Fault-Tolerant Communication Runtime Support for Data-Centric Programming Models

Abhinav Vishnu,* Huub Van Dam,* Wibe De Jong,* Pavan Balaji,‡ and Shuaiwen Song†

* Pacific Northwest National Laboratory, Richland, WA 99352
Email: {abhinav.vishnu, hubertus.vandam, wibe.dejong}@pnl.gov

‡ Mathematics and Computer Science Division
Argonne National Laboratory, Argonne, IL 60439
Email: {balaji}@mcs.anl.gov

† Department of Computer Science
Virginia Polytechnic Institute, Blacksburg, VA 24060
Email: {shuaiwen.song}@cs.vt.edu

***Abstract*—The largest supercomputers in the world today consist of hundreds of thousands of processing cores and many more other hardware components. At such scales, hardware faults are a commonplace, necessitating fault-resilient software systems. While different fault-resilient models are available, most focus on allowing the computational processes to survive faults. On the other hand, we have recently started investigating fault resilience techniques for data-centric programming models such as the partitioned global address space (PGAS) models. The primary difference in data-centric models is the decoupling of computation and data locality. That is, data placement is decoupled from the executing processes, allowing us to view process failure (a physical node hosting a process is dead) separately from data failure (a physical node hosting data is dead).**

In this paper, we take a first step toward data-centric fault resilience by designing and implementing a fault0resilient, one-sided communication runtime framework using Global Arrays and its communication system, ARMCI. The framework consists of a fault-resilient process manager; low-overhead and network-assisted remote-node fault detection module; non-data-moving collective communication primitives; and failure semantics and error codes for one-sided communication runtime systems. Our performance evaluation indicates that the framework incurs little overhead compared to state-of-the-art designs and provides a fundamental framework of fault resiliency for PGAS models.

I. Introduction

The largest systems in the world today already scale to hundreds of thousands of cores. With plans under way for exascale systems to emerge within the next decade, we are likely soon to have systems comprising more than a million processing elements. As researchers work toward architecting these enormous systems, it is becoming increasingly clear that at such scales, resilience to hardware faults is going to be a prominent issue that needs to be addressed. Driven by the needs of large-scale scientific computing applications, a variety of programming models have beenn introduced over the past two decades. While the Message Passing Interface (MPI) [1], [2] has become the de facto standard for writing parallel programs, PGAS models have recently gained popularity as well [3], [4], [5], [6]. Together with these programming models, different communication runtime systems to serve these programming models have also become available [7], [8].

Fault tolerance in MPI has been an area of significant research [9], [10], [11], [12], [13], [14]. Most of this research, however, has focused on allowing the computational processes to survive fault, through either checkpointing or application-level resilience to faults. While a process-driven model for fault tolerance has its benefits, it has the disadvantage that each process manages its data; thus, a failed node implies that the processes residing on those nodes as well as their data are lost and must be recreated or restored. Recently, we have started investigating fault resilience techniques for data-centric programming models such as the partitioned global address space (PGAS) models. The primary difference in data-centric models is the decoupling of computation and data locality. That is, data placement is decoupled from the executing processes, allowing one to view process failure (a physical node hosting a process is dead) separately from data failure (a physical node hosting data is dead).

However, the first obstruction in providing such data-centric fault resilience is that there is a lack of basic fault resiliency in the underlying communication runtime infrastructure for PGAS models and other associated components such as the process manager. Even for the hard faults, there is no low-overhead fault detection framework and no support for even a minimal set of fault-resilient collective communication primitives.

In this paper, we take a first step toward data-centric fault resilience by designing and implementing a fault-resilient, one-sided communication runtime framework. Emphasizing the properties of PGAS models for fault resiliency, we present data redundancy models for continued execution during failure and

a design for a remote node fault detection module that uses a combination of modern network primitives such as remote direct memory access (RDMA) [15], send/receive, and data delivery notification semantics for high-accuracy fault detection. Leveraging this fundamental infrastructure, we design and implement non-data-moving collective communication primitives and provide the foundation for fault-resilient data-moving collectives. We discuss the need for various semantic changes to write-based operations for recovery with data redundancy, and we provide a framework for error notification with one-sided communication primitives. Using Global Arrays (GA) [3] as an example PGAS model, we implement our design with Aggregate Remote Memory Copy Interface (ARMCI) [7], the communication runtime system of Global Arrays [3], and refer to our solution as fault-tolerant ARMCI, or FT-ARMCI. Our performance evaluation shows that FT-ARMCI can provide fault resiliency with low overhead. We are currently designing and implementing a fault-resilient, high-order computational chemistry method using Global Arrays. We plan to present the results in the final version of the paper.

The rest of the article is organized as follows. In Section II, we discuss related work. In Section III, we present the background of our work. In Section IV, we describe the overall design for FT-ARMCI. In Section V, we present a performance evaluation of FT-ARMCI. In section VI, we conclude with a brief summary and discussion of future directions for research.

II. Related Work

Fault tolerance with high-performance computing has been studied by multiple researchers with various programming models and applications. Fagg and Dongarra et al. introduced FT-MPI [11], discussing the process model on occurrence of a failure. Approaches include respawning of MPI processes and patching them with the original communicator or continuing with the holes in the communicator. Our proposed approach is similar to the latter model proposed by FT-MPI; however, fault recovery is done for Global Address space models using Global Arrays [3]. Gropp and Lusk argued that the statement "MPI is not fault tolerant" is unfounded because fault tolerance is a property of the combination of an MPI program and MPI implementation [9]. While most MPI implementations choose to abort on a fault, this behavior is not mandated by MPI standard. Gropp Lusk also discussed methods of recovery using dynamic process creation and intercommunicators [9]. Our proposed approach performs graceful degradation and does not require a process-based fault recovery algorithm at the application level; rather, it requires task-based re-execution.

