File I/O for MPI Applications in Redundant Execution Scenarios

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

***Abstract*—As multi-petascale and exa-scale high-performance computing (HPC) systems inevitably have to deal with a number of resilience challenges, such as a significant growth in component count and smaller circuit sizes with lower circuit voltages, redundancy may offer an acceptable level of resilience that traditional fault tolerance techniques, such as checkpoint/restart, do not. Although redundancy in HPC is quite controversial due to the associated cost for redundant components, the constantly increasing number of cores-per-processor is tilting this cost calculation toward a system design where computation, such as for redundancy, is much cheaper and communication, needed for checkpoint/restart, is much more expensive. Recent research and development activities in redundancy for Message Passing Interface (MPI) applications focused on availability/reliability models and replication algorithms. This paper takes a first step toward solving an open research problem associated with running a parallel application redundantly, which is file I/O under redundancy. The approach intercepts file I/O calls made by a redundant application to employ coordination protocols that execute file I/O operations in a redundancy-oblivious fashion when accessing a node-local file system, or in a redundancy-aware fashion when accessing a shared networked file system. A proof-of concept prototype is presented and a number of coordination protocols are described and evaluated. The results show the performance impact for redundantly accessing a shared networked file system, but also demonstrate the capability to regain performance by utilizing MPI communication between replicas and parallel file I/O.**

***Keywords*-high-performance computing; fault tolerance; resilience; redundancy; Message Passing Interface;**

I. Introduction

Resilience in computing systems, i.e., providing efficiency and correctness of a in the presence of faults, errors, and failures, can be achieved through avoidance, masking, and recovery. Redundancy is a particular resilience method that provides masking of unexpected system behavior, such as failures, through the use of multiple redundant systems and a fail-over mechanism, in case of active/standby, or a voting technique, in case of active/active. While redundancy has been used in mission critical systems for decades [1], it has not been used in parallel and distributed systems due to the associated costs for redundant hardware. With the constantly increasing number of cores per processor, it may be time to reconsider redundancy as an alternative resilience approach.

Today's high-performance computing (HPC) systems assure resilience against hard errors in the same way since the early 1990s using application-level checkpoint/restart to/from a parallel file system [2]. Similarly, they ensure resilience against soft errors in the same way since the early 1980s using error correcting code (ECC) throughout the memory hierarchy [2]. The path of technological progression toward multi-petascale and exa-scale scientific computing systems involves a number of challenges [3], [4], [5], [6]. In terms of resilience, these challenges include the significant growth in component count (1,000,000 nodes with 1,000 cores/node for an exascale system [7]) and the resulting higher hard error rates, as well as, smaller circuit sizes (22.5 nm) in conjunction with lower circuit voltages and the resulting higher soft error rates. As these error rates increase, today's HPC resilience technologies may not be able to provide adequate correctness at reasonable efficiency, i.e., an acceptable level of resilience [2], [8], [9].

Redundancy, although quite controversial, may offer an acceptable level of resilience in an extreme-scale HPC system. Recent investigations (see Section II) focused on models and early implementations of redundancy solutions for Message Passing Interface (MPI) applications. While the models targeted an analysis of the impact in terms of overhead increase and resilience gain in comparison to traditional checkpoint/restart, the implementations aimed at exploring different redundancy algorithms and various positions of the replication layer in the software stack. All of this prior work relies on state machine (or active) replication [10] for the communicating processes of a parallel application. Each process of the application, or only a subset, is involved in a communication (message) replication algorithm requiring deterministic behavior of the parallel application.

As redundancy for MPI applications is an ongoing research topic, this paper takes a first step toward solving certain open research problems associated with running a parallel application redundantly. Apart from the complexities of process replication itself, interfacing replicated processes that represent a single redundant entity with the outside world adds another level of complexity. This includes user interaction at the command line interface, as well as, file input/output (I/O) to/from a file or storage system.

This paper presents an initial solution for file I/O in

redundant parallel application scenarios. It considers the two common use cases in HPC: (1) file I/O to a node-local file system, such as for temporary files, and (2) file I/O to a shared file system, such as for input/output data. The approach interposes Portable Operating System Interface for Unix (POSIX) file I/O system calls between the MPI application and the operating system (OS) to intercept file I/O operations invoked by the parallel application and to implement them in a redundancy-aware fashion if needed. The implementation offers different algorithms for redundant or unified file I/O that are in part able to take advantage of low-latency/high-bandwidth MPI communication between redundant processes to improve file I/O performance.

This paper is structured as follows. Section II discusses related work in redundancy for MPI applications. Section III illustrates the approach and Section IV provides design details of the proof-of-concept implementation. Section V shows the results obtained from an experimental study, while Section VI concludes the paper with a short summary of the presented work and a brief discussion of future work.

