the overhead of the redundant runs is identical for each of the three replica node mappings tested.

The coordination overhead between leader and replica nodes becomes more severe when `MPI_ANY_SOURCE` is used. Recall from the discussion in Section 2, `MPI_ANY_SOURCE` causes replica nodes to delay the posting of receives until the leader node has received its message and informed the redundant node. In Figure 7 we see the result of this. Latency increases by a factor of 1.5 across the board over native.

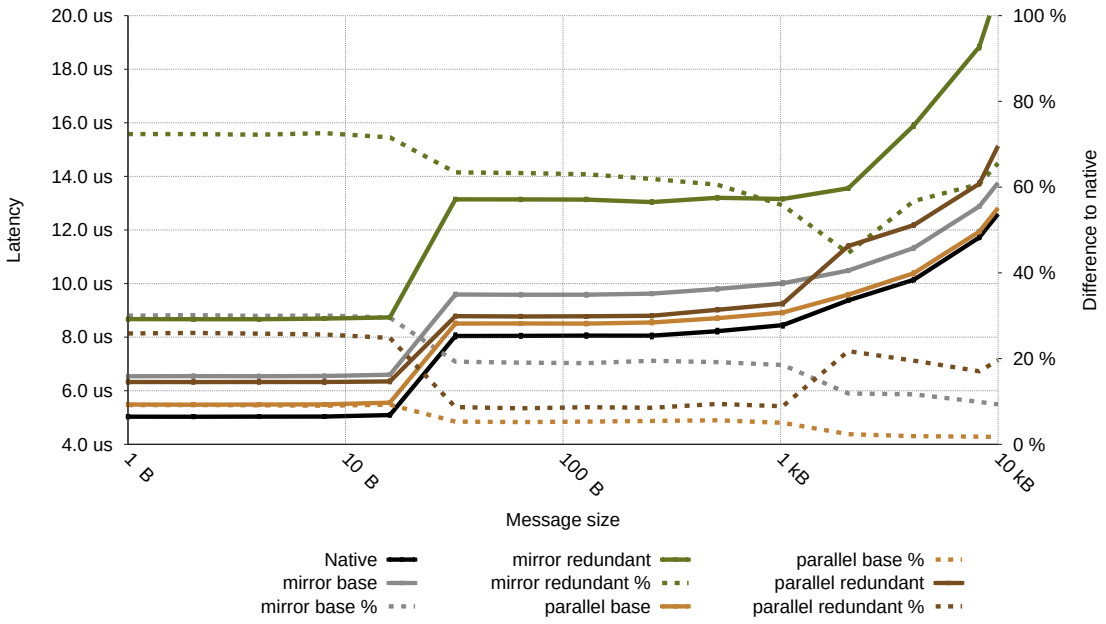


Figure 6. Latency comparison. For this test the performance of forward, reverse, and shuffle is equivalent.