With MPI [1], [2], fault tolerance using application-transparent/assisted approaches have been discussed widely [10], [16], [13], [14]. Gao et al. observed that user-transparent checkpointing using the Berkeley Lab Checkpoint Restart is beneficial for NAS Parallel Benchmarks and that checkpoint aggregation can reduce the overall time of checkpointing significantly; however, the overhead increases significantly after a small number of processors [10]. In our approach, we leverage user-assisted data redundancy for fault recovery and perform continued execution with graceful degradation. Bosilca et al. [13] and Bouteiller et al. have shown that pessimistic message-based logging can be used for recovery on volatile nodes using MPICH-V; however, this approach may not be applicable for one-sided communication libraries such as ARMCI [7] considered in this work. Another benefit of using the replicated approach is continued execution, whereas message logging requires rollback, although it is useful for applications that are more suited for process-based models.

Researchers have also focused on providing fault tolerance using virtual machine–based approaches Recent work includes Xen over high-speed networks [12] in addition to the classical work based on virtual machines using TCP/IP; however, these approaches require either checkpointing for postfailure execution [17] or proactive fault tolerance [12], which depends on the accuracy of fault prediction. Our proposed approach, on the contrary, does not depend on the accuracy of fault prediction, as it is able to perform continued execution upon occurrence of a failure.

III. Background

In this section, we present the background of our work. We begin with a description of PGAS models, modern interconnects, and their primitives.

A. PGAS Models

Partitioned plobal address space models provide a logical abstraction of global memory space, which is partitioned in remote and local data for leveraging the locality of reference. Many language and library implementations are being designed and implemented under this model, such as X10 [5], Chapel [6], High Performance Fortran, Unified Parallel C (UPC) [4], ZPL, and Titanium. These languages provide mechanisms for accessing data in global space and provide efficient compiler-based implementations to coalesce multiple such data accesses by resolving write dependencies. They also provide mechanisms to allow computation to be executed on a remote node (by using active messages or similar mechanisms). PGAS models provide data-centric abstractions and decouple computation from process-based models. We use this property of PGAS models to provide fault resiliency with FT-ARMCI. Library-based implementations such as Global Arrays [3] and SHMEM [18] have also been designed and implemented to serve the purpose of remote memory accesses. The user of Global Arrays [3] is expected to explicitly request remote/local data pointers and make explicit requests to coalesce data transfers.

The Global Arrays programming model exposes to the programmer the non-uniform memory access (NUMA) characteristics of high-performance computers. Accesses to a remote portion of the shared data is slower than to the local portion. The locality information for the shared data is available, and a direct access to the local portions of shared data is provided.

Global Arrays uses ARMCI [7] as the runtime system for communication.

B. Modern Interconnects

In the past decade, high-speed interconnects—both open standard and proprietary—have become available. In the open standard community, InfiniBand [19] has become popular because of its high performance, including features such as RDMA [15], hardware-assisted collective communication primitives, and atomic operations. Similarly, 10 Gigabit Ethernet is becoming popular to support legacy sockets-based applications, while providing primitives for RDMA and zero-copy communication using send/recv primitives [20], [21]. In the proprietary interconnects domain, interconnects such as the IBM Blue Gene Torus network and Cray Seastar [22] have become popular, providing support for RDMA as well as for most features using connectionless transport semantics. High Performance Switch (HPS)IBM's fourth-generation switch, provides support for connectionless RDMA; reliability is implemented by using IBM's LAPI access layer [23], [24], [25].

In this paper, we use a combination of RDMA semantics and reliable notification provided by these interconnects to detect remote-node failure(s). The networks discussed above can be classified on the basis of data delivery notification. Networks such as InfiniBand provide "exact once notification" of data delivery with the reliable connection semantics. This results in a guaranteed notification of data delivery, offloaded in the hardware. Other networks such as IBM HPS provide "maximum once notification." Typically, this results in notification when the data delivery is successful; however, data delivery failures are not notified. Keeping these properties in mind, we design and implement the remote fault detection layer presented in Section IV. Our reference implementation over InfiniBand uses "exact once notification" semantics provided by the network for remote-node fault detection.

C. Communication Runtime Systems

Communication runtime systems play an important role in providing efficient, high-performance support to programming models. For PGAS models such as Global Arrays [3] and Berkeley Unified Parallel C [4], the communication runtime systems used are ARMCI [7] and GASNet, respectively. Each of the communication runtime systems provides general-purpose, efficient, and widely portable remote memory access (RMA) operations (one-sided communication) optimized for contiguous and noncontiguous (strided, scatter/gather, I/O vector) data transfers. Native network communication interfaces and system resources (such as shared memory) are utilized to achieve the best possible performance of the remote memory access/one-sided communication. Optimized implementations for each of these communication runtime systems are available for Cray Portals, Myrinet (GM and MX) [20], Quadrics [21], GigaNet (VIA), and InfiniBand (using OpenFabrics and Mellanox Verbs API) [19]. In addition, they are available for leadership-class machines including Cray XT4/XT5 and Blue Gene/P [26].

IV. Overall Design

In this section, we present the overall design of our fault-resilient communication runtime system for PGAS models. We begin with a presentation of the properties of PGAS models suited for fault resiliency. We follow this with discussion of our application-level data redundancy model. The overall design is presented in Figure 1.

