

# Transparent Redundant Computing with MPI

Ron Brightwell, Kurt Ferreira, and Rolf Riesen

Sandia National Laboratories\*  
Albuquerque, NM USA  
{rbbright,kbferre,rolf}@sandia.gov

**Key words:** MPI, fault tolerance, redundant computing, profiling interface

**Abstract.** Extreme-scale parallel systems will require alternative methods for applications to maintain current levels of uninterrupted execution. Redundant computation is one approach to consider, if the benefits of increased resiliency outweigh the cost of consuming additional resources. We describe a transparent redundancy approach for MPI applications and detail two different implementations that provide the ability to tolerate a range of failure scenarios, including loss of application processes and connectivity. We compare these two approaches and show performance results from micro-benchmarks that bound worst-case message passing performance degradation. We propose several enhancements that could lower the overhead of providing resiliency through redundancy.

## 1 Introduction

It is widely accepted that future extreme-scale parallel computing systems will require alternative methods to enable applications to maintain current levels of uninterrupted execution. As the component count of future multi-petaflops systems continues to grow, the likelihood of a failure impacting an application grows as well. Current methods of providing resiliency for applications, such as checkpoint/restart, will become ineffective, largely due to the overhead required to checkpoint, restart, and recover lost work. As such, the research community is pursuing several alternative approaches aimed at providing the ability for an application to survive in the face of failures and to continue to make efficient computational progress.

One of the fundamental approaches for masking errors and providing fault tolerance is redundancy. Replicating state and repeating operations occurs in many parts of modern computing systems; e.g., RAID has become commonplace. In order for redundancy to be viable for parallel computing, the potential performance degradation has to be offset by significant benefits. An important consideration for existing parallel computing systems and applications is the

---

\* 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-AC04-94AL85000.

amount of invasiveness that will be required to provide fault tolerance. Incremental approaches that minimize modifications to applications, system software, and hardware are more likely to be adopted.

We are exploring approaches to providing redundancy for MPI applications. We are seeking to answer several important research questions: a) Can we employ redundant computing with MPI transparently? b) What missing functionality, if any, is needed? c) What is the worst-case overhead? d) Are there any possible software or hardware enhancements that could reduce this overhead?

In this paper, we present two approaches for providing transparent redundancy for MPI applications. Both of these approaches double the number of processes in the application but use different schemes for recognizing failed processes and lost messages.

## 2 Implementation

We implemented redundant computing as a library that resides between an application and an MPI implementation. The *r*MPI library is described in detail in [1]. Here we provide only a brief description and highlight the parts which are relevant for the remainder of this paper.

### 2.1 Design choices

Future, large-scale machines where redundant computing may be of advantage [2] will have many nodes and will run MPI between these nodes to achieve the desired performance and scalability. Therefore, we implemented *r*MPI using the profiling interface of MPI. This provides us with portability across MPI implementations.

The second reason for implementing *r*MPI at the profiling layer is that we wanted to have a mechanism that is transparent to the application. Other than at job submission time when a user requests additional nodes for redundant computing, the application is not aware of the mechanism. It only sees, and interacts with, the active ranks and is unaware of the additional ranks and communication behind the scenes.

Using *r*MPI, we start an application on  $n \dots 2n$  nodes. During `MPI_Init()` we set up a new communicator for the first  $n$  active nodes and substitute that communicator whenever the application uses `MPI_COMM_WORLD`. Any nodes beyond  $n$  become redundant nodes for active nodes in a one-to-one fashion. Each redundant node performs the exact same computation as its active partner. The *r*MPI library ensures that it sees the same MPI behavior as the active node. That means if the active node is rank  $x$ , then the redundant node will also be rank  $x$ . The *r*MPI library performs the necessary translations to the actual ranks used by the underlying MPI library. Both nodes would send to rank  $y$ , even though there might be actually two nodes that have been assigned rank  $y$ .

Maintaining consistency for receives using `MPI_ANY_SOURCE` or `MPI_ANY_TAG` requires a consistency protocol between active and redundant nodes. For example,

MPI guarantees message order between node pairs, but *r*MPI must make sure that the message order seen on an active node is the same on its redundant node. Otherwise, computation on these two nodes could diverge.