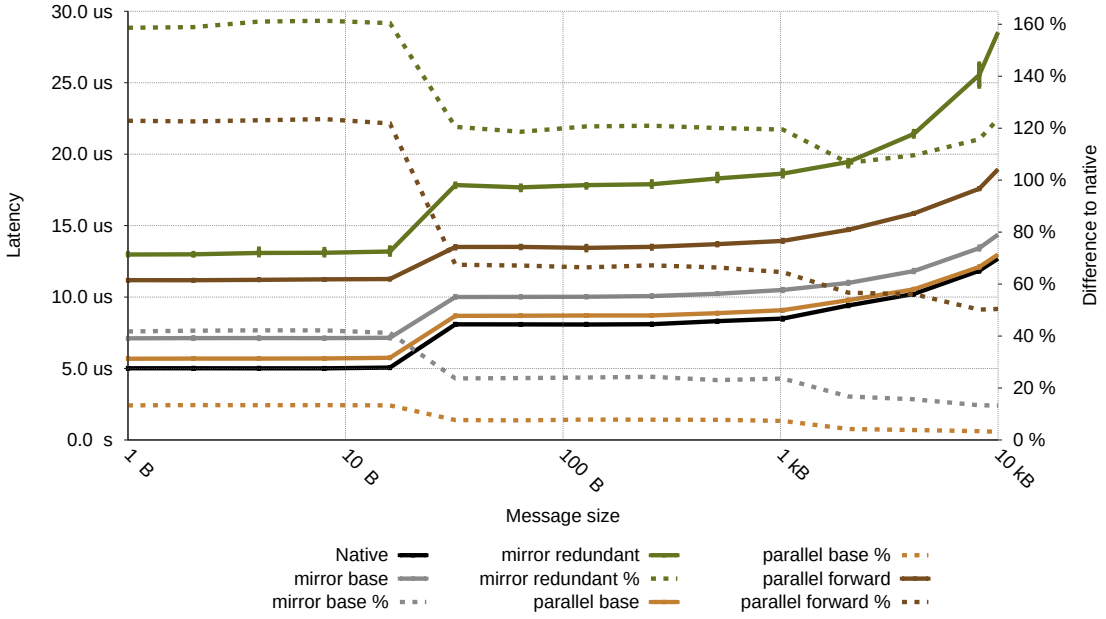


Figure 7. Latency comparison using `MPI_ANY_SOURCE`. For this test the performance of forward, reverse, and shuffle is equivalent.

4.2 Application Performance

In the previous section we showed the overhead of our two rMPI consistency protocols to be quite large for the two micro-benchmarks tested. In this section we investigate how the described protocols affect application performance. For each of the applications tested we show plots with node counts up to 2,048 nodes. For the native and baseline runs this means the application will use 2,048 nodes. For the redundant runs this means the application will use a total 4,096 nodes, of which 2,048 are leader nodes and 2,048 replicas.

LAMMPS Figure 8 shows the performance impact of rMPI with both the mirror and parallel protocol. The impact of each redundancy protocol is less than 5%, independent of the nodes used, while the baseline overhead for each is negligible.

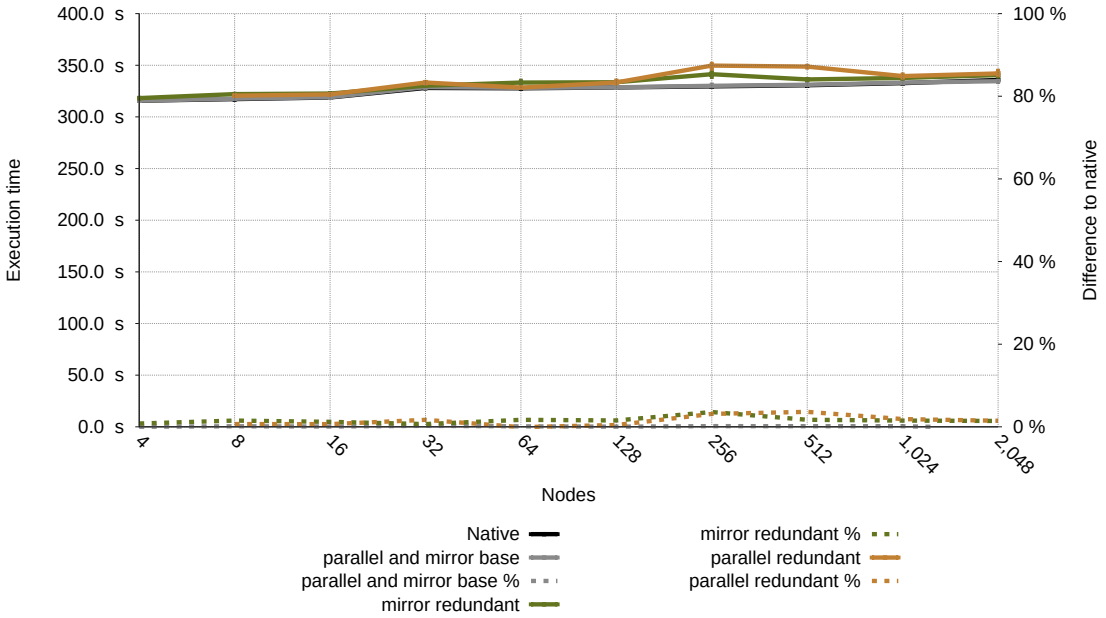


Figure 8. LAMMPS rMPI performance comparison. For both mirror and parallel baseline performance overhead is equivalent. For this application the performance of forward, reverse, and shuffle fully redundant is equivalent.