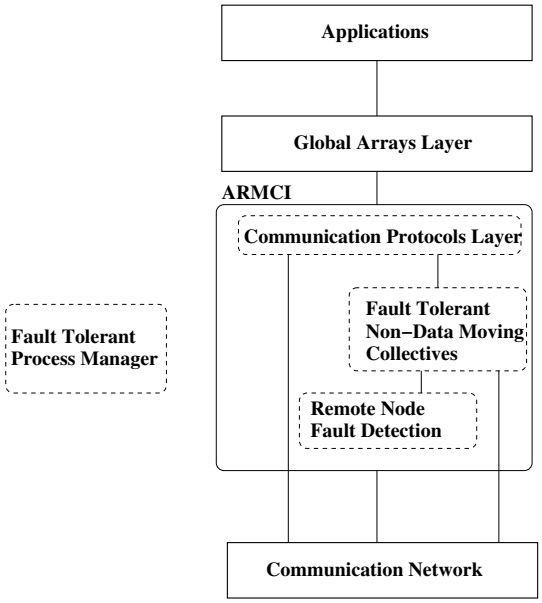

Fig. 1. Overall Design of FT-ARMCI

Efficient and accurate detection is critical in designing fault-resilient communication runtime systems. In addition, designing a fault-resilient process manager is important, since most process managers abort on occurrence of a node failure. Moreover, a framework for fault-resilient collective communication primitives is needed for data-centric models to maintain the control flow of the application. We have designed a low-overhead, high-accuracy remote-node fault detection module to achieve this purpose. Using this module, we designed and implemented fault-resilient process manager and fault-resilient non-data-moving collective communication primitives. The details are presented in the following subsections.

A. Why PGAS Models?

MPI [1], [2] is the predominantly used programming model for most scientific applications. A broad range of applications, requiring frequent synchronization and precisely described in terms of processes rather than data, naturally fit under this model. However, one class of applications, classified as data centric, follow a task-based execution model and define dependencies in terms of task execution; these applications naturally fit PGAS models.

An important property of PGAS models is the data-centric nature of the algorithm and independence from the number of

processes in the execution model. The nature of the algorithm allows the data requests to be served from arbitrary processes, as long as data consistency is maintained. Thus, the total number of processes can grow and shrink arbitrarily. Assuming a fault model giving a notion of node failure, the data-centric property of the algorithm allows continued execution with minimal changes, compared with the process-centric execution model. Hence, we use PGAS models with Global Arrays as the candidate programming model for implementing our design. However, the generic nature of the design is applicable to runtime systems of other PGAS models such as GASNet [8].

We also use a graceful degradation approach, under which we continue with the available number of processes, rather than respawning the processes of the lost node. The fault recovery with the reincarnation of lost processes becomes expensive, since the new processes may require a synchronous continuum on the failure, which can be prohibitive on exascale systems. The graceful degradation approach also allows to continue execution when the spare nodes are not available—a likelihood on exascale systems with capability-class loads.

B. Application-Level Data Redundancy Model

In this section, we discuss the expected data redundancy model from applications using FT-ARMCI. While the design and development of these applications is on-going, the data redundancy model provides guidelines for designing FT-ARMCI.

Global Arrays [3] use the master process on a node [27], [28] to allocate buffer for global address space; other processes on the node attach to the shared-memory segment. Hence the data redundancy can be achieved at a node level. This can be done by allocating a shadow copy of the global address space on the neighboring node. This is shown in the Figure 2 using four nodes.

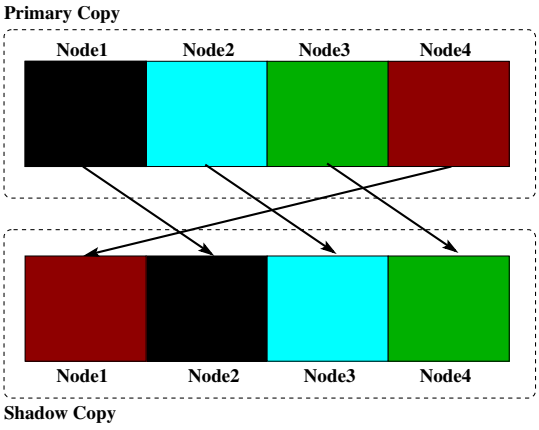

Fig. 2. Global Address Space Redundancy on Multiple Nodes

Figure 2 shows that the shadow copy of the local portion of global address space is present on the logical neighboring node. Another assumption in the redundancy model is that each node has enough memory to store the primary and shadow copy completely. This is a limitation of the current model, and we plan to propose space-efficient data redundancy models.

An important implication of the redundancy is that at least one of the primary or shadow copies should be in a consistent state at the time of recovery. To this end, the communication runtime system must ensure that the same patch of the data is not updated simultaneously in the primary and the shadow copies. We discuss this situation in Section IV-E1.

C. Fault Tolerance Management Infrastructure

The fault tolerance management infrastructure consists of two primary modules: a fault-resilient process manager and a remote-node fault detection module. We present each of these components in detail in the following subsections.

1) Remote-Node Fault Detection Module: Highly accurate detection of component failure is key to designing a fault-resilient communication runtime system. Methods for remote-node fault detection have been proposed in the literature using TCP/IP-based sockets [29], [30]. The kernel involvement and dependency on an entity on a remote node result in greater inaccuracy with this method.

Modern interconnects provide memory and channel semantics for data transfer with RDMA and send/receive capabilities, respectively. The send/receive semantics bypasses the kernel but requires involvement of remote entity (process/thread on the remote node) to respond to the health checks. The RDMA-based approach does not require involvement of a remote entity. Hence, we use the RDMA-based approach for remote-node fault detection. Using RDMA Read primitives provided by most modern networks and data delivery notification semantics, we conclude that remote node is dead if a failure is received during the read. The details are network specific, and we present the details for our reference implementation with InfiniBand in the following subsection.

