SAND2014-16631C

Toward Local Failure Local Recovery Resilience Model using MPI-ULFM

Keita Teranishi
Sandia National Laboratories
P.O. Box 969, MS 9159
Livermore, CA 94551-0969, U.S.A.
knteran@sandia.gov

Michael A. Heroux
Sandia National Laboratories
P.O. Box 5800, MS 1320
Albuquerque, NM 87185-1320, U.S.A.
maherou@sandia.gov

ABSTRACT

The current system reaction to the loss of a single MPI process is to kill all the remaining processes and restart the application from the most recent checkpoint. This approach will become unfeasible for future extreme scale systems. We address this issue using an emerging resilient computing model called Local Failure Local Recovery (LFLR) that provides application developers with the ability to recover locally and continue application execution when a process is lost. We discuss the design of our software framework to enable the LFLR model using MPI-ULFM and demonstrate the resilient version of MiniFE that achieves a scalable recovery from process failures.

Categories and Subject Descriptors

D.1.3 [**Software, Programming Techniques**]: Parallel Programming

Keywords

MPI, Fault Tolerance, User Level Fault Mitigation, PDE solvers, Scientific Computing

1. INTRODUCTION

As leadership class computing systems increase in their complexity and the component feature sizes continue to decrease, the ability of an application code to treat the system as a reliable digital machine diminishes. In fact, there is a growing concern in the reliability of extreme scale systems in future [2], exemplified by a significant reduction in mean time between failures (MTBF) to less than an hour. For such unreliable systems, it is essential for application users to manage resilience issues beyond those provided by systems and hardware.

For application users, the majority of failures are manifested as single node failures. According to Moody et al, 85% of application interrupts are related to single node failures in large PC clusters [16]. Similarly, several anecdotal evidences indicate the predominance of single node failures on Jaguar and Titan at Oak Ridge National Laboratory.

EuroMPI/ASIA '14, September 9-12 2014, Kyoto, Japan

http://dx.doi.org/00.0000/0000000.0000000.

In the current programming model, single node failures are typically handled by checkpoint/restart (C/R); it kills all the remaining processes of a program execution and then restarts the program from the most recent global snapshot of the execution. This approach fits to the current Message Passing Interface (MPI) 3.0 standard because a single process failure triggers termination of all the remaining processes. However, such a globalized reaction to single node (local) failures will be infeasible for future extreme systems because we are already running applications using more than 100,000 MPI processes. Although improvements in C/R techniques may keep its feasibility under a short MTBF, the nature of disproportional recovery for local failures needs to be addressed for efficient system use.

We address this scaling issue through an emerging resilient programming model called Local Failure Local Recovery (LFLR) that provides application developers with the ability to recover locally and continue application execution when a process is lost. In order to achieve this model, we design and implement a software framework using a prototype MPI with *User Level Fault Mitigation* (MPI-ULFM) [3], a fault tolerance capability proposed for the MPI-4.0 standard, to improve the resilience of the existing SPMD execution model.

The organization of this paper is as follows. The background information is covered in Section 2 followed by the architecture of our LFLR framework in Section 3. The recovery mechanism of LFLR-enabled applications is described in Section 4. For evaluating our preliminary implementation, a resilient version of MiniFE code is presented in Section 5, including performance evaluations up to 2,048 processes. Finally, the summary of our work and future research direction are discussed in Section 6.

2. BACKGROUND

Checkpoint/restart (C/R) has been studied for a long time in the context of HPC systems [6, 8, 14, 16, 20, 21], and successful implementations are available for distributed memory systems [8, 16, 17]. The recent work achieved a signif-

`OMPI_Comm_revoke`	Communicator Revocation
`OMPI_Comm_shrink`	Communicator Fix
`OMPI_Comm_agree`	Resilient global agreement

