REDUNDANT EXECUTION OF HPC APPLICATIONS WITH MR-MPI

Christian Engelmann and Swen Böhm
Computer Science and Mathematics Division
Oak Ridge National Laboratory, Oak Ridge, TN, USA
email: engelmannc@ornl.gov, bohms@ornl.gov

ABSTRACT

This paper presents a modular-redundant Message Passing Interface (MPI) solution, MR-MPI, for transparently executing high-performance computing (HPC) applications in a redundant fashion. The presented work addresses the deficiencies of recovery-oriented HPC, *i.e.*, checkpoint/restart to/from a parallel file system, at extreme scale by adding the redundancy approach to the HPC resilience portfolio. It utilizes the MPI performance tool interface, PMPI, to transparently intercept MPI calls from an application and to hide all redundancy-related mechanisms. A redundantly executed application runs with $r * m$ native MPI processes, where r is the number of MPI ranks visible to the application and m is the replication degree. Messages between redundant nodes are replicated. Partial replication for tunable resilience is supported. The performance results clearly show the negative impact of the $O(m^2)$ messages between replicas. For low-level, point-to-point benchmarks, the impact can be as high as the replication degree. For applications, performance highly depends on the actual communication types and counts. On single-core systems, the overhead can be 0% for embarrassingly parallel applications independent of the employed redundancy configuration or up to 70-90% for communication-intensive applications in a dual-redundant configuration. On multi-core systems, the overhead can be significantly higher due to the additional communication contention.

KEY WORDS

high-performance computing, fault tolerance, resilience, redundancy, Message Passing Interface

1 Introduction

With petascale scientific computing systems in production and exascale systems on the horizon, computing centers at national laboratories are at the forefront of extreme-scale scientific computing. Ensuring that these leadership systems operate at an acceptable efficiency and productivity is of utmost importance for scientific discovery through computation. In order to deal with anticipated high failure rates, resilience has become an urgent priority for future-generation extreme-scale systems. The 2009/2010 Scientific Grand Challenges Workshop series (http://extremecomputing.labworks.org) and International Exascale Software Project workshops (http://www.exascale.org) identified several existing and emerging threats for the planned extreme-scale systems that require a departure from the business-as-usual approach for high-performance computing (HPC) resilience:

- The significant growth in component count, such as the up to 50× increase in compute nodes (1,000,000 nodes with 1,000 cores/node by 2018), results in a respectively higher system error rate.
- Smaller circuit sizes (22.5 nm half-pitch by 2016) as well as lower circuit voltages increase soft error vulnerability (bit flips caused by thermal and voltage variations as well as radiation).
- Dynamic power management cycling, such as to reduce the power consumption of underutilized cores, decreases component lifetimes due to thermal and mechanical stresses.
- Hardware fault detection and recovery, like error correcting code (ECC) for registers, cache and main memory, is limited by power consumption requirements and deployment costs (chip space needed).
- Heterogeneous architectures, *e.g.*, different core types on the same chip or in a co-processor setup, add more complexity to fault detection and recovery.

With a few exceptions, today's HPC systems assure fault tolerance in the same way since the early 1990s [3]. Application-level checkpoint/restart (C/R) to/from a parallel file system is the dominant HPC fault tolerance method for hard errors, while error correcting code (ECC) throughout the memory hierarchy is the prevalent HPC soft error resilience technique. System-level C/R to/from a parallel file system is employed at a few HPC centers, *e.g.*, using Berkeley Lab Checkpoint Restart (BLCR) [9]. However, none of the current petascale HPC centers do support system-level C/R. Diskless C/R has only recently been used in production HPC systems using the Scalable C/R (SCR) library at Lawrence Livermore National Laboratory, USA [2]. Message logging, algorithm-based fault tolerance, proactive fault tolerance, and Byzantine fault tolerance are not available in production HPC systems.

Recent reports [3, 5, 11] indicated that existing HPC fault tolerance technologies are insufficient for providing ***resilience***, *i.e.*, ***efficiency and correctness in the presence of failures***, at extreme scale. Ongoing efforts mainly focus on improving the capabilities of recovery-oriented computing by decreasing respective C/R overheads, such as through compute-node-local non-volatile memory (flash,

solid-state or phase-change technology). However, this comes at the cost of global synchronization for coordinated C/R or system noise in case of uncoordinated C/R utilizing message logging. Furthermore, there is currently no mitigation strategy to protect against silent data corruption (SDC) for end-to-end data integrity.

The presented work addresses the deficiencies of recovery-oriented HPC at extreme scale by adding the redundancy approach to the HPC resilience portfolio. By aiming at redundant computation instead of rollback recovery, this work offers a revolutionary, disruptive alternative to existing technologies and planned efforts. Redundancy has been used for decades to ensure uptime (availability) as well as correctness (reliability and data integrity) in information technology, aerospace, and command and control systems (see Section 2). The effort presented in this paper focuses on a software-only approach at the system software layer, *i.e.*, operating system and parallel programming environment, that is completely transparent to numerical libraries and scientific applications. It centers on redundancy at the Message Passing Interface (MPI) layer.