II. Related Work

rMPI [11] is a prototype for redundant execution of MPI applications. It is a library that gets inserted during link time between an application and the MPI library using MPI's profiling interface (PMPI). Using rMPI, an MPI application is started on up to $2n$ nodes and sees ranks $0 \ldots n-1$. rMPI transparently provides redundancy using the remaining nodes. It maintains each redundant node and duplicates the work of its active partner. In case of a failure, the redundant node continues without interruption. The application fails only when two corresponding replicas fail. The synchronization protocols and the additional messages incur overhead that is significant in low-level, point-to-point benchmarks. The 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 modular-redundant MPI (MR-MPI) [12] is a similar solution for transparently executing HPC applications in a redundant fashion. It also utilizes PMPI to transparently intercept MPI calls from an application and to hide all redundancy-related mechanisms. In MR-MPI, messages are replicated between redundant nodes and partial replication for tunable resilience is supported. The results 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., 100% overhead for dual and 200% for triple redundancy. In realistic scenarios, the overhead can be 0% for embarrassingly parallel or up to 70-90% for communication-intensive applications in a dual-redundant configuration.

RedMPI [13], is a recent effort that combines the achievements of the rMPI and MR-MPI projects, is additionally capable of silent data corruption (SDC) online detection and correction without requiring any changes to scientific application source code. Using a more efficient replication mechanism, redMPI transparently detects corrupt MPI messages originating from an SDC-affected MPI process. While a detected SDC event results in an application abort (and possible restart from a non-corrupted checkpoint) under dual redundancy, redMPI further offers SDC correction using a voting mechanism under triple redundancy.

VolpexMPI [14] is an MPI library implemented from scratch that offers redundancy internally. It uses a polling mechanism by the receiver of point-to-point messages to avoid message replication. If a polled sender fails to respond, a different sender is chosen until the receive is successful. Messages are matched with a logical timestamp to allow for late message retrieval. VolpexMPI achieves close to 80% of Open MPI's point-to-point message bandwidth, while the latency of small message increases from 0.5ms to 1.8ms. Using the NAS Parallel Benchmark [15], there is no 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 feature intense communication.

None of these redundant MPI approaches consider file I/O during redundant execution with one small exception. VolpexMPI does provide the capability for one replica to perform file I/O writes, while the other replicas skip this operation. Intuition suggests that performance gains may be achieved by rotating the responsibility to write out from one replica to the next, while letting the non-writing replicas to go ahead. However, these performance gains may be easily negated by synchronizing replication protocols and collective MPI communication.

III. Technical Approach

File I/O in redundant computing scenarios needs to deal with the fact that there are multiple replicas of the same process and that each of them may fail at any point in time. The first challenge is do devise appropriate algorithms to either unify file I/O by redundant processes, such that only one file is read or written, or to fully support redundant file I/O, such that each replica reads in the same non-replicated or its corresponding replicated file (as multiple files) and writes out a replicated file (as multiple files). The second challenge involves fault tolerance, i.e., compensating for the loss of a replica outside and during file I/O operations.

HPC system file I/O typically involves the compute-node local file system, such as for temporary files, and a shared networked file system, e.g., for input/output data. The use case in a replicated execution scenario in both instances is completely different. While the node-local file I/O approach operates completely oblivious to any employed redundancy, shared networked file I/O needs to consider the implications

of redundant processes accessing the same shared file system. In addition to a redundancy-aware networked file I/O that resolves redundant file system access, a redundancy-aware file I/O scheduling algorithm is potentially able to improve performance by utilizing inter-node communication in addition to networked file I/O.

The technical approach taken in the presented work is applicable to any of the described redundant MPI solutions (see Section II). The core assumption is that there is a set of replicated processes that needs to perform node-local or shared network file I/O. The presented solution is independent from the replication mechanism. It is assumed that all replicated processes are deterministic and execute exactly the same POSIX file I/O operations. Our solution either executes these file I/O operations in a redundancy-oblivious fashion when accessing a node-local file system, or in a redundancy-aware fashion employing a coordination protocol when accessing a shared networked file system. In order to take advantage of low-latency/high-bandwidth communication between redundant processes, MPI communication is used for the coordination protocol and optionally for data distribution in the more advanced redundancy-aware file I/O protocols. MPI-IO, a file I/O layer with MPI support for parallel file system access, is not used or supported at this point due to the inherent complexities involving MPI-IO and missing MPI-IO fault tolerance capabilities.