SAGE Figure 9 shows the rMPI performance for SAGE. Similar to LAMMPS, the baseline performance degradation is negligible. Also similar to LAMMPS, the parallel protocol performance remains nearly constant and performance decrease is negligible in the tested node range; with performance overhead generally less than 5%. In contrast, full redundancy for the mirror protocol loses about 10% performance over native, with performance increasing with scale. We attribute the performance degradation for SAGE to the factor of two increase of large network messages sent by SAGE and the limited available network bandwidth.

CTH In Figure 10 we see the impact of our consistency protocols for CTH at scale. Again, baseline for both mirror and parallel shows little performance difference. For CTH, mirror has the greatest impact on performance with full redundancy. This impact, which is nearly 20% at the largest scale, is due to CTH's known sensitivity to network bandwidth [29] (the greatest of each of the applications tested) and the increased bandwidth requirements of the mirror protocol. Interestingly, the parallel protocol version of

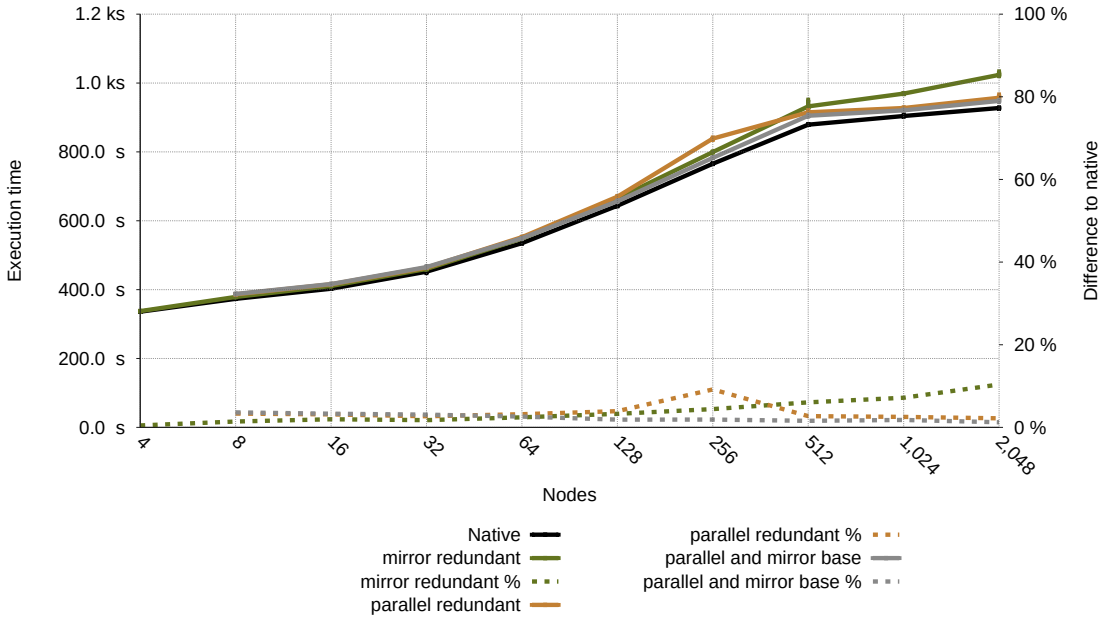


Figure 9. SAGE *r*MPI performance comparison. For both mirror and parallel baseline performance overhead is equivalent. For this application the performance of forward, reverse, and shuffle fully redundant is equivalent.

CTH runs slightly *faster* then the native versions (around 5-8%). We see this increase for all three forward, reverse, and shuffle replica node mappings. Though further testing is needed, current performance analysis results suggest this decrease in application runtime is due to unexpected messages. A CTH run is known to contain a large number of unexpected messages [3]. With parallel's sender side synchronization protocol, many of these unexpected messages will now arrive after the receive has been posted and therefore slightly increase CTH's performance.