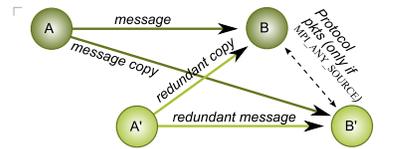
## 2.2 Mirror protocol

We started implementing *r*MPI with a straightforward protocol called *mirror*. As the name implies, it duplicates every message an application sends by transmitting it first to the original destination and then one more time to the destination’s redundant partner, if it exists.

Each receiver posts two receives into the same buffer for every application receive, if the sending node has a redundant partner that will also send. When nodes fail, *r*MPI is notified and stops sending to disabled nodes or posting receives for messages that will no longer be sent.

Figure 1 illustrates the process. As long as either the active node or its redundant partner are still alive, the application can continue. Only when both nodes in a bundle die, or a node without a redundant partner dies, will the application be interrupted and must restart.

Mirroring message order on two independent nodes in case of a `MPI_ANY_SOURCE` receive requires that only one node post the receive, and informs the second node of the actual receive order so it can post specific tag- and source-field receives to duplicate that order. If any `MPI_ANY_SOURCE` receives are pending, the second node must queue all subsequent receives, without letting the MPI implementation see them, until all current `MPI_ANY_SOURCE` receives have been satisfied.



**Figure 1.** In the mirror protocol each sender transmits the user messages twice and additional consistency protocol exchanges are needed in the case of `MPI_ANY_SOURCE`.

## 2.3 Parallel protocol

The mirror protocol consumes a lot of bandwidth when an application sends many large messages. To reduce this overhead we designed a second protocol named *parallel*. It is illustrated in Figure 2. Other than short protocol messages, *r*MPI only sends the original application messages between nodes. However, a larger number of protocol messages are now needed because the sender and its redundant partner must ensure that each of the destinations receive one copy of the message.

If one of the sender fails, the other must take over and send the additional copy. The parallel protocol somewhat resembles a transaction protocol where the two sending partners must ensure that both receivers get exactly one copy each of the application message. While this works well for large application messages where the overhead of a few additional short messages makes little difference, it

is a problem for applications which send a lot of short messages. In that case the message rate that can be achieved by the application is reduced.

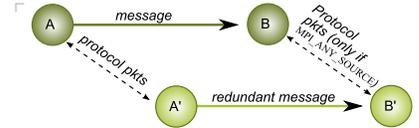
## 2.4 Issues

Initially we did not know whether transparent redundant computing could be achieved at the MPI level. We have shown [1] that it is possible, with relatively minor demands on the RAS system. Applications experience a performance impact that is in general less than 10%. Of course, micro-benchmarks clearly show the overhead present in the two protocols we have described.

While moving lower than MPI in the networking stack is not necessary, doing so may have advantages. The purpose of this paper is to further narrow the issues with our current prototype of *r*MPI and propose new solutions. To start, we list some issues that we have identified:

1. No redundant I/O. MPI-I/O and standard I/O are not currently handled by *r*MPI.
2. Missing integration with a RAS system. There is no standard way of doing this, but *r*MPI needs to know which nodes are alive, and the MPI implementation needs to survive the disappearance of individual nodes.
3. *r*MPI is an almost full re-implementation of the underlying MPI library.
4. Collective operations are reduced to point-to-point transmissions eliminating many of the optimization efforts performed by the underlying MPI library.
5. Mirroring message order on independent nodes for `MPI_ANY_SOURCE` receives causes consistency protocol overhead.
6. Delayed posting of `MPI_ANY_SOURCE` receives can increase the number of unexpected messages.
7. The mirror protocol consumes twice the bandwidth that the application needs and increases latency for small messages.
8. The parallel protocol is more frugal in its bandwidth consumption, but limits message rate.

In this paper we want to address items 3 through 8 and propose some ideas that could improve the performance of *r*MPI and limit some of its other shortcomings. In particular, we are interested in exploiting an intelligent network interface controller (NIC) and router designs to off-load some *r*MPI functionality. Furthermore, it would be interesting to design a solution that combines intra-node communication among the cores of a node with the necessary inter-node communication to reach redundant nodes which should be, for reliability purposes, physically as far away as possible inside the machine. The following sections describe our measurements and solutions.