Table 1: A partial list of APIs in MPI-ULFM

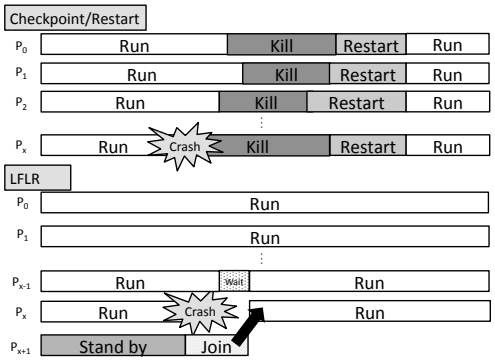

Figure 1: Execution Model of LFLR

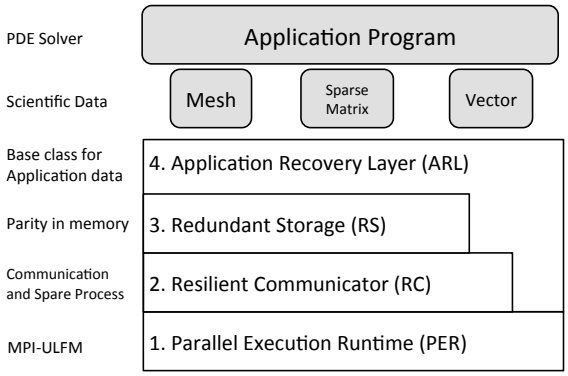

Figure 2: Architecture of LFLR Framework

icant performance improvement, reducing the overhead for accessing global file systems [16, 17].

The design of the current MPI-3.0 standard, which aborts a program for any process/node failures, has made C/R a method of choice for resilience. Despite a few fault-tolerant MPI implementations proposed for alternative schemes [1, 9], none of them has been integrated into the MPI standard. Recently, Fault Tolerance Work Group of MPI Forum proposed MPI-ULFM [3, 4] to integrate resilience capabilities into the MPI-4.0 standard, and a prototype implementation is available in public. ULFM provides the following capabilities: (1) continuing program execution through process failure, (2) process failure detection and notification, (3) communicator revocation and (4) communicator correction. Interestingly, ULFM does not support any special functions to restore failed processes, leaving the users to design their own recovery scheme with a set of APIs listed in Table 1. Despite such a low-level API support, there have already been a few use cases of the resilient version of parallel dense matrix algorithms [15] and Monte Carlo method [18].

3. LOCAL FAILURE LOCAL RECOVERY (LFLR)

Local Failure Local Recovery (LFLR) coined by Heroux [10] is a resilient programming model to overcome the disproportional recovery for single node failures practiced by C/R. LFLR permits a local recovery for a local failure to keep the remaining processes alive during the recovery. The local recovery operation is not limited to single process computation, allowing some assistance from the remaining processes. This loose restriction permits several design options for implementing LFLR.

In this paper, we adapt the LFLR model to the existing MPI SPMD model as illustrated in Figure 1; we employ the idea of spare processes reserve in order to keep the number of computing processes constant after a loss of processes. This eliminates the need for load balancing and maintaing the correctness of an application running with fewer processes. To enable this programming model, we identify several requirements listed below:

1. Runtime and middleware that permit parallel program execution to continue under process failures.

2. Runtime and middleware that provide replacement processes for the failed ones, in order to mitigate complications by running a program with fewer processes. For the process replacement, they allow application programs to query the status of all the processes (alive or lost).

3. Redundant persistent storage for restoring the data associated with failed processes.

4. Tools and frameworks to build application specific recovery schemes. These services would provide flexible options for re-constituting the lost local state of a given application.

In the following sections, we discuss our implementation for each of the requirements.

3.1 LFLR Framework

Based on these requirements discussed in the previous section, we design a software framework to enable the LFLR model for large scale parallel applications as illustrated in Figure 2; the numbers (1-4) in the individual components correspond to the numbers associated with the requirement listed above. Our framework, object-oriented C++ code, provides a seamless integration of all the four requirements through abstraction of each requirement. The labels on the left of Figure 2 indicate our implementation choice for each layer to demonstrate how the existing technologies can be assembled to build an LFLR model.

The bottom two layers denoted as Parallel Execution Runtime (PER) and Resilient Communicator (RC) manage program execution and resource allocation to handle process failures, respectively. In our approach, the user is responsible for allocating extra processes at job launch because typical HPC systems do not always support dynamic process spawning to fill failed processes. At runtime, RC manages a number of parallel execution contexts by separating processes into groups, each of which serves for application execution, application data redundancy and process recovery respectively. This requires several MPI communicators including global MPI communicator (`MPI_COMM_WORLD`) and sub-communicators as illustrated in Figure 3. In this figure, all message passing calls for the existing application are made through `Compute_Comm`. The other sub-communicators such as `Group_Comm` and `Compute_Group_Comm` are split from the global communicator to serve for the recovery purpose such as `commit` and `restore` in the Redundant Storage; they are not relevant to application-specific communicator splitting, and the management of these split communicators is left to the future work.

