5-2022

Scalable and Reliable Sparse Data Computation on Emergent High Performance Computing Systems

Zheng Miao
zmiao@clemson.edu

Scalable and Reliable Sparse Data Computation on Emergent High Performance Computing Systems

---

A Dissertation
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
Computer Science

---

by
Zheng Miao
May 2022

Accepted by:
Dr. Rong Ge, Committee Chair
Dr. Jon C. Calhoun, Committee Co-chair
Dr. Amy Apon
Dr. Shuangshuang Jin
Dr. Jiajia Li

Abstract

Heterogeneous systems with both CPUs and GPUs have become important system architectures in emergent High Performance Computing (HPC) systems. Heterogeneous systems must address both performance-scalability and power-scalability in the presence of failures. Aggressive power reduction pushes hardware to its operating limit and increases the failure rate. Resilience allows programs to progress when subjected to faults and is an integral component of large-scale systems, but incurs significant time and energy overhead. The future exascale systems are expected to have higher power consumption with higher fault rates. Sparse data computation is the fundamental kernel in many scientific applications. It is suitable for the studies of scalability and resilience on heterogeneous systems due to its computational characteristics.

To deliver the promised performance within the given power budget, heterogeneous computing mandates a deep understanding of the interplay between scalability and resilience. Managing scalability and resilience is challenging in heterogeneous systems, due to the heterogeneous compute capability, power consumption, and varying failure rates between CPUs and GPUs. Scalability and resilience have been traditionally studied in isolation, and optimizing one typically detrimentally impacts the other. While prior works have been proved successful in optimizing scalability and resilience on CPU-based homogeneous systems, simply extending current approaches to heterogeneous systems results in suboptimal performance-scalability and/or power-scalability.

To address the above multiple research challenges, we propose novel resilience and energy-efficiency technologies to optimize scalability and resilience for sparse data computation on heterogeneous systems with CPUs and GPUs. First, we present generalized analytical and experimental methods to analyze and quantify the time and energy costs of various recovery schemes, and develop and prototype performance optimization and power management strategies to improve scalability for sparse linear solvers. Our results quantitatively reveal that each resilience scheme has its own

advantages depending on the fault rate, system size, and power budget, and the forward recovery can further benefit from our performance and power optimizations for large-scale computing. Second, we design a novel resilience technique that relaxes the requirement of synchronization and identicalness for processes, and allows them to run in heterogeneous resources with power reduction. Our results show a significant reduction in energy for unmodified programs in various fault situations compared to exact replication techniques. Third, we propose a novel distributed sparse tensor decomposition that utilizes an asynchronous RDMA-based approach with OpenSHMEM to improve scalability on large-scale systems and prove that our method works well in heterogeneous systems. Our results show our irregularity-aware workload partition and balanced-asynchronous algorithms are scalable and outperform the state-of-the-art distributed implementations. We demonstrate that understanding different bottlenecks for various types of tensors plays critical roles in improving scalability.

Acknowledgments

First and foremost, I am incredibly grateful to my PhD advisors, Dr. Rong Ge and Dr. Jon C. Calhoun, for their guidance, patience, expertise, and support. I would like to thank Dr. Rong Ge and Dr. Jon C. Calhoun for leading me into the research area of High Performance Computing. They gave me the freedom to choose research topics that interest me and taught me how to do research of high quality and make good presentations in many details. Without their help, I would not have been here.

I am very grateful to Dr. Jiajia Li for advising me during my internship at Pacific Northwest National Laboratory and for the following research work. She provided me with valuable suggestions to correct my shortcomings and taught me how to make efficient work schedules. I would like to thank Dr. Jiajia Li for connecting me with good job opportunities.

I would like to thank Dr. Amy Apon and Dr. Shuangshuang Jin for serving on my PhD dissertation committee. They provided me with insightful comments and valuable suggestions to help me finish a well-structured and high-quality thesis.

I am very lucky to work with all members of the Scalable Computing and Analytics Lab at Clemson University, Dr. Xizhou Feng, Dr. Pengfei Zou, Tyler Allen, Tim Ransom, Thomas Randall, Bennett Cooper, and Naman Kulshreshtha. I would like to thank everyone for their good advice, collaboration, and assistance.

Last but not least, I would like to thank my family and my girlfriend for all their love and encouragement. For my parents who raised me with a love of science and supported me in all my pursuits. For my girlfriend Yu Li, who has been supporting me all these years and has motivated me to work hard for our bright future together. Thank you!

I acknowledge that part of this thesis was published previously and submitted in the following conferences and journals:

- **Zheng Miao**, Jon C. Calhoun, Rong Ge. "Relaxed Replication for Energy Efficient and Resilient GPU Computing", 2021 IEEE/ACM 11th Workshop on FTXS. IEEE, 2021.
- **Zheng Miao**, Jon C. Calhoun, Rong Ge, Jiajia Li. "Sparsity-Aware Distributed Tensor Decomposition", The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC), ACM Student Research Poster, 2020.
- **Zheng Miao**, Jon C. Calhoun, Rong Ge. "A Framework for Resilient and Energy-efficient Computing for GPU-accelerated Systems", The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC), ACM Student Research Poster, 2019.
- **Zheng Miao**, Jon C. Calhoun, Rong Ge. "Energy Analysis and Optimization for Resilient Scalable Linear Systems", 2018 IEEE International Conference on Cluster Computing (CLUSTER). IEEE, 2018.
- **Under Review: Zheng Miao**, Jon C. Calhoun, Rong Ge. "Towards Resilient and Energy Efficient Scalable Linear Solvers", IEEE Transactions on Parallel and Distributed Systems, 2021.
- **Under Review: Zheng Miao**, Jon C. Calhoun, Rong Ge, Jiajia Li. "Performance Implication of Tensor Irregularity and Optimization for Distributed Tensor Decomposition", ACM Transactions on Parallel Computing, 2021.
- **Under Review: Zheng Miao**, Jiajia Li, Jon C. Calhoun, Rong Ge. "BA-CPD: Balanced Asynchronous Tensor Decomposition on Distributed Memory Systems", European Conference on Parallel Processing, 2022.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 The Interplay between Scalability and Resilience Challenges in HPC

Scalability, including performance-scalability and power-scalability, is always the major objective of high performance computing (HPC). As the top supercomputers' performance keeps increasing fast, their power consumption is also increasing rapidly due to the expansion of the system size. The U.S. Department of Energy (DOE) has set 20 MW as the power limit for future exascale systems [14]. However, since 2010 to 2021, the performance of the top supercomputers has grown from 2 Peta FLoating-point OPerations per Second (PFLOPS) to 400 PFLOPS, while their power consumption has grown from 7 megawatts (MW) to 30 MW based on the TOP500 list [3]. Future exascale systems are expected to have a higher power budget under the projection of the performance per Watt today. Reducing energy consumption while maintaining or increasing the performance of HPC is an urgent problem for both economic and environmental reasons.

Resilience is another major challenge in emergent HPC systems. Today's top Petascale computers have a mean time between failure (MTBF) in 1-7 days [48]. Future exascale systems are expected to have an MTBF within an hour, due to massive concurrency and unreliability induced mainly by huge numbers of hardware components and miniaturizing feature size. Resilience techniques allow programs to continue progressing in the presence of failures in hardware or software through redundancy and recomputing, The design of appropriate resilience solutions is indispens-

able in exascale systems because the current solutions used on petascale computers do not scale up to exascale systems. Unfortunately, power and resilience are intertwined challenges that can not be addressed separately. Running hardware components at lower voltages will lower power but also increase the effects of noise sources, and thus increasing the failure rate in HPC. Improving resilience incurs significant power and time overhead and exacerbates the power challenge. Thus, emergent HPC systems mandate simultaneously addressing power-scalability and resilience.

GPUs provide a majority of the computational capacity for HPC systems. They appear in five of the top ten supercomputers [3], and will appear in the first exascale computers [2, 1]. scalability issues and fault situations are more complex in GPU-accelerated systems [124]. Balancing power and resilience is more complex on GPU-accelerated systems. In such systems, complexities stem from heterogeneity in multiple aspects. First, compared to host CPUs, GPUs have different computation capabilities, power consumption, and energy efficiency. Second, GPUs are mainly used to accelerate the execution of kernels offloaded by CPUs and stay idle otherwise, while CPUs handle communication, services, and kernel offloading. Third, GPUs have different failure patterns and higher failure rates, and are more sensitive to temperature [124]. Such heterogeneity requires sophisticated management that distinguishes GPUs and CPUs and leverages their differences, in order to optimally achieve performance goals and meet power constraints.

Previous studies have mainly focused on improving either resilience or power- scalability for HPC, usually at the expense of one another. Resilience technologies aim to reduce time-to-solution (TTS) for programs in case of failures without considering energy requirements. For example, checkpoint-restart (CR) investigates and balances checkpointing time and rollback distance [46], and triple modular redundancy consumes 3× the power to provide error detection and correction. Algorithm based fault tolerance [64, 36] exploits partial redundancy, and forward recovery [67, 7] explores approximations of lost or corrupted data to recover from faults. Meanwhile, power management technologies — e.g., near-threshold voltage — generally increases the cost of resilience by making computing software and device more complex and unreliable [9].

Sparse data computation is the fundamental kernel in many scientific applications. It is suitable for the studies of scalability and resilience due to its computational characteristic. Previous works have studied resilience or scalability for sparse data computation in isolation. It is difficult to co-optimize resilience and scalability because optimizing one typically detrimentally impacts the other. To address the challenges of scalability and resilience, We need to understand the interplay

between performance-scalability, power-scalability and resilience for sparse data computation. We propose novel resilience and energy-efficiency technologies to optimize scalability and resilience for sparse data computation on heterogeneous systems with CPUs and GPUs. First, we present generalized analytical and experimental methods to analyze and quantify the time and energy costs of various recovery schemes on homogeneous systems with CPUs, and develop and prototype performance optimization and power management strategies to improve scalability for sparse linear solvers. Second, we design a novel resilience technique that relaxes the requirement of synchronization and identicalness for processes, and allows them to run in heterogeneous resources with power reduction. Third, we propose a novel distributed sparse tensor decomposition that utilizes an asynchronous RDMA-based approach with OpenSHMEM to improve scalability on heterogeneous systems.

1.2 Model-based Approach for Resilient and Energy-efficient Sparse Linear Slovers

Sparse data computation is a key kernel in many scientific applications spanning a wide range of domain areas, such as machine learning, computer vision, and fluid-dynamics. The linear system $Ax = b$ is solved by either direct methods [45] or iterative methods [103]. Direct methods incur increasingly long execution time and large computational resources as the problem size increases. While iterative algorithms take advantage of these sparse systems and solve the equation iteratively, approximate the solution. Krylov subspaces are used in iterative algorithms for finding approximate solutions to linear algebra problems [96]. In this dissertation, we consider one of the main Krylov subspace methods, namely conjugate gradient (CG), and evaluate various resilience schemes on it.

Prior work investigates the energy cost of resilience, but is limited to checkpointing and message logging [86] and the energy impact of checkpointing frequency [15] and Dynamic Voltage and Frequency Scaling (DVFS) [88]. More comprehensive studies are needed to answer multiple prominent *research questions*: (1) what is the resilience ability of various recovery mechanisms? (2) what is the power requirement of resilience and how does power management help? (3) what are the time and energy costs of resilience? and (4) how does the resilience cost scale with system size and MTBF?

Answering these questions requires deep understanding of performance, energy efficiency, resilience, and their interplay in faulty environments. Different resilience techniques incur different

amounts of time and energy. Moreover, a resilience technique responds differently to algorithms, workload characteristics, failure rate, hardware, and power allocation. For example, forward recovery for iterative Krylov solvers approximates lost or corrupted data in multiple ways [7]. Typically, better approximations take longer time and more energy to be constructed but allow faster progress to the solution than poor approximations. Quantifying the time and energy costs accurately is necessary to evaluate the time-energy trade-offs and identify the optimal resilience technique for a given situation.

Analytical models built from fine-grain measurement data are a promising approach to project time, energy, and resilience on large-scale systems for multiple reasons. First, generalized analytical models capture the first-order cost factors for various resilience techniques. Second, analytical models can be customized to reflect unique features of specific techniques. Third, fine-grained measurement of performance, power and resilience at a thread-level and of computer components can accurately capture model parameters. Fourth, fine-grain models can be used to predict the effect of power management at the system and component levels.

In this dissertation, we present a set of analytical models that describe the performance-scalability, power-scalability and resilience of scientific applications under faults on homogeneous systems with CPUs. We examine the impact of various fault recovery schemes for iterative linear solvers and further propose techniques to minimize time and energy overhead. We propose the matrix-aware optimization for forward recovery to achieve a better trade-off between scalability and resilience. We find that the input matrix impacts the reconstruction cost, the main computation cost, and their trade-off. In general, a more accurate reconstruction takes more time to build but less main computation time to continue with this reconstruction. Depending on the input matrix, the extra time taken to improve reconstruction accuracy is more or less than the time saved in the main computation. The condition number of a matrix can be used to determine the cost of the linear solver and therefore the trade-off between the reconstruction cost and the main computation cost. Nevertheless, obtaining the condition number of a high-dimension matrix is computationally costly, i.e., with the complexity of $O(n^3)$ where n is the dimension size. To address this challenge, we propose a new practical metric — *convergence factor*. Based on this metric, we evaluate the given matrix and approximate its specific optimal reconstruction accuracy that minimizes the total time cost. To the best of our knowledge, this is the first work to explore matrix-awareness to control the resilience overhead and guide low-cost optimization. Based on model parameters we derive from experiments on a cluster, we parameterize a model and use weak scaling to project program behavior

for large-scale systems.

1.3 Relaxed Replication for Energy Efficient and Resilient Sparse Linear Solvers on Heterogeneous Systems

There are several commonly accepted fault detection and recovery techniques for soft and hard faults on homogeneous systems with CPUs. Checkpoint-restart (CR) [109] periodically saves data to levels of storage devices and rollbacks to previous states when failures occur. Studies demonstrate that CR-based solutions are prohibitively costly on extreme-scale systems with frequent failures [30]. Redundancy-based approaches replicate computations and use replicas to replace faulted processes. Prior work has shown that redundancy-based approaches are more energy-efficient than CR at extreme-scales [110, 53], but current solutions such as RedMPI [56] and rMPI [54] only support parallel programs running on CPU-based homogeneous systems. Algorithm-based fault-tolerance (ABFT) is limited in matrix operations with checksums verification [33]. Existing redundancy based solutions typically rely on identical replicas to support fault detection and recovery. There are two main challenges for previous redundancy solutions: First, exact replication requires the same hardware, resources, execution environment, and thus power for the replicas. Second, fine-grained synchronization between main and replica processes is expensive because the replicas need to be concurrent at every message for MPI workloads. Simply extending such exact replication to GPU computing would waste GPU resources and computational capacity, and suffer severe energy inefficiencies, as GPUs consume more power than CPUs. Other recent work of shadow replication [89] shows better energy-efficiency among fault tolerance mechanisms in homogeneous systems with replicas running at lower processor speeds. However, their solution is not developed and optimized for heterogeneous systems with GPUs due to more expensive synchronization of main and replica processes in different devices.

To address the above limitations, we explore the feasibility of energy-efficient techniques to optimize replication for GPU-accelerated systems running workloads parallelized with MPI. Particularly, we explore a novel replication scheme that relaxes the synchronization and identicalness requirements. Our scheme supports periodical synchronization called rejuvenation or asynchronization between replica and main processes, thus allowing replica processes to run in lower precision, on lower power devices, and at lower performance states than the main processes. Our scheme pro-

vides different solutions to address both hard and soft faults. For soft faults, we recover faulty main processes by receiving dynamic data from the corresponding replicas. For hard faults, we replace faulty main processes with the corresponding replicas. This relaxed replication mechanism enables a reduction of resources and power requirements using today's GPU architecture and hardware. For example, GPUs support multiple levels of precision — e.g., double, single, half — for floating-point operations, where double precision delivers the highest accuracy and half precision consumes the least power. We run replicas in low precision for energy saving. In addition, GPUs transition among many performance states through the dynamic voltage and frequency scaling (DVFS) technology. Running GPUs at a low performance state significantly reduces power consumption [89]. Furthermore, there are plenty of hardware resources in GPU-accelerated systems to choose for running replica processes, e.g., running a kernel on one or more CPU cores consumes less power than on a powerful GPU card.

By relaxing synchronization and identicalness between the main and replica processes, our replication scheme complicates fault detection and recovery. First, as replicas use fewer resources and less power, they typically make less computation progress than the corresponding main processes. As a result, using them to recover faulty main processes loses progress, which takes time and consumes power and energy to recompute. Second, with replicas different from the main processes, we can not directly compare replicas against the main processes for silent data corruption (SDC) detection. To address these issues, we first introduce a periodic update that rejuvenates replicas with the progress of main processes. This periodic update controls the progress gap and thus progress loss during recovery. To support SDC detection, we leverage application specific detection and introduce an application specific discrepancy threshold, and flag SDC only when the gap is larger than the threshold.