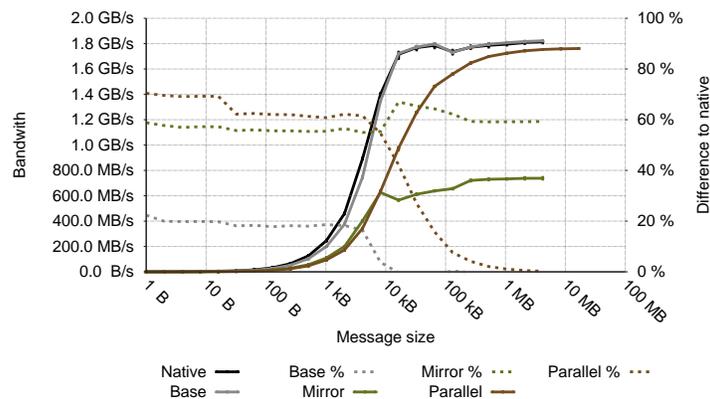
**Figure 2.** Message flow in the parallel protocol.

### 3 Results

In this section, we present the performance impact of our two protocols using a latency, bandwidth, and a message rate microbenchmark. Latency and bandwidth tests are from the OSU MPI benchmark suite(OMB) [3] while the message rate test is from the Sandia MPI microbenchmark suite. Due to the protocols special handling of `MPI_ANY_SOURCE`, we created another microbenchmark similar to the OMB latency test which uses wild-card receives.

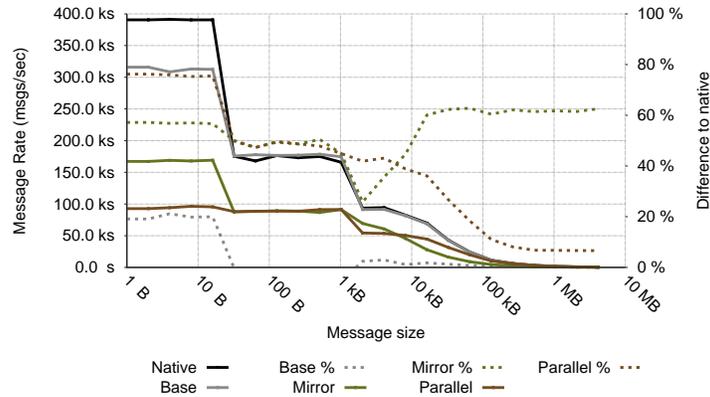
We conducted our tests on the Cray Red Storm system at Sandia National Laboratories. Each data points corresponds to the mean of five runs with error bars shown. In each of the following plots *native* refers to the performance of the benchmark without the *rMPI* library. *Base* for each of the protocols refers to performance with the *rMPI* library linked in but no redundant nodes used. The *parallel* and *mirror* lines are the performance of the application with a full set of replica nodes. Dashed lines show the overhead of keeping these replicas consistent.

Figures 3 and 4 illustrate the performance impact of the protocols on both bandwidth and message rate. Bandwidth in Figure 3 behaves as expected from the protocol descriptions in the previous section. The mirror protocol achieves about half of the observed bandwidth of native and the parallel protocol reaches nearly native bandwidth for large messages but for smaller messages the increased protocol message traffic hinders achievable bandwidth. Similarly, Figure 4 shows that for smaller messages mirror is able to achieve a higher message rate than parallel (with mirror’s rate around half of that of native), but as message size increases parallel’s rate approaches to within 10% of native.



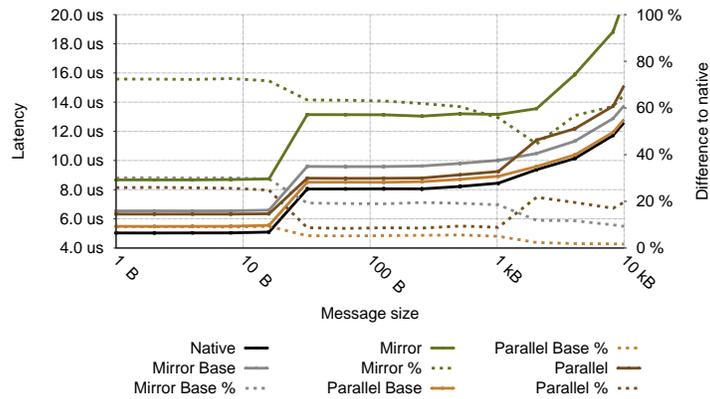
**Figure 3.** Bandwidth measurements for the two protocols compared to native and baseline. Native is the benchmark without *rMPI*, base has *rMPI* linked in but does not use redundant nodes.