IV. Design

The general design of the presented solution relies on a library that interposes POSIX file I/O system calls, like `open()`, `read()`, `write()` and `close()`, such that these calls are intercepted by the library when coming from an application, execute the needed coordination protocol, and perform related file I/O using the original POSIX system calls. The mechanism to intercept the system calls may rely on (1) compile-time redirection using `#define` preprocessor statements in code parsed by the C preprocessor, (2) run-time redirection by symbol overloading via a shared library using the `LD_PRELOAD` mechanism of the dynamic linker, or (3) run-time redirection by symbol overloading via the `-Wl,--wrap` option of the GNU linker (not available for Mac OS due to BSD linker). The implemented prototype uses `-Wl,--wrap` whenever possible and provides support for the other methods as fall-back.

The redundant file I/O library interfaces with a redundant MPI layer as it requires certain runtime information, such as the redundancy degree and replica mapping, to execute the coordination protocol among replicas. The implemented prototype uses MR-MPI (see Section II) for running MPI applications redundantly as the authors where granted access to the source code by the MR-MPI developers. MR-MPI itself was not modified, instead the redundant file I/O library accesses data structures and basic functions within MR-MPI to identify the redundancy degree and replica mapping. The experimental evaluation of the implemented proof-of-concept prototype (see Section V) uses the latest MR-MPI version, before it was merged with rMPI into redMPI (see Section II). Modifying the redundant file I/O library to use a different redundant MPI layer, such as rMPI, redMPI or VolpexMPI, is relatively simple as each redundancy solution needs to maintain this information internally.

Since the POSIX file I/O calls use file descriptors as handles, the redundant file I/O library maintains a mapping between "virtual" file descriptors given out to the application and actual file descriptors used by the library for file I/O. Note that depending on the coordination protocol (explained in the following) a replica may not have an actual file descriptor associated with each "virtual" file descriptor as the data may come from and/or go to a different replica. The redundant file I/O library also supports the `f...()` file I/O calls, such as `fopen()`, `fread()`, `fwrite()` and `fclose()`. The corresponding `FILE` handle is dealt with in a similar way. The redundant file I/O library does recognize non-redundant descriptors/handles and executes the corresponding system call without invoking a coordination protocol in a pass-through fashion.

A. Coordination Protocols

In the following, individual coordination protocols for redundancy-oblivious and redundancy-aware file I/O are described. Each protocol has its own advantages and disadvantages in terms of performance and resilience. It is assumed that the file I/O subsystem itself assures fault notification in case of file I/O hard errors, such as a failing networked file system service, or file I/O soft errors, like data corruption. Detection of file I/O hard errors is typically employed in hardware, while detection of file I/O soft errors is normally implemented in software, e.g., with checksums or Hamming codes. The coordination protocols are separate for read and write operations to provide better performance.

None: As mentioned earlier, there is a significant distinction between accessing a node-local vs. a shared networked file system. As node-local files are not intended to be shared between processes of a parallel application, the presented redundant file I/O library does not interfere with any file I/O involving node-local storage. The previously described system call pass-through capability is employed for node-local file I/O based on the file path given by the application and the knowledge of the redundant file I/O library about shared file system mounts. Any file I/O error is handled like a normal file I/O error, while any replica fault is handled by the redundant MPI layer and does not impact the node-local file I/O of the surviving replica(s).

Simple Read: The most basic coordination for file I/O read operations, Simple Read (see Figure 1), allows each replica to read the same file content at the same time from a shared network file system. As each replica reads its data separately from the file system, the resulting

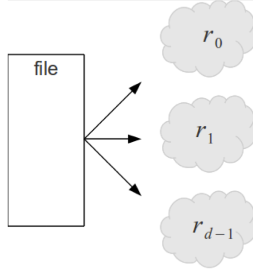

Figure 1. The Simple Read protocol.

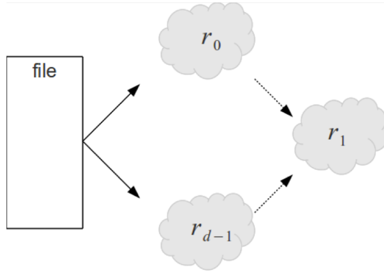

Figure 2. The Extended Simple Read protocol.

performance impact due to contention at the storage system and/or I/O network can be quite significant. An advantage, however, is that data is read redundantly, i.e., an additional protocol that calculates checksums over the read data and exchanges the checksums reliably between the replicas adds more comprehensive soft error resilience. Any file I/O error is handled using an additional coordination between the replicas and consistently reported to the application. Any replica fault is handled by the redundant MPI layer and does not impact the file I/O of the surviving replica(s). This is the only read protocol that does not involve coordination by the redundant file I/O library for replica faults.