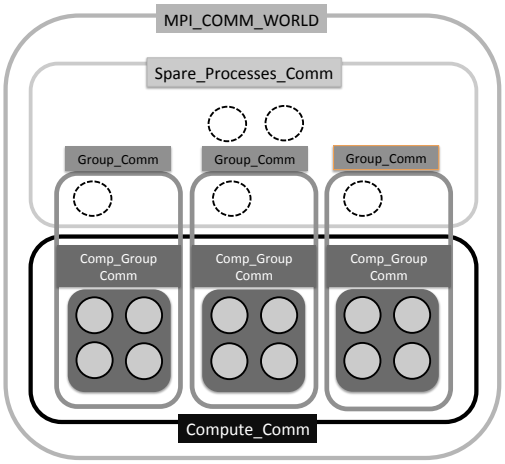

Figure 3: Splitting of `MPI_Comm` by Resilient Communicator: the circles with dashed line indicate the spare processes.

For message passing, RC provides two functionalities: (1) direct access to all the MPI communicators to allow MPI calls directly from an application program and (2) wrapper functions to perform message passing calls including one-to-one send/receive and collectives with different types of resilience capabilities not directly supported by MPI-ULFM. The design of the wrapper functions is similar to those in BLACS [7], which supports different message passing software and parallel computing runtime other than MPI.

The Redundant Storage (RS) layer provides a temporary space for every local process so that the spare processes can retrieve the data for failed ones. In traditional C/R, the storage for checkpoint involves file I/O, which allows the restarted program to retrieve the data to restore its original state. Instead, our framework leverages the user-space memory to minimize the performance impact of applications while providing data persistence unaffected by process failures. To meet this goal, we employ the ideas from diskless checkpointing [19, 20] in which the spare processes accommodate a space for data redundancy combined with local checkpointing. These spare processes, controlled by RC, dedicate their memory space to keep the parity of individual data structures distributed across the processes. The storage cost per spare process never exceeds any of the computing processes in the associated process group. The parity

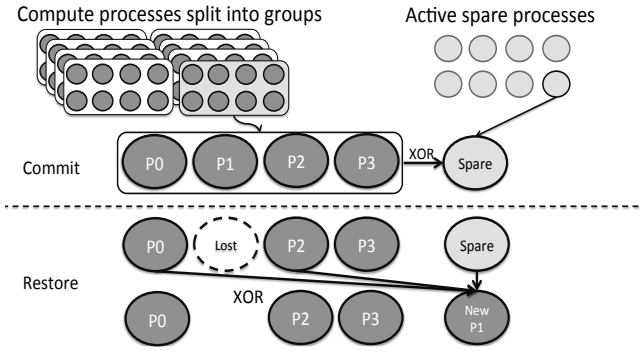

Figure 4: Commit and Restore using dedicated Parity.

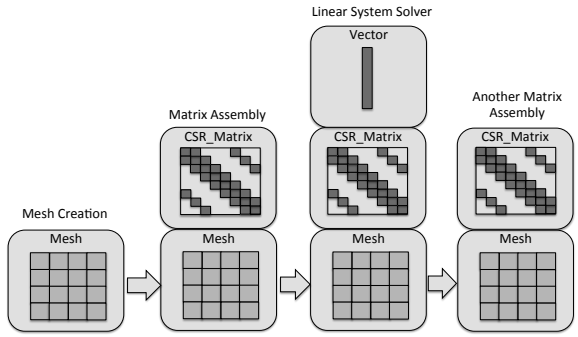

Figure 5: Progress of the stack in `LFLR_registry` for a typical PDE-based application.

operations are implemented with `MPI_Reduce` using binary XOR operation as illustrated in Figure 4. In the RS component, we provide APIs for `commit` and `restore` to recover a specified memory region using a spare process associated with the process group defined by `Group_Comm` in Figure 3.

