Performance Efficient Multiresilience using Checkpoint Recovery in Iterative Algorithms

Rizwan A. Ashraf and Christian Engelmann

Computer Science and Mathematics Division,
Oak Ridge National Laboratory,
Oak Ridge, TN, 37831, USA
{ashrafra, engelmannc}@ornl.gov

Abstract. In this paper, we address the design challenge of building multiresilient iterative high-performance computing (HPC) applications. Multiresilience in HPC applications is the ability to tolerate and maintain forward progress in the presence of both soft errors and process failures. We address the challenge by proposing performance models which are useful to design performance efficient and resilient iterative applications. The models consider the interaction between soft error and process failure resilience solutions. We experimented with a linear solver application with two distinct kinds of soft error detectors: one detector has high overhead and high accuracy, whereas the second has low overhead and low accuracy. We show how both can be leveraged for verifying the integrity of checkpointed state used to recover from both soft errors and process failures. Our results show the performance efficiency and resiliency benefit of employing the low overhead detector with high frequency within the checkpoint interval, so that timely soft error recovery can take place, resulting in less re-computed work.

Keywords: high-performance computing, resilience, soft errors, process failures, design patterns, performance, analytical models

1 Introduction

Reliable operation of extreme-scale computing systems is a significant challenge due to evolving system architectures, hardware components and software, and

This work was sponsored by the U.S. Department of Energy's Office of Advanced Scientific Computing Research. This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).

sheer scale of these systems. Since it is difficult and costly to build reliable high-performance computing (HPC) systems, its users or application developers need to devise solutions which can ensure predictable outcome in the presence of an array of faults, errors, and failures. Broadly, HPC applications are affected by soft errors and hard errors. Soft errors are transient in nature, caused mostly due to cosmic radiation particles interacting with components in the computing system. On the other hand, permanent failures in the system can affect components such as memory, processor, system software, and eventually the applications executing on the HPC system.

Complete resilience of HPC applications requires tackling both soft errors and process failures, or hereafter referred to as *multiresilience*. The manner in which both errors impact applications is unique. Some kind of soft errors can eventually cause a process failure, such as via corruption of a pointer variable, or loop index variable. Most soft errors will corrupt the data of the application. Hard errors which eventually cause a process failure are relatively easy to detect. Whereas, it may not always be possible to detect soft errors, i.e., they may silently corrupt the state of the application with no obvious indicators, which is usually referred to as *silent data corruption (SDC)*. SDCs can have significantly varying consequences on the outcome of the application, ranging from no effect to unusable results at the completion of an execution. Therefore, it is important to be able to design applications which can tolerate and make useful forward progress in the presence of both soft errors and process failures.

We leverage a design pattern oriented approach to the design of multiresilient HPC applications [1]. Design patterns provide concrete and repeatable solutions to commonly occurring problems. Based on this idea, previous work [9] identifies and formalizes design patterns for solution of resilience problems occurring in HPC systems. In this work, we focus on performance models for design patterns used for soft error and process failure resilience. These models serve as a guide to build optimal, efficient and reliable HPC applications. Specifically, we focus on iterative HPC applications, which can tolerate soft errors by taking additional time to converge to a solution [4, 10]. We are interested in the combination of soft error detection and checkpoint-based recovery which has minimal impact on application execution time and provides acceptable level of tolerance to soft errors. Previous work has identified combination of soft error detection with checkpoint by identifying the optimal number of verifications to perform within a checkpoint interval [2]. However, the prior work assumes ideal soft error detectors. In this work, we loosen the definition of an ideal soft error detector, and focus on practical detectors which may let through some soft errors undetected and corrupt checkpointed state, yet provide a satisfactory solution at the cost of additional iterations beyond the error free case.