HPCCG Figure 11 shows the performance impact of *r*MPI on the HPCCG mini-application. In contrast to the other results presented in this section we present the mirror and parallel results separately. Though the results presented in Figure 11(a) and Figure 11(b) represent the same computational problem, the native results of each vary due to different node allocations between the two plots. Allocation issues aside we see that mirror has very little impact. Parallel on the other hand shows a significant impact at higher node counts, with slowdowns of around 10% at 1,024 nodes. Also, in contrast to all the other applications tested, impact from the parallel protocol is greater than that of mirror. The reason for this slowdown for parallel is due to the fact that at the largest node counts, HPCCG communication in addition to the synchronization messages are hitting the maximum messaging rate of a node.

4.3 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.

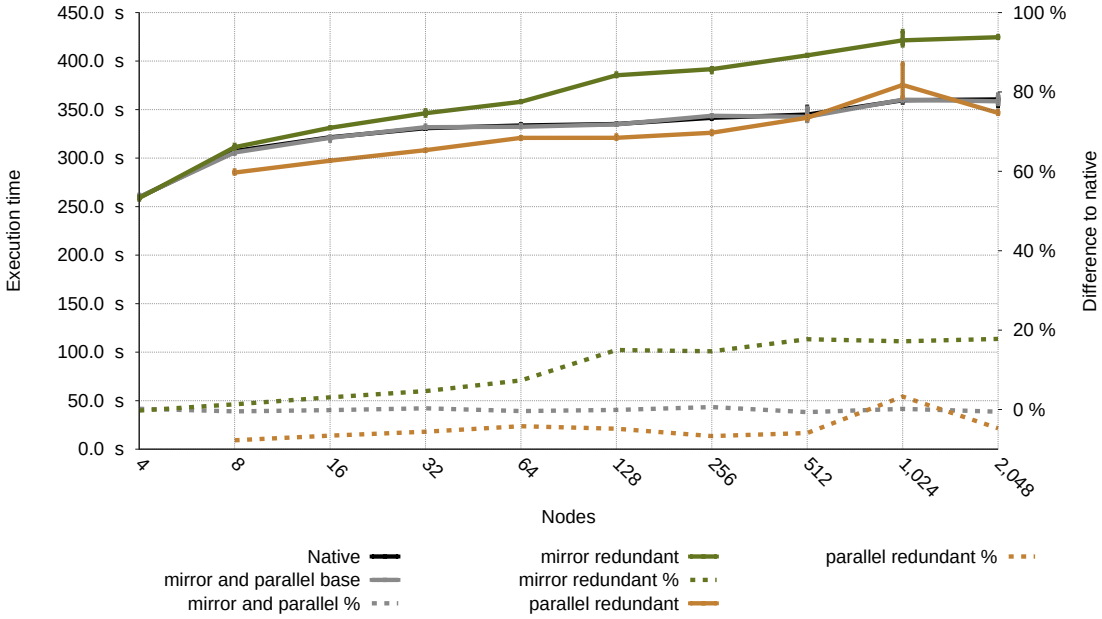
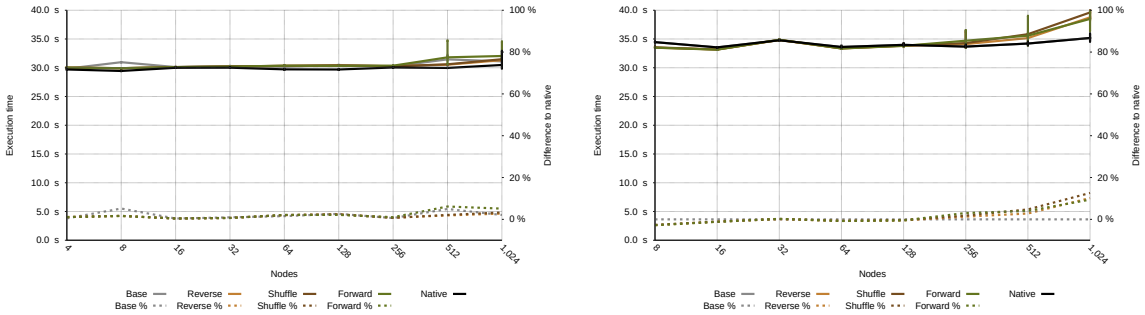


Figure 10. CTH *r*MPI performance comparison. For both mirror and parallel baseline performance overhead is equivalent. For this application the performance of forward, reverse, and shuffle fully redundant is equivalent.