This paper is structured as follows. Section 2 discusses related work, including other efforts in redundancy for MPI applications. Section 3 illustrates the technical approach, while Section 4 details the implementation. Section 5 shows the results, and Section 6 summarizes the presented work and describes ongoing and planned efforts.

2 Related Work

2.1 Modular Redundancy

In general, modular redundancy is able to transparently mask reported errors, such as detected hard and soft errors, without recovery. It is also able to detect silent errors, like SDC, through comparison and recover them using majority voting if more than two replicas exist.

Historically, modular redundancy with specialized hardware has been used to ensure availability and reliability in information technology, aerospace and command & control [16, 17]. Software-only solutions that eliminate the need for expensive hardware, such as thread-level, process-level and state-machine replication, emerged recently [8, 13, 15, 4]. Another recent trend [10] embraces the use of cheap, off-the-shelf instead of expensive, certified components as the price gap is constantly expanding due to market forces (gaming and PC vs. special purpose) and increasing soft error vulnerability (expensive mitigation techniques).

A computing system generally consists of multiple components that can fail independently and even within a component there are multiple units that can fail. The sphere of replication [14] is the logical boundary of redundancy for a replicated system. Components within the sphere are protected, those outside are not. Data entering the sphere (input) is replicated, while data leaving the sphere (output) is compared. It can be as small as a few processor units and as large as a computing system. Independent from the extent of the sphere, input replication and output comparison may be performed at different granularities, *i.e.*, at different I/O interfaces. Instruction-level replication is the smallest and often referred to as lock stepping. Network communication replication is the largest and sometimes described as virtual synchrony [4].

2.2 Modular Redundancy in HPC

Our recent study [6] made the case for modular redundancy in large-scale HPC systems with an availability analysis. First, modular redundancy for single compute nodes was explored and then modular redundancy for all compute nodes was evaluated. The results show that static modular redundancy, *i.e.*, a failed node is not rebooted or replaced, allows lowering compute node mean-time to failure (MTTF) by a factor of 100-1,000 for dual redundancy and by 1,000-10,000 for triple redundancy without lowering overall system MTTF. In case of dynamic redundancy, where a failed node is rebooted or dynamically replaced with a hot spare, compute node MTTF can be lowered by a factor of 1,000-10,000 for dual redundancy and by 10,000-100,000 for triple redundancy.

Instead of a 7-nine node rating for a simplex exascale HPC system with 1,000,000 compute nodes, a dynamic dual-redundant system with 2,000,000 3-nine nodes or a dynamic triple-redundant system with 3,000,000 2-nine nodes provides 90% overall system availability. It is important to understand that the deployment costs for a single compute node at 2- and 3- nine rating and at 7-nine rating significantly differs, thus the cost of deploying more nodes can be recuperated by the cheaper per-node cost, as exemplified by other recent redundancy solutions [10].

2.3 Redundant Execution with rMPI

The System-Directed Resilience for Exascale Platforms project at Sandia National Laboratories recently completed an initial prototype for redundant execution of MPI applications, called rMPI [7]. The rMPI library gets inserted during link time between an application and the MPI library using MPI's profiling interface. rMPI transparently provides redundancy for MPI applications, which are started on up to $2n$ nodes. The application sees ranks $0 \ldots n-1$ and the rMPI library uses the remaining nodes for redundancy. Each redundant node is maintained by rMPI and duplicates the work of its active partner. When one fails, the other continues without interrupting the application. Only when both nodes fail, does the application need to restart.

Letting one MPI rank mirror another requires that it receives the same messages as the active node and that the code running on the mirror uses the same rank value as the active node. In order to preserve MPI semantics, the rMPI library coordinates and replicates messages between redundant nodes and forces an active node and its mirror to return the same values to the application for MPI function

calls, including those that probe for incoming messages, query the current time, or MPI_ANY_SOURCE receives. Without this, the computation on those two ranks could diverge from each other. The synchronization protocols and the additional messages incur overhead that is significant in low-level, point-to-point benchmarks.

The reported impact on actual applications is for the most part negligible. The overhead for LAMMPS is less than 4%, for SAGE less than 10%, for CTH less than 20% at 2,048 nodes, and for HPCCG less than 5%. The current implementation of rMPI requires more work to fully support all of MPI and needs to be better integrated with the reliability, availability and serviceability (RAS) and runtime system of the underlying machine. Recent enhancements to rMPI include partial replication, where not all MPI processes have replicas. Due to implementation problems, rMPI does not support certain complex MPI communicator operations, such as `MPI_Comm_split()`. rMPI also depends directly on the MPICH library for implementing collective communication operations, using rMPI's point-to-point communication primitives.