The Application Recovery Layer (ARL) bridges between data structure (Vector, Matrix and Mesh) and the RS and RC layers bringing these elements together to implement application specific recovery. This design enables to add a recovery mechanism to the existing scientific data structures and classes as seen in the software frameworks such as Trilinos [11]. ARL has two capabilities to enable application-specific recovery: (1) abstract class to allow users to implement a recovery scheme specific to individual data structure and (2) registry class to monitor the status of every recoverable data objects. An abstract class, `recoverable`, permits the users to design data-structure specific recovery scheme using several method calls for accessing to the RS. For each data class, `recoverable` encapsulates the commit and recovery schemes in its class functions, `commit` and `restore`, respectively. The object monitor class, `LFLR_registry`, maintains a stack of pointers to the active data objects as illustrated in Figure 5. The details of our application data recovery scheme is described in Section 4.2.

4. RECOVERY OF APPLICATIONS

In the current LFLR framework, MPI-ULFM is responsible for failure detection and notification to trigger application recovery. For detection, MPI-ULFM returns an error flag when receiving messages from a failed process. Failure notification can be implemented using APIs such as `OMPI_Comm_revoke` and `OMPI_Comm_agree`. In our use case described in Section 5, we have found `OMPI_Comm_agree` suitable for iterative linear system solvers to stop iterations across all the processes in the same iteration. More generalized failure detection/notification is left to the future work.

When recovering an application, our LFLR framework restores three entities: process, data and state. The following subsections describe how our framework handles them, respectively.

4.1 Process Recovery

For process recovery, the application code invokes RC to make a `recover` call to correct its internal MPI communica-

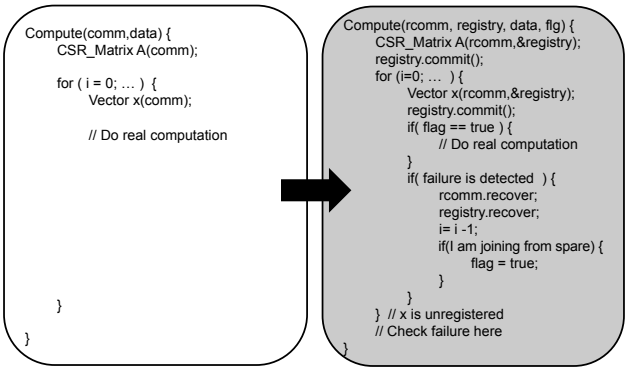

Figure 6: Code modification to enable LFLR. A spare process has `flg=false` to skip the real computation. Once it joins the computing processes, `flg` is changed to `true` to perform real computation for the lost process.

tors. The correction begins with `MPI_Comm_shrink` for the global communicator (copied from `MPI_COMM_WORLD`). Then, a spare process takes over the lost process through rank reordering by `MPI_Group_incl` and `MPI_Comm_create`. After this correction, `Compute_Comm` is created from the new global communicator. The other sub-communicators follows the correction process after the global communicator is fixed.

4.2 Data Recovery

After all MPI communicators are updated, a new process joining from the spare reserve needs to recover the data and state of the application. The `LFLR_registry` object in ARL then iterates its own stack of pointers for the allocated objects from the bottom. This bottom-up ordering ensures the recovery of the primary data objects prior to the recovery of the objects dependent on these. For distributed data structures, the recovery involves `restore` calls to recover the local data as indicated by Figure 4. For non-distributed data structures such as application parameters, the recovery schemes can be implemented without RS.

4.3 Application State Recovery

For application-based C/R [16, 17, 19, 20], users need to design a way to locate the most recent successful checkpoint of a given application code so that it can restart with an appropriate roll-back. In our approach, the spare processes keep abreast to the application state by executing a "skeletonized" code of the application, by which we mean that the spare processes participate in the program logic execution, but have no portion of the distributed data. In the application program source, these spare processes execute the same program of the compute process, but skip the real computation except initialization of data objects that requires binding to `LFLR_registery`. Figure 6 presents a source code modification to enable LFLR. This involves some coding effort for the users to write extra `if` statements, but it can be mitigated by writing pre-built classes for basic scientific data and compute kernel functions. Many robustly implemented applications already contain some logic that handles this situation, since partitioning of data may naturally result in a process have no portion of the distributed data.