Reference Implementation with InfiniBand: We leverage the "exact once" data delivery notification semantics of reliable connection transport provided by InfiniBand to check the status of the remote node [31]. A helper thread is created by the master process (only one thread is created per node) during initialization of the communication runtime system. A small buffer is allocated by each thread, which is registered with the InfiniBand hardware; and the associated information is exchanged so that the helper threads can perform RDMA reads on the buffer. The helper thread also performs periodic RDMA reads from nodes to check their health. In addition, itt responds to the health check requests (pongs) from arbitrary nodes. This functionality is also used at multiple execution points explicitly by the ARMCI library to check the status of the remote node, such as fence and collective communication primitives.

The reference implementation creates an unreliable datagram-based communication channel between all helper threads. As Koop et al. have presented, the unreliable datagram-based approach scales well [32]. Since reliable connection transport semantics require pairwise connections [27], a user-specified topology of reliable connections is also created

between the helper threads, with the default being the ring topology. The status of a node can be checked by using RDMA if it is available directly. This is illustrated in Figure 3. For checking the status of an arbitrary node, an unreliable datagram-based virtual connection is used if reliable connection is not available to the node. This is illustrated in the Figure 4. The status of a node can be checked by sending a ping message to that node and sending a ping message to a neighboring node simultaneously. A quorum-based protocol is used to test whether the remote node is healthy.

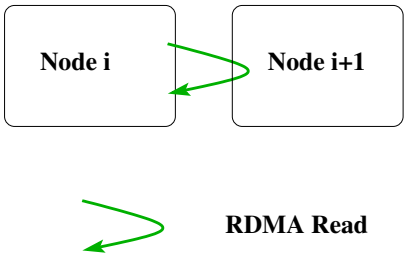

Fig. 3. Remote-Node Fault Detection When RDMA Is Available Directly

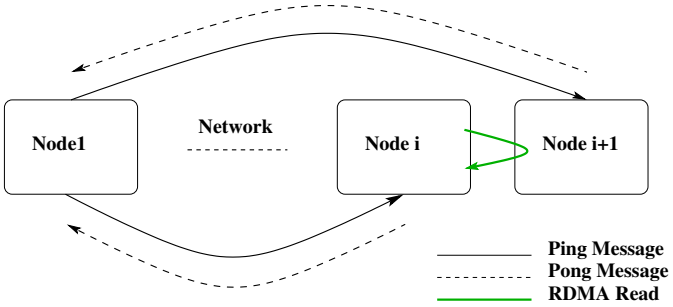

Fig. 4. Remote-Node Fault Detection When RDMA Is Available Indirectly

2) Fault Resilient Process Manager: Process management plays a crucial role in providing fault resiliency to one-sided communication runtime systems. While there has been an effort to standardize process management interfaces with PMI [33], most of the PMI implementations are geared to MPI [1], [2]. While fault tolerance is not a property of the MPI standard or MPI programs, but rather is a combination of these, state-of-the-art MPI implementations abort on occurrence of a failure. As a result, process management implementations are geared toward aborting on occurrence of a failure. Clearly, this solution does not suffice for the needs of our fault-tolerant communication runtime system.

To address this issue, we use a process manager distributed with the MVAPICH/MVAPICH2 libraries [34], and we enhance the implementation to prevent aborting of the whole job on occurrence of a failure. We plan to integrate the changes required to make the process manager fault resilient with PMI [33].

D. Fault-Resilient Non-Data-Moving Collective Communication Primitives

Collective communication primitives are widely used by programming models to provide abstractions for processes in a group to perform an operation. MPI [1], [2] provides a wide variety of data-moving collective communication primitives such as all-to-all broadcast, all-to-all personalized exchange, barrier, reduction, broadcast, and variants of these primitives. However, applications using PGAS models typically use only a small subset of these primitives. Computational chemistry codes such as NWChem [35] use barrier indirectly by executing a *sync* operation, which performs active target fence and barrier operatons. Hence, we design and implement a fault-resilient barrier primitive as a critical component for our fault-resilient one-sided communication library. We will address the topic of fault-tolerant data moving collectives in future work.

We begin with using the hypercube algorithm for the barrier primitive. The key challenge is to continue the execution of barrier in occurrence of a failur. When a failure occurs, the processes participating with the failed process during the step does not receive a message from the failed process. The participating processes use the remote-node fault detection module presented above to detect the fault. Using this information, these processes communicate with the process, which the failed process would have communicated to in the next step. This is illustrated in Figure 5.

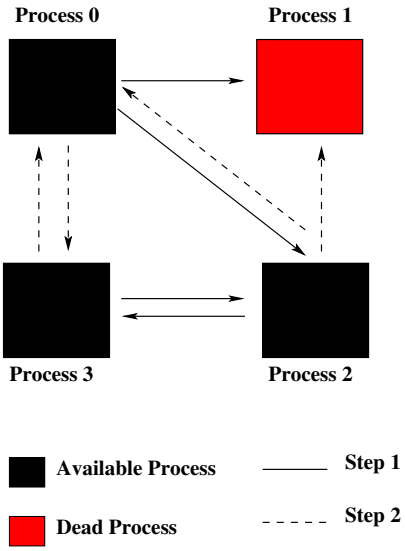

Fig. 5. Execution of Fault-Tolerant Hypercube Algorithm for Barrier

As shown in the figure, step 1 finishes successfully for process 2 and process 3, while process 0 does not receive any response from process 1, since process 1 is not alive. Process 0 calculates the destination of process 1 in the next step, process 2, and sends the message to it. Process 2 responds to it in the next step, when it receives the message. This algorithm can esaily be extended for an arbitrary number of failures during execution.

E. Semantics for One-Sided Communication Primitives

In this section, we present the semantics of one-sided communication primitives, which are required to provide fault resiliency to the PGAS models.