2.4 Redundant Execution with VolpexMPI

VolpexMPI [12], developed at the University of Houston, USA, is an MPI library implemented from scratch that offers redundancy for MPI process fault tolerance. It supports around 40 functions of the MPI-1 specification. In contrast to rMPI, VolpexMPI uses a polling mechanism by the receiver of point-to-point messages to avoid message replication. If a polled sender (of a replicated sender-receiver pair) does not exist due to a prior failure or does not respond in time due to performance degradation, a different sender (replica of the original sender) is chosen until the receive is successful. Messages are matched with a logical timestamp to allow for late message retrieval. MPI_ANY_SOURCE receives are not supported due to the missing synchronization mechanisms.

VolpexMPI can achieve close to 80% of the point-to-point message bandwidth of Open MPI. The small message latency increases from 0.5ms with Open MPI to 1.8ms with VolpexMPI. Using the NAS Parallel Benchmark (NPB) suite (http://www.nas.nasa.gov/Resources/Software/npb.html), there is no noticeable overhead for BT and EP for 8 and 16 processes. SP shows a significant overhead of 45% for 16 processes. The overhead for CG, FT and IS is considerably higher as these benchmarks are communication heavy.

2.5 Redundant Execution with MMPI

The recently proposed MMPI [18] offers a set of protocols for redundant execution of MPI applications with different replica partitioning and comparison schemes. It relies entirely on cumbersome source code modifications for implementing the redundancy protocols. It also has not yet been adequately evaluated.

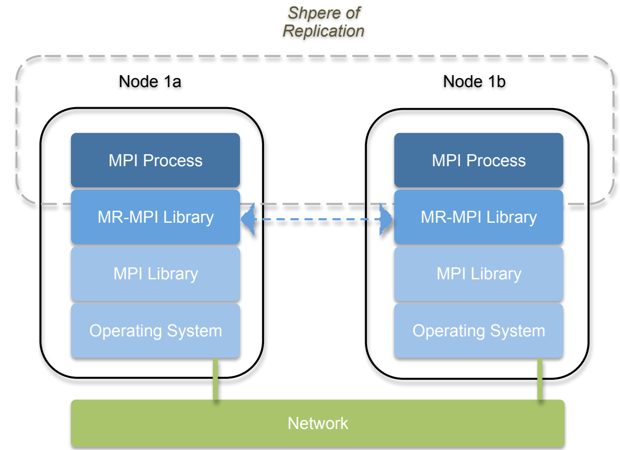

Figure 1. Architecture and sphere of replication for redundant execution of MPI applications with MR-MPI: Nodes 1a and 1b contain replicated MPI processes and non-replicated software stacks below

3 Technical Approach

Modular-redundant MPI, MR-MPI, targets MPI-level redundancy similar to rMPI and VolpexMPI to support redundant execution of MPI processes (a) on the same processor using different cores, (b) on different processors in the same compute node, and (c) on separate compute nodes. Input replication and output comparison is enforced at the application programming interface (API) layer between the MPI library and the application (Figure 1). The MR-MPI approach is similar to rMPI, such that the MPI itself becomes the I/O interface of the shpere of replication. Data entering the sphere, *i.e.*, MPI call returns and/or messages received into application-supplied buffers, are replicated to different MPI processes.

The current MR-MPI solution does not offer comparison of data leaving the sphere, such as MPI call arguments and/or messages sent from application-supplied buffers. However, it supports future extension for this feature. The fault model is fail-stop, *i.e.*, a faulty MPI process stops to communicate. More complex fault models, such as practical Byzantine fault tolerance covering unexpected deviation, are targeted in the future using more advanced mechanisms, such as output comparison.

In contrast to rMPI, MR-MPI does not rely on a specific MPI library, such as MPICH for rMPI. Instead, it is able to provide redundancy atop any MPI library. For for collective communication calls that involve calling user-defined operations, a recent addition to the MPI standard, `MPI_Reduce_local()`, needs to be supported by this MPI library. Also opposite to rMPI, MR-MPI does support certain complex MPI communicator operations, such as `MPI_Comm_split()`. MR-MPI distinguishes itself from VolpexMPI by not reimplementing an MPI layer. Experts spent years on optimizing MPI layers for efficiency. MR-MPI allows to utilize the native MPI library, which is typically optimized for a particular system.