5. USE CASE: RESILIENT MINIFE

We present a use case of the LFLR framework with MiniFE from the Mantevo mini-applicaition collection [12]. MiniFE is a parallel finite element analysis code for thermal PDEs on 3D regular mesh written in C++. The code includes three major functions (1) mesh generation, (2) construction of the sparse linear system and (3) single linear system solution using Conjugate Gradient (CG) iterations. In real PDE applications, a number of linear systems are solved to understand nonlinear or time-dependent behavior of physical systems. Exploring a single linear system solution is, therefore, oversimplified to understand the behavior of such applications from the resilience perspective. For this reason, we modify the source code to emulate a time dependent PDE solver, which iterates a number of linear system solutions with a right hand side updated by the solution of the previous linear system as shown in Algorithm 1. The sparse matrix data is kept constant during the time stepping. The source of MiniFE is template based C++ code to describe all data classes and methods, making it straightforward to integrate with our LFLR framework.

The process failure is emulated by `kill` system call on a randomly chosen process at any matrix or vector operations in the linear system solution. In the solver code of MiniFE, a failure is detected by `MPI_Wait` for non-blocking receive at sparse matrix vector multiplication (SpMV) and `MPI_Allreduce` in vector dot product. For failure notification, `OMPI_comm_agree` is called at the end of the iteration to terminate the solver.

Algorithm 1 Resilient Time Step MiniFE

```
Create Mesh Ω
Commit
Create Matrix A from Ω
Create Initial x_0 from Ω
Commit
while i = 1 until the last time step do
    Commit
    Create b_i from x_{i-1}
    Solve A x_i = b_i (process failure occurs here)
    if Process failure is detected then
        Recover, i := i − 1
    end if
end while
```

5.1 Performance of Resilient MiniFE

The performance testing is conducted on Sandia's TLCC2 cluster that comprises 1,272 nodes (19,712 cores). Each compute node has dual sockets of 2.6Ghz 8-core Intel SandybrdigeEP CPUs with 64 Gbyte 1,600MHz DDR3 RAM. The interconnect is 4xQDR QLogic Infiniband in Fat-Tree topology. For MPI-ULFM, we applied a modification to the source of the latest commit (`b24c2e4` as of May 1, 2014) to apply tree-based resilient collectives [13] primarily for

	512 procs	1,024 procs	2,048 procs
Original	5.65 sec	16.53 sec	30 min+ (hang?)
Tree-based	**4.04** sec	**6.55** sec	**12.07 sec**

Table 2: Performance of Communicator Fix: the cost of `OMPI_Comm_shrink` and a few `MPI_Comm_Create` calls.

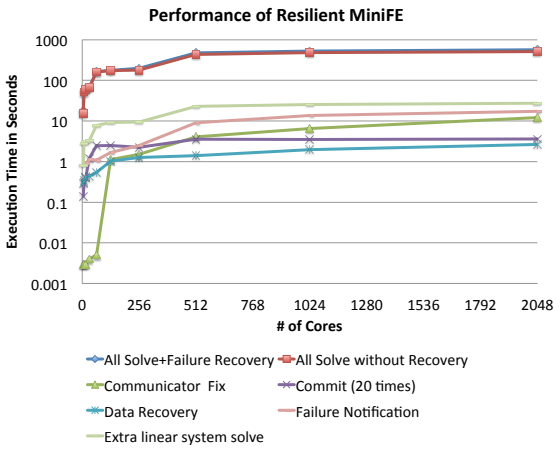

Figure 7: Execution Time of Resilient MiniFE including all recovery cost. The cost for `All solve + recovery` and `All solve without recovery` are hard to distinguish in the logarithmic scale.

communicator creation calls used in `MPI_Comm_create` and `OMPI_Comm_shrink`. The original code uses one-to-all and all-to-one algorithm, which exhibits a poor scalability. The performance improvement with the tree-based code is shown in Table 2; we have observed that the original code fails to finish in 30 minutes on 2,048 processes. The parameters for the collective communications for `mpiexec` are set to `tuned,basic,ftbasic`. All the source code (including MPI-ULFM) has been compiled with GNU-4.7.2 compilers. The performance of the code was measured as large as 2,048 processes (cores).

We study the weak scalability of the resilient MiniFE from $64 \times 64 \times 64$ grid (262K in matrix size) for 4 processes to $512 \times 512 \times 512$ grid (134M in matrix size) for 2,048 processes. The data size per process is approximately 23 Mbytes. A process group size is set to 128 for the efficiency of `commit` and `restore`. The recovery involves process replacement (communicator fix), data recovery and repeating the same linear system solution. The resilient MiniFE performs 20 time steps, and one of the processes is killed during these time steps.

