Mapped-File Based Techniques for Fault Recovery in Parallel Simulations

Issakar Ngatang, and Masha Sosonkina
Department of Modeling, Simulation and Visualization Engineering
Old Dominion University
Norfolk, VA USA
E-mail: {ingat001, msosonki}@odu.edu

Abstract— *Reliable storage, such as a hard drive, is limited by its read/write speed; thereby affecting the checkpointing mechanisms that rely on such a type of storage. This paper proposes and implements several fault-tolerance mechanisms for high-performance simulations that employ mapped files in memory instead of disks. In particular, one proposed technique avoids checkpointing while the two others improve on diskless checkpointing. These mechanisms were tested in LULESH and CoMD, a hydrodynamics and a molecular dynamics proxy simulations, respectively. The experiments have shown that all the proposed techniques may reduce permanent storage overhead and sufficiently recover from a complete simulation shutdown. The CoMD average checkpointing time using mapped files took about 1/125th of that for the checkpoint to a hard drive using regular files, and the mapped files checkpoints of LULESH took about 9/500th of that for its checkpointing to disk.*

Keywords: Fault Tolerance, Mapped-File Checkpointing, Diskless Checkpointing, CoMD, LULESH

1. Introduction

The race for performance and reaching the exascale era in High Performance Computing (HPC) is a road full of challenges. They include, in particular, the problem of energy consumption [1], [2] and fault occurrences [3], [4], [5], [6] that can limit the performance of supercomputers. This paper focuses on fault tolerance. Failures may increase due to the growing number of computation units working on a problem solution. Failure is detrimental to large scale simulations as it could mean a loss of all the effort produced from their start up to the moment of occurrence of the fault. There are two main types of failures [5]: Silent Data Corruption, also called *soft* faults, and *hard faults*, which are those involving hardware or operating-system failure. This paper focuses on hard faults, which directly result in stopping the execution of the program, and, thus, are known as fail-stop failures. There are two main challenges with faults: They first need to be detected then they need to be dealt with.

Checkpoint/restart is the most used approach in fault tolerance mechanisms. It can deal, to some extent, with both soft and hard faults. The principle of checkpointing is to periodically save, in some way, the state of the simulation. This may be a small set of variables from the program or the whole memory footprint allocated to the processes belonging to the simulation. Making checkpoints incurs an overhead that is often substantial in comparison to the overall simulation performance. Finding new approaches that can reduce or nullify this overhead is key to good performance for HPC simulations. There are several ways to save checkpoints. Depending on system reliability, it might be needed to have checkpoints saved to permanent storage, or a volatile storage such as the main memory. The checkpoint storage medium has a great impact on the performance of the simulation. Writing to a disk has a bigger overhead compared to writing to the main memory. The disk is permanent while main memory data is volatile. For this reason, checkpoints stored in disks can withstand entire node failures, power failures and failures that lead to a computational system restart. Main memory checkpoints would be lost the moment power is cut off. The work presented in this paper explores a hybrid checkpointing medium consisting of memory-mapped files. In this work, checkpoints may have a longer life due to the properties of this medium. The main contributions of this work are in:

- Exploring the applicability of memory-mapped files along with their properties as an alternative to main-memory checkpoints.
- Introducing a checkpointing-free technique to reduce the overhead and memory footprint of checkpointing.

1.1 Related Work

Fault tolerance is a major concern in HPC. Several authors have produced works on different techniques to build resilience in simulations [7], [3], [4], [5], [6]. Large body of the research is directly addressing checkpoint/restart mechanisms, which is the most used strategy to fault tolerance. Nicolae *et al.* presented a two-stage checkpointing mechanism for cloud environments in [8]. The first stage consists on saving simulation state to virtual machine disk. After this, the second stage takes a snapshot of the virtual machine disk and saves it in a pool of checkpoints. This mechanism uses permanent storage as checkpoint medium, which is limited by the speed of read and write operations. A