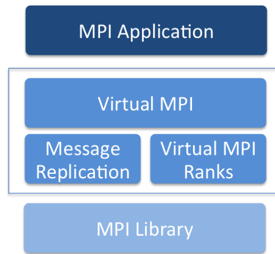

Figure 2. Architecture of the MR-MPI library: Sitting between the application and the MPI library, intercepting MPI calls and providing a virtual MPI

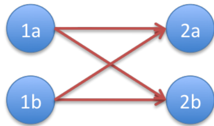

Figure 3. Message replication scheme of MR-MPI: All sender replicas send the same message to all receiver replicas to avoid recovery cycles in case of dropped messages

4 Implementation

To enable transparent redundant execution of MPI applications, MR-MPI utilizes the MPI performance tool interface, PMPI, to intercept MPI calls from the application and to hide all redundancy-related mechanisms (Figure 2). This is the same mechanism that rMPI uses to provide replication. The PMPI allows another library to override MPI library calls, which are defined as weak symbols, and to call the PMPI equivalents in the MPI library instead. The MPI standard defines that an MPI library has to provide weak `MPI_...()` calls and equivalent `PMPI_...()` calls to enable this feature. The MR-MPI library is written in C and provides 53 C and 53 Fortran MPI calls, *i.e.*, the MR-MPI library becomes the MPI layer of the application, while the native layer MPI is used by the MR-MPI library via the PMPI API. Since both, the application and the MR-MPI library, are compiled with and linked against the same MPI library, they are application binary interface compatible.

With MR-MPI, a redundantly executed application runs with $r * m$ native MPI processes, where r is the number of MPI ranks visible to the application (virtual MPI ranks) and m is the replication degree. To enable seamless operation without recovery cycles, all application communication is redundant (Figure 3), such that each of the m replicas of rank r_y sends a message to each of the m replicas of rank r_z. Application progress is guaranteed as long as at least one replica for each rank is alive. To provide compliance with the MPI standard, application startup with redundancy is supported through application command line arguments, *i.e.*, `mpirun -np <procs> <application> -mrmpi-np <vprocs>`. The `<vprocs>` parameter sets the number of virtual MPI ranks exposed to the application. MR-MPI automatically uses any additionally provided resources for redundancy.

Partial replication, *e.g.*, a replication degree of 1.3 or 2.3 instead of just 1, 2 or 3, for tunable resilience is supported. This permits the utilization of free resources to enhance resilience and to improve time to solution. For example, extreme-scale systems are typically managed by a batch job scheduler, which tends to schedule jobs at a certain efficiency. In cases, where there are unallocated resources available due to the number and size of submitted jobs, these free resources may be allocated with a job to enable redundant execution or to add even more replicas. This increases the resilience of a particular job and decreases its likelihood to fail and to be restarted at a later time.

The core of the MR-MPI library performs message replication for nonblocking point-to-point communication and offers the respective MPI primitives, `MPI_Isend()`, `MPI_Irecv()`, `MPI_Wait...()` and `MPI_Test...()`. For MPI point-to-point communication requests, MR-MPI does some additional book keeping as more than one native communication request may exist for each application initiated virtual MPI request due to the message replication scheme.

The blocking point-to-point and collective communication calls are based on these nonblocking point-to-point communication calls. A recent addition to the MPI standard, `MPI_Reduce_local()`, is used for collective communication calls that involve calling user-defined operations, such as `MPI_Reduce()`. `MPI_Reduce_local()` allows to execute a user-defined operation that is registered with the MPI library, such as the predefined `MPI_SUM` operation, without performing a collective communication operation. MR-MPI uses this feature to implement the collective communication calls atop its nonblocking point-to-point communication primitives. Also, persistent point-to-point communication requests, *e.g.*, using `MPI_Recv_init()` and `MPI_Request_free()`, are supported as well as replicated persistent point-to-point communication requests.

Specific consideration is needed in cases where there is the possibility that replicas may not receive the same input due to certain nondeterminism. Examples are `MPI_ANY_SOURCE` receives, `MPI_Wtime()`, `MPI_Test()`, and `MPI_Probe()`. These cases are related to a message delivery order on the network or to local unsynchronized time. An internal master with fail-over mechanism is employed that performs the operation on the lowest ranking replica and informs the higher ranking replicas about the result. The higher ranking replicas then either return the respective result, such as for `MPI_Wtime()`, or execute the operation in a deterministic way to get the same result, such as for `MPI_ANY_SOURCE` receives. In case of a failure, the next replica in-line is performing the operation and informs its higher ranking replicas about the result.

The MR-MPI library has its own internal book keeping for MPI groups and MPI communicators as these differ between the underlying MPI library and the virtual MPI exposed to the application. MR-MPI translates between the native and virtual MPI ranks, such that for