1) Synchronized Write-Based, One-Sided Communication Primitives: One-sided communication primitives provide semantics for buffer reusability with variants for blocking and nonblocking interfaces similar to MPI [1], [2]. Fence is typically used to ensure the completion of a data transfer operation at a remote node. For Get-based primitives, completion at the initiator side results in completion of the data arrival as well. As presented in Section IV-B, applications using Global Arrays [3] can leverage a redundant copy of the global address space to ensure that recovery is possible when a node fault occurs.

A key issue during recovery is that at least one of the copies should be in a consistent state to recover. This requires that any write-based one-sided communication primitive should be "fenced," resulting in each write-based primitive also having to be fenced. This approach guarantees that if a failure occurs during the write, at least the data from one copy is r thatecoverable. However, local completion semantics of Get-based one-sided communication primitives do not require this change. In Section V, we evaluate the impact of this change for microbenchmarks with and without faults.

2) Error Propagation and Return Codes: Error return codes are a key issue for fault-resilient one-sided communication runtime systems. For our design, we return an error in transmission only during the Get-based primitives, but we do not return error codes during the write-based one-sided communication primitives, put, and accumulate, respectively.

In addition, changes are required in various protocols of one-sided communication to ensure continued execution during occurrence of a failure. Once a process has failed, we cache the information about the failure, and any further write requests from application are ignored. However, any read-based requests result in a failure.

V. Performance Evaluation of FT-ARMCI

In this section, we present a performance evaluation of FT-ARMCI. We compare this with the latest release of Global Arrays, version 4.3, which we refer to as "Original" for the rest of the section. We have used Chinook [36], an AMD Barcelona-based Supercomputer at Pacific Northwest National Laboratory as the experimental testbed for our evaluation.

A. Experimental Testbed

Chinook [36] is a 160 TFlops system that consists of 2310 HP DL185 nodes with dual socket, 64-bit, Quad-core AMD 2.2 GHz Barcelona processors. Each node has 32 Gbytes of memory and 365 Gbytes of local disk space. Communication between the nodes is performed by using InfiniBand with Voltaire [37] switches and Mellanox [38] adapters. The system runs a version of Linux based on Red Hat Linux Advanced Server. A global 297 Tbyte SFS file system is available to all the nodes.

B. Performance Evaluation without Faults

In this section, we present the performance evaluation of FT-ARMCI, in the absence of faults. The primary objective is to understand the overhead of FT-ARMCI when no faults occur. We design simple microbenchmarks using one-sided communication primitives and compare the performance of FT-ARMCI with the Original implementation. We use two processes for evaluation, with one process on each node.

Figures 6 and 7 show the performance evaluation of ARMCI Put contiguous and Put strided one-sided communication primitives, respectively. In the tests process 0 initiates the blocking variant of the communication primitive for a small number of iterations and reports the observed bandwidth. We observe that the peak bandwidth achieved by each of the primitives is similar. However, FT-ARMCI incurs significant overhead for small message latency, as presented in the Figures 8 and 9. For each of these primitives, FT-ARMCI increases the 8-byte message latency to approximately three times. Since each write-based primitive results in an additional exchange of data transfer with the remote node, as presented in Section IV-E1, this overhead is incurred. There are possible performance improvements, such as combining the communication primitive and data transfer for reducing the latency, which we plan to extend in the near term.

C. Performance Evaluation with Faults

In this section, we present a performance comparison of the Original implementation with that of FT-ARMCI in the absence and presence of faults. We evaluate the benchmarks presented in the previous section on a larger number of processes. We modify the benchmarks to report the latency observed at every iteration. At every iteration, each process invokes a one-sided communication primitive in displaced ring communication [39], followed by a fence. We compare the Original implementation with no faults, FT-ARMCI No Fault, and FT-ARMCI One Fault. To invoke the faults, we manually kill all the processes on a node at arbitrary points during benchmark execution, in order to emulate a node failure. The point of failure is the middle iteration on the charts. We have chosen this arbitrary communication pattern to show the worst communication pattern. Other patterns reflecting MPI-style collective communication primitives should incur less overhead.

Figure 10 shows the results for the put communication primitive with 512 processes and 8 bytes. The observed latency is relative to the latency observed in the first iteration of the Original implementation. As explained previously, FT-ARMCI No Fault incurs significant overhead compared to the Original implementation for small messages. At the point of failure, the observed latency is high. A finer-grained analysis of the overheads during the node failure shows that the most time is taken by the timeout before remote-node fault detection module is used. The other overhead is due to the communication protocol of the remote-node fault detection module. While not noticeable in the chart, the overall latency decreases after fault occurrence, because the overall number of