We present, in detail, this relaxed replication redundancy mechanism and the energy-efficient techniques and discuss its employment on real systems. We implement a prototype and evaluate the impact on energy efficiency and resilience under both process/node failures and silent data corruption. Our evaluation environment includes a 16-GPU cluster and three representative HPC applications: CG [21], MiniQMC from QMCPACK [73], and DeepBench [92]. Results show that our scheme reduces energy by up to 19% for unmodified programs and 32% for programs that are able to adapt the precision of the replicas over a direct extension of the redundancy based resilience framework RedMPI for CPU systems.

1.4 Scalable Algorithms for Large-Scale Sparse Tensor Decomposition

Tensors are multidimensional arrays and often sparse that are utilized by applications spanning a wide range of domain areas, such as quantum chemistry, (healthcare, social network, brain signal, electrical grid) data analytics, signal processing, machine learning, and recommendation systems [70, 97, 95, 5, 12, 98, 74, 40, 63]. Tensor decompositions are a class of tensor methods for data analytics, low-rank approximation, data compression, and so on. In this work, we study the CANDECOMP/PARAFAC decomposition (CPD), one of the most popular tensor decompositions.

Large data generated from these applications requires distributed memory implementations due to the large amount of memory requirements and the need for fast execution time. For example, the *amazon* tensor comprises reviews and contains more than 1 billion nonzeros, the state-of-the-art CPD implementation could not analyze it on fewer than 8 CPU nodes. Some studies show impressive performance for sparse distributed CPD algorithms [114, 72, 39]. The previous works present medium-grained decomposition that performs a N-dimensional decomposition of the tensor, where N is the number of modes, and one-dimensional decompositions of the factor matrices [114, 17, 101]. They have achieved good performance and scalability in CPD for tensors with relatively regular dimension sizes and nonzero distribution because both computation and communication are balanced well. However, the sparsity and irregularity features and their influence on stages of the CPD algorithm have not been well investigated, which hinders further performance improvement and machine scalability. Other recent works use a fine-grained decomposition of tensors to co-optimize computation and communication [72, 71]. But they require significant time overhead in hypergraph partitioning.

We categorize the *irregularity* of a sparse tensor based on two aspects: very different dimension sizes and a non-uniform nonzero distribution. Analyzing sparse tensors from various data sources, we observe a tensor could have *dimension(s) much longer relative to the others*. This phenomenon is typical for tensors from real data because of different information contained in diverse dimensions: short dimensions could come from a small range of time-stamps, types of relations, etc., while long dimensions could be users, pages, keywords, papers, etc. Sparse tensors from real applications tend to have a *non-uniform nonzero distribution*; while different dimension sizes make it worse. The nonzeros could be extremely dense in a couple of regions, but much sparser in other

regions in an irregular tensor.

There are three types of load imbalance that play critical roles in what bottlenecks performance on sparse tensors: tensor nonzero, communication volume, and matrix computation imbalance. To measure these imbalances, we introduce three ratios as metrics. The state-of-the-art works such as medium- [114, 17, 101] and fine-grained [72, 71] distributed CPDs have made efforts to optimize these three types of imbalance. Medium-grained distributed CPD chooses to optimize them separately. However, when it focuses on balancing tensor nonzero, the other imbalances increase significantly for irregular tensors. Fine-grained distributed CPD [72, 71] utilizes hypergraph partitioners to co-optimize these imbalances, but it requires significantly more time overhead in partitioning than actual CPD computation for large tensors. To address these limitations, we present irregularity-aware CPD that co-optimizes different types of imbalance with a low overhead during preprocessing. Our solution provides two insights: First, by evaluating SPLATT theoretically and experimentally, we reveal that these two irregularities lead to unacceptable load imbalance when distributing a sparse tensor among multiple computing nodes. Furthermore, we outline four findings that influence the performance of existing methods. These findings demonstrate that two stages in the preprocessing *grid configuration* and *distribution policy* are critical for the overall CPD performance-scalability. Second, we leverage the sparsity and irregularity information that reflects in the large imbalance of matrix computation. The matrix computation is usually the bottleneck of performance and scalability. Therefore, we identify the dominant imbalance ratio as matrix computation imbalance for irregular tensors and optimize it with higher priority. However, focusing only on the balancing of matrix computation makes other imbalances worse. It is important to achieve the best trade-offs between different imbalances to improve performance-scalability.

1.5 Summary of Contributions

To summarize, this dissertation focuses on improving scalability and resilience for sparse data computation on emergent HPC systems. Our contributions are as follows:

- We present a generalized analytical model to quantitatively co-study performance-scalability, power-scalability and resilience for common resilience technologies to enable resilient and energy-efficient large-scale scientific computing on homogeneous systems with CPUs. The proposed analytical models capture the first order time and energy cost factors for various fault

recovery schemes and can be used to identify the best recovery schemes for given fault situations. This work investigates matrix-aware optimization for forward recovery and proposes the convergence factor metric, which makes it practical to determine the optimal lost data reconstruction for high-order matrices. Our results quantitatively reveal that each resilience scheme has its own advantages depending on the fault rate, system size, and power budget, and the forward recovery can further benefit from matrix-aware optimizations for large-scale computing.

- We propose a novel redundancy mechanism that relaxes the requirement of synchronization and identicalness for replicas for MPI programs running on GPU-accelerated systems. Our mechanism supports replicas to run in lower-precision and at lower power/performance states with periodical rejuvenation or asynchronization. Our redundancy mechanism supports both asynchronous and synchronous execution between the main and replica processes, where the former is capable of fast progress, and the latter is capable of detecting errors. We prototype the relaxed replication mechanism and evaluate it on a cluster. Our results can benefit real applications and GPU-accelerated systems.

- We investigate the common algorithm structure of state-of-the-art distributed tensor decomposition from theoretical and experimental analysis and observe important findings to guide optimization for performance-scalability. We demonstrate that the imbalance of computation and communication, and their trade-offs, are critical to the overall CPD performance-scalability. We identify the dominant imbalance ratio as matrix computation imbalance for irregular tensors. We propose irregularity-aware CPD that co-optimizes these imbalances with high priority in matrix computation imbalance in grid configuration and distribution policy with a low time overhead. To achieve better scalability of CPD on large-scale systems, we optimize the communication overhead by designing and implementing an asynchronous RDMA-based approach with OpenSHMEM, and prove that our method works well in heterogeneous systems.

Chapter 2

Background and Related Work

In this chapter, we present the background about resilience and energy-efficiency of HPC, sparse linear solver and sparse tensor decomposition. We introduce existing works related to the dissertation and discuss their limitations.

2.1 Fault Tolerance for HPC

2.1.1 Faults and Fault tolerance

Faults are caused by incorrect states of various factors, including hardware, software, and environment. For example, processor failure and hard disk drive malfunction are common faults in hardware. Bugs, race conditions, and deadlock are examples of faults in software. Faults can be classified into hard and soft faults based on their impacts. Hard faults cause an application or system to crash [16] Soft faults cause an erroneous deviation in applications but without an interruption or include bit-flips which can lead to silent data corruption (SDC) [117]. Soft fault is usually not visible to the application or end user. In this dissertation, we consider both hard and soft faults in hardware and assume that the software environment is faultless. We assume that the software environment is still able to propagate errors generated by hardware faults. Soft faults are commonly grouped into three categories [117]: Detected and Corrected Error (DCE), Detected but Uncorrected Error (DUE), and Silent Data Corruption (SDC). Hard faults have more categories. For example, there are three common and frequent hard faults at system level [48]: System-Wide Outage (SWO),

Single Node Failure (SNF), Link and Node Failure (LNF).

Faults are expected to occur more frequently on exascale systems as the hardware and software become more complex. A common measurement of fault frequency is MTBF. Let the MTBF of a single node be λ, Λ—MTBF of the whole system with N nodes—is estimated as $\Lambda = \lambda/N$ with a Poisson distribution of faults [117]. Figure 2.1 indicates that the MTBF of an exascale system is within an hour if projected from Petascale systems [48]. Here we assume a petascale machine consists of 20K compute nodes built with today's technology and an exascale machine consists of 1M compute nodes with 11 nm technology [14, 117]. We use the same method as in [48, 117] to estimate the MTBF of various fault classifications on a single node or the whole system. We conservatively assume that MTBF is only affected by system size and node-level technology. The actual situation might be worse [30, 117].

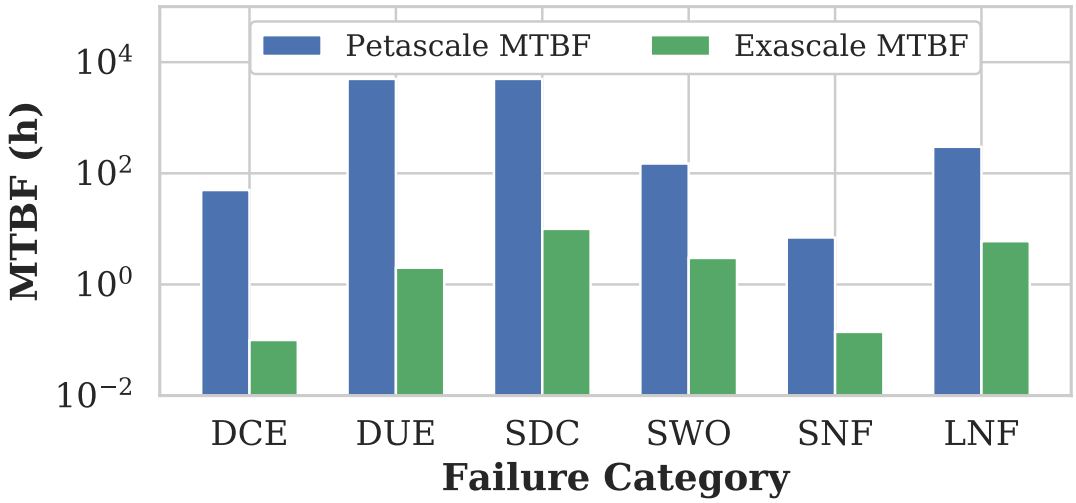


Figure 2.1: Estimated MTBF for exascale systems from petascale systems.

Fault tolerance is indispensable at exascale systems where MTBF is small — i.e., within hours or minutes [48]. Without resilience, most applications will make little forward progress in computation or return erroneous results. Resilient computing requires fault detection and recovery. In this work, we focus on fault recovery and assume that faults are detected and confined to a subset of data structures [29]. There are three main recovery approaches for soft and hard faults: Double Modular Redundancy (RD), Checkpoint-restart (CR), and Forward-recovery (FW).

2.1.1.1 Rollback Recovery

Rollback recovery reverts the application to the previous correct state when a fault occurs. It is widely used to provide resilience in current HPC environments. Mechanisms of rollback recovery can be classified into checkpoint-restart approaches [117, 61] and message logging approaches [69]. Checkpoint-restart periodically saves data to levels of stable storage devices and rollbacks to the latest correct state in case of a fault. To avoid rollback of a global state, message logging restarts only the failed processes. Message logging protocols require that each process periodically saves its local state and log the messages received since the previous state. When a fault occurs, the failed processes are replaced by new processes, and the rest of processes will re-send the messages to the new processes and make progress or wait idle until recovery is finished. In this dissertation, we evaluate the checkpointing/restart technique as a typical example of rollback recovery.

In large-scale systems, fault tolerance mechanisms involve checkpointing /restart (CR) from a parallel file system [117]. In related research for CR, Berkeley Lab Checkpoint Restart (BLCR) [61] are system-level CR with disks for HPC applications that communicate through MPI. Disadvantages of classical CR strategy exist as the unacceptable time to checkpoint, and the global restart even if only one process fails. Diskless checkpointing [100] is proposed to reduce the checkpointing overhead by storing the checkpoints locally in CPU memory. However, diskless checkpointing can not survive node failures or the failures of the whole system. To reduce the checkpoint time with disks and support resilience for more types of failures, Fault Tolerance Interface (FTI) [18] and Scalable CR (SCR) [91] uses multi-level CR that combines several storage technologies to store the checkpoint, including local storage, storage on another partner node, distributed storage in multiple nodes, and the parallel file system. In addition to the above optimizations by using faster checkpoint storage, another approach to reduce the checkpointing overhead is reducing the checkpoint size. Recent works explore compression techniques to reduce the checkpoint data size, but at the expense of adding error into the checkpoint. Sasaki et al. [105] proposes a lossy compression approach based on wavelet transformation to reduce the checkpoint size for climate applications while minimizing the errors. Calhoun et al. [28] leverages the numerical properties of partial differential equation (PDE) simulations to evaluate the feasibility of using lossy compression in checkpointing PDE simulations. These optimizations of reducing the checkpoint size can not provide adaptive data protection because they require the understanding of the critical data in some particular scientific application.

2.1.1.2 Redundancy

Modular redundancy duplicates computation, communication, and data to recover a faulted process with the healthy replication. Modular redundancy can be partial, dual, or triple based on the number of replicas for each process. Typically, dual modular redundancy (DMR) can detect soft errors via replica comparison, and recover the faulty replica identified by another technique [47]. Triple modular redundancy (TMR) can detect and correct the faulty replica via majority voting. Prior work shows that redundancy-based approaches are more energy-efficient than CR at extreme-scales [53].

Studies of modular redundancy have been developed significantly in recent years, which focus on thread-level [58], process-level [110], and state-machine replication [118]. Previous studies make efforts to reduce the replication overhead from the management of extra messages required for replication. rMPI [54], MR-MPI [53], and RedMPI [56] are proposed to provide transparent fault detection and recovery to MPI programs via the MPI profiling interface. rMPI implements protocols that ensure identical message ordering between replicas. It addresses the replication overhead by reducing the number of communications between replicated processes. MR-MPI utilizes the MPI performance tool interface to intercept MPI calls from the application and to hide all redundancy related mechanisms. Unlike rMPI and MR-MPI which focus only on hard faults, RedMPI compares the messages sent by replicated processes for the detection of silent data corruptions. RedMPI still has the limitation of extra communication overhead because it synchronizes main and replicated processes whenever there is inter-process communication.

Recent works on redundancy mainly focus on reducing overhead instead of consuming $2\times$ or $3\times$ the resources. Previous studies on partial replication [19, 57] explores asymmetry of more fragile resources in reliability and smartly replicates only those processes. Other similar ideas such as selective redundancy [119, 120] and intra-parallelization [102] studies how to identify and selectively replicate only the reliability-critical tasks or computation instead of complete replication. Another scheme of shadow or lazy replication [42] executes the replicated process at a reduced CPU rate to save power and energy.

2.1.1.3 Forward Recovery

Instead of rollback recovery or redundancy, forward-recovery schemes [67, 7] have been studied to create a new correct state and avoid resilience overhead when no fault occurs. In forward recovery mechanisms, the application runs forward recovery steps to reconstruct a correct or partially correct state via past or remote data after the detection of a fault. Therefore, forward recovery mechanisms make sense when a relatively small portion of the global state changes and requires that the runtime environment stays alive. Forward recovery can not handle system-level failures, but they can be widely used in parallel applications where data in most processes is correct.

Previous studies of forward recovery have explored to reconstruct the lost data with extra computation and focus more on the recovery accuracy. Bland et al. [24] proposes an approach that handles the failure and executes some actions to recover failed data without relying on periodic checkpointing. Their protocol allows each surviving MPI process to create a checkpoint only after a fault and to recover failed processes based on it. Jaulmes et al. [67] presents a general resilience solution for detectable, but uncorrectable errors based on the algorithmic properties. They exact redundancy relations in iterative solvers and leverage them to recover lost data after faults. Agullo et al. [7] utilizes interpolation strategies to reconstruct the lost data for Krylov subspace linear solvers. Their reconstruction via direct solvers has non-negligible computational costs associated as the expense of achieving higher recovery accuracy. In this dissertation, we explore the trade-off between the accuracy and the overhead of forward recovery and optimize forward recovery strategies with low overhead.