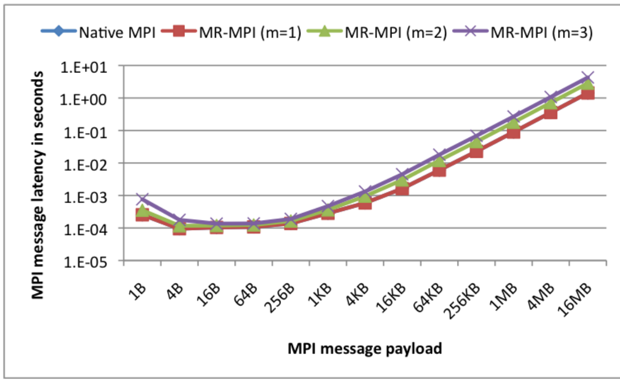

Figure 4. Point-to-point latency performance

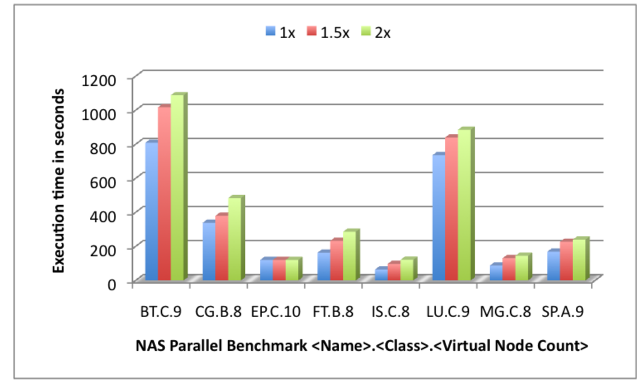

Figure 6. NPB performance

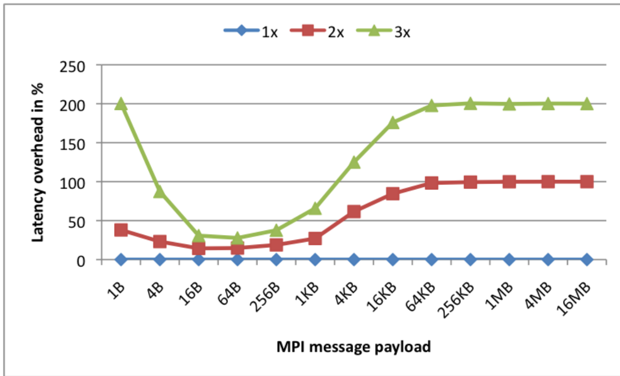

Figure 5. Normalized point-to-point latency performance

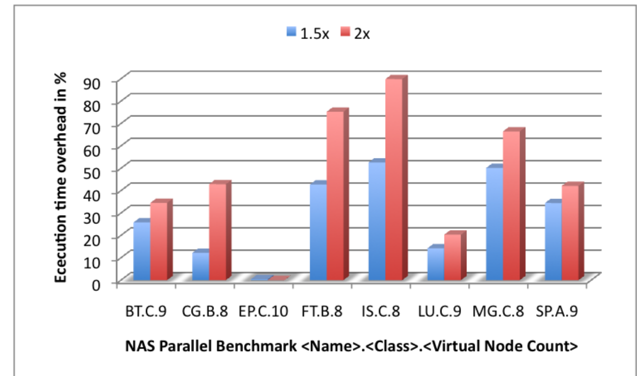

Figure 7. Normalized NPB performance

each native rank there is a virtual/replica rank pair in the respective group or communicator. In order to be able to utilize the native MPI's communicator and group operations, the ranks are always organized in a predetermined order: $1_1 \ldots (r-1)_1, \ldots, 1_m \ldots (r-1)_m$. This allows the MR-MPI library to simply map ranks, even in the case of partial replication, and to implement its own MPI communicator and group operations on top of the native MPI's communicator and group calls. Rather complex operations, such as `MPI_Comm_split()` and `MPI_Group_intersection()` are supported through this mechanism, again, even for partial replication, where certain replicas do not exist.

5 Results

The developed MR-MPI prototype was deployed on an older 32-node Intel-based Linux cluster. Each node has an Intel Pentium 4 2GHz processor and 0.75GB of memory. The system is running the Ubuntu 8.04 32-bit Linux distribution without swap and Open MPI 1.5.rc3 with the recently added `MPI_Reduce_local()` support. All nodes are connected via Fast Ethernet (100Mbps). Excluding the fact that generally more resources are used in a replication scenario, the performance of redundant MPI application execution with MR-MPI is highly influenced by the $O(m^2)$ messages between replicas.

In the first experiment, MR-MPI's point-to-point message latency performance was evaluated. Figures 4 and 5 clearly show the impact of the message replication scheme on point-to-point message latency performance. As already discovered with rMPI, the impact is rather significant. For small messages (16B-1kB), the latency is lower due to the fact that MR-MPI uses nonblocking communication. For all other messages (<16B,>1kB), network contention occurs on the node network cards caused by the replicated message send and receive operations. Note that this latency increase is also additionally influenced by the native MPI's switch from the eager to the rendezvous protocol for bigger messages, which adds an additional contention penalty at the native MPI library for the message replication.