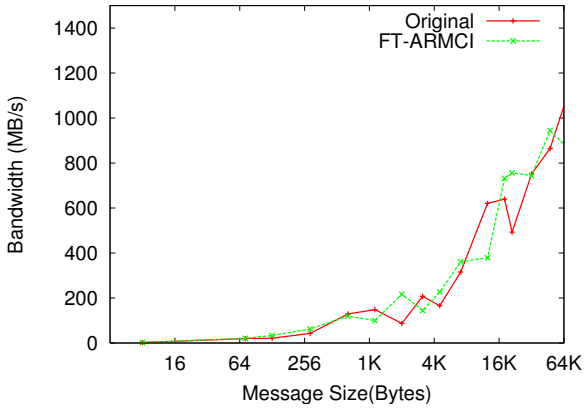

Fig. 6. ARMCI Put Unidirectional Bandwidth

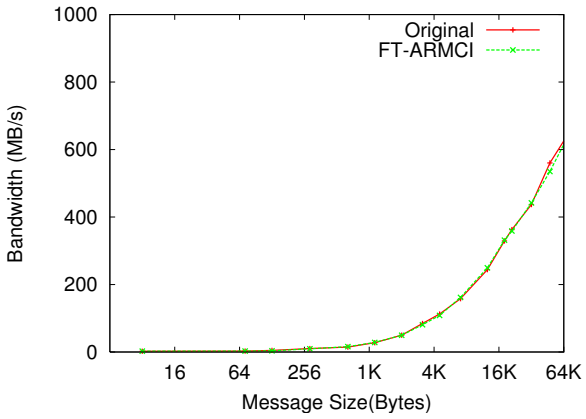

Fig. 7. ARMCI Put Strided Unidirectional Bandwidth

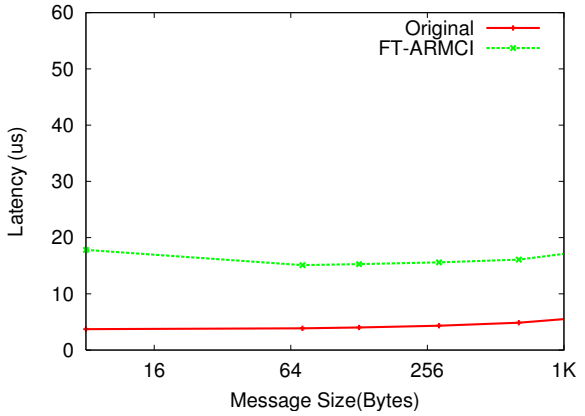

Fig. 8. ARMCI Put Latency

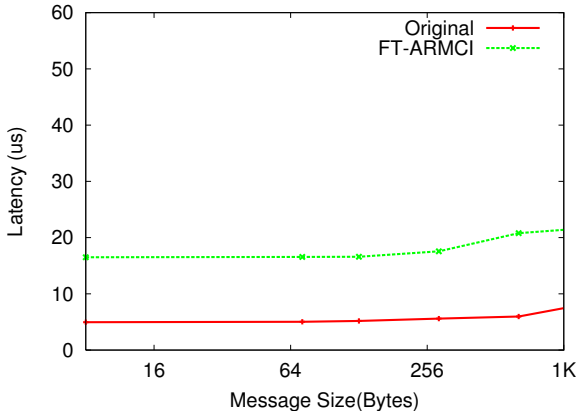

Fig. 9. ARMCI Accumulate Latency

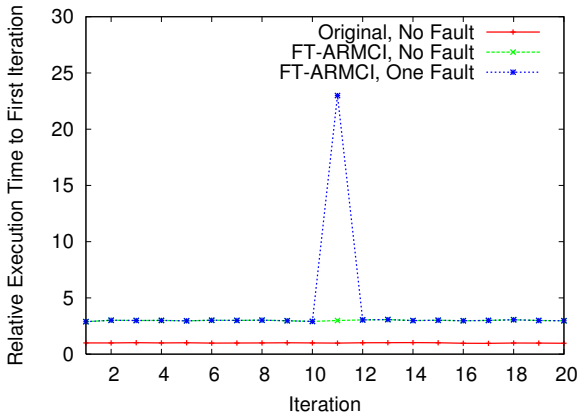

Fig. 10. ARMCI Put Latency, 512 Processes, 8 Bytes

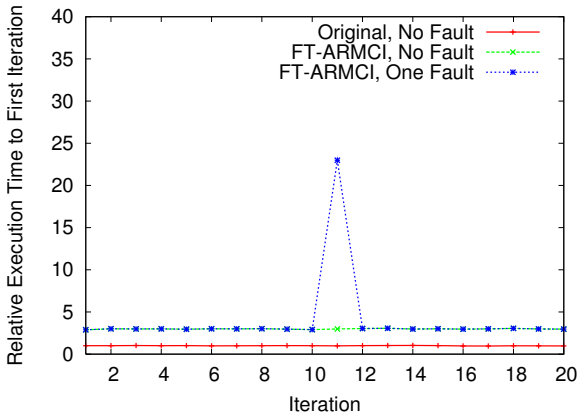

Fig. 11. ARMCI Put Latency, 1024 Processes, 8 Bytes

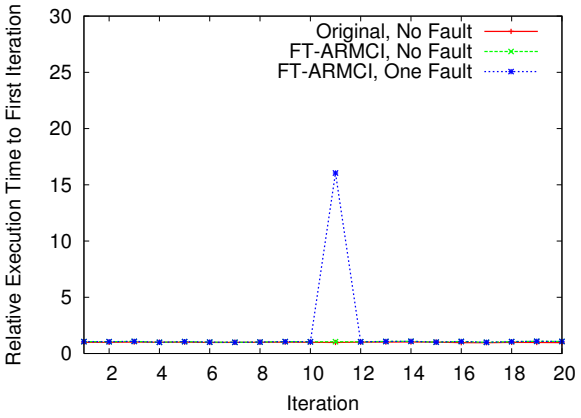

Fig. 12. ARMCI Get Latency, 512 Processes, 1 MBytes

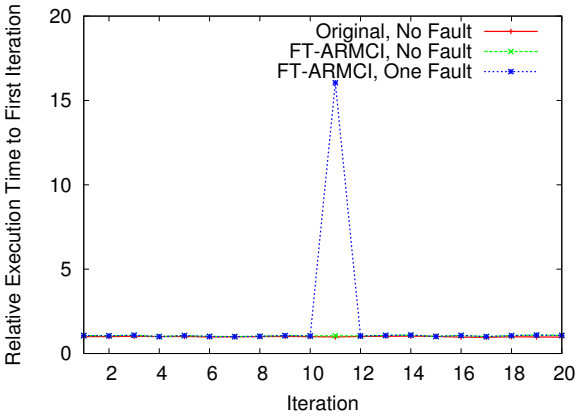

Fig. 13. ARMCI Accumulate Latency, 1024 Processes, 1 MBytes

nodes decrease. Since FT-ARMCI caches the information of failed nodes, overhead is not incurred on subsequent iterations.