Checkpointing is commonly used in HPC applications to recover from process failures [9]. It involves checkpointing the application state to a stable storage at regular intervals and its utilization in the event of a process failure. The application rollbacks to the last known good checkpoint and continues execution. A checkpointing based approach becomes complicated with the presence of soft

errors since the state which is being checkpointed may be corrupted, and the iterative application can become stuck and make no progress. Therefore, it is important to verify the integrity of the checkpoint before it is stored to a stable storage. This check provides a loose guarantee that the application will keep on making forward progress.

In this paper, we develop performance models for multiresilience to both soft and hard errors using checkpoint-based recovery, which is a well-utilized method in the field. In our experiments, we compare the performance of two distinct soft error detectors. One detector is high overhead and high accuracy, whereas the other detector is low overhead and low accuracy. We investigate experimentally and analytically, is it better to use a high overhead detector less often or a low overhead detector more often. A tradeoff exists since the low overhead detector can take more iterations to converge as compared to the case when high overhead detector is used. This tradeoff is investigated in our work. We perform our experiments with a Generalized Minimal Residual (GMRES) solver implemented using Trilinos and Open MPI User Level Failure Mitigation (ULFM) [3] in C++ programming language. The experiments are performed on an in-house Linux cluster with 960 processing cores as described in Section 5.

2 Soft Error Resilience

In this section, the two distinct soft error resilience design patterns utilized in our work are discussed. Without loss of generality and encompassing the scope of our work to iterative applications, we introduce the patterns based on a linear solver. The solver solves for the solution vector x in the system of equations of the form: $Ax = b$, where matrix A and right hand vector b are known. Soft errors can corrupt the state of the solver, which is composed of both static and dynamic states. The static state in this case forms the matrix A and the vector b, whereas the dynamic state is represented by the solution vector x. The remaining state of the solver which is required for achieving computational results is the environment state. The environmental state includes the variables associated with the runtime system of the message passing library (e.g., Open MPI), pointers, index variables, etc. Corruption of any of the above mentioned state categorizations can cause slowdowns, unbounded errors or fatal crashes.

The SDC detection patterns assist in catching these abnormalities exploiting the algorithmic characteristics of the solver. The two patterns namely: "Monotonicity Violation" and "Bounded Computations" are listed in Tables 1 and 2 respectively. In the first case, the detection relies on the property of the solver that it is always making forward progress with increasing iteration count, i.e., a characteristic of the iterative algorithms. This pattern can be utilized in all iterative applications which use a quality metric to determine convergence of the algorithm. To reduce the possibility of a false positive detection using this pattern, the difference from prior iteration can be bounded within certain limit. For detection in the GMRES solver, we utilize the residual which is a measurement of the error in the current solution. The residual at iteration or time step k is defined as: $r_k = b - Ax_k$. The residual has the property of being monotonically

Table 1. Resilience Pattern: Monotonicity Violation

Pattern Name	**Monotonicity Violation**
Problem	SDC Detection in iterative algorithms
Context	Check the progress of algorithm at each iteration by inspecting the quality metric
Forces	Applicable for iterative algorithms where quality metric is supposed to be monotonically non-increasing
Solution	Calculate quality metric at each iteration and check violation by comparing the quality metric from previous iteration
Capability	The need to calculate quality metric frequently increases computation and communication between parallel processes
Protection Domain	SDCs in static and dynamic state can be detected
Resulting Context	Enables timely recovery of iterative algorithm state
Rationale	Inexpensive method as compared to redundant computation

decreasing in the GMRES solver [8]. Over the course of iterative computations, if for any reason, this property is violated, we infer the presence of soft errors, and initiate recovery. The calculation of residual is a costly operation because it involves matrix vector multiplication, Ax_k. The matrix multiplication is a global operation across all the parallel processes and involves both parallel computation and communication. However, this high overhead detector is able to catch soft errors with high accuracy.