Extended Simple Read: In the Extended Simple Read protocol (see Figure 2), only a subset of the replicas reads in the same file content at the same time from a shared network file system. The other replicas receive the data from this subset. In a triple redundant scenario, for example, two replicas read the file content and the third receives it from one of the two (or both). In contrast to the Simple Read protocol, less data is transferred from the shared network file system, which results in less contention and higher performance for file I/O. However, the additional step for sending the data to the other replica(s) does add latency. Similar to the Simple Read protocol, an additional protocol calculating and exchanging checksums adds more comprehensive soft error resilience. Any file I/O error is coordinated between replicas and consistently reported to the application. Any replica fault is handled by the redundant MPI layer and the coordination protocol that includes an optional fall back to the Simple Read protocol.

Read & Distribute: The Read & Distribute protocol (see Figure 3) is a special variant of the Extended Simple Read protocol. Only one master replica is reading the file content and distributing it to the other replica(s). Depending on the inter-node network, the performance impact can be quite extensive as the data needs to be multicast to the other replica(s) using point-to-point messages or a native multicast. While this adds a significant latency, pressure at the shared networked file system is reduced. Further, as the data is only read once, soft error resilience is not provided by this protocol and needs to be supplied by the file I/O and MPI layers. Any file I/O error is coordinated between replicas and consistently reported to the application. Any replica fault is handled by the redundant MPI layer and the coordination protocol that includes a fail-over of the master.

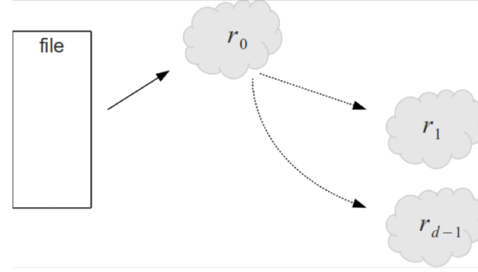

Figure 3. The Read & Distribute protocol.

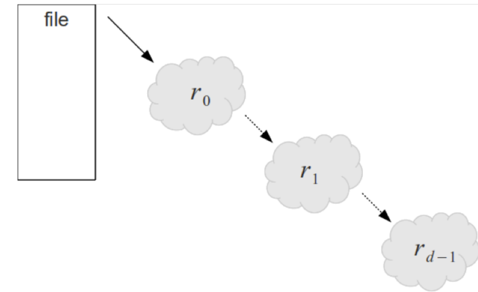

Figure 4. The Chained Read protocol.

Chained Read: The Chained Read protocol (see Figure 4) is a special variant of the Read & Distribute protocol. The multicast mechanism is a point-to-point chain, which incurs the highest latency. Failure handling is performed as in the Read & Distribute protocol. This protocol is only described as a progression step toward the next protocol.

Chunked & Chained Read: The Chunked & Chained Read protocol (see Figure 5) reduces the latency of the Chained Read protocol by cutting the data buffer to be read into multiple chunks and reading/distributing each chunk separately. Performance is gained due to the overlapping file I/O and replica messaging. Although a tree-based multicast may seem to be an intuitively better fit for distributing the chunks, the message scheduling among the replica(s) is actually an optimal pipeline using the chained multicast. File I/O and replica failure handling is performed in this protocol as in the Read & Distribute protocol.

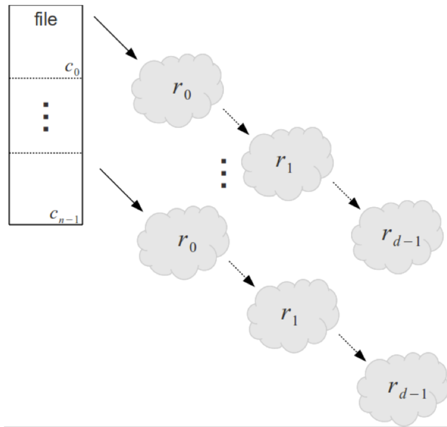

Figure 5. The Chunked & Chained Read protocol.

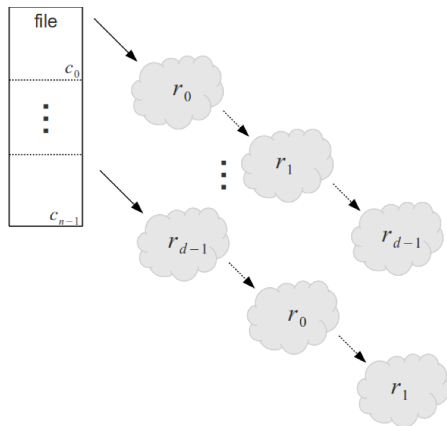

Figure 6. The Distributed Chunked Read protocol.