(a) Mirror protocol

(b) Parallel protocol

Figure 11. HPCCG *r*MPI performance comparison. Varying performance for native and baseline between mirror and parallel protocols is due to different node allocations.

While micro-benchmarks clearly show the overhead introduced by the two consistency protocols for rMPI, the impact on applications is much less severe. It ranges from 20% overhead for the mirror protocol on CTH, a communication intensive application, to almost no overhead for LAMMPS, to over 10% overhead for the parallel protocol with HPCCG, a message rate intensive problem. For all tests except HPCCG, the parallel protocol outperforms the bandwidth intensive mirror protocol. Though mirror has the advantage of virtually no performance degradation as nodes fail, parallel performance will degrade proportionally to that of mirror as nodes fail.

5 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 with regard to when redundant computing pays off from a resource usage perspective as well as the system requirements for redundant computing.

5.1 RAS Requirements

Initially, we thought a library such as rMPI would require substantial support from the underlying RAS system. Our experience implementing rMPI has taught us that the requirements are relatively few (see Section 2). Most existing RAS systems and current frameworks such as [19] provide the features necessary for rMPI.

It would be beneficial if failed nodes could be repaired and reintegrated into the running application. The latter would require changes to rMPI 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 rMPI library, it would also make sense to integrate it into an existing MPI implementation. This would reduce overhead and improve the performance of may operations, especially collectives.

5.2 When does redundant computation make sense

There is an inherent trade-off between the cost of using additional resources for redundancy and the savings of redundant computation by reducing the I/O required for checkpoint and restart. One way to evaluate this trade-off is to approximate, through analytic modeling, the total execution time of applications with and without redundant computation.

Figures 12 shows the approximate wallclock runtime of a fully-redundant and non-redundant application that uses checkpoints to ensure resilience. For both cases, we use Daly's equations [10] to calculate the wallclock time $T_W(\tau, n)$, Equation 1, and the optimal checkpoint interval τ_{opt}. For the fully-redundant application, we use the *Birthday Problem*[24, 32] as a way to approximate the number of node failures the application can survive without a restart. We then use this result to adjust the MTTI of the application, thus increasing τ_{opt} and reducing the checkpoint/restart overhead. See [32] for details on applying the birthday problem to redundant computation.

A third line in Figure 12 shows the runtime of the redundant application as a percentage of the non-redundant application. When the line falls below 50%, the benefit of redundant computation exceeds the cost of non-redundant computation. For a system with MTTI/node of 5 year, redundant computation makes sense for applications larger than 50,000 nodes. Figure 13 shows how the same plot for systems with different MTTI/node.