Table 2. Resilience Pattern: Bounded Computations

Pattern Name	**Bounded Computations**
Problem	SDC Detection in critical computations
Context	Check the progress and integrity of algorithm by inspecting the outputs produced during critical computations
Forces	Applicable for algorithms with identifiable critical computations and deterministic lower and upper bounds
Solution	Compare key outputs produced during critical computations against lower and/or upper bounds
Capability	Utilize implicit calculations and local invariant checking
Protection Domain	SDCs in static and dynamic state can be detected
Resulting Context	Enables timely recovery of the iterative algorithm state
Rationale	Inexpensive method as compared to redundant computation

The GMRES solver does not need to calculate the residual at every iteration to determine convergence since it can use the 2-norm of the result obtained from solving the least squares problem [8] as an indicator for convergence. The residual only needs to be calculated after convergence has been indicated and it is used to certify the convergence criteria has been met, i.e., the residual falls below a certain user-specified threshold value. Even though the value of 2-norm can be used as an indicator of errors, we rely on the more accurate residual as a quality metric for use within the monotonicity violation resilience pattern. As

far as our low overhead detector is concerned, we rely on inexpensive invariant checks, as highlighted by the bounded computations design pattern in table 2. It involves checking of an algorithmic condition which is performed locally. The GMRES solver utilizes an orthogonalization phase during every iteration of the solver, where the projection lengths are bounded by Frobenius norm of matrix A. This condition on the projections can be checked relatively inexpensively, since each parallel process iterates over its projection lengths locally. These are a good indicator of the corruption of state due to soft errors [8]. However, it is not a high accuracy detector.

Once soft errors are detected, checkpoints can be utilized for soft rollback. To be able to minimize the amount of re-computations, it is best to perform soft error detections regularly such that the rollback takes place quickly, i.e., we fail fast. Otherwise, if only a single detection or verification is performed prior to a checkpoint, then the whole interval which is usually composed of multiple iterations needs to be re-computed. This is because soft errors cause corruption unlike process failures which cause disruption in the parallel environment and are relatively easy to detect. The resilience to process failures and aspects for multiresilience are discussed in the next section.

3 Process Failure Resilience and Multiresilience

In distributed applications based on the message passing programming model, the failure of even one process in the parallel environment causes a fatal crash of the application in most implementations. Recent proposal to integrate ULFM in Message-Passing Interface (MPI) addresses some of the challenges associated with handling process failures [3]. For example, an implementation of ULFM based on Open MPI provides the ability to reliably detect process failures in a MPI application using a consensus algorithm. It also provides the ability to continue execution despite the presence of process failures, by reconstructing communication objects. However, it does not provide the ability to recover application state and it is left on to the user because each application has unique traits. Multiple methods exist to recover application state including forward and backward recovery of application state [9]. Methods which use application-oblivious checkpointing tend to have high storage and performance overheads as compared to an approach which only considers the minimal state required to resume computation. In this work, we utilize application-assisted checkpointing. For example, we only checkpoint the dynamic state of the solver at regular intervals, whereas the static state of the solver only needs to be checkpointed at the start and readjusted after each process failure to sustain future failures.

The checkpoint restart design pattern to recover from process failures is widely utilized [9]. Recovery from process failure can be accomplished via spares or using only the surviving processes. In this work, we utilize spare processes to recover from failures since it avoids the need to re-balance the workload among surviving processes. We also use in-memory checkpointing [12], whereby the highly optimized point-to-point connectivity between nodes in the HPC system

is utilized to store the checkpoints in the memories of assigned nodes in the system. We maintain two copies of the checkpoint, one is maintained locally, and the other one is maintained at a neighboring process. This arrangement helps to recover checkpointed state in case of failure of one process. The approach is extendable to handle multiple process failures by maintaining extra level of redundancy at more than two processes. This is beyond the scope of this work. In a multiresilient implementation, the local checkpoints can be used to recover from soft errors relatively inexpensively compared to the communication overhead required in case of process failure. Thus, performing multiple soft error detections within the interval is feasible since errors can be caught early and timely recovery can be performed.