Distributed Chunked Read: The Distributed Chunked Read protocol (see Figure 5) is an extension of the Chunked & Chained Read protocol, in which each replica reads in a different chunk and distributes it to the others. This protocol has the potential to improve throughput and decrease latency as individual chunks may be located on different storage servers in a networked shared file system and the data distribution among the replicas may take advantage of the low-latency/high-bandwidth communication between replicas. File I/O failure handling is performed as in the Read & Distribute protocol. Any replica fault is handled by the redundant MPI layer and the coordination protocol that includes a fail-over of the master replica with the difference that there is a different master for each data chunk.

Simple Write: In the Simple Write protocol (see Figure 7) a master replica writes out the file I/O buffer and communicates the result (bytes written and possibly a checksum) to the remaining replicas. As only one replica performs file I/O and minimal result data is communicated between replicas, the performance of this protocol is quite high. It does offer soft error resilience using checksum exchanges between replicas. Any file I/O error is coordinated between replicas and consistently reported. Any replica fault is handled by the redundant MPI layer and the coordination protocol that includes a fail-over of the master.

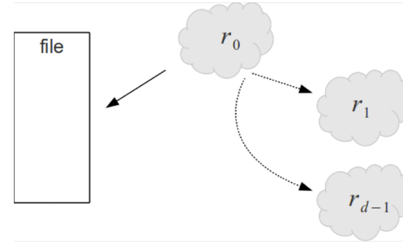

Figure 7. The Simple Write protocol.

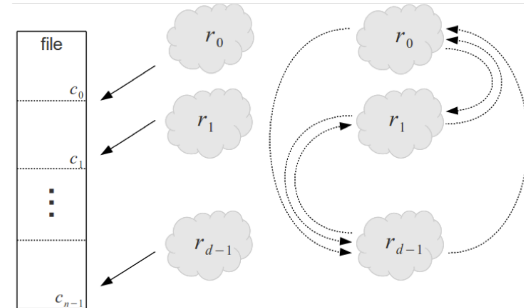

Figure 8. The Distributed Write protocol.

Distributed Write: The Distributed Write protocol (see Figure 8) allows each replica to write out a specific chunk of the data and to communicate the result (bytes written and possibly a checksum) to the other replicas. This rather complex operation not only requires support from the networked shared file system for multiple clients writing different parts of a file, but also a very careful handling of any failures. This protocol does offer soft error resilience using checksums. Any file I/O error is coordinated between replicas and consistently reported to the application. Similar to the Distributed Chunked Read protocol, the Distributed Write protocol protocol may offer better performance as individual chunks may be located on different storage servers in a networked shared file system.

Distributed Separate Write: The last protocol described is the Distributed Separate Write (see Figure 9). It is a variant of the Distributed Write protocol with a mechanism that writes each chunk coming from a replica into a separate file. While this protocol requires post-processing to join these files, this operation may be as simple as joining metadata records without moving file content in an advanced networked shared file system. In this case, performance may be even better than using the Distributed Write protocol, depending on the capabilities of the networked shared file system. Failure handling is performed as in the Distributed Write protocol.

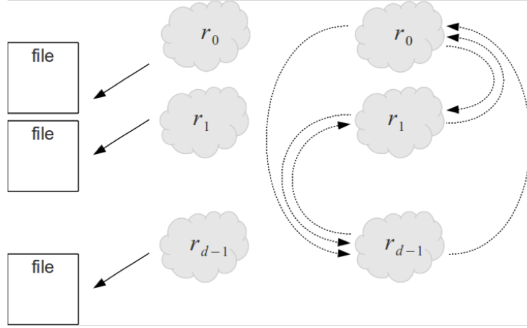

Figure 9. The Distributed Separate Write protocol.

V. Experimental Evaluation

The developed proof-of-concept prototype was deployed on a 16-node cluster together with the latest version of MR-MPI and a set of MPI micro benchmarks performing file I/O. Each node has two 2.4GHz AMD Opteron processors with 4-cores each, *i.e.*, 8 cores per node, and 8GB RAM, *i.e.*, 1GB per core. The system has a total of 128 cores and 128 GB RAM. The network interconnect is Gigabit Ethernet with a central non-blocking switch. The cluster is running Ubuntu 10.04.1 LTS Linux and Open MPI 1.5.3. Tests were performed to the local file system as well as using Network File System (NFS) mounts. To show the impact on accessing a single shared vs. a distributed shared file system, a single and multiple NFS mounts (one for each replica) were employed to simulate the effect of parallel file I/O. Each performance test was executed 10 times.