2.2 Resilience for Linear Solvers

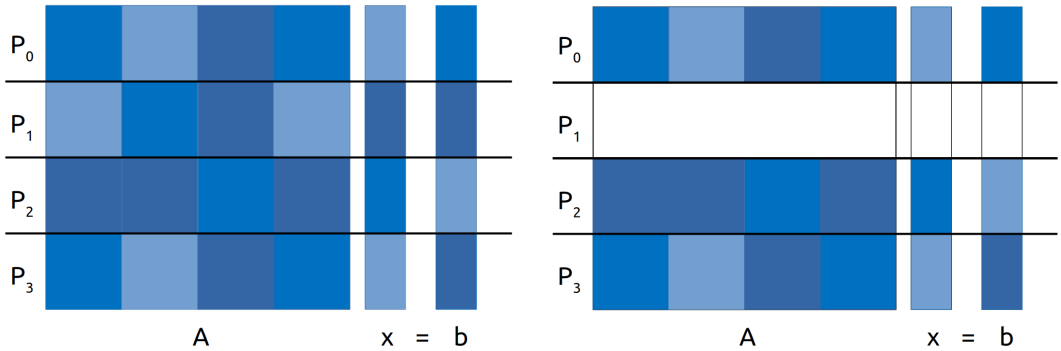
2.2.1 Resilient Linear Solvers

In many HPC applications, the solution of linear systems is the most computational-intensive kernel. We consider one basic form of it as follows:

$$Ax = b \tag{2.1}$$

where the matrix $A \in \mathbb{R}^{n \times n}$ is symmetric, positive-definite (SPD) and real, the solution x is a column vector with n entries and the right-hand side b is a column vector with n entries. For the

solution of a linear system like this, we focus on parallel iterative solvers as the common method in many HPC applications. And we take the Conjugate Gradient (CG) method (Algorithm 1) as an example to research fault recovery of the iterative solver. We assume that all A, x, b are parallelized to several processors via a block-row partition, as Figure 1 (a) shows. Let $A_{;,p_i}$, x_{p_i}, and b_{p_i} be the block-rows stored in processor i.



(a) Before faults (b) After a fault in P_1

Figure 2.2: Recovery pattern of an iterative solver. (a) Matrix A, Vector x and b are parallelized to four processors. (b) When a fault occurs in Processor P_1, the data stored in it is erroneous or lost.

Algorithm 1: The conjugate gradient method (CG)

Require: A symmetric positive definite (SPD) matrix $\mathbf{A} \in \mathbb{R}^{I \times I}$;
Ensure: Vector $\mathbf{x}$;
1: Initial guess $\mathbf{x}_0$;
2: Compute $\mathbf{r}_0 = \mathbf{A}\mathbf{x}_0 - \mathbf{b}$;
3: $\mathbf{p}_0 = \mathbf{r}_0$;
4: **for** $k = 1, 2, ...,$ until convergence **do**
5: $\quad \boldsymbol{\alpha}_k = \mathbf{r}_k^T \mathbf{r}_k / \mathbf{p}_k^T \mathbf{A} \mathbf{p}_k$;
6: $\quad \mathbf{x}_{k+1} = \mathbf{x}_k + \boldsymbol{\alpha}_k \mathbf{p}_k$;
7: $\quad \mathbf{r}_{k+1} = \mathbf{r}_k - \boldsymbol{\alpha}_k \mathbf{A} \mathbf{p}_k$;
8: $\quad \boldsymbol{\beta}_k = \mathbf{r}_{k+1}^T \mathbf{r}_{k+1} / \mathbf{r}_k^T \mathbf{r}_k$;
9: $\quad \mathbf{p}_{k+1} = \mathbf{r}_{k+1} + \boldsymbol{\beta}_k \mathbf{p}_k$;
10: **end for**

2.2.1.1 Algorithm-Based Fault Tolerance

Algorithm-Based Fault Tolerance (ABFT) techniques detect and recover from errors in linear algebra operations by the use of checksums [64]. Significant research has been proposed for different ABFT schemes to address soft faults [35, 44] or hard faults [8, 36, 67] or both [129, 37, 41]. These ABFT schemes can address faults in a certain type of matrices or linear systems. One important

research branch of ABFT focuses on algorithms involving sparse matrices with iterative methods because they are often the most computational intensive kernel in many scientific applications. To overcome the major limitation of ABFT for its considerable computational overhead, in [49, 34, 78], related research is proposed as "lossy approach", which leverages recomputation of the lost data and describes an approach redundancies of a parallel linear solver implementation instead of recovering by checkpoint or checksum. Sloan et al. [112] propose the use of fine-grained partial recomputation based on finding the location of errors efficiently to achieve forward recovery. Agullo et al. [8] address hard faults by exploiting interpolation to define a new guess of the lost data to restart and recover the Krylov solvers. A related approach by Chen et al. [35] can detect soft errors in Krylov methods by using the properties of the algorithm. Some research of ABFT focus on Fast Fourier Transform [83], Matrix Factorizations [128], and general iterative methods [122]. ABFT has more general type and could also work in FT-GMRES [52] and multigrid [90]. In the latest research of ABFT, Huber et al. [65] combines domain partitioning with geometric multigrid methods to obtain resilient solvers based on the redundant storage of ghost values. Scholl et al. [107] proposes a fault tolerance approach to implicitly provide error locations and to enable partial recomputations for erroneous outputs after error detection. These ABFT algorithms are often combined with a rollback-recovery mechanism, which brings an overhead for a fault-free situation. In this dissertation, we propose optimizations based on forward-recovery schemes in [7] to avoid an overhead when no fault occurs.

2.3 Interplay between Power-scalability and Resilience

Besides resilience, another major concern for HPC is to reduce energy consumption for both economic and environmental reasons. Previous studies mainly focus on improving either resilience or energy efficiency, usually at the expense of one another. However, resilience incurs power and time overheads and exacerbates the power challenge. A deep understanding of the interplay among scalability, energy-efficiency and resilience are required to achieve the best trade-off between them.

2.3.1 Energy-efficiency for resilience mechanisms

Prior works investigate the energy cost of resilience, but is limited to the execution time of checkpointing and message logging [51, 86] and the energy impact of checkpointing frequency [15] and Dynamic Voltage and Frequency Scaling (DVFS) in checkpointing [88]. Diouri et al. [51] eval-

uates CR and other existing fault tolerance protocols from energy consideration by running real HPC applications and monitoring energy consumption. They conclude that the difference in energy consumption mainly depends on the execution time because operations of computing, checkpointing, and message logging consumes similar amounts of power. Meneses et al. [86] evaluates how fault tolerance and energy consumption interplay for three rollback-recovery schemes, including CR, message logging and parallel recovery. Their paper shows that parallel recovery consumes less energy because it reduces the restart time. Aupy et al. [15] explores energy reduction for CR via optimization on the checkpointing period. Their approach obtains different optimal periods for either minimizing the total execution time or minimizing the total energy consumption. Mills et al. [88] utilizes DVFS to throttle CPU speed during checkpoint writes to achieve energy savings with little impact in time to solution. Rajachandrasekar et al. [32] proposes a power-aware checkpointing framework via efficient utilization of I/O and CPU by data funneling and selective core power capping. Their approach addresses the problem that the naive use of power capping during checkpointing phases can incur considerable performance degradation.

Recent studies explore energy-efficiency for resilience mechanisms other than CR. Grant et al. [59] proposes a power-reliability metric that imposes a quantifiable penalty to energy savings techniques for increased time providing reliability. They compare energy saving techniques that take into account reliability of HPC systems at scale and account for the probability of failure increase due to time overhead from energy saving techniques. Yetim et al. [131] presents an energy optimization framework based on Mixed-Integer Linear Programming (MILP) that meets performance and resilience constraints. Their framework demonstrates energy, performance, and resilience can be flexibly exploited as needed for different HPC applications. Scholl et al. [108] adapts the underlying precision in Preconditioned Conjugate Gradient (PCG) solvers on approximate computing hardware to gain energy-efficiency.

Instead of only looking into execution time or power individually, we investigate energy optimizations of reducing the communication overhead via localized reconstruction and DVFS during the recover/restart phase. We further extend our recovery schemes with the matrix-aware optimization to lower overhead.

2.3.2 Model-based analysis for energy-efficiency and resilience

Analytical models built from fine-grain measurement data are promising approaches to project time, energy, and resilience on large-scale systems. Scalability models are widely used to evaluate the performance of parallel applications. Amdahl's law [11] and Gustafson's law [60] are two well-known models to capture fault-free parallel execution time for fixed-size problems or fixed-time problems, respectively.

Recent works of scalability models evaluate failure impact and therefore can account for application performance in the presence of failures. Since CR is widely used for HPC applications in large scale systems, several models have been studied to capture job execution time with CR [132, 43, 133]. Young's and Daly's models [132, 43] compute the optimal checkpoint interval and predict the execution time with faults. Their models are applicable for single-level checkpointing and typically imply expensive checkpointing overhead for large-scale jobs. Wang et al. [126] presents a model that takes into account failures during checkpointing and recovery, and correlated failures. They leverage Stochastic Activity Networks (SAN) to model coordinated checkpointing for large-scale systems. Their model also defines the optimal number of processors that maximize the amount of total useful work with CR. Models by Zheng et al. [133] focus on extending Amdahl's law and Gustafson's law by considering system failures, and use Daly's model to derive reliability-aware scalability models with checkpointing. They use the models to demonstrate the benefits of fast recovery and proactive failure prevention via process migration. Tan et al. [121] models the integrated effects of energy-efficiency and resilience for CR with DVFS in large-scale systems by extending the Amdahl's Law and the Karp-Flatt Metric. Their model demonstrates that typical HPC parameters, such as the number of CPU cores, frequency/voltage, and failure rates, have an inherent causal relationship with each other.

Unlike these models focusing on energy-efficiency and resilience for checkpointing, this dissertation presents generalized models to evaluate and compare time and energy across several resilience schemes, such as CR, module redundancy, and forward recovery.

2.4 Performance-scalability for Sparse Tensor Decomposition

2.4.1 Tensor and tensor decomposition

Tensors are multidimensional arrays and often sparse that are utilized by applications spanning a wide range of domain areas, such as quantum chemistry, (healthcare, social network, brain signal, electrical grid) data analytics, signal processing, machine learning, and recommendation systems [70, 97, 95, 5, 12, 98, 74, 40, 63]. Tensor decompositions are a class of tensor methods for data analytics, low-rank approximation, data compression, and so on. In this work, we study the CANDECOMP/PARAFAC decomposition (CPD), one of the most popular tensor decompositions.

Symbols	Description
$\mathcal{X}$	A sparse tensor
$\mathbf{X}_{(n)}$	Matricized tensor $\mathcal{X}$ in dimension-n
$\mathbf{A}, \mathbf{B}, \mathbf{C}, \tilde{\mathbf{A}}$	Dense matrices
$\mathbf{a_r}, \mathbf{b_r}, \mathbf{c_r}$	Dense vectors
$\boldsymbol{\lambda}$	Weight vector
N	Tensor order
I_n	Tensor dimension sizes
M	#Nonzeros of the input tensor $\mathcal{X}$
R	Approximate tensor rank (usually a small value)
I_l	Layer dimension size
I_p	#Local matrix rows
P	#MPI processes

Table 2.1: Symbols and notations.

We use different fonts for tensors ($\mathcal{X} \in \mathbb{R}^{I \times J \times K}$), matrices ($\mathbf{A} \in \mathbb{R}^{I \times J}$), and vectors ($\mathbf{x} \in \mathbb{R}^{I}$) in this paper following the work [74]. A nonzero (i, j, k)-element of tensor $\mathcal{X}$ is x_{ijk}. Figure 2.3 shows a sparse third-order tensor with dots representing nonzero entries. We assume a Nth-order sparse tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ with M nonzeros in the subsequent context, sometimes we use a third-order tensor for simplicity. If a tensor $\mathcal{X}$ has one or more dimension(s) that are very small relative to the other dimensions or the nonzero values are not uniformly distributed in one or more dimensions, then we call it an *irregular tensor*. A *slice* is a two-dimensional cross-section of a tensor, obtained by fixing all indices but two, e.g., $\mathbf{S}_{::k} = \mathcal{X}(:, :, k)$. We summarize the symbols and notations in Table 2.1.

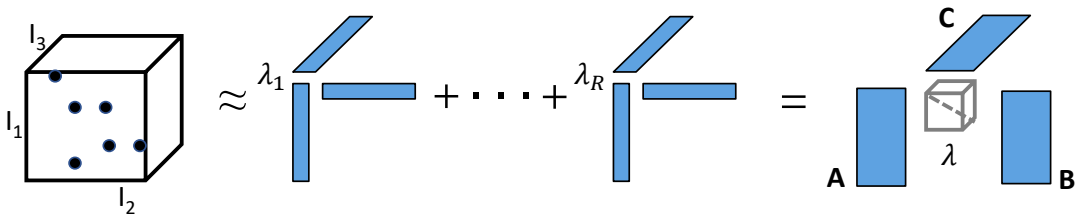


Figure 2.3: CPD for a third-order sparse tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times I_3}$.

2.4.2 Distributed Cpd

CANDECOMP/PARAFAC decomposition (CPD) factorizes a tensor into a sum of component rank-one tensors [74]. Figure 2.3 illustrates a third-order CPD. In general, CPD approximates a Nth-order tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$ as

$$\mathcal{X} \approx \sum_{r=1}^{R} \lambda_r \mathbf{a}_r^{(1)} \circ \cdots \circ \mathbf{a}_r^{(N)} \equiv [\![\boldsymbol{\lambda}; \mathbf{A}^{(1)}, \ldots, \mathbf{A}^{(N)}]\!], \tag{2.2}$$

where R is the canonical rank of tensor $\mathcal{X}$, the number of component rank-one tensors [74]. In a low-rank approximation, R is usually chosen to be a small number less than 100. The outer product of the vectors $\mathbf{a}_r^{(1)}, \ldots, \mathbf{a}_r^{(N)}$ produces R rank-one tensors. $\mathbf{A}^{(n)} \in \mathbb{R}^{I_n \times R}, n = 1, \ldots, N$ are the *factor matrices*, each formed by taking the corresponding vectors as its columns. We normalize these vectors to unit magnitude and store the factor weights in the vector $\boldsymbol{\lambda} = \{\lambda_1, \ldots, \lambda_r\}$. Typically, the factor matrices $\mathbf{A}^{(n)}$ are given initial values and solved iteratively.

Data decomposition and distribution. For large tensors, the number of nonzeros M and the resulting factor matrices $\mathbf{A}^{(n)}$ are large and easily exceed the memory capacity of a single node. To meet the needs of large-scale data processing, distributed CPD algorithms, such as coarse-grained [39], medium-grained [114, 6], and fine-grained [72, 71] strategies, have been developed. Medium-grain is one of the most successful from the studies [114, 101, 6, 17] and is the baseline for this work (described in Section 2.4.3). To efficiently store large tensors, we consider one state-of-the-art tensor format, Compressed Sparse Fiber (CSF) for general unstructured sparse tensors. CSF [114] is a hierarchical and fiber-centric format that effectively generalizes the Compressed Sparse Row (CSR) sparse matrix format to tensors.

Distributed algorithm. We focus on the most popular medium-grained, bulk-synchronous distributed CPD algorithms [114, 17, 101, 6], adopted in multiple libraries including SPLATT, the

Surprisingly ParalleL spArse Tensor Toolkit [116] and ENSIGN [77]. It has shown outstanding performance and scalability as well as efficient memory usage compared to the counterparts [39, 72], evaluated in the work [114, 17, 101, 6]. Medium-grained tensor distribution, an N-dimensional partitioning (N as tensor order) on a tensor, corresponds to a 2D stationary algorithm in traditional dense matrix multiplication [106] which has been proven to be performance efficient in the SUMMA algorithm [125] included in ScaLAPACK [38] and PLAPACK [10] libraries.

Algorithm 2: Medium-grained, bulk-synchronous distributed CPD-ALS algorithm (MGBS-CPD).