In a multiresilient solution, it is also important to verify the integrity of the state being checkpointed, since use of corrupt state in recovery can hinder the ability of the iterative application to make forward progress. It may be possible to store multiple checkpoints, and jump back to older ones, in case no forward progress is determined, however it results in high overheads as well as the challenge of determining when the soft error might have started the corruption of checkpoints. Therefore, it is important to perform soft error detection to look for obvious abnormalities in the state being checkpointed. Herein, we assume that the checkpoint state stored in the memory will not be corrupted. However, with double in-memory checkpoints it is possible to drop this assumption.

4 Performance Model for Multiresilience

In this section, we develop analytical models to investigate performance characteristics of multiresilient iterative applications. We focus on the combination of soft error detection and mitigation patterns, and process failure mitigation patterns that reduce the time-to-solution. Specifically, we are interested in finding which kind of detector to use and how often to use it within a single checkpoint interval given their overheads. We assume two types of soft error detectors D_1, a high overhead and high accuracy detector, and D_2, a low overhead and low accuracy detector, with overheads quantified in software implementation as T_{D_1} and T_{D_2}, respectively. Here, T_{D_1} is mostly composed of the overhead of calculating the residual and T_{D_2} is composed of the overhead of iterating through multiple projection lengths and performing the comparisons. It is noteworthy to mention that if we assume both D_1 and D_2 to be ideal detectors, i.e., they can detect each and every soft error, then choosing the low-overhead detector D_2 is the obvious choice. As opposed to prior work [2], we define our detectors to be non-ideal and are therefore interested in the overall impact on time-to-solution. With a generic software detector having a overhead T_D, the time-to-solution for an iterative application in the error- and failure-free case (T_{FF}) is quantified as:

$$T_{FF} = T_{work} N_{FF} + \lfloor N_{FF} \gamma_{check} \rfloor (T_D N_D + T_{check}) \tag{1}$$

Here, T_{work} represents the time spent doing useful work inside a single iteration of the application, N_{FF} represents the number of iterations required to converge to a solution when no error or failure occurs, γ_{check} represents the factor

or frequency with which checkpoints are taken (assumes checkpoints are only taken at the completion of an iteration, e.g., a value of 1/20 means checkpoint is taken after every 20 iterations), T_{check} represents time spent performing the checkpoint, and N_D represents the number of soft error detections done within a single checkpoint interval. The rate with which to take the checkpoints is dependent on the cost of taking the checkpoint and failure rate of the HPC system [7]. A tradeoff exists between how often to checkpoint which causes overhead in case of failure-free execution and the amount of re-computation in case of failure which may be high if checkpoints are not taken frequently. We assume γ_{check} to be constant for our analysis since it has nothing to do with the choice of the soft error detector. Other parameters are dependent on the application and vary depending on the workload used.

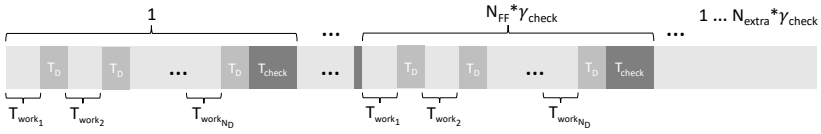

Fig. 1. The multiresilient checkpointing and fail-fast recovery approach.

Now, we model the time-to-solution in the presence of both detected soft errors and process failures, T_{fail}. This is composed of the following components: (1) error- and failure-free total time, (2) overhead incurred due to re-computation after recovery from detected soft-errors and process failures, (3) recovery overheads of detected errors and failures, and (4) extra work done beyond error free case due to presence of bounded errors or undetected soft errors in the state of the application. With these overheads, T_{fail} can be quantified as:

$$\begin{aligned} T_{fail} = T_{FF} + N_{SE}(T_{recomp-SE} + T_{SE-r}) + N_{PF}(T_{recomp-PF} + T_{PF-r}) \\ + T_{work}N_{extra} + \lfloor \gamma_{check} N_{extra} \rfloor (T_D N_D + T_{check}), \quad \text{where,} \end{aligned} \tag{2}$$