different type of checkpoints is presented in [9]. In this work, input data in a task-based simulation are checkpointed before each task. A specific task can restart from its checkpointed input in case of failure. Zheng *et al.* presented a single and a double in-memory checkpointing mechanisms in [10], [11]. Both works replicate checkpoint data in the memory of other nodes, allowing recovery in case of a single node crash. Both methods are limited by the fact that the checkpoints are short-lived, since they are saved in memory. A system shutdown would lead to complete loss of all checkpoints. Another work on diskless checkpointing is presented by Plank *et al.* in [12], where checkpointing is built into the simulation using the parity checkpointing technique. Moody *et al.* presented a multi-level checkpointing mechanism with their library SCR [13]. SCR is capable of making cheap local checkpoints to the local memory, flash, and disk, but also more expensive and more resilient checkpoints to the parallel file system. Di *et al.* present an optimization of the multi level checkpointing in [14]. Their aim is to determine the most efficient use of multi level checkpoints by determining optimal checkpoint period for the different levels, the selection of the levels based on observed failure in a system. Hargrove and Duell presented a different approach in [15]. Their solution is geared toward the system, allowing for job preemption in case of fault prediction. Their method also does not require any code modification for the application. Dongarra *et al.* cover a more extensive set of techniques, but mainly checkpoint/restart in their work [16]. The authors present different variants of checkpoint/restart techniques such as coordinated and uncoordinated checkpoints and establish mathematical foundations such as the optimal checkpointing period. Although all these works present valuable checkpoint/restart techniques, they all use traditional storage to save the checkpoints. Multilevel checkpointing has advantage that it can choose from different storage with different speed but still relies on parallel file system to ultimately save its checkpoints. The work in this paper stands out by aiming to explore non-traditional storage media; specifically to examine the use of mapped files to save checkpoints.

The remainder of this paper is presented as follows: Section 2 overviews interprocess communication techniques specifically memory-mapped files, Section 3 presents implementation details, Section 4 compares several techniques and Section 5 concludes.

2. Memory-Mapped Files

Interprocess communication (IPC), is a wide variety of tools implementing communication among processes at different levels. Belonging to this category are memory-mapped files. The current work is based on the memory-mapped files. Memory-mapped files are regular files mapped to main memory and treated as normal memory blocks[1].

A process needing access to a memory-mapped file would add the corresponding memory block to its memory space through a given pointer. Many processes can simultaneously access the same memory-mapped file, each one of them with its own pointer, different from the other processes, but all pointing to the same memory block. Memory-mapped files have the benefits of regular files, in the fact that they are persistent even after termination of the process which created them. They have to be specifically removed from memory just like C/C++ dynamic memory allocations. This persistent nature of memory-mapped files can be exploited, and this paper presents a work that aims to use memory-mapped files for fault tolerance in HPC simulations. Memory-mapped files are accessible at speeds comparable to main memory, due to the fact that the file is mapped to memory.

The principle experimented in the present work is as follows: a memory-mapped file is created and holds a simulation variable. The program can run as normal, updating all variables as the instructions are executed. When a fault occurs, there is no time to specifically delete the mapped file containing the variable, which is a needed step to deallocate and delete the file mapped to memory. The memory block would not exist at that moment, but the file would. At restart time, the file would be remapped to memory in order to recover the variable from the previous failed run. This mechanism may be applied to checkpointing, which is the focus of this paper.

3. Proposed Techniques

The persistent nature of memory-mapped files can be used in different ways to implement fault tolerance in HPC simulations, mainly in checkpointing fault-tolerance mechanisms. This paper presents three different implementations of checkpoint/restart techniques. The first one is a standard checkpoint/restart, using mapped files to save checkpoints. The second avoids explicitly making checkpoints, and the third replicates checkpoints through process communication in the same fashion as in-memory double checkpoints [11]. In these three implementations, only the set of data that dynamically changes during the course of execution is checkpointed. In particular, variables that are reinitialized at each time step and variables that are only computed at the initialization time are not checkpointed. The implementations presented in this paper all use POSIX mapped files in C/C++ language. The mapped files are created but then deleted only after clean program termination. POSIX implementation of mapped files has *kernel persistence*, which means that the files are deleted after system shutdown or when all attached processes explicitly unmap the file using `shm_unlink` which leads to deletion of the mapped file [17].