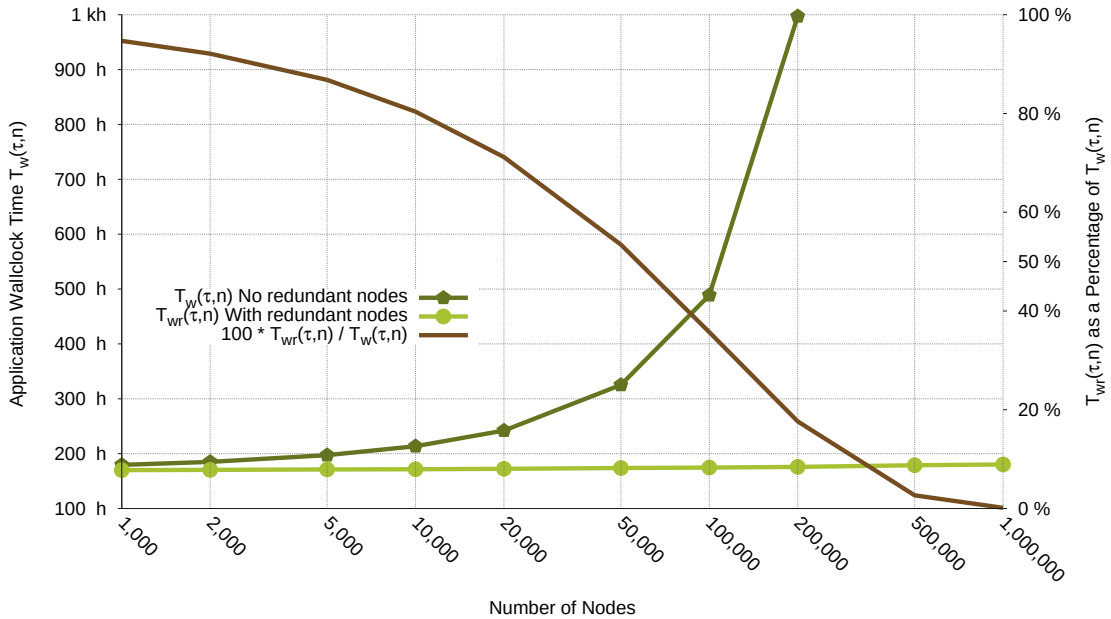


Figure 12. Wallclock running time of a fully-redundant and non-redundant application on a system with a 5-year MTTI/node. A third line shows the runtime of the redundant application as a percentage of the non-redundant application.

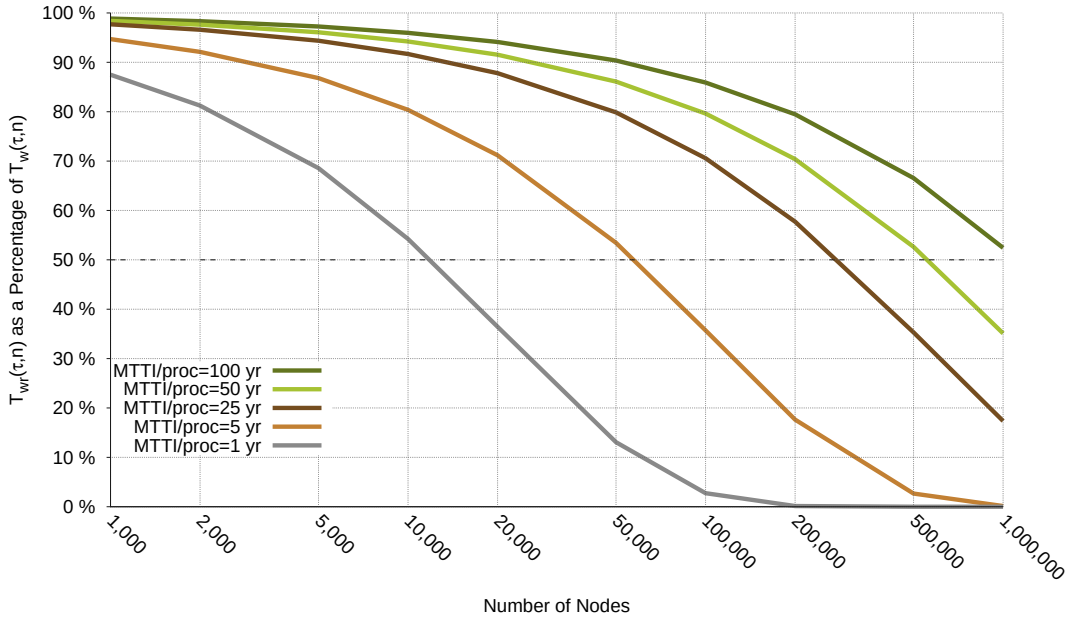


Figure 13. Wallclock time of a fully-redundant application as a percentage of non-redundant application for systems with MTTI/node = 1, 5, 25, 50, and 100 years.

6 Related work

Redundant computation and process replication has a long history and is used extensively in both distributed [16, 8] and mission critical systems [25, 2, 28] as a technique to improve fault tolerance. Previously, this idea had been dismissed in HPC as being too expensive for the meager benefits that are seen at present machine scale. Recently, Schroeder & Gibson [36] and others [38] have suggested redundant computation (also termed process pairs) as an alternative to standard checkpoint/restart fault-tolerance methods due to the high predicted overhead of such methods on next-generation scale machines.

Recently, Engelmann et al. [15] make a case for double and triple modular redundancy for HPC in terms of the increased availability of the system, outlining the dramatic increase with various levels of redundancy. We instead explore mean time to job interrupt, since we believe this parameter is more useful as it allows you to estimate the time to solution for an application, and optimal checkpoint interval [10].