The execution time of MiniFE and its recovery overhead are presented in Figure 7, indicating a small effect to the overall performance by process failure. The cost for commit shows a moderate increase for small process counts, but the growth after 128 processes is relatively small due to the grouping. The data recovery cost is very negligible as it is executed within a single process group and leaves the other group to start roll-back recovery immediately. The cost of the communicator fix is more expensive than the other recovery operations. Despite the performance improvement of the resilient agreement algorithm in `OMPI_Comm_shrink` and `MPI_Comm_create`, the execution time grows almost linearly as indicated by Figure 8. We further investigate the performance of every single resilient agreement performed in the communicator fix routine. Interestingly, a large fluctuation is observed in the execution time as shown in Figure 9. In particular, the first two calls (in `OMPI_Comm_Shrink`) spend a significant amount of time compared to the subsequent agreement calls. We conjecture that MPI-ULFM has some

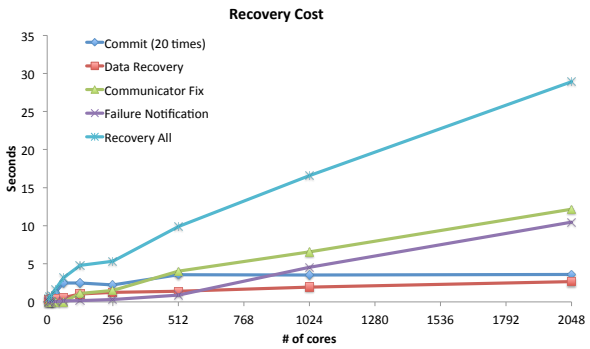

Figure 8: Execution Time of Individual Recovery Components in Resilient MiniFE.

problems in the network setup for managing new communication patterns incurred by a process loss.

6. CONCLUSION

In this paper, we have described a software framework to enable the LFLR resilience model for SPMD programming model. For the backend of our framework, we employ MPI-ULFM, a fault tolerant MPI proposed for the future MPI standard. Our framework allows application developers to extend their MPI programs so that they can run through single process failures, thereby eliminating the need for checkpoint/restart (C/R) and global files systems. The use of hot spare processes combined with the ideas of disk-less checkpointing permits scalable recovery and relaxes the complications for running applications with fewer processes. Future middleware and I/O technology can be plugged into our framework for more flexible options of the recovery schemes. Our preliminary results indicate that a scalable recovery for application data and state is achievable though there are some performance issues in MPI-ULFM, in particular the resilient collective for the communicator modification routines. The current implementation of MPI-ULFM is still a prototype; the performance is the secondary interest at this moment. Performance improvements in future releases would resolve this problem and make LFLR a method of choice over the state-of-art C/R for extreme scale systems.

The future work includes two directions. The first direction is performance tuning and analysis of our LFLR frame-

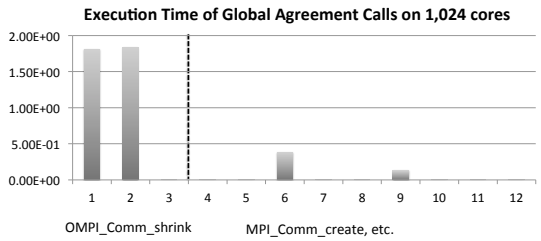

Figure 9: Execution time of a sequence of 12 global agreement calls (tree-based) in the communicator fix. The numbers in x-axis indicate the calling sequence. Many of the calls spend as small as 0.001 seconds.

work for a larger number of processes and multiple process failures. This will also involve more effective use of the hot spare processes to take over some resilience functionalities beyond those described in this paper. The second direction is exploring more sophisticated recovery semantics such as roll-forward and the algorithm-based fault tolerance such as FT-GMRES [5].