[1]Memory Mapped Files `http://beej.us/guide/bgipc/output/html/multipage/mmap.html`

3.1 Mapped-File Checkpointing (MAP-Check)

In most simulations with checkpoint capabilities, the checkpoints are saved to the hard drive for persistence. Saving checkpoints in the main memory is not wise because they are short-lived. A simulation crash would result in a complete loss of all the checkpoints. The POSIX implementation of mapped files has the benefit of file persistence as stated earlier. In the MAP-Check, developed here, checkpoints are periodically made in memory-mapped files. In particular, a simulation will have the key data needed for recovery duplicated and stored in mapped-file variables. MAP-Check is able to recover from the faults that abort the simulation, but not the entire computing platform. Checkpoint content includes a particular set of simulation variables, such as the time step and simulation data, updated dynamically and which are not reinitialized at each time step. Note that, to implement MAP-Check, modifications to the simulation source code are necessary to duplicate the variables to be recovered; thereby, increasing the memory footprint of the simulation. In a nutshell, MAP-Check works as follows: At the initialization, the simulation checks whether or not there are files to be mapped, then, if found, maps them into memory and checkpoints periodically into them. Obviously the periodicity of checkpoints is important, as with any checkpointing mechanism, and has been extensively studied already (see, e.g., [16]). Hence, a reasonable periodicity is assumed here, without its further investigation. Specifically, this work has assumed a periodicity based on the number of simulation time steps.

3.2 Variables Stored in Mapped Files (MAP-Var)

The MAP-Var, developed here, differs from MAP-Check in that the checkpoints are explicitly avoided. The goal is to reduce the checkpointing overhead and memory footprint used in MAP-Check. In MAP-Var, POSIX mapped files are also used for the data to be recovered. At each time step the variables are updated in the mapped files. In particular, for MAP-Var, the fault tolerance is achieved similarly to that for MAP-Check in terms of using the mapped files. The differences lie in manipulating extra pointers for the former as opposed to employing the file I/O for the latter. Observe, that the applicability of MAP-Var is no less than that of MAP-Check because they both target faults in the same category, when computing platform stays operational. Additionally, MAP-Var not only avoids explicit checkpointing overheads but also needs only a single copy of the variables slated for recovery.

3.3 Mapped-File Checkpointing with Replication (MAP-Comm)

It is known [17] that POSIX mapped files cannot survive platform crash. One way to recover a simulation when one or more compute nodes fail is to replicate checkpoints obtained in MAP-Check. Each process keeps a copy of its own checkpoint as well as a copy of the most recent checkpoint from one or more "buddy" process(es) located in other compute node(s). The number of copies to be made depends on how much overhead is acceptable for the simulation. If the computing system is stable, with few faults occurring, it would make sense to save a checkpoint from only one "buddy" process. Conversely, for an unstable system, an extra overhead from replication and communication is needed to recover from multiple node crashes. At the limit, in a system with n nodes, a replication with the highest overhead may recover from $(n-1)$-node crashes.

MAP-Comm extends the in-memory double checkpoints of [11]. Specifically, MAP-Comm checkpoints periodically into mapped files, as MAP-Check does so, followed by the checkpoint replication phase, in which the checkpoints are exchanged with the "buddy" process. The present work differs from [11] in that, additionally, it enables the recovery from the *total* abort of the simulation, i.e., when *all* the processes died. In contrast, [11] recovers from the main memory, hence some processes must be alive to recover the rest. Note that, compared with MAP-Var and MAP-Check MAP-Comm broadens the fault category that it is able to withstand, to some nodes of the computing platform being faulty in addition to the simulation abort. They recover only when some of the processes are dead, because they use main memory to save checkpoints. In the present implementation, recovery can be done when all processes are down but the computational platform did not restart, and can recover when all processes are down and some nodes of the computational platform have been restarted. This work extends the method presented in [11].

In this paper, a light-overhead replication technique is considered as follows: Each node hosts the checkpoints of exactly one "buddy" node in a mapped file, thereby forming a logical ring of neighbor-processor node numbers, as depicted in Fig. 1 for six nodes. Each node has two processes (ranks, depicted as squares in Fig. 1) participating in MAP-Comm to facilitate the checkpoint exchange and recovery, similar to [11]. For example, if `Node 4` crashes, the only checkpoint copies for $rank8$ and $rank9$ are in `Node 5`, so $rank10$ and $rank11$, respectively, have to send these copies back to recover. Note that, in Fig. 1, the red dotted-line circles represent the nodes that may fail and the entire simulation still recovers. Hence, such a ring topology with only one remote "buddy" node may withstand multiple-node failures.