There are a number of excellent studies investigating the efficiency of checkpoint/restart for large scale systems [14, 27, 13]. Traditional checkpoint/restart breaks down into a number of areas based on: where the checkpoints are written, what data is written, and if nodes coordinate when writing.

Where the data is written typically varies from writing checkpoints to various forms of stable storage [11] or disk to diskless methods that write checkpoints to memory [9, 30, 37]. What data gets written varies from checkpointing full memory images to incremental methods that write only data that has changed [33, 5]. Lastly, how nodes coordinate to write the checkpoint varies from full coordination with all nodes writing concurrently, to uncoordinated approaches where each node checkpoints independently. Referring back to Figure 1, in each of these checkpoint cases, to use less resources on a million nodes than redundant computation the node MTTI checkpoint time must land on a green area on the plot.

In addition, several alternative approaches to checkpoint/restart are currently under investigation such as the work done in [26] which uses overlay nodes to improve checkpoint writing. Other approaches include optimistic checkpointing [20] and message logging [21, 7].

The requirements for RAS systems have been studied before [1] and newer work more directly aimed at large-scale systems is under way [19].

7 Summary and future work

In this paper we presented the tradeoffs of redundant computing for HPC. We describe the *r*MPI library which inserts between an application and the native MPI library. *r*MPI enables redundant computation and uses two separate consistency protocols to maintain consistency between rank replicas. In the description of the design and implementation of *r*MPI we detailed these protocols and described the techniques that are necessary to maintain MPI semantics, especially enforcing message ordering across replicas. We presented experiential data from these two protocols and determined that *r*MPI's overhead is quite small in most cases but dependent on the protocol used with mirror showing the greatest overhead for bandwidth sensitive applications and parallel showing greatest overhead to message rate sensitive applications. This library shows that implementing a transparent, rank-level redundant, user-level library is possible. It is important to note that this library represents a worst case overhead for redundancy as a lower level integration of *r*MPI with a MPI library would reduce much of the overhead presented in this work.

In addition to the overheads, we briefly presented the possible benefits of redundant computation. Replication reduces the number of interrupts an application experiences. That means less time performing checkpoint, restart, and the repeated work that was lost at the last interrupt. Reducing this overhead allows more jobs to move through a system and increases 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.

Currently, we are using the *r*MPI library to investigate frequency of occurrence of silent data corruption on a variety of different architectures. With *r*MPI's low run time overhead we can flag silent errors using full dual redundancy and possibly correct these errors with triple redundancy. In addition, we are investigating an alternative method to enable redundant computing that has a lower resource overhead then what is presented here. Rather than having a replica specific to a particular rank we are looking into methods that would aggregate a number of replicated ranks onto one node and spread state throughout all of these aggregate replicas in a job. This would allow an aggregate replica, on demand, to replace a certain rank and could possibly allow the application to avoid checkpointing altogether.

References

[1] Michael Barborak, Anton Dahbura, and Miroslaw 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] Ron Brightwell, Sue P. Goudy, Arun Rodrigues, and Keith D. Underwood. Implications of application usage characteristics for collective communication offload. *Int. J. High Perform. Comput. Netw.*, 4(3/4):104–116, 2006.

[4] Ron Brightwell, Trammell Hudson, Kevin T. Pedretti, and Keith D. Underwood. SeaStar Interconnect: Balanced bandwidth for scalable performance. *IEEE Micro*, 26(3), May/June 2006.

[5] Greg Bronevetsky, Daniel J. Marques, Keshav K. Pingali, Radu Rugina, and Sally A. McKee. Compiler-enhanced incremental checkpointing for openmp applications. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 275–276, New York, NY, USA, 2008. ACM.

[6] William J. Camp and James L. Tomkins. Thor's hammer: The first version of the Red Storm MPP architecture. In *In Proceedings of the SC 2002 Conference on High Performance Networking and Computing*, Baltimore, MD, November 2002.

[7] Sayantan Chakravorty and Laxmikant V. Kalé. A fault tolerant protocol for massively parallel systems. In *Proceedings of the International Parallel and Distributed Processing Symposium*, Santa Fe, NM USA, April 2004. IEEE Computer Society Press.