Require: An Nth-order sparse tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ with M nonzeros, P MPI processes;
Ensure: Vector $\boldsymbol{\lambda}$ and dense matrices $\mathbf{A}^{(n)} \in \mathbb{R}^{I_n \times R}, n = 1, \ldots, N$;

// Variables
Initialize matrices $\mathbf{A}^{(n)}, n = 1, \ldots, N$;
$\mathbf{A}_l^{(n)}$ is the layer-distributed matrix, needed by MTTKRP computation on p.
$\mathbf{U}_n \in \mathbb{R}^{R \times R}, n = 1, \ldots, N$ is local temporary data.
// Preprocessing
Distributedly load $\mathcal{X}$ to P MPI processes' local memory
Grid configuration $\mathcal{G}$: Get rank dimensions $P_n, n = 1, \ldots, N$ decomposed from P and initialize MPI communicator
Determine a **distribution policy** $\mathcal{D}$
▷ Tensor partitioning, $\mathcal{X}_p$ locally owned by process p.
Redistribute $\mathcal{X}$ according to $\mathcal{D}$
▷ Matrix partitioning, $\mathbf{A}_p^{(n)}$ locally owned by process p.
Distribute all $\mathbf{A}^{(n)}$ to $\mathbf{A}_l^{(n)}$ and $\mathbf{A}_p^{(n)}$, $n = 1, \ldots, N$ according to $\mathcal{D}$
Get $\mathcal{X}_p$ after removing empty slices and get index mapping from $\mathcal{X}_p$ to $\mathcal{X}$
Get the indices in $\mathbf{A}_p^{(n)}$ that need to communicate in AlltoAll($\mathbf{A}_p^{(n)}$)
Randomly initialize $\mathbf{A}_l^{(n)}$
// Computation
$\mathbf{A}_l^{(n)} = \text{AlltoAll}(\mathbf{A}_p^{(n)})$; $\mathbf{U}_n = \text{AllReduce}\ (\mathbf{A}_p^{(n)T} \mathbf{A}_p^{(n)})$
do
  for $n = 1, \ldots, N$ **do**
    $\tilde{\mathbf{A}}_l^{(n)} = \text{MTTKRP}(\mathcal{X}_p, \mathbf{A}_l^{(1)}, \ldots, \mathbf{A}_l^{(n-1)}, \mathbf{A}_l^{(n+1)}, \ldots, \mathbf{A}_l^{(N)})$ ▷ MTTKRP
    $\tilde{\mathbf{A}}_p^{(n)} = \text{AlltoAll}(\tilde{\mathbf{A}}_l^{(n)})$
    $\tilde{\mathbf{A}}_p^{(n)} = \tilde{\mathbf{A}}_p^{(n)}\ (\mathbf{U}_1 * \cdots * \mathbf{U}_N)^{\dagger}$ ▷ MAT SOLVE
    $\tilde{\boldsymbol{\lambda}} = \text{Normalize}\ (\tilde{\mathbf{A}}_p^{(n)})$ ▷ MAT NORM
    $\tilde{\mathbf{U}}_n = \text{AllReduce}(\tilde{\mathbf{A}}_p^{(n)T} \tilde{\mathbf{A}}_p^{(n)})$ ▷ MAT $\mathrm{A}^T\mathrm{A}$
    $\tilde{\mathbf{A}}_l^{(n)} = \text{AlltoAll}(\tilde{\mathbf{A}}_p^{(n)})$
  end for
while fit not change or maximum iterations exhausted

2.4.3 Medium-Grained, Bulk-Synchronous Distributed Cpd Algorithm

We extract the general medium-grained, bulk-synchronous distributed CPD algorithm as a template in Algorithm 2, named as MGBS-CPD, extracted from the state-of-the-art work [114, 17, 6].

Medium-grained data distribution. The medium-grained decomposition uses a nonzero-oriented data decomposition strategy. After loading a tensor file into each process' memory in a distributed way, two performance critical steps follow: process grid configuration and distribution policy determination. Based on these two steps, a tensor $\mathcal{X}$ is N-dimensional partitioned into subtensors in a non-overlapping fashion and distributed to processes; each factor matrix $\mathbf{A}^{(n)}$ is distributed to the processes according to the distribution policy on each dimension-n.

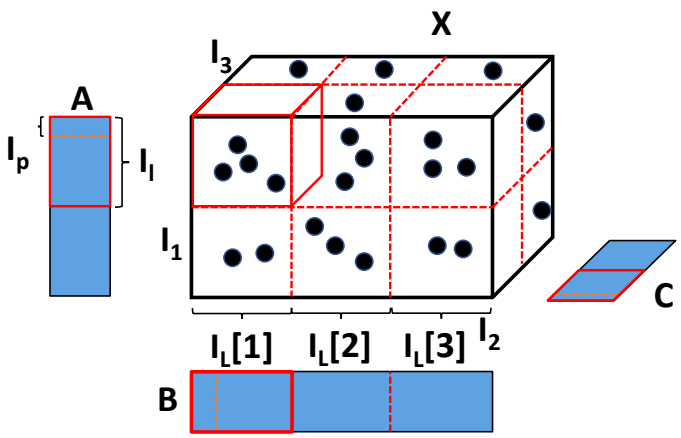


Figure 2.4: Tensor and matrix distribution over a $12 = 2 \times 3 \times 2$ process grid. Dotted lines on matrices indicate local matrix storage in one process. The tensor is partitioned into $2 \times 3 \times 2$ subtensors, each mapped to a process. Each factor matrix is first partitioned by the layers affiliated with tensor partition, and then evenly split among the corresponding process subgrid.

Take a $P = 2 \times 3 \times 2$ process grid [1] in Figure 2.4 as an example. The tensor $\mathcal{X}$ is partitioned to $2 \times 3 \times 2$ subtensors, each associated with a process and saved in its memory. Meanwhile, each $\mathbf{A}^{(n)}$ is partitioned to P submatrices along its dimension with two levels: the *layer-level* corresponds to the tensor computation and splits each matrix to sub-matrices $\mathbf{A}_l^{(n)}$ affiliated to its row dimension (blank boxes on $\mathbf{A}$), and the *process-level* further evenly splits $\mathbf{A}_l^{(n)}$ to $\mathbf{A}_p^{(n)}$ for each process p in the corresponding subgrid (dashed lines on $\mathbf{A}$). Note that $\mathbf{A}_p^{(n)}$ is the actual local matrix storage per process, while $\mathbf{A}_l^{(n)}$ is only stored during tensor-matrix computation (MTTKRP, described below).

Bulk-synchronous parallel algorithm. Computation is accordingly partitioned with

[1] Due to our hybrid MPI+OpenMP implementation, the MPI processes count is referred in grid configuration.

the above data decomposition — i.e., each process only does local tensor/matrix computation and updates its own matrix partition $\mathbf{A}_p^{(n)}$. Thus, the grid configuration and distribution policy, which determines the data decomposition, play critical roles in the performance of the CPD algorithm.

Algorithm 2 shows the bulk-synchronous parallel algorithm for an Nth-order tensor using a traditional alternating least square algorithm [74]. The bulk-synchronous parallel algorithm is generalized from almost all existing distributed CPD-ALS implementations [114, 17, 101, 6, 72, 71, 39, 22]. This is an iterative implementation. In each iteration, matrices are updated one-by-one; each time, all but one matrix are fixed to update the matrix $\tilde{\mathbf{A}}^{(n)}$. The algorithm comprises four main computation kernels. MTTKRP is the only kernel that computes on the sparse tensor, and has been studied most for optimization in previous work [114, 72, 71]. The other three compute on dense matrices only. Note that all the four steps except MAT SOLVE have mixed computation and communication.

- MTTKRP: each process computes the Khatri-Rao product of its subtensor with all but one layer-partitioned $\mathbf{A}_l^{(1)}, \ldots, \mathbf{A}_l^{(n-1)}, \mathbf{A}_l^{(n+1)}, \ldots, \mathbf{A}_l^{(N)}$, which are obtained from remote memory by communicating with other processes.
- MAT SOLVE: each process updates $\tilde{\mathbf{A}}_p^{(n)}$ using the Cholesky method based on the temporary results from MTTKRP.
- MAT NORM: each process normalizes $\tilde{\mathbf{A}}_p^{(n)}$ locally and then performs a parallel reduction to obtain $\boldsymbol{\lambda}$.
- MAT $\mathrm{A}^T\mathrm{A}$: each process uses symmetric matrix multiplication locally and then performs a reduction to form the new $\tilde{\mathbf{U}}_n$ for the next iteration.
- MTTKRP COMM: $\tilde{\mathbf{A}}_p^{(n)}$ is updated by communicating $\tilde{\mathbf{A}}_l^{(n)}$ after local MTTKRP computation. Consequently, communications are involved to update $\tilde{\mathbf{A}}_l^{(n)}$ from $\tilde{\mathbf{A}}_p^{(n)}$ to prepare the layer-partitioned $\tilde{\mathbf{A}}_l^{(n)}$ for the next MTTKRP.

The complexity lies in both communication and local computations influenced by the grid configuration and distribution policy from the preprocessing steps. All communication within CPD computation is for dense matrices, while sparse communication only exists in preprocessing for sparse tensors. Due to the sparsity of the tensor, the communication volume for dense matrices could be very imbalanced.

2.4.4 Related Work of distributed Cpd

Three major bulk-synchronous distributed CPD algorithms have been proposed: coarse-grained [39], medium-grained [114, 6, 17], and fine-grained [72]. DFacTo [39] designs a coarse-grained distributed CPD implementation. DFacTo uses an efficient MTTKRP algorithm that is posed as a series of sparse matrix vector multiplications (SpMVs). DFacTo consists entirely of SpMV operations and therefore can take advantage of a wealth of existing research that can be applied to an efficient parallel implementation. SPLATT [116] is a popular sparse tensor library which includes medium- and fine-grained distributed CPD implementations. The medium-grained decomposition uses an m-mode decomposition over the tensor and related 1D decompositions on the factor matrices. HyperTensor [72] uses a fine-grained decomposition that nonzeros of a tensor are individually assigned to processes. The most successful computation of fine-grained decomposition relies on hypergraph partitioning. A balanced partitioning of the hypergraph leads to a load-balanced computation with low communication volume.

In addition to these three major bulk-synchronous distributed CPD algorithms, recent works explore optimization in distributed CPD. ENSIGN [77] uses special sparse tensor data structures mode-specific sparse (MSS) and mode-generic sparse (MGS) with an optimization that improves data reuse and reduces redundant computations in tensor decompositions [17]. But ENSIGN requires significantly higher memory usage due to its special data structures. Other efforts employed MapReduce/Hadoop or Spark programming models on cloud platforms, such as GigaTensor [70], HaTen2 [68], and CSTF [22]. These works utilize the parallelism of MapReduce by reformulating MTTKRP as a series of Hadamard products. There are no dependencies during a Hadamard product, and each element of the output can be computed in parallel.

Prior studies [114, 101] have shown that medium-grained CPD generally obtains the optimal state-of-the-art performance. The medium-grained decomposition addresses the limitations of coarse-grained methods by avoiding complete replication and communication of the factors. In addition, the medium-grained decomposition does not require computationally expensive pre-processing such as hypergraph partitioning to have a low communication volume. Our work develops upon medium-grained distributed CPD and through optimizing grid configuration and distribution policy to improve performance and scalability.

Chapter 3

Resilient and Energy Efficient Scalable Linear Solvers

Exascale computing must simultaneously address both energy efficiency and resilience as power limits impact scalability and faults are more common. Unfortunately, energy efficiency and resilience have been traditionally studied in isolation and optimizing one typically detrimentally impacts the other. To deliver the promised performance within the given power budget, exascale computing mandates a deep understanding of the interplay among energy efficiency, resilience, and scalability.

In this chapter, we propose novel methods to analyze and optimize the costs of common resilience techniques, including checkpoint-restart and forward recovery. We focus on sparse linear solvers as they are the fundamental kernels in many scientific applications. In particular, we present generalized analytical and experimental methods to analyze and quantify the time and energy costs of various recovery schemes on computer clusters, and develop and prototype performance optimization and power management strategies to improve energy efficiency. Moreover, we take a deep dive into the forward recovery that recently started to draw attention from researchers, and propose a practical matrix-aware optimization technique to reduce its recovery time. The result shows that while the time and energy costs of various resilience techniques are different, they share the common components and can be quantitatively evaluated with a generalized framework. This analysis framework can be used to guide the design of performance and energy optimization technologies. While

each resilience technique has its advantages depending on the fault rate, system size, and power budget, the forward recovery can further benefit from matrix-aware optimizations for large-scale computing.

3.1 Performance, Energy, and Resilience Co-Modeling

We focus on three performance metrics for a given workload w: time-to-solution T, power P, and energy-to-solution E. T had been the sole measure in parallel computing until power and energy began to constrain performance and scalability [82]. These metrics interact and their interplay depends on workload characteristics, performance optimization, and power and energy saving technologies.

Each of the metrics is altered by faults and the resilience techniques employed to tolerate faults. Faults, if occurring frequently, can have a dominating effect in large scale computing. In this work, we analytically model the impact of faults and evaluate the inherent time and energy costs of different resilience techniques.

We use CG as a case of study and examine workload properties commonly in parallel computing. Particularly, we focus on sparse banded matrices. We investigate weak scaling to project the performance and costs for large-scale systems. Specifically, we adopt the *fixed time scaling* approach [111], i.e., the execution time is constant for scaled workloads if parallel overhead is negligible. In our context, the number of non-zero entries and the number of degrees-of-freedom per process remains constant.

3.1.1 Generalized Models

We first present general models to capture the time, power, and energy costs of all resilience techniques under study. The metrics and parameters are presented in Table 3.1.

Time-to-solution for the original workload $T_1(w)$: the amount of time to complete a workload w sequentially on a single core. We denote this time as T_{solve}:

$$T_1(w) = T_{solve} \tag{3.1}$$

Time-to-solution for the scaled workload $T_N(w')$: the amount of time to complete a scaled

	Symbol	Description
Metric	T	Time to solution
	P	Power consumption
	E	Energy to solution
Workload	w	Original workload
	w'	Scaled workload (fixed time)
Parameter	λ	Failure rate
	N	Number of cores

Table 3.1: Metrics and Model Parameters.

workload w' on a system with $N \geq 1$ CPU cores. Equation 3.2 includes the time to solve the scaled problem and parallel overhead in a fault-free situation.

$$T_N(w') = T_{solve} + T_O(N) \tag{3.2}$$

Here T_O is the parallel overhead and is a function of N. Note the same problem-solving time for the original and scaled workloads when the parallel overhead is not considered. The scaled workload w' has the same characteristics as the original workload w, but requires $N\times$ the computation. In the CG case, the size of the matrix A scales accordingly to keep a constant amount of work per process.

In faulty environments with a failure rate of λ, resilience incurs extra cost. We focus on recovery and assume fault detection is performed by other techniques [29] and the detection overhead is factored into the base running time for the solver. Therefore, we extend Equation 3.2:

$$T_N(w') = T_{solve} + T_O(N) + T_{res}(w', N, \lambda) \tag{3.3}$$

where T_{res} is the total time overhead for resilience, including time to checkpoint, recompute lost progress, reconstruct an approximate state, and restart the external environments.

Power consumption for the original workload $P_1(w)$: the amount of power consumed by a workload w during a sequential execution. Conceptually, the power consumption is summed over all computer components. For simplicity, we only account for the CPU core's power for two reasons: (1) cores are the dominant power consumer; (2) their power varies the most across resilience techniques.

Power consumption for the scaled workload $P_N(w')$: the amount of power consumed by N processor cores when executing the scaled workload w'. For the fixed time workload scaling, each

processor core maintains the same computational intensity and thus power. Therefore,

$$P_N(w') = N \times P_1(w) \tag{3.4}$$

In a fault-free situation, the application execution consists of useful problem progress periods and parallel overhead time such as communication and synchronization. Since we are more interested in the impacts of resilience, we assume that the application power profile is the same during progress phases and parallel overhead.

In faulty environments, the power profiles may alter between disjoint normal execution phases and recovery phases, and overlapped execution-recovery phases.

$$P_N(w') = \begin{cases} N \times P_1(w) & \text{execution phase} \\ P_{N,res} & \text{recovery phase} \\ N \times P_1(w) + P_{N,res} & \text{overlapped phase} \end{cases} \tag{3.5}$$

The power consumption during the recovery phase $P_{N,res}$ is discussed and quantified for each resilience technology in Section 3.1.2 and Section 3.3.

Energy-to-solution for the original workload $E_1(w)$: the total amount of energy to complete the workload w on a single core in a fault-free situation. It is the product of power and time — i.e.,

$$E_1(w) = P_1(w) \cdot T_1(w) \tag{3.6}$$

Energy-to-solution for the scaled workload $E_N(w')$: the total amount of energy to complete the scaled workload w' with N processor cores.

In a fault-free situation, it accounts for the energy to solve the problem and the parallel overhead.

$$E_N(w') = N \cdot P_1(w) \cdot \Big(T_{solve} + T_O(N)\Big) \tag{3.7}$$

In faulty environments with a failure rate of λ, additional energy is consumed to support resilience (see Section 3.1.2).

$$E_N(w') = P_N(w')_{avg} \cdot \Big(T_{solve} + T_O(N) + T_{res}(w', N, \lambda)\Big) \tag{3.8}$$

3.1.2 Specific Models for Recovery Schemes

We analyze the recovery cost of a hard or a soft fault, which causes data loss or corruption on a single process p_i. Recovery is needed for the computational environment, lost static data, and lost dynamic data [78]. We assume that the computational environment and the lost static data are recovered immediately, as in [7]. Thus, the challenge is to recover the lost dynamic data — i.e., $x_{p_i}^k$ on the failed process p_i for CG — see Figure 2.2.

We discuss several recovery schemes: Checkpoint/Restart (CR), Redundancy (RD), and Forward Recovery (FW) as shown in Table 3.2. CR and FW include multiple variations. The general models are applicable to all these schemes. However, $T_{res}(w', N, \lambda)$ and $P_{N,res}$ are further refined for each resilience technique.