We examine the protocols’ impact on application latency next. Figure 3 shows the results of the latency microbenchmark without `MPI_ANY_SOURCE` receives, and Figure 3 repeats the experiment with `MPI_ANY_SOURCE` receives. Again, latency for



**Figure 4.** Message rate measurements.

parallel is significantly lower than mirror. This is due to the fact that mirror must either wait for both messages to arrive or receive one of the messages and cancel the other before proceeding. Each of these operations require much more time than the one receive that the parallel protocol must wait for. Note in Figure 3 that this difference in performance between the two protocols is smaller when an `MPI_ANY_SOURCE` receive is posted. This is because the overhead is dominated by the consistency protocol to enforce message order on the redundant nodes.



**Figure 5.** Latency without `MPI_ANY_SOURCE` for the two protocols.

These microbenchmark results show that parallel is better for bandwidth and latency sensitive applications, and mirror has the advantage of having a higher achievable message rate for smaller messages and places less demand on RAS system functionality [1]. In either case, it is clear that the performance of both protocols could be improved with some additional support from both the underlying hardware and the MPI implementation.

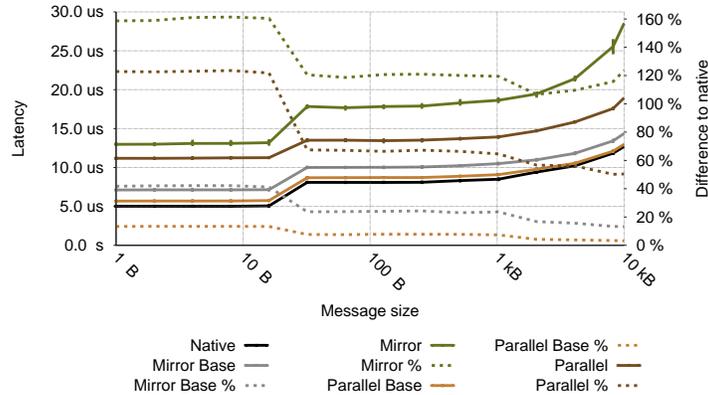


Figure 6. Latency with MPI\_ANY\_SOURCE for the two protocols.

## 4 Accelerating redundant computing

In Section 2.4 we identified issues with the current implementation of *r*MPI and in Section 3 we measured some additional properties that are affected by the *r*MPI implementation and the protocols it uses. We are now ready to address issue items 3 through 8 from our list on page 4.

### 4.1 Integrating *r*MPI into an MPI implementation

While implementing *r*MPI as a layer between the application and the MPI implementation allowed for quick prototyping, it has performance and code maintenance drawbacks that can be addressed by moving *r*MPI inside an existing MPI implementation. For example, this approach would allow for providing fully optimized collective operations.

### 4.2 Bandwidth and latency consumption

The mirror protocol sends each application message twice. Since the messages are identical, save for the different destinations, it would make sense to let the NIC duplicate the message and send two copies out, potentially reducing bandwidth consumption on the local NIC- to-memory connections. An even better approach would be to have the first router where the two message paths diverge, do this task. The message would have to be flagged as a redundant message and contain the address of, or routes to, both destinations. This mechanism would be an incremental increase in complexity inside a router or NIC. It would help the parallel protocol when it is operating in degraded mode.

Currently, *r*MPI operates below the collective operations and uses the MPI implementation underneath as a point-to-point transport layer. This leads to poor collective performance as can be seen in Figure 7. This figure shows the

consistency protocol overhead for a barrier operation for both mirror and parallel. Similar slowdown can be seen for other collective operations [1]. Instead, *r*MPI should make use of the provided and optimized collective operations, and, for example, use one broadcast operation to deliver data to all nodes – active and redundant. This would help both protocols to take advantage of topology optimized MPI features.