[8] J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosiu, and A. Gupta. Hive: fault containment for shared-memory multiprocessors. In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 12–25, New York, NY, USA, 1995. ACM.

[9] Tzi-Cker Chiueh and Peitao Deng. Evaluation of checkpoint mechanisms for massively parallel machines. In *Annual Symposium on Fault Tolerant Computing*, pages 370–379, Sendai, Japan, June 1996. IEEE Computer Society Press.

[10] J. T. Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Gener. Comput. Syst.*, 22(3):303–312, 2006.

[11] Xiangyu Dong, Naveen Muralimanohar, Norm Jouppi, Richard Kaufmann, and Yuan Xie. Leveraging 3d pcram technologies to reduce checkpoint overhead for future exascale systems. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–12, New York, NY, USA, 2009. ACM.

[12] 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.

[13] E. N. Elnozahy, L. Alvisi, Y. M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375 – 408, SEP 2002.

[14] 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.

[15] Christian Engelmann, Hong H. Ong, and Stephen L. Scott. The case for modular redundancy in large-scale high performance computing systems. In *Proceedings of the* 8th *IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN) 2009*, pages 189–194, Innsbruck, Austria, February 16-18, 2009. ACTA Press, Calgary, AB, Canada.

[16] F. C. Gärtner. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Computing Surveys*, 31(1):1–26, March 1999.

[17] William Gropp. MPICH2: A new start for MPI implementations. In Dieter Kranzlmuller, 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] D. J. Kerbyson, H. J. Alme, Adolfy 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.

[23] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[24] Frank H. Mathis. A generalized birthday problem. *SIAM Review*, 33(2):265–270, June 1991.

[25] Dennis McEvoy. The architecture of tandem's nonstop system. In *ACM '81: Proceedings of the ACM '81 conference*, page 245, New York, NY, USA, 1981. ACM.

[26] 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.

[27] 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.

[28] Hewlett Packard. HP NonStop computing. `http://h20338.www2.hp.com/NonStopComputing/cache/76385-0-0-0-121.html`.

[29] Kevin T. Pedretti, Courtenay Vaughan, Karl Scott Hemmert, and Brian Barrett. Application sensitivity to link and injection bandwidth on a Cray XT4 system. In *Proceedings of the 2005 Cray User Group Annual Technical Conference*, Helsinki, Finland, May 2008.

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

[31] Steve J. Plimpton. Fast parallel algorithms for short-range molecular dynamics. *J Comp Phys*, 117(1):1–19, 1995.

[32] Rolf Riesen, Kurt Ferreira, and Jon Stearley. See applications run and throughput jump: The case for redundant computing in HPC. In *To Appear, 1st International Workshop on Fault-Tolerance for HPC at Extreme Scale (FTXS 2010)*, Chicago, IL, June 2010.

[33] Jose; Carlos Sancho, Fabrizio Petrini, Greg Johnson, Juan Fernandez, and Eitan Frachtenberg. On the feasibility of incremental checkpointing for scientific computing. *Parallel and Distributed Processing Symposium, International*, 1:58b, 2004.

[34] Sandia National Laboratory. LAMMPS molecular dynamics simulator. `http://lammps.sandia.gov`, Apr. 10 2010.

[35] Sandia National Laboratory. Mantevo project home page. `https://software.sandia.gov/mantevo`, Apr. 10 2010.

[36] Bianca Schroeder and Garth A Gibson. Understanding failures in petascale computers. *Journal of Physics: Conference Series*, 78(1):012022, 2007.

[37] Chen Yu, Du Zhi-Hui, Peng Liu, and Li San-Li. Os kernel supported fault tolerant mpi. *Journal of Shanghai University*, 5(SUP):18 – 21, 2001.

[38] Ziming Zheng and Zhiling Lan. Reliability-aware scalability models for high performance computing,. In *Cluster'09: Proceedings of the IEEE conference on Cluster Computing*, 2009.

DISTRIBUTION:

1	MS 1319	Kurt Ferreira , 1423
1	MS 1319	Ron Oldfield , 1423
1	MS 1319	Kevin Pedretti , 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 0359	D. Chavez, LDRD Office, 1911

Sandia National Laboratories