Simple Read and Distributed Chunked Read: The first evaluation focused on the file read performance of the Simple Read and Distributed Chunked Read protocols without (1x), with double (2x) and with triple (3x) redundancy in comparison to the non-redundant native (1x) performance exemplified by the None protocol. An MPI micro benchmark was developed and executed as one non-, dual- or triple-redundant MPI process that reads the data and measures the file read throughput.

An 8 GB file was read from a single NFS server with various read buffer sizes (see Figure 10). The chunk size for the Distributed Chunked Read protocol was determined by dividing the read buffer by the number of replicas. The results are quite interesting. The first to note is that with read buffer sizes of 2 GB and below, the performance is better than the theoretical bandwidth of the Gigabit Ethernet (120 MB/s). This can be entirely attributed to read-ahead and caching by the OS that was purposely not disabled to demonstrate performance under realistic conditions. At 4 GB read buffer size, these effects subside as the total node memory is 8 GB and the OS is throttling read-ahead and caching to conserve resources. Without redundancy, the Simple Read (1x) protocol performs slightly worse in the read-ahead and caching zone than the None (1x) protocol due to the added internal bookkeeping disturbing these

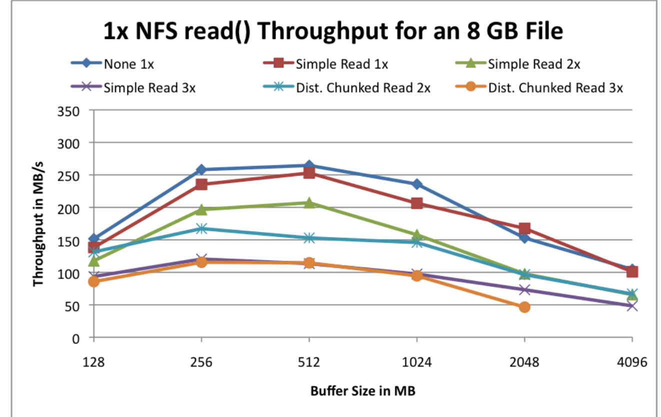

Figure 10. The None, Simple Read and Chunked & Chained Read protocols reading an 8 GB file from a single NFS server with various block sizes and without (1x), with double (2x) and with triple (3x) redundancy.

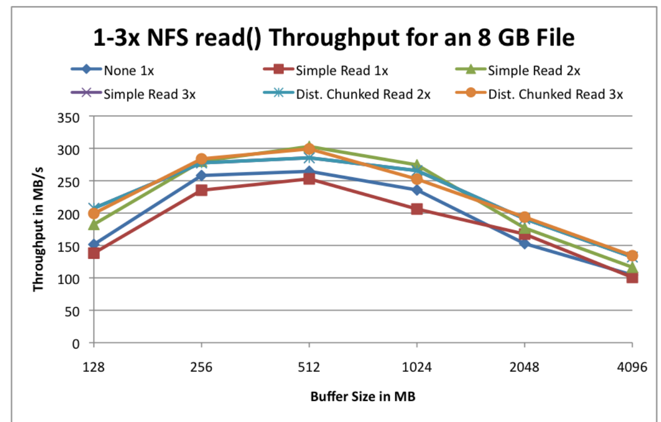

Figure 11. The None, Simple Read and Distributed Chunked Read protocols reading an 8 GB file from multiple NFS servers with various block sizes and without (1x), with double (2x) and with triple (3x) redundancy.

effects. At 4 GB read buffer size, the performance is the same. With double redundancy, the Distributed Chunked Read (2x) protocol performs worse in the read-ahead and caching zone than the Simple Read (2x) protocol due to the added coordination and data distribution between replicas using MPI disturbing these effects. At 4 GB read buffer size, the performance is the same. With triple redundancy, both protocols perform similar in the read-ahead and caching zone. The performance of the Distributed Chunked Read (3x) drops off significantly at 2 GB read buffer size. As the network performance between nodes is the same as to the single NFS server, the data distribution of the Distributed Chunked Read (3x) causes this performance penalty.

To evaluate the performance of the Simple Read and Distributed Chunked Read protocols when accessing a parallel networked file system with multiple storage servers using RAID, the number of NFS servers was increased with the degree of redundancy and each replica read from its corresponding NFS server. While the single NFS scenario

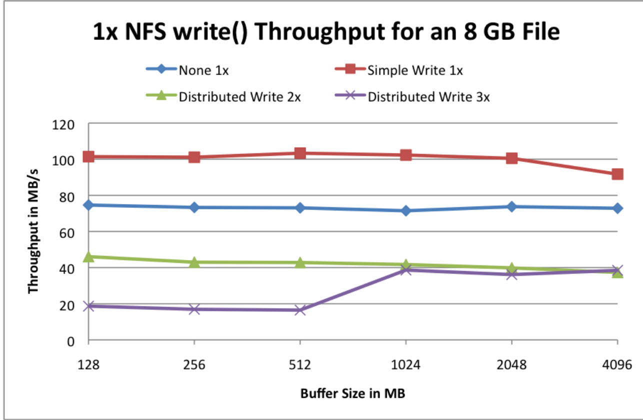

Figure 12. The None, Simple Write and Distributed Write protocols writing an 8GB file to a single NFS server with various block sizes and without (1x), with double (2x) and with triple (3x) redundancy.