Type	Scheme	Description
CR	CR-D	Checkpoint to/rollback from disk
	CR-Mul	Multi-level Checkpoint/rollback
RD	DMR	Double modular redundancy
FW	F0	Assign 0 to $x_{p_i}^k$
	FI	Assign initial guess to $x_{p_i}^k$
	LI	linearly interpolate lost $x_{p_i}^k$
	LSI	Interpolate lost $x_{p_i}^k$ with least squares

Table 3.2: Recovery schemes under study

Checkpoint/Restart. The iterative solution, e.g., vector x of CG, is checkpointed to storage periodically at certain iterations and recovered from the most recent correct checkpoint after a fault. Let x^{C_m} be the most recent checkpointing of x performed after the mth iteration when a fault occurs in the kth ($k \geq m$) iteration, the resilience cost $T_{res}(w', N, \lambda)$ with CR includes time to checkpoint the solution vector x and the time lost to compute from x^{C_m} to x^k.

$$T_{res}(w', N, \lambda) = T_{chkpt}(w', N, \lambda) + T_{lost}(w', N, \lambda) \tag{3.9}$$

where T_{chkpt} is the total time spent checkpointing, and T_{lost} is the total time spent in recomputing to arrive at the state before the failure/error occurred.

T_{chkpt} is the product of per checkpointing cost t_C and the number of checkpoints taken.

The latter is derived from the total execution time and checkpointing interval I_C, i.e.,

$$T_{chkpt} = t_C \cdot \frac{T_N(w')}{I_C} \tag{3.10}$$

t_C differs with the checkpoint storage — e.g. local-memory(cheap) or remote disk(expensive). The optimal checkpointing interval, I_C, is a function of failure rate and is commonly approximated with Young's and Daly's approaches [132, 43].

T_{lost} is dependent on the failure rate and the average amount of recomputation time t_{lost}. The latter is approximated as a half of the checkpointing interval. For a failure rate λ, T_{lost} is derived as

$$T_{lost} = t_{lost} \cdot \lambda \cdot T_N(w') \approx \frac{I_C}{2} \cdot \lambda \cdot T_N(w') \tag{3.11}$$

In general, CPUs are not highly utilized during checkpointing and thus consume less power than in the computation phase. That is, $P_{N,res} < N \cdot P_1(w)$. For cases when checkpointing takes a long time, transitioning the CPU's power to a lower power state saves power.

Redundancy. A dual-modular redundancy (DMR) resilience scheme requires $2X$ CPUs to support redundant computation. Assuming an unlimited number of CPUs without a power budget and two independent sets, the recovery time for x^k from the redundant replica after a fault is negligible. Nevertheless, the resilience phases are always concurrent with the normal program progress phases. Resilience causes additional power $P_{N,res}$ for the duration of the application by requiring double the power.

$$P_{N,res} = N \cdot P_1(w) \tag{3.12}$$

Forward Recovery. Forward recovery approximates lost data with simple assignments or reconstruction techniques. A more precise approximation of x^k takes more time/energy to construct but takes fewer extra iterations to converge to the final solution.

The time cost for FW resilience is modeled as:

$$T_{res}(w', N, \lambda) = T_{const} + T_{extra} \tag{3.13}$$

Where T_{const} captures the cost of reconstructing an approximation for x^k, and T_{extra} captures the cost of extra iterations required to converge. The former is the product of the reconstruction count

and the cost per reconstruction t_{const}.

$$T_{const} = \lambda \cdot T_N(w') \cdot t_{const} \tag{3.14}$$

Constructing an approximation of the lost data may or may not involve all CPUs depending on the recovering algorithm. For example, $\widetilde{N} = 1$ during reconstruction for the FW methods under study. Given $\widetilde{N} \leq N$ processes actively constructing the approximation and $N - \widetilde{N}$ CPUs idle, the power during construction is less than that during normal execution.

$$\begin{cases} P_{N,const} = \widetilde{N} \cdot P_1(w) + (N - \widetilde{N}) \cdot P_{idle}, & \text{if constructing} \\ P_{extra} = N \cdot P_1(w), & \text{if extra iter.} \end{cases} \tag{3.15}$$

here P_{idle} is the power consumption when the core is idle.

The energy cost for resilience is the sum of the reconstruction and extra iterations, i.e.,

$$E_{N,res} = P_{N,const} \cdot T_{const} + N \cdot P_1(w) \cdot T_{extra} \tag{3.16}$$

We investigate four FW schemes: filling $x_{p_i}^k$ with all zeros (F0) and the initial guess (FI), linear interpolation (LI) [78] and least squares interpolation (LSI) [7]. These schemes have different reconstruction costs and accuracy. F0 and FI are assignment based and thus do not incur a construction cost — i.e., $T_{const} = 0$. However, they incur a large T_{extra} to converge. On the contrary, LI and LSI are interpolation based and take time to construct more accurate approximations, but require fewer extra iterations to converge. The specific construction cost and extra iteration cost are determined by the workload and matrix properties.

Let $x_{p_i}^{LI}$ be the approximation of $x_{p_i}^k$ for the linear system solved by CG, LI constructs it with linear interpolation:

$$\begin{cases} x_{p_i}^{LI} = A_{p_i,p_i}^{-1}(b_{p_i} - \sum_{j \neq i} A_{p_i,p_j} x_{p_j}^k) & \text{for } j = i \\ x_{p_j}^{LI} = x_{p_j}^k & \text{for } j \neq i \end{cases} \tag{3.17}$$

LSI uses a more complex interpolation scheme and provides a more accurate approximation

than LI. Let $x_{p_i}^{LSI}$ be the interpolation of $x_{p_i}^k$, LSI approximates it with:

$$\begin{cases} x_{p_i}^{LSI} = \min_{x_{p_i}} \|b - \sum_{j \neq i} A_{:,p_j} x_{p_j}^k - A_{:,p_i} x_{p_i}\| & \text{for } j = i \\ x_{p_j}^{LSI} = x_{p_j}^k & \text{for } j \neq i \end{cases} \tag{3.18}$$

The analytical modeling distinguishes between different resilience schemes. Corresponding model parameters are derived from experimental data in the following section.

3.2 Minimizing Recovery Cost

The four FW options (F0, FI, LI, LSI) for filling $x_{p_i}^k$ have two extremes: minimum reconstruction time or minimum convergence time. None is likely to minimize the total recovery time for all workloads. As we show, there is a trade-off between the construction cost and extra iteration cost, and their best combination is determined by workload and matrix properties.

In this section, we present several optimization strategies to reduce the overhead of the LI and LSI recovery schemes. We dismiss the assignment based options as they are workload agnostic. Our optimizations include localized approximate reconstruction, matrix-aware accuracy selection, and power reduction.

3.2.1 Localized Approximate Reconstruction

LI Approximation: LI reconstructs the lost dynamic data $x_{p_i}^{LI}$ by solving a linear system. Let $y = b_{p_i} - \sum_{j \neq i} A_{p_i,p_j} x_{p_j}^k$. The failed process p_i solves the following equation:

$$A_{p_i,p_i} x_{p_i}^{LI} = y \tag{3.19}$$

where all entries of A_{p_i,p_i} are static and are recovered from local storage on process p_i, and y is calculated using entries of x from all the other processes. After a communication step, this problem is solved locally on process p_i.

Previous work [7] uses a sequential LU factorization of A_{p_i,p_i} to get the *exact* solution of $x_{p_i}^{LI}$. LU factorization requires a large amount of memory [62], and incurs high time and energy costs due to the complexity of $O(n^3)$ for the matrix size n. A possible faster alternative is to parallelize

LU factorization. However, parallelization increases communication time and can increase energy consumption by using all the cores.

We propose a more efficient approach to solve Equation 3.19. The key idea is to derive an approximation of $x_{p_i}^{LI}$ locally on process p_i, i.e., using the CG iterative algorithm. The rationale is that the exact solution is not necessary because itself is an approximation of the lost data $x_{p_i}^{k}$. Its benefit comes from two sources: (1) the local execution eliminates communications, and (2) the other processes can enter into sleep states for power savings.

LSI Approximation: LSI reconstructs $x_{p_i}^{LSI}$ by solving a least-squares linear systems, where $\beta = b - \sum_{j \neq i} A_{;,p_j} x_{p_j}^{k}$:

$$(A_{;,p_i}^{T} A_{;,p_i}) x_{p_i}^{LSI} = A_{;,p_i}^{T} \beta. \tag{3.20}$$

$A_{;,p_i}$ is a parallel matrix across N processes. Previous work [7] uses a parallel sparse QR factorization of $A_{;,p_i}$ to get the exact solution of $x_{p_i}^{LSI}$ with a high communication volume depending on the sparsity pattern of A.

We use CG to locally solve for $x_{p_i}^{LSI}$ on process p_i. We first transform the problem to enable local computation. Given the SPD matrix A, then $A_{;,p_i} = A_{p_i,;}^{T}$. Thus, we transform Equation 3.20 as follows:

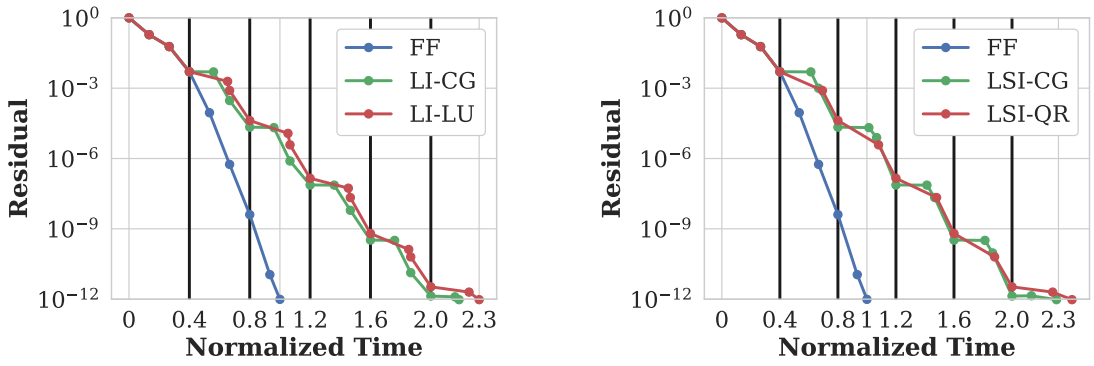
$$(A_{p_i,;} A_{p_i,;}^{T}) x_{p_i}^{LSI} = A_{p_i,;} \beta \tag{3.21}$$

Figure 3.1 shows that CG-based LI and LSI have a shorter time-to-solution than the respective exact solutions. The improvement is 4–15%, depending on the tolerance. By computing a less accurate approximation, CG-based LI and LSI require less recovery time and total time than LU-based LI and QR-based LSI.

3.2.2 Matrix-Aware Approximation Accuracy Selection

This subsection discusses limitations in the CG-based LI and LSI algorithms in [87], and illustrates the challenges to addressing these limitations. It then presents a practical new convergence factor to describe matrices and a matrix-aware overhead control.

Limitations of accuracy oriented reconstruction. The previously proposed CG-based LI and LSI focus on maximizing the accuracy of the lost data reconstruction, disregarding



(a) LI based

(b) LSI based

Figure 3.1: Reconstruction algorithms for Matrix Kuu with 5 faults (black vertical lines). LI/LSI-CG: CG-based LI/LSI forward recovery; LI-LU: LU-based LI forward recovery; LSI-QR: QR-based LSI forward recovery.

the overhead associated with the target accuracy [7, 87]. For example, the exact reconstruction [7] achieves the highest possible accuracy for reconstruction with the complexity of $O(n^3)$, where n is the matrix size. And the CG-based approximation algorithms in Section 3.2.1 uses low residual values or large iteration numbers, e.g., $1e^{-12}$ or 5,000 respectively, to terminate the iterations for all input matrices.

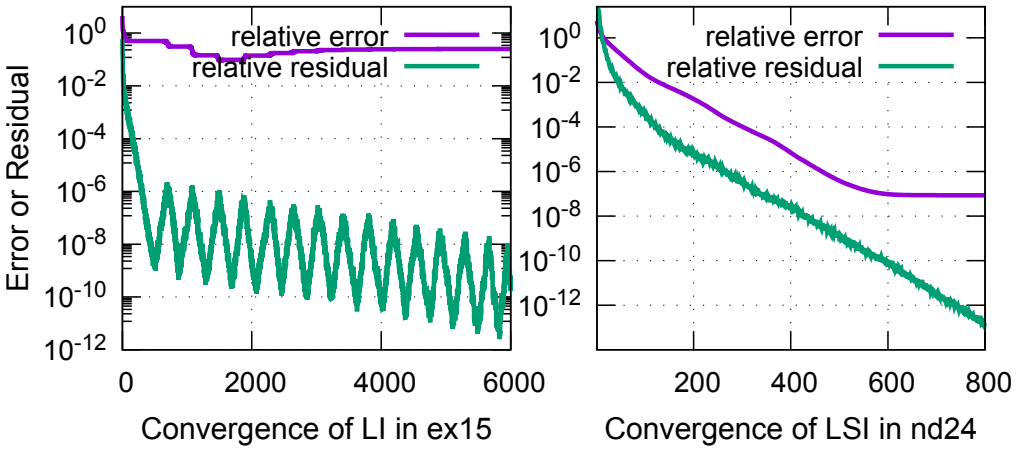


Figure 3.2: Convergence of LI or LSI in matrix $ex15$ and kuu

There are two major limitations for the accuracy oriented reconstruction algorithms. First, the previous algorithms have one certain and costly criterion for all matrices. They typically set a

certain tolerance for relative residual or maximum iterations across all cases, without differentiating the input matrices and reconstruction algorithms. As such, the default values must be conservative to achieve a sufficiently accurate approximation, which incurs unnecessary excessive time and energy costs without improving the quality of the final problem solution for many cases. For example, a matrix with a uniform nonzero distribution may be able to tolerate large residuals and thus only need a coarse approximated reconstruction, but a matrix with non-uniform nonzero distribution needs fine approximations for efficient recovery because it is more difficult to converge. To support both types of matrices, prior work has to set the default tolerance low, which wastes computation resources and time for the matrices with uniform nonzero distribution. Second, the residual used by the previous algorithms doesn't truthfully reflect the accuracy of reconstruction. The relative residual r_k in iteration k is defined as $||Ax_k - b||/||b||$, where x_k and b are column vectors. The ultimate evaluation of accuracy instead should be the relative error to solution, i.e., $err_k = ||x_k - x_t||/||x_t||$, where x_t is the solution vector before a fault and the target reconstruction of LI or LSI recovery. r_k and err_k are different but correlated by the matrix A in that $r_k \cdot ||b|| = A \cdot err_k||x_t||$, and $r_k = 0$ if and only if $err_k = 0$ by the uniqueness of the solution. Because x_t is unobtainable in practice to calculate err_k due to the failure, alternatively, the residual is adopted to evaluate accuracy. However, err_k might have already reached convergence when r_k still varies due to the impact of A [26]. For this reason, the residual-based method would incur more cost than necessary.

As shown in Figure 3.2, the residual-based evaluation takes more time overhead than the error-based evaluation to reach a small target change for the LI-CG and LSI-CG algorithms. For the matrix $ex15$ with non-uniform nonzero distribution, LI takes more than 6,000 iterations to reach the tolerance for the residual of $1e^{-12}$ and 3,000 iterations to reach the tolerance for the error. That is, the residual-based evaluation takes about 50% more time overhead than the error-based evaluation with the LI recovery for matrix $ex15$. The example of the matrix $nd24k$ with uniform nonzero distribution and the LSI algorithm demonstrates the same conclusion.

Challenges of Matrix-Awareness. Unlike the accuracy oriented reconstruction, better solutions should aim to minimize the resilience overhead, which is $T_{res} = T_{const} + T_{extra}$ as in Eq 3.13. Typically, a more accurate reconstruction incurs larger T_{const} and requires smaller T_{extra}. Nevertheless, the addition of T_{const} may be larger or smaller than the saving of T_{extra}, depending on the matrix under study and the accuracy improvement. As reconstruction accuracy typically has diminishing gain, consequently the accuracy oriented methods incur excessive T_{const}. Better

solutions should target the reconstruction accuracy that achieves the best trade-off specific to the matrix under study.

It is challenging to identify the matrix-aware optimal reconstruction in practice, due to the data loss of target reconstruction. In addition, the LSI algorithms typically converge slower than the LI algorithms for the same input matrix. As shown in Figure 3.3 from one fault for two different matrices on 8 nodes, where point D shows the global minimal value of the relative error. The relative error converges to $1e^{-1}$ at D in LI with more than 3500 iterations for matrix $ex15$ in Figure 3.3(a), to $1e^{-5}$ at D with 600 iterations for LSI in matrix $nd24k$ in Figure 3.3(b). These examples demonstrate that matrix-awareness is needed for not only the convergence of errors, but also the difference between LI and LSI algorithms.