4. Experiments

The techniques presented in Section 3 were implemented for two HPC simulations, CoMD [18] and LULESH [19]. These simulations were chosen because they represent proxy of realistic computations in classical molecular dynamics and

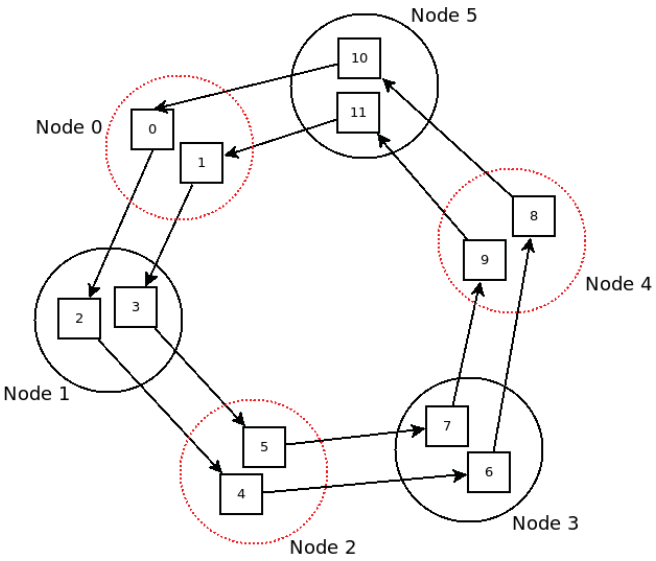

Fig. 1: Checkpoint replication topology for six compute nodes with two processes each.

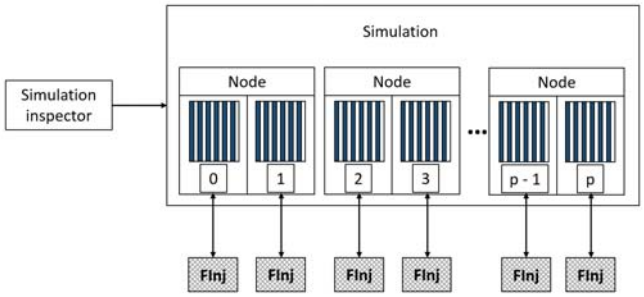

Fig. 2: Experiment architecture.

hydrodynamic shock calculation, respectively, that are both computation- and memory-intensive. CoMD is implemented in C with MPI/OpenMP hybrid model. CoMD implements two different potentials, embedded atom model (EAM) and Lennard Jones (LJ). The work in this paper was based on the LJ potential. LULESH also uses a hybrid MPI/OpenMP programming model. LULESH implements a Lagrangian method, and is written in C++.

4.1 Experimental Setup

The experiments presented here were conducted on Edison, a supercomputer at the National Energy Research Scientific Center (NERSC). Edison is a XC30 Cray cluster computer with a peak performance of 2.57 petaflop/s [20]. Edison is composed of 5576 compute nodes each equipped with two sockets, hosting a 12-core Intel "Ivy Bridge" architecture clocked at 2.4GHz each. Preliminary experiments were also performed on the Turing High Performance Cluster at Old Dominion University.

Figure 2 shows the high-level architecture of the experimental setup, but more specifically, each Edison compute node hosts two MPI processes with 12 OpenMP threads each. Along with these, sits there is a fault injector `FInj` process per one MPI process; and a `Simulation Inspector` process per entire execution. The rôle of the latter two process types are as follows: **Fault injection.** It is used randomly introduce a hard fault into the running simulation. The fault injector is composed of a set of light processes set to sleep for a randomly calculated period of time and then killing a specified process upon waking up. The random numbers are drawn from an exponential distribution in order to simulate fault occurrence time. The fault injector stands outside of the simulation and is light enough not to affect the simulation performance. The fault injector introduces a single fault, terminating one process. This single fault results in termination of remaining running processes by the scheduler. Following this, recovery attempt is engaged by the simulation inspector. MAP-Comm had a slight difference in its fault injector. In this version, after injection and abortion of the simulation, mapped files in some nodes are deleted in order to simulate the crash effect of these nodes, forcing the fault tolerance mechanism to handle the case by requesting saved checkpoints from a neighboring process. **Simulation inspector.** The simulation inspector is a simple stand-alone script that examines the output files from the previous simulation execution and determines if the termination was clean or not. If the termination is found to be the result of a fault, a recovery is initiated and the simulation is set to be restarted.