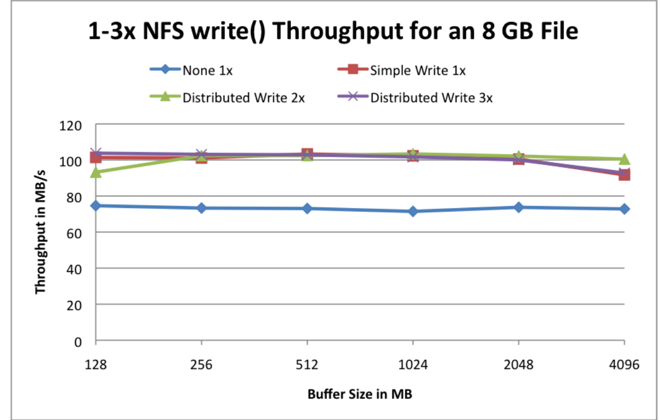

Figure 13. The None, Simple Write and Distributed Write protocols writing an 8GB file to multiple NFS servers with various block sizes and without (1x), with double (2x) and with triple (3x) redundancy.

provided lower performance bound, this scenario offers an upper performance bound. The results (see Figure 11) show the same read-ahead and caching effects. However, the None (1x) protocol is outperformed by all other protocols, except for the non-redundant Simple Read (1x) protocol. In this performance evaluation, redundancy actually provides a performance gain as data is read for each replica from a different storage server. The Distributed Chunked Read outperforms the Simple Read under both redundancy configurations (1x and 2x).

Simple Write and Distributed Write: The next evaluation aimed at the file write performance of the Simple Write and Distributed Write protocol without (1x), with double (2x) and with triple (3x) redundancy in comparison to the non-redundant Simple Write (1x) protocol and native (1x) performance exemplified by the None protocol. The redundant Simple Write protocols (2x and 3x) were omitted for clarity as there is only one writer as in the non-redundant Simple Write (1x) protocol. An MPI micro benchmark was developed and executed as one non-, dual- or triple-redundant MPI process that writes the data and measures the file write throughput.

An 8 GB file was written to a single NFS server with various write buffer sizes (see Figure 12). The results show a better performance for the non-redundant Simple Write (1x) than for the non-redundant None (1x) protocol. This may be caused by slight performance interruptions the interposing file I/O library introduces that ultimately reduce the network pressure created at the NFS server. The results also demonstrate the significant impact of the Distributed Write (2x and 3x) on the single NFS server as different parts of a file are written by each replica at the same time, creating contention at the NFS server.

In a corresponding series of write tests the number of NFS servers was increased with the degree of redundancy and each replica wrote to its NFS server. Similar to the file read performance evaluation, the file write results (see Figure 13) show again a performance increase under redundancy. All redundancy protocols perform close to or slightly better than the non-redundant Simple Write (1x). The non-redundant None (1x) suffers again from creating too much network pressure at the NFS server.

VI. Summary and Future Work

With the research and development presented in this paper, we offered a first step beyond availability/reliability models and process replication protocols in the relatively new area of redundancy for MPI applications. The issue of file I/O under redundancy is quite important as MPI applications do perform file I/O, mostly for input/output data. The described proof-of-concept prototype intercepts file I/O calls from the MPI application and employs the needed coordination protocols to enable redundancy-oblivious or redundancy-aware file I/O, depending on the targeted file system. While the proof-of-concept prototype and the experimental evaluation relies on MR-MPI, which the authors were granted source code access to, using a different redundant MPI layer is as simple as rewriting the redundant file I/O library calls to identify the redundancy degree and the replica mapping.

The performance evaluation focused on the Simple Read, Distributed Chunked Read, Simple Write and Distributed Write protocols with various block sizes and without, with double and with triple redundancy. The results clearly indicate the performance impact under redundancy when accessing a single networked file system. They also demonstrate the capability to offset this performance impact in a parallel networked file system using parallel file I/O and MPI communication between replicas for data distribution.