7. ACKNOWLEDGMENTS

The authors would like to thank George Bosilca, Robert Clay, Pedro Diniz and Mark Hoemmen for interesting discussions related to this work. This work was supported by the U.S. Department of Energy (DOE) National Nuclear Security Administration (NNSA) Advanced Simulation and Computing (ASC) program. Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

8. REFERENCES

[1] R. Aulwes, D. Daniel, N. Desai, R. Graham, L. Risinger, M. A. Taylor, T. Woodall, and M. Sukalski. Architecture of LA-MPI, a network-fault-tolerant MPI. In *Proceedings of 18th International Parallel and Distributed Processing Symposium.*, pages 15–, April 2004.

[2] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, K. Yelick, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Keckler, D. Klein, P. Kogge, R. S. Williams, and K. Yelick. Exascale computing study: Technology challenges in achieving exascale systems peter kogge, editor & study lead. Technical report, Defence Advanced Research Projects Agency Information Processing Technology Office (DARPA IPTO), September 2008.

[3] W. Bland, A. Bouteiller, T. Herault, G. Bosilca, and J. Dongarra. Post-failure recovery of mpi communication capability: Design and rationale. *International Journal of Highe Performance Computing Applications*, 27(3):244–254, 2013.

[4] W. Bland, A. Bouteiller, T. Herault, J. Hursey, G. Bosilca, and J. Dongarra. An evaluation of user-level failure mitigation support in mpi. In *Proceedings of Recent Advances in Message Passing Interface – 19th European MPI Users' Group Meeting, EuroMPI 2012*, Vienna, Austria, September 2012. Springer.

[5] P. Bridges, K. Ferreira, M. Heroux, and M. Hoemmen. Fault-tolerant linear solvers via selective reliability, 2012, `http://arxiv.org/abs/1206.1390`.

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

[7] J. J. Dongarra and R. C. Whaley. A user's guide to the BLACS v1.1. Technical Report 94, LAPACK Working Note, 1997.

[8] J. Duell, P. Hargorive, and E. Roman. The design and implementation of berkeley lab's linux checkpoint/restart. Technical Report LBNL-54941, Lawrence Berkeley National Laboratory, 2002.

[9] G. E. Fagg and J. J. Dongarra. Building and using a fault-tolerant mpi implementation. *Int. J. High Perform. Comput. Appl.*, 18(3):353–361, Aug. 2004.

[10] M. A. Heroux. Toward resilient algorithms and applications, `http://arxiv.org/abs/1402.3809`, 2014.

[11] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, and K. S. Stanley. An overview of the trilinos project. *ACM Trans. Math. Softw.*, 31(3):397–423, Sept. 2005.

[12] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich. Improving performance via mini-applications. Technical Report SAND2009-5574, Sandia National Laboratories, 2009.

[13] J. Hursey, T. Naughton, G. Vallee, and R. L. Graham. A log-scaling fault tolerant agreement algorithm for a fault tolerant mpi. In *Proceedings of the 18th European MPI Users' Group Conference on Recent Advances in the Message Passing Interface*, EuroMPI'11, pages 255–263, Berlin, Heidelberg, 2011. Springer-Verlag.

[14] K. Li, J. Naughton, and J. Plank. Low-latency, concurrent checkpointing for parallel programs. *Parallel and Distributed Systems, IEEE Transactions on*, 5(8):874–879, Aug 1994.

[15] P. Luszczek, G. Bosilca, A. Bouteiller, and J. J. Dongarra. *FT-LA*, `http://icl.cs.utk.edu/ft-la/software/index.html`.

[16] A. Moody, G. Bronevetsky, K. Mohror, and B. de Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, pages 1–11, 2010.

[17] A. Moody and K. Mohror. Scalable Checkpoint Restart, `http://sourceforge.net/projects/scalablecr/`.

[18] S. Pauli, M. Kohler, and P. Arbenz. A fault tolerant implementation of multi-level monte carlo methods. Technical Report 793e, Institute of Theoretical Computer Science, ETH Zürich, August 2013.

[19] J. S. Plank, Y. Kim, and J. Dongarra. Algorithm-based diskless checkpointing for fault tolerant matrix operations. In *25th International Symposium on Fault-Tolerant Computing*, pages 351–360, Pasadena, CA, June 1995.

[20] J. S. Plank, K. Li, and M. A. Puening. Diskless checkpointing. *IEEE Transactions on Parallel and Distributed Systems*, 9(10):972–986, October 1998.

[21] E. Roman. A survey of checkpoint/restart implementations. Technical Report LBNL-54942, Lawrence Berkeley National Laboratory, 2002.