In a second experiment, the NAS Parallel Benchmark (NPB) [1] suite was run to measure the overhead introduced by MR-MPI with different benchmarks. The NPB benchmarks with class A, B, or C problem size were executed at a replication degree of 1, 1.5 (partial replication at 50%) and 2, and averaged over 10 test runs. The classes and number of nodes differ for the benchmarks due to individual processor count (square vs. power of two) and memory requirements. Figures 6 and 7 clearly show the different impact MR-MPI has on the various NPB benchmark applications. The embarrassingly parallel (EP) benchmark did not had any overhead under any redundancy configuration, while the communication-heavy integer sort (IS) and fast

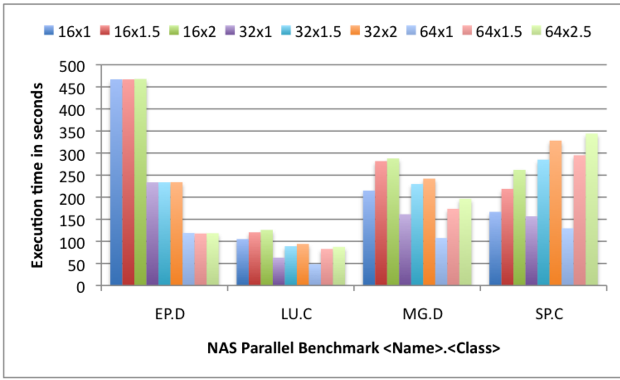

Figure 8. NPB core-scaling performance

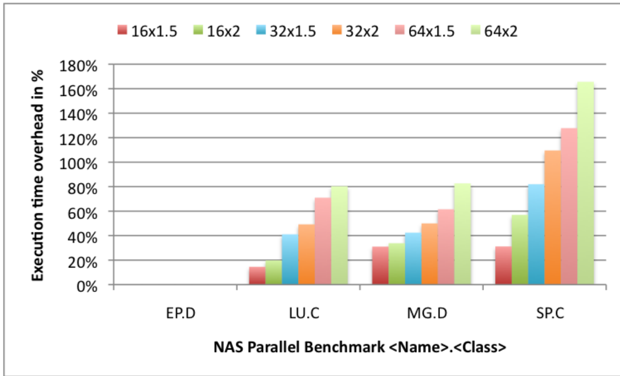

Figure 9. Normalized NPB core-scaling performance

Fourier transform (FT) benchmarks have an enormous 70-90% overhead under dual-modular redundancy. Again, this overhead is caused by the $O(m^2)$ messages between replicas. In terms of MR-MPI functionality, it is important to note that FT, BT and SP use `MPI_Comm_split()`, and BT and SP use `MPI_Comm_dup()`. These communicator operations are supported in MR-MPI.

The developed MR-MPI prototype was also deployed on a new 16-node AMD-based Linux cluster. Each node has two 4-core AMD Opteron 2378 2.4GHz processors and 8GB of memory. The system is running the Ubuntu 10.04 64-bit Linux distribution with swap and Open MPI 1.5 with the `MPI_Reduce_local()` support. All nodes are connected via a non-blocking Gigabit Ethernet switch.

In the third experiment, a subset of the NPB suite was run to measure the overhead introduced by MR-MPI with different benchmarks on multi-core systems. The class C NPB benchmarks LU and SP, and the class D NPB benchmarks EP and MG were executed at a replication degree of 1, 1.5 (partial replication at 50%) and 2, and averaged over 10 test runs. The benchmark classes are different from the prior experiment as this system has much better computation and communication capabilities. The benchmarks were executed on all 16 nodes using 2, 4, and 8 cores, such that the lower 8 nodes executed one entire replica and the upper 8 nodes executed another half or full replica. Due to the non-blocking nature of the Gigabit Ethernet switch,

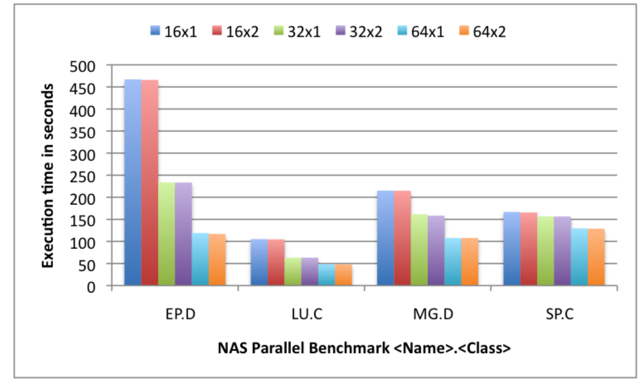

Figure 10. NPB performance with simplistic redundancy

all introduced overhead stems from the redundancy protocol. Figures 8 and 9 clearly show the different impact the MR-MPI redundancy protocol has on the NPB benchmark subset with core scaling. They also show that with increasing per-node core counts, the overhead is increasing as well. The only exception is the embarrassingly parallel (EP) benchmark, which did not had any overhead under any redundancy configuration and scaled perfectly.

There are two interesting aspects obtained from this experiment. First, the non-redundant LU, MG and SP benchmarks do not scale perfectly with this system setup, *i.e.*, they are communication bound. Second, all MPI processes on the same node share a single network card, which causes local contention. In both cases, any additional communication introduced by a redundancy protocol likely results in a significant performance degradation.