CoMD runs were executed on 2, 4, 8, and 16 compute nodes occupying 48, 96, 192, and 384 cores, respectively, with 16, 32, 64, and 128 million of particles each. LULESH was run on 4, 32, 108, 256, and 500 nodes occupying 96, 768, 2592, 6144, and 12000 cores, respectively, for problem sizes of 8, 64, 216, 512, and 1000 million elements. Each simulation was run for 100 time steps, taking checkpoints every 10 steps, to meaningfully compare load and read/write speeds for the different implementations. Along with these was implemented a simple disk checkpointing method, denoted here as File-Check, to roughly estimate a performance difference of the proposed methods with respect to writing to a disk using the parallel file system. Comparisons with a more efficient disk checkpointing are desirable and are left as near-term work. It is expected, however, that these comparisons will not change drastically the observations and conclusions presented in Sections 4.2 and 5.

4.2 Results

Tables 1 and 2 depict average times (in seconds) to make a single checkpoint, where the column headings represent the total number of cores (12 per one MPI process) used to run the simulation. Firstly, observe that saving the checkpoints to the disk takes much more time compared to any of the mapped files implementations, which is not surprizing.

Table 1: Average checkpointing times for CoMD (in seconds) across different implementations.

	Number of cores			
	48	**95**	**192**	**384**
MAP-Check	0.26	0.25	0.25	0.25
MAP-Comm	1.67	1.72	1.63	1.66
File-Check	31	29.16	32.12	27.25

Table 2: Average checkpointing times for LULESH (in seconds) across different implementations.

	Number of cores				
	96	**768**	**2592**	**6144**	**12000**
MAP-Check	0.12	0.12	0.12	0.12	0.12
MAP-Comm	0.64	0.64	0.64	0.64	0.64
File-Check	6.54	6.46	6.52	6.5	6.52

This is due to the low read/write speed (48 GB/s of I/O bandwidth [21]) of hard drives. Even though the file system is tuned for high performance, the limitations are still visible for the obtained checkpoint sizes, which are approximately 91.55 MB and 750.49 MB for CoMD and LULESH, respectively. From Tables 1 and 2 it may be inferred that checkpointing CoMD to the hard drive takes about 119 times longer on average than what it takes to checkpoint CoMD to a memory-mapped file, while disk checkpointing LULESH takes on average *only* 54 times longer to save as compared to memory-mapped files. Note that, in this work, mapped files were not explicitly created. Although a careful study of how disk speeds affect the presented mapped-file based approaches is warranted, it has been left as future work, while its initial experiments indicate low overheads from explicit file creation.

Secondly, average checkpoint times in Tables 1 and 2 indicate that MAP-Comm incurred a higher cost than that of MAP-Check mainly because MAP-Comm replicates checkpoints across nodes through communication. For CoMD, the checkpoint communication cost averages at around 1.45 seconds per checkpoint for 384 cores. Such a low communication overhead was attained due to an efficient MAP-Comm implementation leveraging the communication/computation overlap as developed in authors' earlier work [22]. Specifically, here checkpoint messages were held until the simulation-specific (halo) communication was ready to take place to overlap both communications. In LULESH, on the other hand, it was not possible to leverage such an overlap, and its average cost for one checkpoint was observed at 0.49 seconds on 12000 cores.