Figure 11 shows the 8-byte latency for the put one-sided communication primitive for 1,024 processes. We observe similar trends in the overhead with the FT-ARMCI No Fault and the FT-ARMCI One Fault casies. Similar trends are observed with the other one-sided communication primitives, as presented in the Figures 12 and 13.

VI. Conclusions and Future Work

In this paper, we have taken a first step toward data-centric fault resilience by designing and implementing a fault-resilient one-sided communication runtime framework. Emphasizing the properties of PGAS models for fault resiliency, we have presented data redundancy models for continued execution during failure and a design for a remote-node fault detection module that uses a combination of modern network primitives such as remote direct memory access (RDMA) [15], send/receive, and data delivery notification semantics for high-accuracy fault detection. Leveraging this fundamental infrastructure, we have designed and implemented non-data-moving collective communication primitives and provided the foundation for fault-resilient data-moving collectives. We have discussed the need for various semantic changes to write-based operations for recovery with data redundancy, and we have provided a framework for error notification with one-sided communication primitives. Using Global Arrays (GA) [3] as an example PGAS model, we have implemented our design with Aggregate Remote Memory Copy Interface (ARMCI) [7], the communication runtime system of Global Arrays [3]; we refer to our solution as fault-tolerant ARMCI, or FT-ARMCI. Our performance evaluation has shown that FT-ARMCI is able to provide fault resiliency with low overhead. We are currently designing and implementing a fault-resilient, high-order computational chemistry method using Global Arrays, and we plan to present the results in the final version of the paper.

We will also finish the remaining components for providing fault tolerance. Our immediate goal is to complete the writing of a fault-resilient high-order computational chemistry method such as coupled cluster. In addition, we are working on fault-tolerant data-moving collectives with data-centric abstractions, rather than process-centric abstractions such as MPI. As we continue to develop these components, we will conduct large-scale evaluations and will make performance improvements to protocols and components.

References

[1] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard," *Parallel Computing*, vol. 22, no. 6, pp. 789–828, 1996.

[2] A. Geist, W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. L. Lusk, W. Saphir, T. Skjellum, and M. Snir, "MPI-2: Extending the message-passing interface," in *Euro-Par, Vol. I*, 1996, pp. 128–135.

[3] J. Nieplocha, R. J. Harrison, and R. J. Littlefield, "Global Arrays: A Nonuniform Memory Access Programming Model for High-Performance Computers," *Journal of Supercomputing*, vol. 10, no. 2, pp. 169–189, 1996.

[4] P. Husbands, C. Iancu, and K. A. Yelick, "A Performance Analysis of the Berkeley UPC Compiler," in *International Conference on Supercomputing*, 2003, pp. 63–73.

[5] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: An Object-Oriented Approach to Non-Uniform Cluster Computing," in *OOPSLA '05: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 2005, pp. 519–538.

[6] B. Chamberlain, D. Callahan, and H. Zima, "Parallel Programmability and the Chapel Language," *International Journal on High Performance Computing Applications*, vol. 21, no. 3, pp. 291–312, 2007.

[7] J. Nieplocha and B. Carpenter, "ARMCI: A Portable Remote Memory Copy Library for Distributed Array Libraries and Compiler Run-Time Systems," in *Lecture Notes in Computer Science*. Springer-Verlag, 1999, pp. 533–546.

[8] D. Bonachea and J. Duell, "Problems with Using MPI 1.1 and 2.0 as Compilation Targets for Parallel Language Implementations," *International Journal on High Performance Computing Applications*, vol. 1, no. 1-3, pp. 91–99, 2004.

[9] W. Gropp and E. Lusk, "Fault Tolerance in Message Passing Interface Programs," *International Journal on High Performance Computing Applications*, vol. 18, no. 3, pp. 363–372, 2004.

[10] Q. Gao, W. Yu, W. Huang, and D. K. Panda, "Application-Transparent Checkpoint/Restart for MPI Programs over InfiniBand," in *International Conference on Parallel Processing*, 2006, pp. 471–478.

[11] G. E. Fagg and J. Dongarra, "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, London, UK, 2000, pp. 346–353.

[12] J. Liu, W. Huang, B. Abali, and D. K. Panda, "High Performance VMM-Bypass I/O in Virtual Machines," in *ATEC '06: Proceedings of the annual conference on USENIX '06 Annual Technical Conference*. USENIX Association, 2006, pp. 3–3.

[13] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Neri, and A. Selikhov, "MPICH-V: Toward a Scalable Fault Tolerant MPI for Volatile Nodes," in *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, 2002, pp. 1–18.

[14] A. Bouteiller, F. Cappello, T. Herault, G. Krawezik, P. Lemarinier, and F. Magniette, "MPICH-V2: A Fault Tolerant MPI for Volatile Nodes based on Pessimistic Sender Based Message Logging," in *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, 2003, p. 25.

[15] R. A. F. Bhoedjang, T. Rühl, and H. E. Bal, "User-Level Network Interface Protocols," *IEEE Transactions on Computers*, vol. 31, no. 11, pp. 53–60, 1998.

[16] J. Hursey, J. M. Squyres, and A. Lumsdaine, "A Checkpoint and Restart Service Specification for Open MPI," Indiana University, Bloomington, Indiana, USA, Tech. Rep. TR635, July 2006.