In a last experiment, the MR-MPI replication protocol was modified to execute MPI applications independently from each other, *i.e.*, messages are not sent to all replicas, just to the intended target. This protocol has two disadvantages: (1) processes are not actively replicated, and (2) partial replication is not supported. The missing active replication causes any single MPI process failure to stall the entire execution of one out of the set of many redundantly run MPI applications. The missing partial replication results in coarse-grain resource utilization. The third experiment was re-executed with the modified MR-MPI. The results in Figure 10 show that there is no performance overhead at all as the network switch is non-blocking and replication is performed on different resources without coordination.

6 Summary and Future Work

This paper presents the implementation details and performance results of a recently developed modular-redundant MPI solution, MR-MPI. The work addresses the deficiencies of recovery-oriented HPC, *i.e.*, checkpoint/restart to/from a parallel file system, at extreme scale by adding the redundancy approach to the HPC resilience portfolio. Redundancy has been used for decades to ensure uptime (availability) as well as correctness (reliability and data integrity) in other areas, typically for single services. Re-

cent efforts in redundancy for HPC focused on an availability analysis that convincingly makes the case for redundant computation in HPC environments and on two prototypes, rMPI and VolpexMPI. The presented solution, MR-MPI, follows the rMPI model and architecture by utilizing the MPI performance tool interface, PMPI, to intercept MPI calls from the application and to hide all redundancy-related mechanisms, and by replicating messages between redundant nodes. In contrast to rMPI, MR-MPI does not rely on a specific MPI library, such as MPICH for rMPI. It also supports certain complex MPI communicator operations, such as `MPI_Comm_split()`, which rMPI does not. MR-MPI distinguishes itself from VolpexMPI by not reimplementing the MPI layer.

The performance results clearly show the negative impact of the $O(m^2)$ messages between replicas. For low-level, point-to-point benchmarks, the impact can be as high as the replication degree, *i.e.*, message latency can be three times as much in a triple-redundant configuration. For application benchmarks, such as the NAS Parallel Benchmark suite, performance highly depends on the communication types and counts of a particular benchmark. For the embarrassingly parallel (EP) benchmark the overhead is 0%, while the communication-heavy integer sort (IS) and fast Fourier transform (FT) benchmarks experience 70-90% overhead in a dual-redundant configuration on single-core systems. On multi-core systems, the overhead can be significantly higher due to the additional communication contention.

The presented work is only a first step toward redundancy in HPC environments. For MR-MPI, planned improvements include removing the costly message replication scheme and implementing a VolpexMPI-like message caching without reimplementing the MPI layer. As mentioned earlier, MR-MPI improvements will also focus on output comparison to support more complex fault models, such as to detect SDC. Other planned work focuses on combining redundancy with checkpoint/restart to cover faults where all replicas fail and to evaluate the synergistic effect. The key challenge here is to provide seamless resilience capabilities and to integrate the two orthogonal resilience schemes. In conjunction with this effort, redundant checkpoints and checkpoint comparison for SDC detection will be considered as well.

7 Acknowledgements

Research sponsored by the Laboratory Directed Research and Development Program of Oak Ridge National Laboratory (ORNL), managed by UT-Battelle, LLC for the U. S. Department of Energy under Contract No. De-AC05-00OR22725.

This manuscript has been authored by UT-Battelle, LLC, under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes.

References

[1] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications*, 5(3): 63–73, Fall 1991. URL citeseer.ist.psu.edu/article/bailey94nas.html.

[2] G. Bronevetsky and A. Moody. Scalable I/O systems via node-local storage: Approaching 1 TB/sec file I/O. Technical Report TR-JLPC-09-01, Lawrence Livermore National Laboratory, Livermore, CA, USA, Aug. 2009. URL http://dx.doi.org/10.2172/964079.

[3] N. DeBardeleben, J. Laros, J. T. Daly, S. L. Scott, C. Engelmann, and B. Harrod. High-end computing resilience: Analysis of issues facing the HEC community and path-forward for research and development. Whitepaper, Dec. 2009. URL http://www.csm.ornl.gov/~engelman/publications/debardeleben09high-end.pdf.

[4] D. Dolev and D. Malki. The Transis approach to high availability cluster communication. *Communications of the ACM*, 39(4):64–70, 1996. ISSN 0001-0782. URL http://doi.acm.org/10.1145/227210.227227.

[5] E. N. M. Elnozahy, R. Bianchini, T. El-Ghazawi, A. Fox, F. Godfrey, A. Hoisie, K. McKinley, R. Melhem, J. S. Plank, P. Ranganathan, and J. Simons. System resilience at extreme scale. Technical report, Defense Advanced Research Project Agency (DARPA), 2008. URL http://institutes.lanl.gov/resilience/docs/Toward%20Exascale%20Resilience.pdf.