Thirdly, for several core counts, Figs. 3 and 4 show the execution times of CoMD and LULESH, respectively, normalized by the time of the corresponding *clean* execution[2] across the four fault-tolerance mechanisms, MAP-Var, MAP-Check, MAP-Comm, and File-Check. MAP-Var is noticeably the fastest, mostly because the time taken to make checkpoints for others adds up over time. It can be seen that sometimes MAP-Var takes about the same time as a clean execution (down time is not taken into consideration). MAP-Check is next best performing because it does not require communication, contrary to MAP-Comm. It is not a surprise that File-Check incurs the largest overhead overall, such that the time to solution by more than three-fold in each case. Hence, checkpointing to a file may not be beneficial in general.

It may be also observed that the simulations preserve their scalability when the fault-tolerant techniques are applied, even for the File-Check, which may be attributed to the use of a parallel file system [23], [21]. Note that the tested checkpoint techniques themselves are scalable as shown in Tables 1 and 2, which is also desirable to preserve overall scalability, especially for MAP-Comm, where the increase in the number of nodes may cause additional communication overhead. The findings in Tables 1 and 2 also indicate that MAP-Comm is scalable. Hence, it may be inferred that, when the number of checkpoint copies is constant, the increase in node count does not affect significantly communication overhead. Conversely, when the number of checkpoint copies increases, the communication overhead grows.

[2]A clean execution is defined here as a fault-free run of the original simulation that does not have fault-tolerant mechanisms.

5. Conclusions

This work explored the feasibility and potential benefits of using mapped files as a medium to save checkpoints and as a way to recover from a failure without having to make explicit checkpoints. Experiments have shown not only that the proposed MAP-Var method incurs negligible overheads but also that the mapped-file checkpointing may incur low overheads, which are on average, just a fraction of those for the file-checkpointing counterpart. In particular, For CoMD, the mapped-file (MAP-Check) and mapped-file with replicated (MAP-Comm) checkpointing took less than 1% and 5.6%, respectively, of the time for the checkpoints made to a file (File-Check). For LULESH, these comparisons result in 1.8% and 9.8%, respectively.

To summarize, mapped files appear to have two major benefits: The speed of the main memory and, to a reasonable extent, the permanence of files, both of which are necessary ingredients in fault tolerance.

In the future, the proposed applicability of mapped files may be broadened to more diverse computing-platform failures by developing a means to periodically and seamlessly backup the mapped files to hard drives, which will help achieve the simulation resilience on par with the standard persistence storage. Future work will also include comparisons to state-of-the-art libraries such as SCR and using mapped files in checkpoint/restart libraries, at a system level, so that no modifications are required for the applications to take advantage of mapped files.