Looking beyond the presented work, we will target a more comprehensive evaluation using HPC applications and larger-scale systems. Future work not only includes a more mature implementation that can be used in production, but

also dynamic protocol selection based on application demands and system capabilities. The current proof-of-concept prototype only supports static protocol switching separately for each application run. For more efficient protocol selection, a dynamic redundant file I/O implementation would consider runtime information, such as file size and access patterns/frequency, system configuration, like network latencies/bandwidths and I/O bottlenecks, application configuration parameters, such as node count and location, and, most importantly, application user guidance. Further work may focus on built-in support for redundant applications in advanced parallel file systems, such as Lustre [16] and PVFS [17], especially considering the fact that these file systems do employ redundancy protocols for Redundant Array of Independent Disks (RAID) mechanisms across storage servers.

VII. Acknowledgment

This research is sponsored by the Office of Advanced Scientific Computing Research; U.S. Department of Energy. The work was performed at the Oak Ridge National Laboratory, which is managed by UT-Battelle, LLC under Contract No. De-AC05-00OR22725. 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. P. Siemwiorek, "Architecture of fault-tolerant computers: An historical perspective," *Proceedings of the IEEE*, vol. 79, no. 12, pp. 1710–1734, 1991.

[2] 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, 2009.

[3] E. N. 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," Defense Advanced Research Project Agency (DARPA), Tech. Rep., 2008.

[4] 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," Defense Advanced Research Project Agency (DARPA) Information Processing Techniques Office (IPTO), Tech. Rep., 2008.

[5] F. Cappello, A. Geist, W. D. B. Gropp, L. V. S. Kale, W. T. C. B. Kramer, and M. Snir, "Toward exascale resilience," University of Illinois at Urbana-Champaign (UIUC) - Institut National de Recherche en Informatique et en Automatique (INRIA) Joint Laboratory on PetaScale Computing, Tech. Rep. TR-JLPC-09-01, 2009.

[6] A. Geist and R. F. Lucas, "Major computer science challenges at exascale," International Exascale Software Project, Tech. Rep., 2009, whitepaper.

[7] J. Dongarra, P. Beckman, T. Moore, J.-C. Andre, J.-Y. Berthou, T. Boku, F. Cappello, B. Chapman, X. Chi, A. Choudhary, S. Dosanjh, A. Geist, B. Gropp, R. Harrison, M. Hereld, M. Heroux, A. Hoisie, K. Hotta, Y. Ishikawa, F. Johnson, S. Kale, R. Kenway, B. Kramer, J. Labarta, B. Lucas, B. Maccabe, S. Matsuoka, P. Messina, B. Mohr, M. Mueller, W. Nagel, H. Nakashima, M. E. Papka, D. Reed, M. Sato, E. Seidel, J. Shalf, D. Skinner, T. Sterling, R. Stevens, W. Tang, J. Taylor, R. Thakur, A. Trefethen, M. Snir, A. van der Steen, F. Streitz, B. Sugar, S. Sumimoto, J. Vetter, R. Wisniewski, and K. Yelick, "International exascale software project roadmap (draft 0.93)," 2009.

[8] B. Schroeder and G. A. Gibson, "Understanding failures in petascale computers," in *Journal of Physics: Scientific Discovery through Advanced Computing Program (SciDAC) Conference*, vol. 78, 2007, pp. 2022–2032.

[9] J. T. Daly, L. A. Pritchett-Sheats, and S. E. Michalak, "Application MTTFE vs. platform MTTF: A fresh perspective on system reliability and application throughput for computations at scale," in *IEEE International Symposium on Cluster Computing and the Grid (CCGrid): Workshop on Resiliency in High Performance Computing (Resilience)*, 2008.

[10] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys (CSUR)*, vol. 22, no. 4, pp. 299–319, 1990.

[11] 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," Sandia National Laboratories, Technical report SAND2009-6753, 2009.

[12] C. Engelmann and S. Böhm, "Redundant execution of HPC applications with MR-MPI," in *IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN)*, 2011, pp. 31–38.

[13] D. Fiala, "Detection and correction of silent data corruption for large-scale high-performance computing," Poster at the 25th IEEE International Parallel and Distributed Processing Symposium (IPDPS) 2011, Anchorage, AK, USA, 2011.

[14] 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: European PVM/MPI Users' Group Meeting (EuroPVM/MPI)*, vol. 5759, 2009, pp. 124–133.

[15] 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*, vol. 5, no. 3, pp. 63–73, 1991.

[16] Sun Microsystems, Inc., Santa Clara, CA, USA, "Lustre file system – High-performance storage architecture and scalable cluster file system," 2007, white paper.

[17] P. H. Carns, W. B. L. III, R. B. Ross, and R. Thakur, "PVFS: A parallel file system for linux clusters," in *Annual Linux Showcase and Conference*, 2000, pp. 317–327.