[6] C. Engelmann, H. H. Ong, and S. L. Scott. The case for modular redundancy in large-scale high performance computing systems. In *Proceedings of the 27th IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN) 2009*, pages 189–194, Innsbruck, Austria, Feb. 16-18, 2009. ACTA Press, Calgary, AB, Canada. ISBN 978-0-88986-784-0. URL http://www.csm.ornl.gov/~engelman/publications/engelmann09case.pdf.

[7] K. Ferreira, R. Riesen, R. Oldfield, J. Stearley, J. Laros, K. Pedretti, R. Brightwell, and T. Kordenbrock. Increasing fault resiliency in a message-passing environment. Technical report SAND2009-6753, Sandia National Laboratories, Oct. 2009.

[8] A. Golander, S. Weiss, and R. Ronen. DDMR: Dynamic and scalable dual modular redundancy with short validation intervals. *IEEE Computer Archi-*

tecture Letters, 7(2):65–68, 2008. URL http://doi.ieeecomputersociety.org/10.1109/L-CA.2008.12.

[9] P. H. Hargrove and J. C. Duell. Berkeley Lab Checkpoint/Restart (BLCR) for Linux clusters. In *Journal of Physics: Proceedings of the Scientific Discovery through Advanced Computing Program (SciDAC) Conference 2006*, volume 46, pages 494–499, Denver, CO, USA, June 25-29, 2006. Institute of Physics Publishing, Bristol, UK. URL http://www.iop.org/EJ/article/1742-6596/46/1/067/jpconf6_46_067.pdf.

[10] C. A. Hulme, H. H. Loomis, A. A. Ross, and R. Yuan. Configurable fault-tolerant processor (CFTP) for spacecraft onboard processing. In *Proceedings of the IEEE Aerospace Conference 2004*, volume 4, pages 2269– 2276, Big Sky, MT, USA, Mar. 6-13, 2002. IEEE Computer Society. ISBN 0-7803-8155-6. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1368020.

[11] P. Kogge, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, and K. Yelick. ExaScale computing study: Technology challenges in achieving exascale systems. Technical report, Defense Advanced Research Project Agency (DARPA) Information Processing Techniques Office (IPTO), 2008. URL http://users.ece.gatech.edu/~mrichard/ExascaleComputingStudyReports/exascale_final_report_100208.pdf.

[12] T. LeBlanc, R. Anand, E. Gabriel, and J. Subhlok. Volpexmpi: An MPI library for execution of parallel applications on volatile nodes. In *Lecture Notes in Computer Science: Proceedings of the* 16th *European PVM/MPI Users‘ Group Meeting (EuroPVM/MPI) 2009*, volume 5759, pages 124–133, Espoo, Finland, Sept. 7-10, 2009. Springer Verlag, Berlin, Germany. ISBN 978-3-540-75415-2. URL http://dx.doi.org/10.1007/978-3-642-03770-2_19.

[13] N. Madan and R. Balasubramonian. Power efficient approaches to redundant multithreading. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 18(8):1066–1079, 2007. ISSN 1045-9219. URL http://doi.ieeecomputersociety.org/10.1109/TPDS.2007.1090.

[14] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt. Detailed design and evaluation of redundant multithreading alternatives. In *Proceedings of the* 29th *Annual International Symposium on Computer Architecture (ISCA) 2002*, pages 99–110, Anchorage, AK, USA, May 25-29, 2002. IEEE Computer Society. ISBN 0-7695-1605-X. URL http://doi.ieeecomputersociety.org/10.1109/ISCA.2002.1003566.

[15] A. Shye, J. Blomstedt, T. Moseley, V. J. Reddi, and D. A. Connors. PLR: A software approach to transient fault tolerance for multicore architectures. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 6(2):135–148, 2009. ISSN 1545-5971. URL http://doi.ieeecomputersociety.org/10.1109/TDSC.2008.62.

[16] D. P. Siemwiorek. Architecture of fault-tolerant computers: An historical perspective. *Proceedings of the IEEE*, 79(12):1710–1734, 1991. ISSN 0018-9219. URL http://dx.doi.org/10.1109/5.119549.

[17] A. Wood, R. Jardine, and W. Bartlett. Data integrity in HP nonstop servers. In *Proceedings of the* 2nd *Workshop on System Effects of Logic Soft Errors (SELSE) 2006*, Urbana-Champaign, IL, USA, Apr. 11-12, 2006. URL http://selse2.selse.org/papers/wood.pdf.

[18] X. Yang, Z. Wang, and Y. Zhou. MMPI: A scalable fault tolerance mechanism for MPI large scale parallel computing. In *Proceedings of the* 10th *IEEE International Conference on Computer and Information Technology (CIT) 2010*, pages 1251–1256, Bradford, UK, June 29 - July 1, 2009. IEEE Computer Society. ISBN 978-0-7695-4108-2. URL http://doi.ieeecomputersociety.org/10.1109/CIT.2010.226.