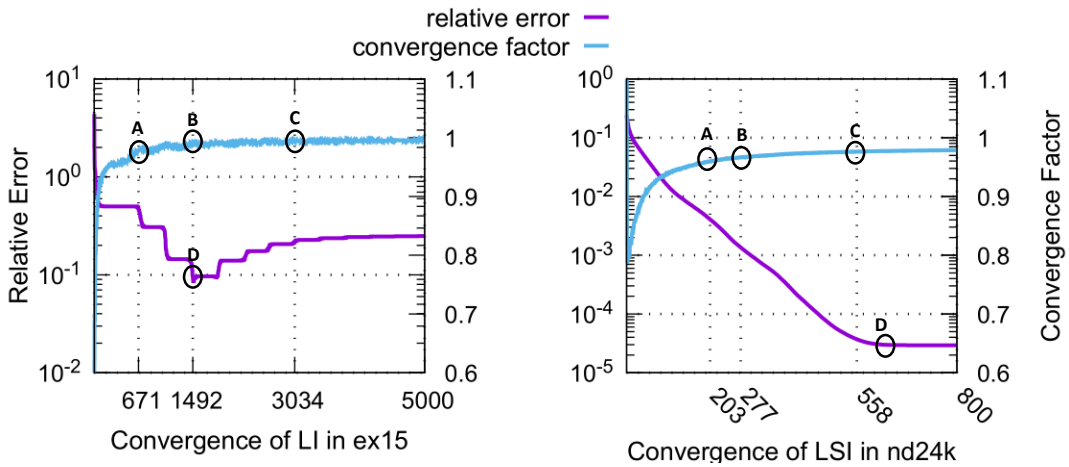


Figure 3.3: Error and Convergence factor in matrix $ex15$ and $nd24k$

Our New Convergence Metrics. To evaluate reconstruction for a given matrix, an alternative to the convergence of error is the condition number based on the Gershgorin circle theorem [127]. However, computing the condition number for a matrix usually takes several hours as its computational complexity is $\mathcal{O}(n^3)$ where n is the matrix size. This time overhead is too expensive, as it is much larger than solving the linear system.

In this work, we propose an on-line and low-cost method and corresponding metric to evaluate the recovery reconstruction of lost dynamic data. The metric emulates the convergence of error. We define the *convergence factor* as $(r_k/r_0)^{1/k}$ at iteration k, where r_0 is the initial residual [103]. The computational complexity is $\mathcal{O}(I)$ where I is the number of iterations the linear solver needs to converge. This real-time metric captures the convergence of the error better than the

residual for several reasons. First, the convergence factor has a direct correlation with the residual and an inherent correlation with the error due to the definition of r_k and err_k. The term r_k/r_0 captures the relative difference between the current residual to the initial residual. r_k/r_0 has the same trend of decreasing as r_k in Figure 3.2 because r_0 is a constant regardless of the difference in x_0. Second, the power of $1/k$ is able to reflect this small difference in r_k/r_0 as k increases. Therefore, the convergence factor converges smoothly regardless of the difference of the error trend from various matrices.

Figure 3.3 plots the convergence factor of LI and LSI for matrix $ex15$ and $nd24k$. We mark three points in the convergence factor. Point A indicates the beginning to converge. From point B, it changes very slowly and point C approximates that it has almost converged. When the convergence factor is about to converge at point C, the error also approaches to convergence with the local minimal value. For matrix $nd24k$, point D indicates both the global and local minimal value. While the error reaches the global minimal value when the convergence factor is near B and then reaches the global minimal value when the convergence factor is near C for matrix $ex15$. Any of the three points in the convergence factor might match better with the global minimal value of the error for different types of matrices. We can terminate the recovery at each of the three points, as the recovery is not required to be very accurate. These observations demonstrate that we can use the derivative of the convergence factor to approximate the convergence of error.

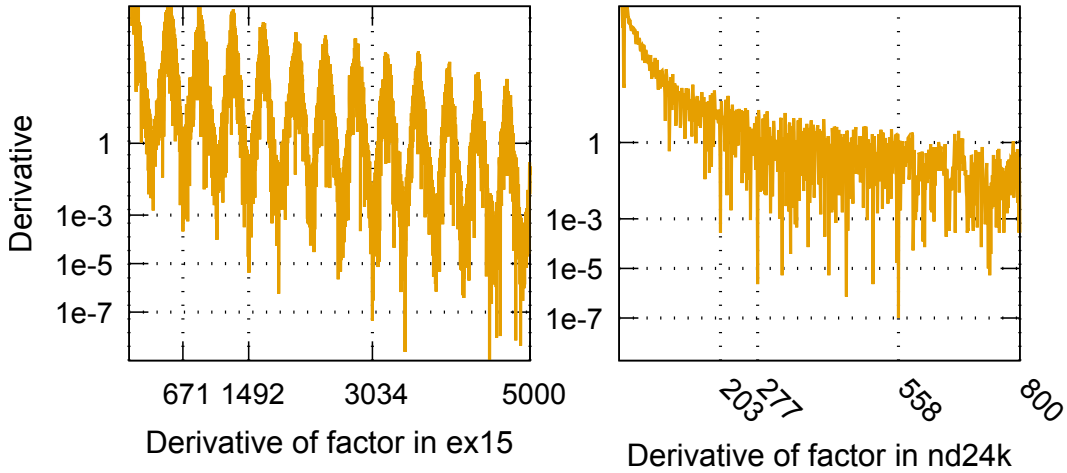


Figure 3.4: Average values of last c derivatives of convergence factors

We assume that the derivative of the convergence factor indicates the speed of convergence

of error, and a value approaching zero suggests a stabilized solution. Figure 3.4 shows the derivatives of the convergence factor in LI and LSI for matrices $ex15$ and $nd24k$. The values are averaged over the last c iterations of the derivatives of the convergence factor. We set c=5 by the experimental results, and different values of c show similar behaviors. The derivative generally decreases, i.e., the error to solution decreases, as the number of iterations increases. There is the oscillation in the derivative for matrix $ex15$ due to the inherent property of the derivative and the residual. However, its overall trend is decreasing for both matrices. Thus, we can set a threshold value and use it to select the lost data reconstruction. A low threshold selects a more accurate reconstruction, which takes more iterations in construction but less time in extra solving time.

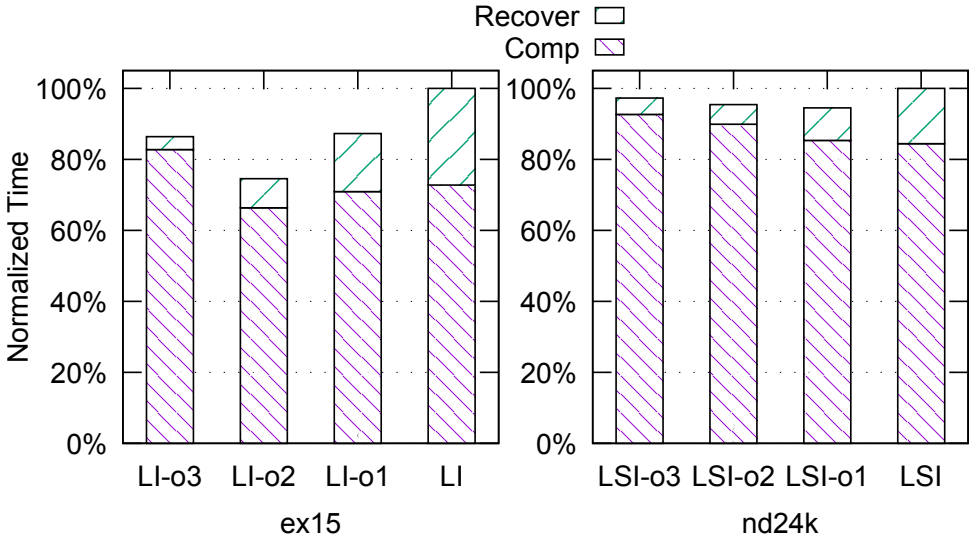


Figure 3.5: Normalized time breakdown of LI and LSI for $ex15$ and $nd24k$.

Matrix-Aware Reconstruction and Accuracy Selection. We propose to select the reconstruction and its accuracy by using the derivative of the converge factor. We search the number of iteration where the error has the global minimal value based on a smaller derivative. We set multiple different thresholds to estimate the points A, B and C in Figure 3.3, i.e., three thresholds $1e^{-7}$, $1e^{-5}$ and $1e^{-3}$ as three options *LI-o1* to *LI-o3* and *LSI-o1* to *LSI-o3* respectively, to evaluate different trade-offs in Eq 3.13. These three thresholds are obtained by experimental results of all matrices in our dataset.

Algorithm 3 shows our matrix-aware reconstruction based on CG algorithm [7]. Our method shares the same steps for CG loops and has extra steps for the convergence factor and derivative

Algorithm 3: Matrix-aware reconstruction (LI-MA, LSI-MA)

Require: LI-CG, LSI-CG, thresholds $th = \{th_1, th_2, th_3\}$;
Ensure: LI-MA = {LI-o1, LI-o2, LI-o3}, LSI-MA = {LSI-o1, LSI-o2, LSI-o3};
1: Initial guess x_0;
2: **for** $k = 1, 2, ...,$ until $der < th$ **do**
3: Same steps in CG loops
4: $cf_k = (r_k/r_0)^{1/k}$
5: $der_k = |cf_k - cf_{k-1}|$
6: $der = (der_k + der_{k-1} + ... + der_{k-c-1})/c$
7: **end for**

computation in Step 4 to 6. A lower threshold selects a more aggressive and accurate reconstruction. Thus, when changing from *LI-o1* to *LI-o3*, the construction time T_{const} decreases while the extra time T_{extra} increases. Figure 3.5 shows this trade-off for various matrices on 64 processors with one fault injected. For a smaller matrix $ex15$ with non-uniform nonzero distribution, the recovery time accounts for a larger portion of the total time. *LI-o2* performs best for $ex15$ and achieves the best trade-off. The reconstruction with low accuracy with *LI-o3* reduces the reconstruction time but incurs more extra solve time. For a larger matrix $nd24k$ with uniform nonzero distribution, the recovery time accounts for a small portion of the total time. All three thresholds lead to the reduced total time for LSI, and *LSI-o1* performs best.

Matrix $ex15$ obtains 25% maximum reduction for the total time, while matrix $nd24k$ obtains 6% maximum reduction. In the above cases, our optimization obtain 69% and 42% reduction in recovery cost for matrix $ex15$ and $nd24k$. Matrix $ex15$ and $nd24k$ have different convergence factors in both the main CG computation and recovery. The main CG computation solves the initial linear system $Ax = b$, while the recovery solves a linear system using the matrix A_{p_i,p_i} in LI or $A^T_{;,p_i}A_{;,p_i}$ in LSI. The convergence factors of the main CG computation tend to be stable. After a fault injection and recovery, the convergence factor has an immediate change in derivative. These derivatives are decreasing from *LI-o3* to LI because the recovery is more and more accurate. For matrix $ex15$, the convergence factor of the main CG computation shows a significant impact from the fault recovery only for *LI-o3*. The resilience and computation time obtains the best trade-off with *LI-o2* for $ex15$. While in matrix $nd24k$, the derivatives from *LI-o3* to LSI are decreasing smoothly, and the resilience and computation time obtains the best trade-off with *LI-o1*. Therefore, the trade-off between resilience and computation time needs to be considered, particularly for different matrices. We explore this with more experimental results in Section 3.3.3.

The recovery cost of LI and LSI is different for various matrices with our three thresholds. Figure 3.6 shows these behaviors for all matrices in our benchmark. Generally, all matrices can obtain a significant reduction in recovery cost with our three thresholds. Some larger matrices reach the maximum iterations (5000) we set for LI or LSI, while most smaller matrices require fewer than 600 iterations to recovery for LI. Our optimization obtains a larger reduction in recovery if the convergence factor takes longer to be stable, like matrix $t2dahe$. The reduction in recovery is smaller if the convergence factor is easier to be stable, like matrix $ex15$. The differences among matrices come from several features comprehensively, such as size, density, and nonzero distribution.

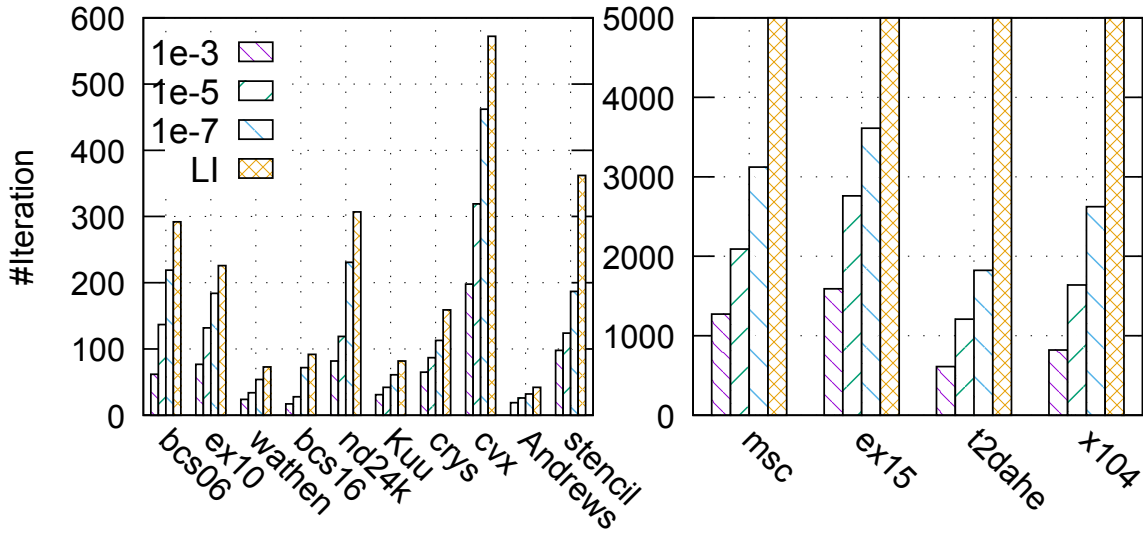


Figure 3.6: Iterations of LI with different thresholds for all matrices.

3.2.3 Power Reduction

Besides reducing the time-to-solution, using CG for the LI and LSI schemes provides power saving opportunities during the reconstruction phases. Since only p_i constructs the lost data of x_i, cores running other processes are able to transition to low speed states to reduce power consumption without impacting application performance.

In this work, we exploit DVFS commonly available on HPC CPUs for power reduction [79]. We bind processes to cores and adjust the core speed during the reconstruction phases for the LI and LSI schemes. Process-core binding is a common resource management technique, and typically a one-to-one mapping is adopted for HPC applications. The core with process p_i always runs at the highest CPU frequency, while the other cores scale down to the lowest CPU frequency before

reconstruction and scale up to the highest CPU frequency when reconstruction finishes.

Employing this power optimization techniques reduces power consumption during reconstructions by 40% with the power-aware LI scheme on a 24-core node (detailed results in Section 5.3). During reconstruction, 23 CPUs are idle, and the node power drops to 0.75× of normal power consumption during execution phases without DVFS scheduling, and 0.45× with DVFS scheduling. The power-aware LSI scheme achieves similar power savings.

3.3 Experimental Results

This section evaluates the resilience and energy efficiency of different recovery schemes, and answers the research questions raised in the introduction section. We first present our experimental setup and benchmarks. We then evaluate the resilience of recovery mechanisms, and lastly assess their time and energy costs.

3.3.1 Experiment Setup

The experiment platform consists of 8 dual-socket nodes. Each node has two 12-core Xeon(R) E5-2670v3 processors and 128 GB DDR4 DRAM evenly distributed between the two NUMA sockets. DVFS is controlled using the CPUfreq interface. Each core can independently transition from 1.2 GHz to 2.3 GHz with a step of 0.1 GHz. Each core supports 2-way hyper-threading, which is only enabled for resilience evaluation and disabled for power and energy related experiments. Execution time is collected from benchmark reports, and processor power is collected with the Intel Running Average Power Limit (RAPL) interface.

We focus on symmetric positive definite (SPD) matrices with various sizes, densities, and convergence speeds, as shown in Table 3.3 from the Suite Sparse Matrix Collection [4]. Each matrix is distributed among all parallel MPI processes in our experiment. CG and all resilience schemes are implemented from routines in RAPtor [21].

3.3.2 Resilience of Recovery Mechanisms

What is the resilience of various recovery mechanisms? To answer this question, we investigate how resilient each recovery mechanism is, how it performs for different problems, and how it reacts to multiple faults.

Name	#Rows	NNZ/row	Problem Kind	#Iters
bcsstk06	420	19	structural	4,476
msc01050	1,050	25	structural	35,765
ex10hs	2,548	22	CFD	3,217
bcsstk16	4,884	59	structural	553
ex15	6,867	17	CFD	1,074
Kuu	7,102	24	structural	849
t2dah_e	11,445	15	model reduction	82,098
crystm02	13,965	23	materials	1,154
wathen100	30,401	16	random 2D/3D	355
cvxbqp1	50,000	7	optimization	11,863
Andrews	60,000	13	graphics	216
nd24k	72,000	399	2D/3D	10,019
x104	108,384	80	structure	96,704
5-point stencil	640,000	5	structure	3162

Table 3.3: Properties for matrices taken from Suite Sparse Matrix Collection.