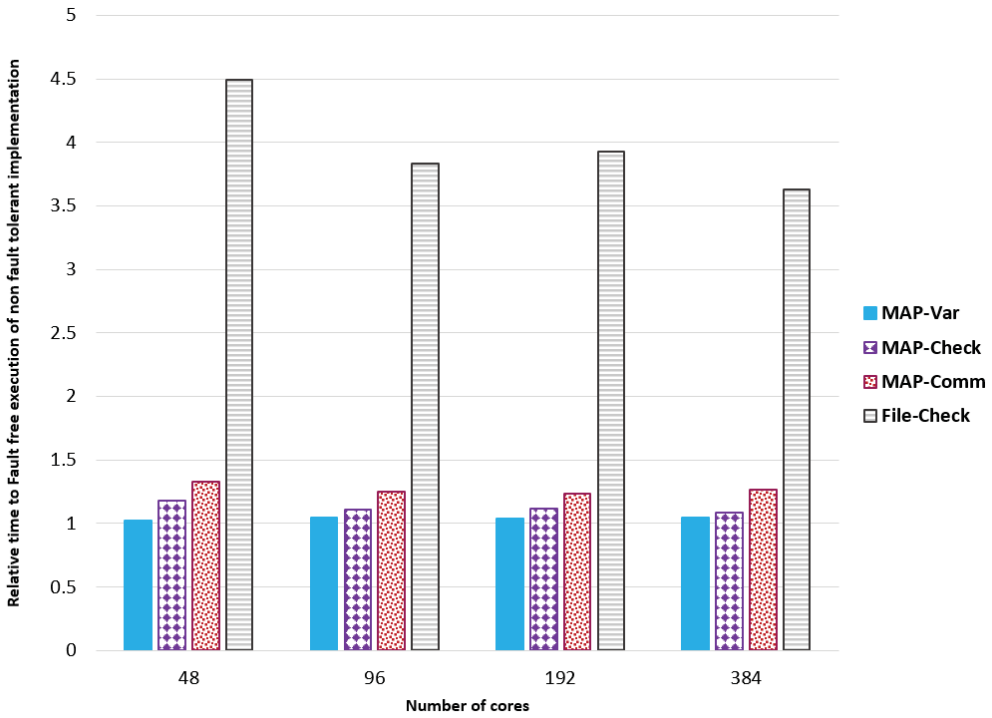

Fig. 3: Total simulation times normalized by the time of the original execution in which no faults occured (denoted as *clean execution*) in CoMD.

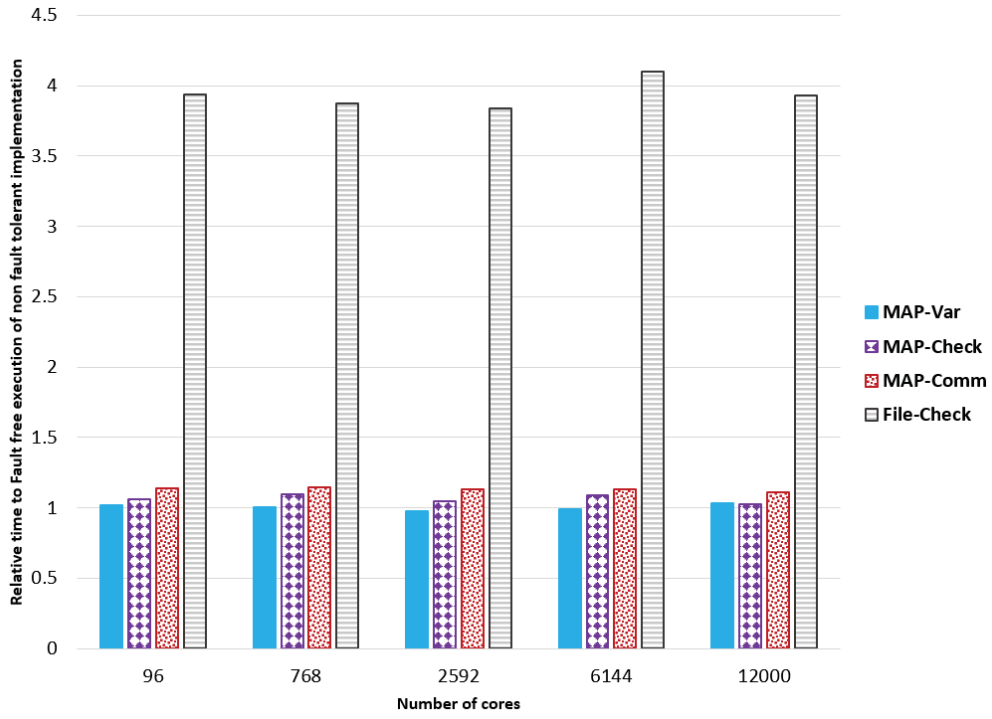

Fig. 4: Total simulation times normalized by the time of the original execution in which no faults occured (denoted as *clean execution*) in LULESH.

Acknowledgments

This work was supported in part by the Air Force Office of Scientific Research under the AFOSR award FA9550-12-1-0476, by the U.S. Department of Energy, Office of Advanced Scientific Computing Research, through the Ames Laboratory, operated by Iowa State University under contract No. DE-AC02-07CH11358, and by the U.S. Department of Defense High Performance Computing Modernization Program, through a HASI grant. This work was also supported by the Turing High Performance Computing cluster at Old Dominion University. This work used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. The authors thank the reviewers for their helpful comments.

References

[1] W. Feng and K. Cameron. The Green500 List: Encouraging Sustainable Supercomputing. *Computer*, 2007.

[2] J. Balladini, R. Suppi, D. Rexachs, and E. Luque. Impact of Parallel Programming Models and CPUs Clock Frequency on Energy Consumption of HPC Systems. In *Conference on Computer Systems and Applications (AICCSA), 2011 9th IEEE/ACS International*, 2011.