$$\begin{aligned} T_{recomp-SE} &= ((1 + 2 + 3 + ... + N_D)/N_D).((T_{work}\gamma_{check}^{-1})/N_D + T_D) \\ &= 0.5\ (N_D + 1).((T_{work}\gamma_{check}^{-1})/N_D + T_D), \quad \text{and,} \end{aligned} \tag{3}$$

$$T_{recomp-PF} = (T_{work}\gamma_{check}^{-1} + T_D N_D)/2 \tag{4}$$

Here, N_{SE} is the expected number of soft errors which are detected and is therefore dependent on the type and frequency with which the detector is utilized (note, this represents each time soft error detector positively flags corruption of state which may include multiple soft errors in practice), $T_{recomp-SE}$ and T_{SE-r} represent the re-computation and recovery overheads after recovery from a single successful soft error detection respectively, N_{PF} represents the expected number of process failures which are detected, $T_{recomp-PF}$ and T_{PF-r} represent re-computation and recovery overheads associated with detected process failures respectively, and N_{extra} represents the expected number of extra iterations taken by the iterative application beyond the error-free case. Fig. 1 shows the placement of soft error detectors within a single checkpoint interval. We assume

that the detectors are placed such that the interval is divided into equal sized chunks which may compose multiple iterations of useful work depending on N_D. Irrespective of the original location of the soft fault, the error only has a chance to be detected upon the execution of a detector. If the first detector after the checkpoint catches an error, then only one chunk of work and one detection need to be recomputed. Similarly, if the second detector catches an error, then two chunks of work and two detections starting from the last restart location need to be recomputed, and so on. Following this observation and assuming the fault is equally likely to strike in each chunk, the average value for re-computation due to detected soft errors can be estimated as in eq. 3. On the other hand, a process failure is detected almost immediately due to its disruptive nature, therefore, based on a uniform distribution, the average amount of work recomputed is estimated as in eq. 4.

Other parameters such as N_{SE} and N_{PF} depend on the system specifications such as error and failure rates, respectively. Similarly, T_{check} can be determined based on the latency of transferring checkpoints over the HPC network and the size of the checkpoint [5]. The values for other parameters are best estimated through statistical fault injection experiments. In the next experiment and results section, we find the value of N_D for each type of soft error detector which minimizes T_{fail}. We also estimate N_{extra} and N_{SE} in terms of type and frequency of detector used, although they are strongly application dependent.

5 Experiments and Results

In our experiments, we utilize the FT-GMRES solver which has been implemented using the Trilinos framework [8]. Trilinos provides the ability to solve large scale problems using an array of parallel programming models on a variety of computing platforms. Our implementation is done using ULFM 1.1 built on top of Open MPI 1.7.1, which provides the ability to detect failed processes and remove them from communication objects. In our previous work, we modified FT-GMRES to support multiresilience including the ability to utilize spare processes to recover from process failures [1]. This work provides an in-depth analysis of how to choose the right soft error detector in a multiresilient setup.

We perform our experiments on a Linux cluster with 40 nodes with 2 AMD Opteron processors each (48 cores per node) interconnected with 1 Gbps ethernet. We solve a linear problem with a sparse matrix A which has about 7 million rows and 186 million non-zero elements using 512 cores. The GMRES solver is able to converge to a solution in 320 iterations in the fault free (N_{FF}) case. We perform fault injection in our experiments to determine the multiresilience of the solver and various parameters of interest. In all cases, the number of process failures and the time window in which these are injected are the same, and are based on an exponential distribution. The checkpoints are also performed at the same rate, e.g., we set $\gamma_{check} = 1/20$ for all our experiments. With this setup, the variables associated with process failures have bounded values. Soft errors are injected randomly into computed data (e.g., the resultant vector produced after a sparse matrix vector multiplication operation) after almost every 10 iterations

of useful work. The error and failure rates are fixed across all our experiments. Enough fault injection experiments (at least 100 for each case) are performed in each case to keep the coefficient of variation low.