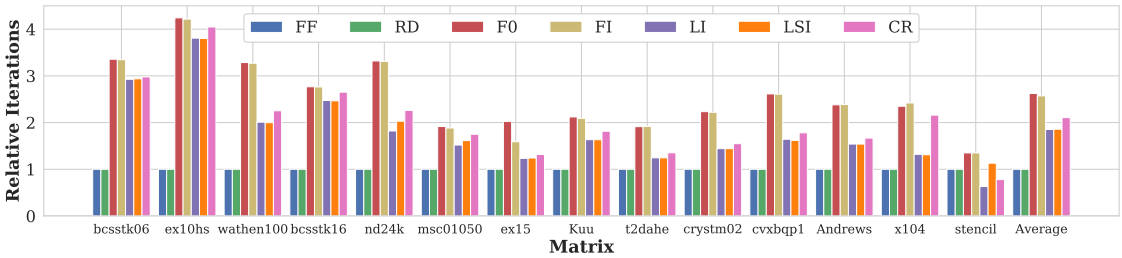


Figure 3.7: Iterations to converge for different matrices using 256 processes with 10 faults. Each matrix uses its own normalization base, which is the fault free case.

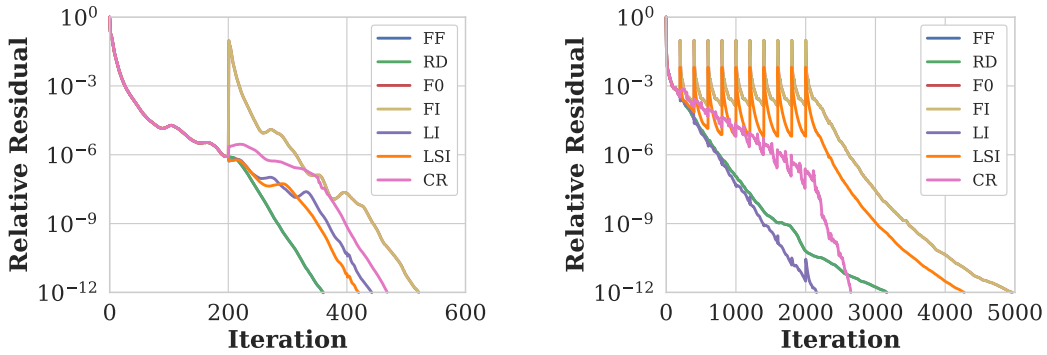
The work evaluates recovery schemes for CG, but our results are applicable to other iterative solvers. CG iteratively refines an initial guess at each iteration. The algorithm terminates when the iterative solution is deemed accurate enough based on a small relative residual or when a fixed number of iterations is reached. In the presence of faults, the number of iterations to reach the same accuracy can increase. A recovery mechanism that takes fewer iterations to reach a desired accuracy is more resilient.

Note the resilience analyses in this subsection only accounts for iterations. Section 3.3.4 extends this discussion to cover time, energy, and power. In the following experiments, 10 faults are inserted evenly over the iterations required by the fault-free execution (no more faults are inserted after the fault-free execution converges). The solver tolerance is set at $1e{-}12$. Since the number of iterations is the same regardless of where the checkpoint is stored, we do not delineate between memory and disk checkpointing in this subsection. Instead, we present results of disk checkpointing with a frequency of every 100 iterations.

Mechanisms vs. Problems. We examine how the recovery mechanisms perform on different matrices. Figure 3.7 presents the number of iterations normalized to the fault-free performance. Overall, F0 and FI take the highest number of iterations (2.5× on average) to converge. RD takes the lowest number of iterations. LI, LSI, and CR perform similar to F0 and FI for matrices such as bcsstk06 and ex10hs, but perform much better for other matrices such as ex15 and t2dah_e. This is due to the fact that LI and LSI construct less accurate solutions for the matrices with an irregular structure. In this experiment CR checkpoints in low frequency. CR requires more iterations than LI and LSI because it rolls back to a prior iteration state. LI and LSI do not require as many iterations because of more accurate reconstruction of x.

Number of Iterations to Converging and Correction. Recovery mechanisms takes a number of extra iterations due to faults/failures. Figure 3.8 shows the variation in the residual history when solving two different linear systems with various numbers of faults and recovery schemes.

With a single fault injected at the 200th iteration, Figure 3.8(a), the residual increases for all recovery schemes except for RD, which overlaps with the FF case. This is due to the fact that RD recovers the exact solution. Different recovery schemes result in a different change in the residual. F0 and FI (overlapped) have the largest increase, while LI and LSI (overlapped) get a minimal increase by constructing a more accurate approximation. Note that CR has a noticeable increase by rolling back to a previously checkpointed result. Figure 3.8(b) shows an example with 10 faults for a



(a) One fault and 8 processes for matrix wathen100.

(b) Ten faults and 64 processes for 5-point stencil matrix.

Figure 3.8: *residual∼#iteration* relation and correction under various recovery mechanisms. FF and RD are overlapped. F0 and FI are overlapped.

5-point stencil matrix. LI and CR take fewer iterations to converge. In CG, reconstructing x forces reconstruction of others renew other variables in each iteration, including CR. In this example, their constructed solution makes the path to converge shorter.

3.3.3 Optimization for Forward Recovery

This subsection presents how resilience overhead and the main CG computation interplay for our optimization for forward recovery. As we discussed in Section 3.2.2, the trade-off between the accuracy of recovery and extra main computation are determined by the convergence factors of both the initial matrix (A) and the matrix in LI (A_{p_i,p_i}) or LSI ($A^T_{:,p_i}A_{:,p_i}$) solving. Our three options of accuracy selection terminate the recovery at different levels to control the trade-offs for various matrices.

Impacts of our optimization. We examine the trade-off between the accuracy of recovery and extra major computation by the detailing time and energy in CG for different matrices under various fault situations. Figure 3.9 shows the normalized time and energy of the computation and recovery kernels in CG for one large matrix *cvx* in different fault rates on 512 processors.

cvx is a large matrix with a low convergence speed. The recovery time accounts for a small portion of the total time, even though it takes thousands of iterations of LI to recover from one fault. Among our three options for LI, *LI-o2* performs best with reduced recovery time by 68.9% and energy by 67.8% compared to LI without optimization in low fault rate. The total reduction

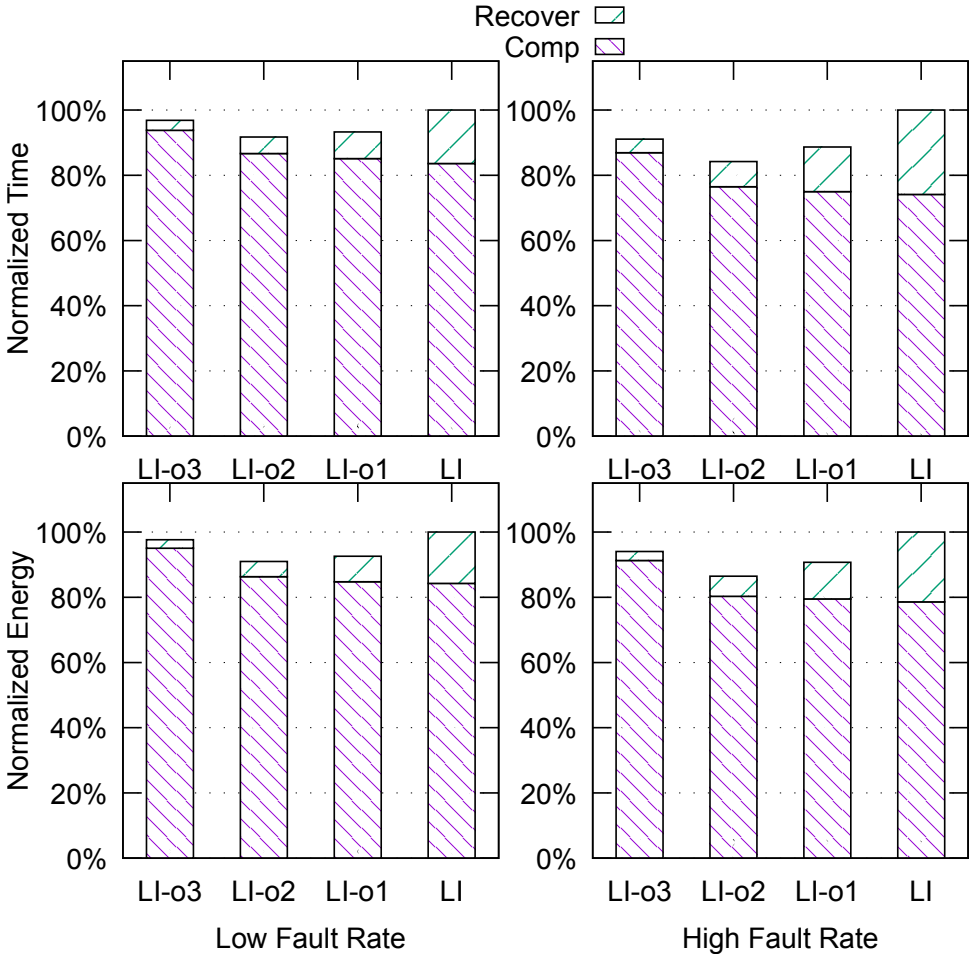


Figure 3.9: Normalized time and energy breakdown of LI for matrix *cvx* in different fault rates on 512 processors.

for time and energy are 9.0% and 8.5%. As the fault rate increases, recovery time accounts for an increased portion of the total time. However, it is still a small portion due to the low convergence speed of matrix *cvx*. In a high fault rate, *LI-o2* performs best with reduced total time by 15.8% and energy by 13.5%. The percentages of reduction for recovery is similar to those in a low fault rate because the reduction for each fault is similar with our matrix-aware optimization. $O2$ performs best because it achieves a better trade-off in Eq 3.13 for matrix *cvx*. The impact of time and energy for LSI is similar to that in LI for our matrix-aware optimizations.

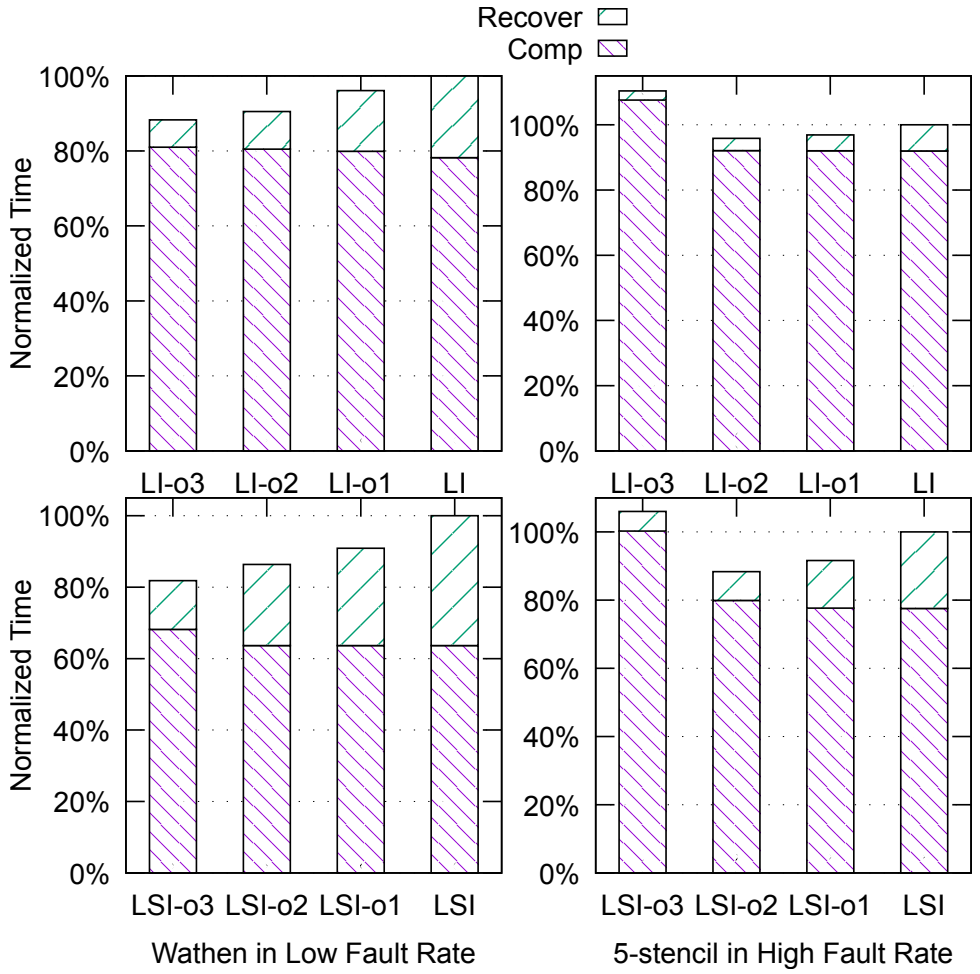


Figure 3.10: Normalized time breakdown of LI and LSI for matrix *wathen* and 5-points stencil in different fault rates.

We focus on time for other matrices, as time and energy show a similar behavior. Figure 3.10 shows the normalized time of major computation and recovery kernels in CG for a small matrix *wathen*100 in low fault rate and the 5-point stencil matrix with 640,000 rows in high fault rate on

512 processors. $Wathen100$ is a small matrix with a high convergence speed. It takes hundreds of iterations to converge in main computation of CG and tens of iterations for LI or LSI to recover from a fault. Therefore, the recovery time accounts for a large portion of the total time. For *wathen*, $o3$ performs best with reduced recovery time by 66.4% in LI and 62.7% in LSI. We obtain reduced total time for all three optimization methods and maximum by 10.6% in LI and by 18.2% in LSI with $o3$ for *wathen*. This is because $wathen100$ is less sensitive from fault and can tolerate a coarse approximated reconstruction. For the 5-point stencil matrix, $o2$ performs best with reduced recovery time by 53.1% in LI and 62.5% in LSI. We obtain reduced total time by 6.2% in LI and by 11.7% in LSI with $o2$. This is because the 5-point stencil matrix with 640,000 rows can tolerate the accuracy of LI until the reconstruction in $o2$. When the accuracy is less as $o3$, the main CG computation has a significant increase.

Guideline for choosing strategies. The results show that the trade-off between T_{solve} and T_{res} in Equation 3.3 is significantly different for various matrices. For large matrices with non-uniform nonzero distribution like *cvx*, a small reduction in T_{res} can result in a large increase in T_{solve}. While for small matrices with uniform nonzero distribution like *wathen*, reducing T_{res} has little impact on T_{solve}.

Figure 3.9 and 3.10 show different behaviors of time and energy among various matrices. When changing from *o1* to *o3*, the construction time T_{const} decreases while the extra time T_{extra} increases. There is no one option in our matrix-aware optimization that always performs best because this trade-off depends on convergence factors in both main CG computation and recovery. It is comprehensively impacted by the size, density and nonzero distribution of the matrix. For example, $o1$ works better for large matrices with the non-uniform nonzero distribution that require accurate recovery, while $o3$ works better for small matrices with uniform nonzero distribution that do not require very accurate recovery. For some matrices, $o2$ achieves a better trade-off. We recommend $o1$ if the user does not have such information of the matrix because it always performs better than LI or LSI without optimization.

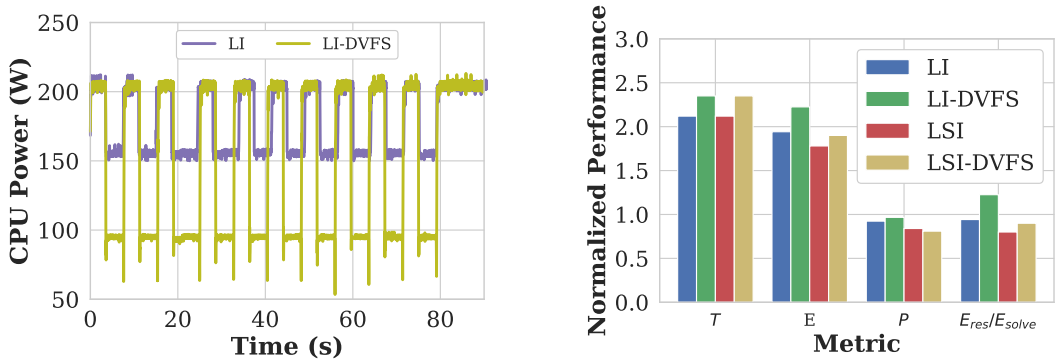
3.3.4 Power Optimization

The previous analysis only captures extra iterations required by resilience. Iterations do not tell the entire time cost. In this subsection, we analyze the time, power, and energy costs of resilience, and begin with power consumption. From this subsection, MTBF is set as the same of

that in Section 3.3.2. The checkpointing frequency of CR is computed via Young's formula [132].