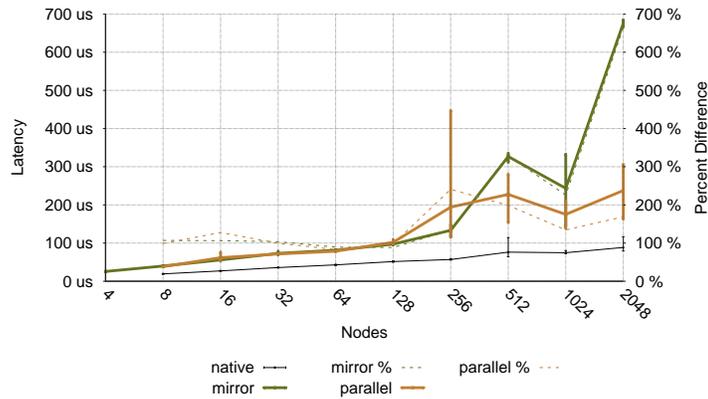


Figure 7. MPI\_Barrier() performance for the two protocols compared to native.

### 4.3 Message order semantics in case of MPI\_ANY\_SOURCE

Both protocols suffer when an application posts receives with the MPI\_ANY\_SOURCE or MPI\_ANY\_TAG marker. We cannot post the receive on the redundant node until we know in which order the message arrived at the active node. This causes overhead due to processing of unexpected messages. In addition, *r*MPI latency is degraded because of the protocol overhead to agree on a message order.

Implementation issues with MPI\_ANY\_SOURCE are well known and some researchers have advocated forbidding it, saying that well-written applications do not need it. *r*MPI can avoid most of the overhead when an application does not use MPI\_ANY\_SOURCE, since both nodes in a pair can detect the use of MPI\_ANY\_SOURCE. However, the fundamental problem remains for a fully compliant MPI implementation. (*r*MPI currently handles MPI\_ANY\_SOURCE properly, but does not support message receives with both MPI\_ANY\_SOURCE and MPI\_ANY\_TAG specified.)

One option we have not evaluated yet, is a modification to the parallel protocol. In Figure 2, instead of node *A*' sending the redundant message, node *B* could send a copy to node *B*'. That way node *B* could control the message order *B*' sees. A redundant node would get all of its messages from its active partner. When sending, a protocol is needed to let the redundant node know that the message has been sent and that the redundant node can skip the transmission. If the active receiver (*B* in the example) fails, the active sender (*A*) would transmit to the redundant receiver (*B*'). If the active sender (*A*) fails, the redundant sender (*A*') would start sending to the active receiver (*B*).

We have not implemented this variation of the parallel protocol because we believe it would not significantly improve performance, and few high-performance applications use `MPI_ANY_SOURCE` in the critical path.

#### 4.4 Use of one-sided operations

It might be possible to use one-sided operations to accelerate data delivery when `MPI_ANY_SOURCE` is used. This method would be useful for larger messages; temporary buffers and memory copies can be used for short messages. The active receiving node would inform the redundant receiver about the message order and let the redundant node get the data in the appropriate order using a pull protocol. This method would increase latency slightly, since the *get* request needs to be sent, and, depending on the architecture, a confirmation that the data has been picked up. This small overhead can be easily amortized for larger messages and the overhead to copy short messages into the correct user buffers is small.

## 5 Related work

Although redundancy is one of the fundamental approaches to masking errors and providing resiliency, it has not been extensively explored or deployed in high-performance computing (HPC) environments. Since HPC applications can scale to consume any of the available resources in a system (e.g., compute power, memory), the cost of duplicating resources for resiliency has been perceived as being too high – especially given the reliability levels of current large-scale systems. However, there are several characteristics of future systems that are motivating the community to explore alternative approaches to resiliency. Recently, redundant computation has been suggested as a possible path [4, 5] to resiliency, and the increasing probability of soft errors in future systems has also lead some to argue that higher levels of redundancy will also be needed [6].

There are several prior and ongoing research projects that are exploring resiliency and fault tolerance for MPI applications [7–11]. The MPI-3 Forum is also considering enhancements to the standard to enable fault-tolerant applications.