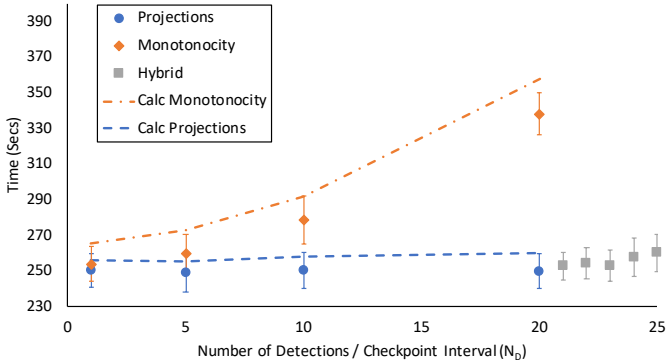

Fig. 2. The effect on total time-to-solution (average and std. dev.) with different number of soft error detections inside a single checkpoint interval. The performance estimations obtained from our analytical models in eqs. 1 and 2 are also plotted.

The overall time-to-solution for FT-GMRES with the two soft error detectors, i.e., monotonicity violation and bounded computations (projections), is shown in Fig. 2. The effect of performing increasing number of soft error detections inside a single checkpoint interval (note: a max of 20 detections can be performed) on time-to-solution shows the runaway effect when using the high-overhead detector, especially when $N_D > 5$. In our experiments, on average we measured about 90 fold higher overhead for monotonicity detector compared to bounded computations detector. Thus, using the high accuracy detector too often starts to dominate the time-to-solution nullifying any other positive effects. However, the disparity among overheads of accurate and inaccurate detectors is high for FT-GMRES solver. Consequently, the conclusions may differ for other applications depending on the tradeoff between penalty of extra work with low accuracy detector and overhead of using the high accuracy detector. The estimations obtained from the performance model proposed in eqs. 1 and 2 are also plotted in Fig. 2, demonstrating a decent bound on time-to-solution with both detectors in FT-GMRES solver. Some observations on the parameters in our models are: T_{check} is dominated by the time to store the checkpoint in remote memory, $T_{SE-r} << T_{PF-r}$ recovering checkpoint from local memory is orders of magnitude faster than process failure recovery, and N_{extra} and N_{SE} depend on the type of the detector and the frequency (N_D) with which it is used.

Based on the performance of the two detectors, we also evaluate an additional type of detector, which is a hybrid of the monotonicity violation and bounded computations detectors. In this case, we perform the low overhead detection at every iteration of the solver, whereas the high overhead detector is performed up to 5 times inside a single checkpoint interval. The time-to-solution with the hybrid soft error detector is shown in Fig. 2. The hybrid detector gives mid-tier overall performance, with interesting implications on total iteration count and soft error detection success rate as discussed hereafter.

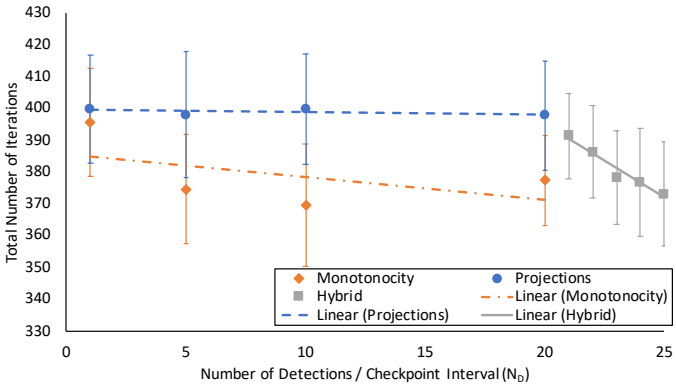

Fig. 3. The effect on total iterations (average and std. dev.) to converge to a solution with different number of soft error detections inside a single checkpoint interval. Includes recomputations with errors and extra iterations beyond error-free case.