[17] O. Villa, S. Krishnamoorthy, J. Nieplocha, and D. M. Brown, Jr., "Scalable Transparent Checkpoint-Restart of Global Address Space Applications on Virtual Machines over InfiniBand," in *CF '09: Proceedings of the 6th ACM conference on Computing frontiers*, 2009, pp. 197–206.

[18] R. Brightwell, "A New MPI Implementation for Cray SHMEM," in *EuroPVM/MPI*, 2004, pp. 122–130.

[19] InfiniBand Trade Association, "InfiniBand Architecture Specification, Release 1.2," October 2004.

[20] N.J. Boden, D. Cohen, R.E. Felderman, A.E. Kulawik, C.L. Seitz, J.N. Seizovic, and W. Su, "Myrinet: A Gigabit-per-second Local Area Network," *IEEE Micro*, vol. 15, no. 1, pp. 29–36, February 1995.

[21] F. Petrini, W. Feng, A. Hoisie, S. Coll, and E. Frachtenberg, "The Quadrics Network: High-Performance Clustering Technology," *IEEE Micro*, vol. 22, no. 1, pp. 46–57, 2002.

[22] R. Brightwell, K. T. Pedretti, K. D. Underwood, and T. Hudson, "SeaStar Interconnect: Balanced Bandwidth for Scalable Performance," *IEEE Micro*, vol. 26, no. 3, pp. 41–57, 2006.

[23] G. Shah, J. Nieplocha, J. H. Mirza, C. Kim, R. J. Harrison, R. Govindaraju, K. J. Gildea, P. DiNicola, and C. A. Bender, "Performance and Experience with LAPI - a New High-Performance Communication Library for the IBM RS/6000 SP," in *IPPS/SPDP*, 1998, pp. 260–266.

[24] R. K. Govindaraju, P. H. Hochschild, D. G. Grice, K. J. Gildea, R. Blackmore, C. A. Bender, C. Kim, P. Chaudhary, J. Goscinski, J. Herring, S. Martin, and J. Houston, "Architecture and Early Performance of the New IBM HPS Fabric and Adapter," in *International Conference on High Performance Computing*, 2004, pp. 156–165.

[25] R. Sivaram, R. K. Govindaraju, P. H. Hochschild, R. Blackmore, and P. Chaudhary, "Breaking the Connection: RDMA Deconstructed," in *Hot Interconnects*, 2005, pp. 36–42.

[26] S. Kumar, G. Dozsa, G. Almasi, P. Heidelberger, D. Chen, M. E. Giampapa, M. Blocksome, A. Faraj, J. Parker, J. Ratterman, B. Smith, and C. J. Archer, "The Deep Computing Messaging Framework: Generalized Scalable Message Passing on the Blue Gene/P Supercomputer," in *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*, 2008, pp. 94–103.

[27] A. Vishnu and M. Krishnan, "Efficient On-Demand Connection Management Protocols with PGAS Models over InfiniBand," in *International Conference on Cluster, Cloud and Grid Computing*, 2010.

[28] A. Vishnu and D. K. P. M. K. Krishnan, "A Hardware-Software Approach to Network Fault Tolerance wwith InfiniBand Cluster," in *International Conference on Cluster Computing*, 2009, pp. 479–486.

[29] R. Gupta, P. Beckman, B.-H. Park, E. Lusk, P. Hargrove, A. Geist, D. Panda, A. Lumsdaine, and J. Dongarra, "CIFTS: A Coordinated Infrastructure for Fault-Tolerant Systems," in *ICPP '09: Proceedings of the 2009 International Conference on Parallel Processing*, 2009, pp. 237–245.

[30] J. Varma, C. Wang, F. Mueller, C. Engelmann, and S. L. Scott, "Scalable, Fault Tolerant Membership for MPI Tasks on HPC Systems," in *ICS '06: Proceedings of the 20th Annual International Conference on Supercomputing*, 2006, pp. 219–228.

[31] A. Vishnu, A. Mamidala, S. Narravula, and D. K. Panda, "Automatic Path Migration over InfiniBand: Early Experiences," in *Proceedings of Third International Workshop on System Management Techniques, Processes, and Services, held in conjunction with IPDPS'07*, March 2007.

[32] M. J. Koop, S. Sur, Q. Gao, and D. K. Panda, "High performance mpi design using unreliable datagram for ultra-scale infiniband clusters," in *ICS*, 2007, pp. 180–189.

[33] W. Yu, J. Wu, and D. K. Panda, "Fast and Scalable Startup of MPI Programs in InfiniBand Clusters," in *International Conference on High Performance Computing*, 2004, pp. 440–449.

[34] Network-Based Computing Laboratory, "MVAPICH/MVAPICH2: MPI-1/MPI-2 for InfiniBand and iWARP with OpenFabrics," http://mvapich.cse.ohio-state.edu/.

[35] R. A. Kendall, E. Aprà, D. E. Bernholdt, E. J. Bylaska, M. Dupuis, G. I. Fann, R. J. Harrison, J. Ju, J. A. Nichols, J. Nieplocha, T. P. Straatsma, T. L. Windus, and A. T. Wong, "High Performance Computational Chemistry: An Overview of NWChem, A Distributed Parallel Application," *Computer Physics Communications*, vol. 128, no. 1-2, pp. 260–283, June 2000.

[36] "Chinook SuperComputer, Environmental Molecular Science Lab, PNNL," http://emsl.pnl.gov.

[37] "Voltaire Technologies," http://www.voltaire.com/.

[38] "Mellanox Technologies," http://www.mellanox.com/.

[39] A. Vishnu, M. J. Koop, A. Moody, A. R. Mamidala, S. Narravula, and D. K. Panda, "Hot-Spot Avoidance with Multi-Pathing Over InfiniBand: An MPI Perspective," in *Cluster Computing and Grid*, 2007, pp. 479–486.