[3] F. Cappello, A. Geist, B. Gropp, L. Kale, B. Kramer, and M. Snir. Toward Exascale Resilience. *International Journal of High Performance Computing Applications*, 2009.

[4] A. Gainaru, F. Cappello, M. Snirand, and W. Kramer. Fault prediction under the microscope: A closer look into hpc systems. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[5] P. G. Bridges, K. B. Ferreira, M. A. Heroux, and M. Hoemmen. Fault-tolerant linear solvers via selective reliability. *CoRR*, 2012.

[6] O. Subasi, J. A. Moreno, O. S. Unsal, J. Labarta, and A. Cristal. Programmer-directed partial redundancy for resilient HPC. In *Proceedings of the 12th ACM International Conference on Computing Frontiers, CF'15, Ischia, Italy, May 18-21, 2015*, pages 47:1–47:2, 2015.

[7] T. Herault and Y. Robert. *Fault-Tolerance Techniques for High-Performance Computing*. Springer, 2015.

[8] B. Nicolae and F. Cappello. Blobcr: Efficient checkpoint-restart for hpc applications on iaas clouds using virtual disk image snapshots. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 34:1–34:12. ACM, 2011.

[9] O. Subasi, J. A., O. Unsal, J. Labarta, and A. Cristal. Nanocheckpoints: A task-based asynchronous dataflow framework for efficient and scalable checkpoint/restart. In *Proceedings of The 23rd International Conference on Parallel, Distributed, and Network-Based Processing*. IEEE, 2015.

[10] G. Zheng, X. Ni, and L. V. Kale. Ftc-charm++: an in-memory checkpoint-based fault tolerant runtime for charm++ and mpi. In *International Conference on Cluster Computing*. IEEE, 2004.

[11] G. Zheng, X. Ni, and L. V. Kale. A scalable double in-memory checkpoint and restart scheme towards exascale. In *International Conference on Dependable Systems and Networks Workshops*, pages 1–6. IEEE, 2012.

[12] 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.

[13] A. Moody, G. Bronevetsky, K. Mohror, and B. R. de Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *Supercomputing 2010*, 2010.

[14] S. Di, M. S. Bouguerra, L. Bautista-Gomez, and F. Cappello. Optimization of multi-level checkpoint model for large scale hpc applications. In *Symposium on Parallel and Distributed Processing*. IEEE, 2014.

[15] P. H. Hargrove and J. C. Duell. Berkeley lab checkpoint/restart (blcr) for linux clusters. In *Proceedings of SciDAC 2006*, 2006.

[16] J. Dongarra, T. Herault, and Y. Robert. Fault tolerance techniques for high-performance computing. In Thomas Herault and Yves (Eds.) n Robert, editors, *Fault-Tolerance Techniques for High-Performance Computing*, chapter 1, pages 3–85. Springer, 2015.

[17] M. Kerrisk. Linux programmer's manual shm_overview(7). `http://man7.org/linux/man-pages/man7/shm_overview.7.html`, 2015. Accessed: 2016-08-17.

[18] J. Mohd-Yusof. Co-Design Molecular Dynamics (CoMD) Overview, 3 2013.

[19] R. D. Hornung, J. A. Keasler, and M. B. Gokhale. Hydrodynamics Challenge Problem, Lawrence Livermore National Laboratory. Technical Report LLNL-TR-490254, 2011.

[20] NERSC. Edison. `http://www.nersc.gov/users/computational-systems/edison/`, 2016. Accessed: 2016-08-19.

[21] NERSC. Edison file storage and i/o. `http://www.nersc.gov/users/computational-systems/edison/file-storage-and-i-o/`, 2016. Accessed: 2016-12-06.

[22] I. Ngatang and M. Sosonkina. Strategies to hide communication for a classical molecular dynamics proxy application. In *HPC '15 Proceedings of the Symposium on High Performance Computing*, pages 210–216, 2015.

[23] NERSC. Optimizing i/o performance on the lustre file system. `http://www.nersc.gov/users/storage-and-file-systems/optimizing-io-performance-for-lustre/`, 2016. Accessed: 2016-12-11.