Results in Fig. 3 show the total number of iterations taken by the solver to converge to a solution with each type of detector while using different number of detections inside a single checkpoint interval. The total iteration count here includes all the recomputations after each soft error and process failure recovery, and the extra iterations taken by the solver to converge to a solution beyond the fault free case. Overall, increasing the use of low overhead detector does not effect the iteration count drastically when compared to the high overhead detector. For example, the total iterations decrease at a rate of 0.08 and 0.71 per detection when using bounded computations and monotonicity patterns, respectively. The hybrid approach seems to provide the fastest decrease in total iteration count among all cases. These results correspond directly to the number of additional iterations taken by the solver beyond the fault free case, N_{extra}. Our estimations for N_{extra} range between 18 and 46 for the high accuracy detector, and between 51 and 60 for the low accuracy detector. These results show that there is more overhead due to additional iterations for the low accuracy detector including extra checkpoints as compared to high accuracy detector.

The detection accuracy of each soft error detector is shown in Fig. 4. Here, the undetected soft errors may cause any of the following cases: no effect on the outcome or convergence of the solver, extra work to converge to a solution, and increased chances of inducing a process failure. Another possibility is that the solver does not converge to a solution in allocated time, which is not listed earlier since sufficient time is given to the solver in our experiments. The results in Fig. 4 also provide a good estimate for expected number of soft errors detected N_{SE} in each case. We estimate N_{SE} by averaging the number of soft errors detected across all runs. On average, the high accuracy detector catches between 1 and 2 soft errors, whereas the low accuracy detector catches between 0 and 1 soft errors in each run depending on the number of detections performed in each interval. In most cases, the bounded computations soft error detection pattern achieves lower accuracy than the monotonicity violation pattern. The low accuracy detector

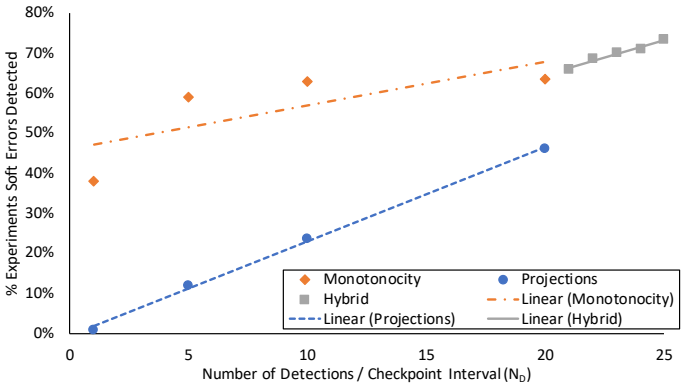

Fig. 4. Variation of soft error detection rate (total number of experiments in which soft errors were successfully detected out of all experiments) when different number of soft error detections are performed inside a single checkpoint interval.

seems to surpass the lowest accuracy achieved by high accuracy detector with $N_D > 15$. The use of high accuracy detector at low frequency combined with low accuracy detector at highest frequency is seen to provide matching or better accuracy than the high accuracy detector in most cases. Overall, the hybrid detector is able to detect soft errors more often reducing the overheads due to extra iterations and therefore is able to provide significantly better resilience than the low overhead detector with up to 4% more performance overhead.

6 Related Work

Previous works in the design of resilient iterative methods have focused on tolerance to either soft errors [4,10] or process failures [6] but not both together. Most recently, design patterns have been utilized for implementation of multiresilient solutions in HPC applications [1]. The catalog of design patterns [9] comprehensively describes various resilience solutions, a layered hierarchy of the patterns and a patterns language. The use of algorithmic approaches to detect soft errors for sparse linear algebra and a linear solver are demonstrated in [11] and [8], respectively. In this work, we develop a performance model which aids in the selection and tuning of soft error detectors in conjunction with a checkpoint-based recovery approach, which is widely applicable [5,12]. Related work [2] develops a performance model which minimizes total execution time by only focusing on a single system-level checkpoint interval and does not consider the special case for iterative algorithms. Furthermore, we also test the efficacy of our performance model through experiments with a linear solver application.