Most similar to *r*MPI is P2P-MPI [12, 13] which provides fault tolerance for grid applications through replication. In contrast to *r*MPI, P2P-MPI does not ensure consistency when wild-cards are used. In addition, P2P-MPI is tied to Java and requires a number of grid based services and protocols, which make this library inappropriate for an HPC environment. Furthermore, the failure analysis of P2P-MPI focuses on the probability of failures in the presence of replication, while in this work we focus on the impact of MTTI for the application which we believe is a more useful metric [2]. The approach we describe in this paper is very different from the other work on providing resiliency in the context of MPI applications, mostly due to the assumption that the cost of using resources for redundant computation will be acceptable for future large-scale systems.

There is some precedent for paying the resource cost of redundancy in high-performance computing. In [14] IBM describes a flow control protocol designed

to efficiently manage limited buffer space for MPI unexpected messages on their BG/L system. Even though the network is reliable, an acknowledgment-based flow control protocol is used to ensure that unexpected messages do not overflow the limited amount of available memory on a node. This protocol can potentially slowdown all applications, but the authors argue that a factor of two increase in runtime is acceptable: *“Nevertheless, the main conclusion is that the overhead is never more than twice the execution time without memory problems, which is not a high [sic] price to pay to make your application run without problems.”*

## References

1. Ferreira, K., Riesen, R., Oldfield, R., Stearley, J., Laros, J., Pedretti, K., Brightwell, R., Kordenbrock, T.: Increasing fault resiliency in a message-passing environment. Technical report SAND2009-6753, Sandia National Laboratories (2009)
2. Riesen, R., Ferreira, K., Stearley, J.: See applications run and throughput jump: The case for redundant computing in HPC. In: 1st International Workshop on Fault-Tolerance for HPC at Extreme Scale (FTXS 2010). (2010)
3. Network-Based Computing Laboratory, Ohio State University: OSU MPI benchmarks (OMB). <http://mvapich.cse.ohio-state.edu/benchmarks/> (2010)
4. Schroeder, B., Gibson, G.A.: Understanding failures in petascale computers. Journal of Physics: Conference Series **78**(1) (2007) 188–198
5. Zheng, Z., Lan, Z.: Reliability-aware scalability models for high performance computing. In: Proceedings of the IEEE conference on Cluster Computing. (2009)
6. He, X., Ou, L., Engelmann, C., Chen, X., Scott, S.L.: Symmetric active/active metadata service for high availability parallel file systems. Journal of Parallel and Distributed Computing (JPDC) **69**(12) (2009) 961–973
7. Fagg, G.E., Dongarra, J.: FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world. In: Proceedings of the 7th European PVM/MPI Users’ Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface. (2000) 346–353
8. Gropp, W., Lusk, E.: Fault tolerance in message passing interface programs. International Journal of High Performance Computing Applications **18**(3) (2004)
9. Bouteiller, A., Cappello, F., Herault, T., Krawezik, G., Lemarinier, P., Magniette, F.: MPICH-V2: a fault tolerant MPI for volatile nodes based on pessimistic sender based message logging. In: Proceedings of the ACM/IEEE International Conference on High Performance Computing and Networking. (2003)
10. Hursey, J., Squyres, J., Mattox, T., Lumsdaine, A.: The design and implementation of checkpoint/restart process fault tolerance for Open MPI. In: Proceedings of the IEEE International Parallel and Distributed Processing Symposium. (2007)
11. Santos, G., Duarte, A., Rexachs, D., Luque, E.: Providing non-stop service for message-passing based parallel applications with RADIC. In: Euro-Par. (2008)
12. Genaud, S., Rattanapoka, C.: P2P-MPI: A peer-to-peer framework for robust execution of message passing parallel programs on grids. J. Grid Comput. **5**(1) (2007) 27–42
13. Genaud, S., Jeannot, E., Rattanapoka, C.: Fault-management in P2P-MPI. Int. J. Parallel Program. **37**(5) (2009) 433–461
14. Farreras, M., Cortes, T., Labarta, J., Almasi, G.: Scaling MPI to short-memory MPPs such as BG/L. In: Proceeding of the International Conference on Supercomputing. (2006) 209–218