7 Conclusion

We demonstrate the design of performance efficient multiresilient linear solver application. Checkpoint restart is shown to be an effective recovery approach in our multiresilient solution. Our approach shows the appropriate combination

of soft error and process failure resilience solutions. We evaluate two different type of soft error detectors in our work and investigate the tradeoffs of using them under non-ideal detection conditions. Results evaluate the affect of using the detectors with different frequency on time-to-solution, the number of extra iterations taken by the solver beyond the fault free case, and the rate of successful soft error detections in simulated fault injection experiments. A hybrid approach which uses the high overhead and high accuracy detector sparingly combined with a low overhead detector and low accuracy detector at every iteration is observed to have similar or better detection success rate as using a high overhead detector at every iteration with significantly less impact on time-to-solution.

Acknowledgements This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, program manager Lucy Nowell, under contract number DE-AC05-00OR22725.

References

1. Ashraf, R.A., Hukerikar, S., Engelmann, C.: Pattern-based modeling of multiresilience solutions for high-performance computing. In: Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering (2018)
2. Benoit, A., Cavelan, A., Robert, Y., Sun, H.: Optimal resilience patterns to cope with fail-stop and silent errors. Report RR-8786, LIP - ENS Lyon (Oct 2015)
3. Bland, W., Bouteiller, A., Herault, T., Bosilca, G., Dongarra, J.: Post-failure recovery of MPI communication capability. The International Journal of High Performance Computing Applications 27(3), 244–254 (2013)
4. Bronevetsky, G., de Supinski, B.: Soft error vulnerability of iterative linear algebra methods. In: 22nd Annual International Conference on Supercomputing (2008)
5. Cao, J., Arya, K., Garg, R., Matott, S., Panda, D.K., Subramoni, H., Vienne, J., Cooperman, G.: System-level scalable checkpoint-restart for petascale computing. In: IEEE 22nd International Conference on Parallel and Distributed Systems (2016)
6. Chen, Z.: Algorithm-based recovery for iterative methods without checkpointing. In: 20th International Symp. on High Performance Distributed Computing (2011)
7. Daly, J.: A higher order estimate of the optimum checkpoint interval for restart dumps. Future Generation Computer Systems 22(3), 303 – 312 (2006)
8. Elliott, J., Hoemmen, M., Mueller, F.: Evaluating the impact of SDC on the GMRES iterative solver. In: 2014 IEEE 28th International Parallel and Distributed Processing Symposium. pp. 1193–1202 (May 2014)
9. Hukerikar, S., Engelmann, C.: Resilience Design Patterns: A Structured Approach to Resilience at Extreme Scale (version 1.2). Tech. Rep. ORNL/TM-2017/745, Oak Ridge National Laboratory, Oak Ridge, TN, USA (August 2017)
10. Jaulmes, L., Casas, M., Moret, M., Ayguad, E., Labarta, J., Valero, M.: Exploiting asynchrony from exact forward recovery for DUE in iterative solvers. In: SC15: Int'l Conf. for High Performance Computing, Networking, Storage and Analysis (2015)
11. Sloan, J., Kumar, R., Bronevetsky, G.: An algorithmic approach to error localization and partial recomputation for low-overhead fault tolerance. In: 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (2013)
12. Zheng, G., Shi, L., Kale, L.V.: FTC-Charm++: an in-memory checkpoint-based fault tolerant runtime for Charm++ and MPI. In: 2004 IEEE International Conference on Cluster Computing. pp. 93–103 (Sept 2004)