What is the power requirement of resilience and how does power management help? Here we focus on the LI and LSI mechanisms and how they benefit from power management. We limit our discussion on power management for checkpointing as it has been previously investigated [86].

Figure 3.11(a) illustrates how DVFS-based power management changes the power profiles of the matrix $nd24k$ on a single node with the LI scheme. We compare our optimization denoted LI-DVFS with the OS-level power management. The OS-level management uses the "ondemand" governor and scales up CPU speed if the CPU utilization is high or scales the frequency down if low. LI-DVFS uses the "userspace" governor. It runs all CPUs at 2.3 GHz in the computation phase, and runs all but one CPU at 1.2 GHz during the construction phase. The one CPU that actively reconstructs an estimation of lost data runs at 2.3 GHz. LI-DVFS reduces power by 39% during the construction phase without performance degradation. While not shown, LSI-DVFS achieves similar power reduction.



(a) Power reduction for nd24k

(b) Energy savings for 14 matrices

Figure 3.11: Power reduction and energy savings with LI-DVFS and LSI-DVFS. (a):Power profile of $nd24k$ with simple LI and LI-DVFS; (b) average time, power, and energy for 14 matrices included in Figure 3.7. T, E, and P are normalized based on the fault-free case. E_{res} / E_{solve} is the ratio of energy cost for resilience and for fault-free case.

Figure 3.11(b) presents the overall performance, power, and energy impact for the 14 matrices presented in Figure 3.7. LI-DVFS and LSI-DVFS maintains the same performance, and reduce energy by 11% and 16% respectively. With these optimizations, more energy is allocated to problem-solving rather than resilience, as demonstrated by E_{res} / E_{solve}.

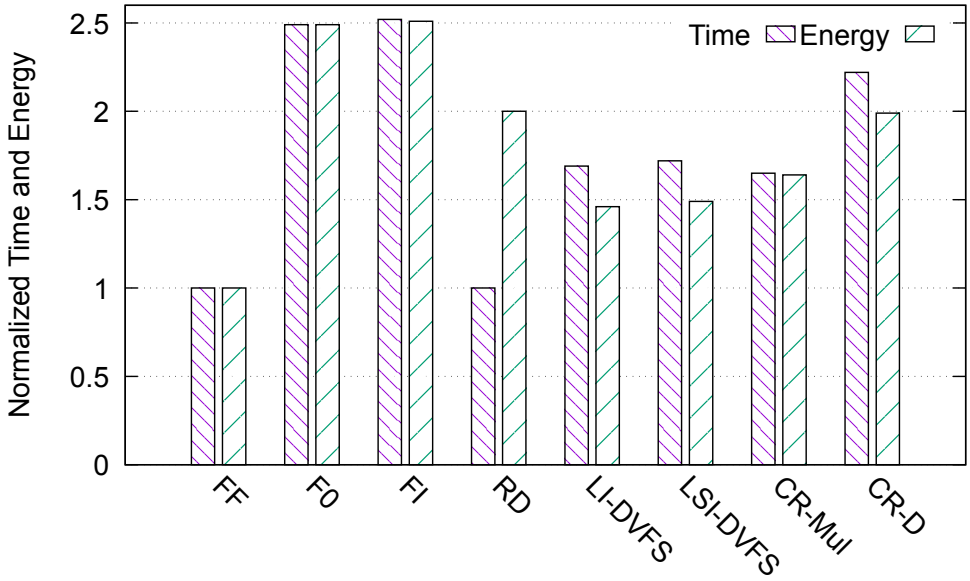


Figure 3.12: Normalized Time and Energy for all matrices, with ten faults on 512 processors.

What are the time and energy costs of resilience? Since LI-DVFS and LSI-DVFS consume less energy than LI and LSI, we only include the former in discussions henceforth. We implement both multi-level CR (CR-Mul) and CR with disk (CR-D) to give a range of checkpointing/restart cost. We apply various recovery mechanisms to the benchmark matrices under study, and analyze the time, energy, and power cost of resilience.

Figure 3.12 presents the normalized time and energy costs of resilience for various schemes. The values are averaged over all the matrices under study. Overall, LI-DVFS incurs the least energy overhead, and CR-Mul incurs the least time overhead except for RD. In contrast, FI takes the most time and energy. We assume that the disk is shared between multiple users and consumes a constant amount of power regardless of configuration.

3.3.5 Scalability of Recovery Mechanisms

How does the resilience cost scale with system size and a decreasing MTBF? To answer this question, we need to use the experimental data and project T_{res} from our experimental platform to a very large system. We implement multi-level checkpointing (CR-Mul) as every ten checkpoints from memory with one checkpoint from disk. Figure 3.13 plots the performance scalability for LI-opt and LSI-opt compared with CR-Mul and CR-D. We assume a constant per-processor

#Nodes	T_{FF}	$t_C disk$	$t_C mem$	t_{const}	t_{extra}
32	9.78	0.24	0.00011	0.04	0.22
128	11.98	0.44	0.00012	0.09	0.29
512	14.72	0.86	0.00010	0.17	0.24
2K	17.95	1.72	0.00011	0.34	0.25
8K	21.67	3.44	0.00011	0.68	0.25
32K	25.83	6.88	0.00011	1.36	0.25
128K	30.61	13.76	0.00011	2.72	0.25

Table 3.4: Time cost of weak scaling in seconds.

MTBF of 6K hours; thus, the application's MTBF decreases as it uses more processors.

Figure 3.13(a) shows strong scalability for one large matrix $x104$ from 32 to 512 processors. We show the best version of our optimization for LI-opt and LSI-opt. All time performance is normalized based on the fault-free case on 32 processors. Generally, all four resilience methods show good, strong scalability. As the number of processors increases, the time overhead for our optimized strategies decreased significantly because the matrix size in LI-opt and LSI-opt solutions halves. CR-D shows the worst performance because the resilience overhead of writing and reading a checkpoint file in NFS by each processor core is heavy as the system size increases.

To evaluate how the resilience cost scale with a larger system size, we compute overhead for a scaled workload via the models from Section 3.1. First, we project T_{FF} for a fault-free baseline, where $T_{FF} = T_{solve} + T_O$. In our measured data, parallel overhead T_O roughly equals the communication overhead. In each CG iteration, communication incurs to transfer data for sparse matrix-vector multiplications (SpMV) and vector-inner products. We use the average communication time cost of a SpMV from experimental data from a large system [20], where the SpMV's weak scaling performance is studied for matrices with 50K nnz per processor ranging from 2K to 128K processes. The time cost of a vector-inner product is linear with system size [130]. We project T_O with the average communication time cost of SpMV and vector-inner product. We run the workload for 10K iterations at each system size from 32 to 512 in the fault-free situation and set the final residual as the target tolerance for other faulty cases. T_{FF} is prjected as Table 3.4 shows.

We project resilience overheads for the best case of FW, CR-Mul, and CR-D from our experimental data to a larger system. We project T_{res} to the large system based on our models in Section 3.1. $t_C disk$ of CR-D increases linearly as system size increases in our experimental data. We assume it continues to increase linearly in the large system. The average $t_C mul$ of CR-Mul is $t_C mem + 0.1 t_C disk$, where $t_C mem$ is stable, and we assume this continues in the large system. t_{const}

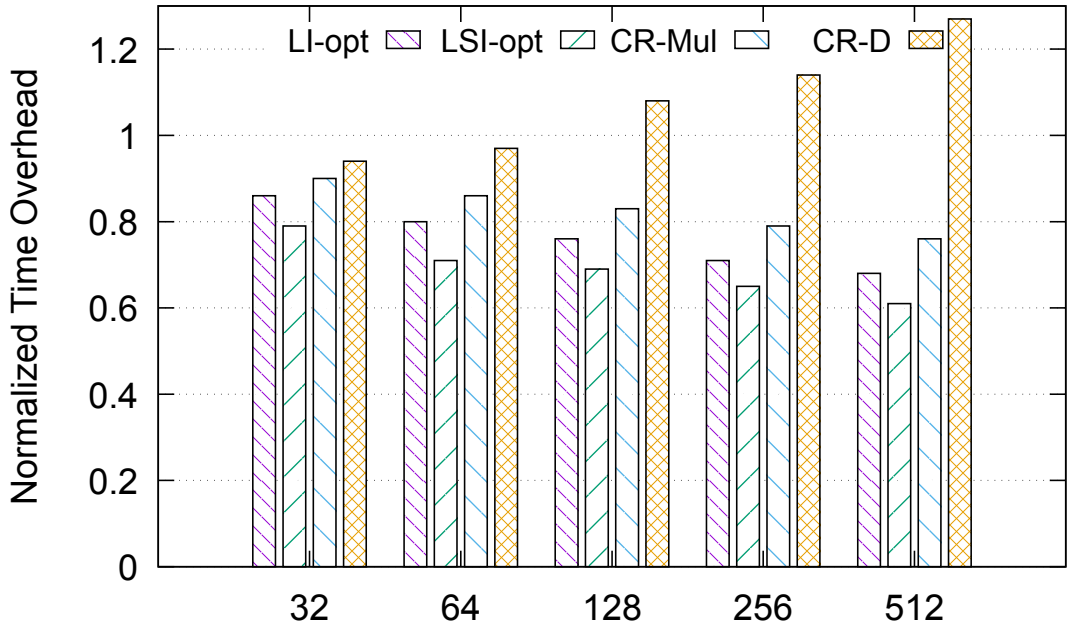
of FW increases linearly as system size increases in our experimental data. We assume that this trend continues in large systems. For t_{extra} of FW, we adopt an average normalized overhead based on the fault-free case. We adopt this average data to project FW in the large system. Table 3.4 shows the above parameters used in weak scalability projection.

Figure 3.13(b) presents weak scalability of normalized resilience time (T_{res}) for 5-point stencil matrices with 50K nnz per processor under the situation we described above. We present experimental data from 32 to 512 processors and project T_{res} from 2K to 128K processors. T_{res} is normalized to the fault-free case for each system size. As system size increases and MTBF decreases, T_{res} of FW increases roughly linearly because t_{const} is linear and t_{lost} per fault is fixed. T_{res} of CR-D increases faster because of t_C and more frequent checkpointing. T_{res} of CR-Mul and CR-D increases faster because of $t_C disk$ and more frequent checkpointing. This projection emphasizes the importance of developments of efficient resilience mechanisms, as resilience overhead keeps increasing on a larger system. This analysis also indicates that CR-Mul can reduce the effect of significant failure rates compared to CR-D, while our optimized forward recovery mechanism shows better efficiency and scalability in CG.

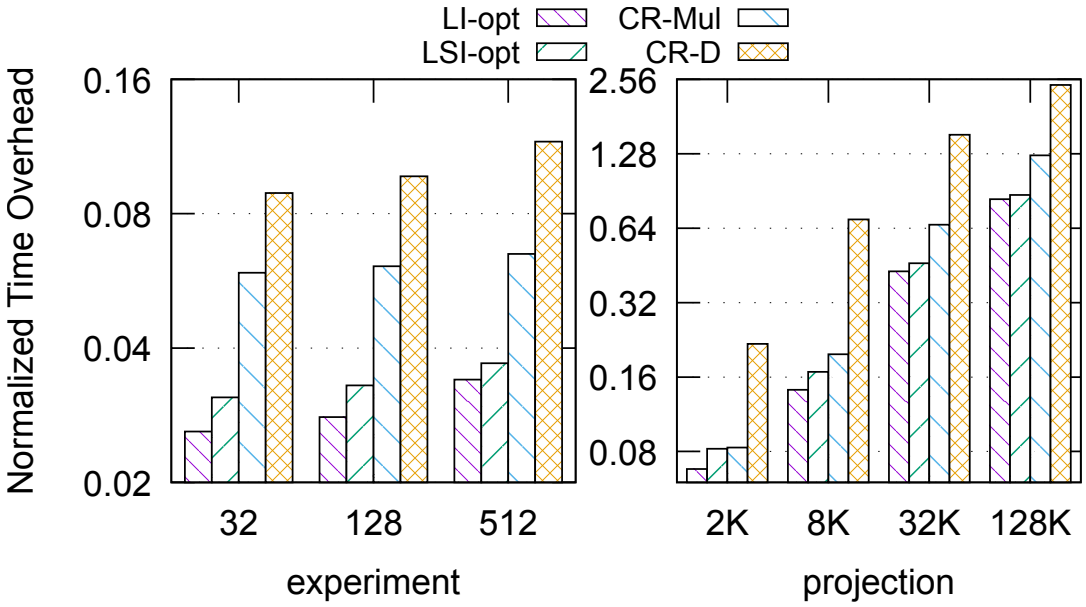
3.4 Summary

This chapter proposes a novel approach to analyze and optimize the cost of resilience techniques for iterative linear solvers. We present a set of models to better understand the resilience and energy overhead of applications in a faulty environment, and we perform power optimizations to reduce the overhead of forward recovery. Our experiments show that our optimized forward-recovery algorithm significantly reduces the resilience overhead and provides insights for selecting recovery schemes for certain workloads. Our projection result reveals trends of resilience cost on large systems and provides direction for optimization of resilience schemes. In future work, we plan to extend our models to capture more resilience mechanisms and study the performance and energy optimization for more applications. Overall, our major findings and contributions include:

- We present a generalized analytical model to quantitatively co-study performance, scalability, resilience and energy efficiency for common resilience technologies to enable resilient and energy-efficient large-scale scientific computing.



(a) Strong scalability for $x104$



(b) Weak scalability for 5-points stencil

Figure 3.13: Scalability of resilience mechanisms

- Our proposed analytical models capture the first order time and energy cost factors for various fault recovery schemes and are customized to fit specific ones. They are used to identify the best recovery schemes for given fault situations.
- Our optimization techniques reduce the time and energy overhead of recovery schemes by 16% for parallel iterative algorithms. We investigate matrix-aware optimization for forward recovery and proposes the convergence factor metric, which makes it practical to determine the optimal lost data reconstruction for high-order matrices. This matrix-aware optimization further improves energy efficiency by 13.5% on large-scale systems.
- We quantitatively reveal that each resilience scheme has its own advantages depending on the fault rate, system size, and power budget, and the forward recovery can further benefit from matrix-aware optimizations for large-scale computing.

Chapter 4

Relaxed Replication for Energy Efficient and Resilient GPU Computing

Power and reliability are two intertwined challenges in GPU-accelerated large-scale computing. Managing power and resilience are challenging, due to the heterogeneous compute capability, power consumption, and varying failure rates between CPUs and GPUs. Previous works have shown that redundancy-based approaches are more energy-efficient than checkpointing/restart at extreme-scales, but current solutions only support parallel programs running on CPU-based homogeneous systems. Simply extending redundancy approaches from CPU-based systems results in suboptimal performance and/or energy efficiency because existing redundancy solutions typically rely on identical replicas with expensive synchronization.

In this chapter, we explore redundancy techniques and energy-efficient techniques for GPU-accelerated systems running MPI parallel workloads. Specifically, we design a novel redundancy technique that relaxes the requirement of synchronization and identicalness for replica processes and allows them to run in lower-precision and at lower power/performance states with periodical rejuvenation or asynchronization, enabling resources and power reduction. This relaxed replication mechanism complicates fault detection and recovery over the homogeneous exact replication. We discuss techniques to handle and mitigate these complexities for both process/node failures and

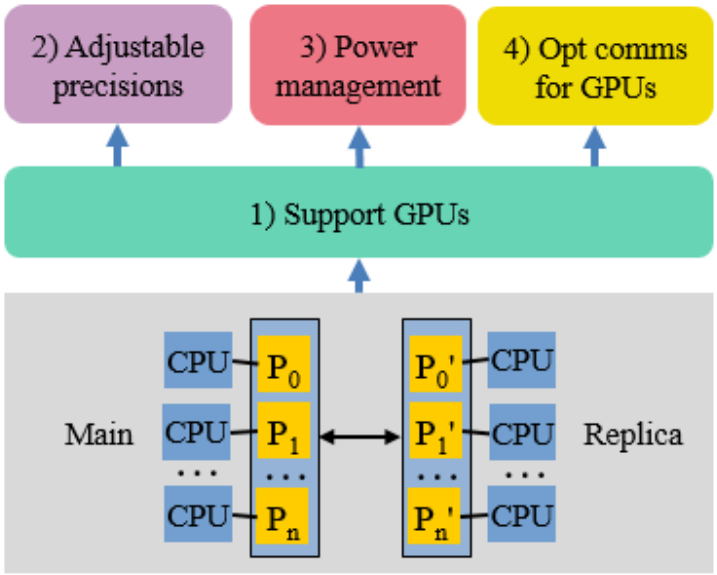


Figure 4.1: Design of REESE. A energy-efficient redundancy-based resilience framework for GPU computing.

silent data corruption. Evaluation results on a 16-GPU cluster show our techniques reduce energy by up to 15% for unmodified programs and 32% for programs that are able to adapt the precision of the replicas.

4.1 Framework Design

We design a Resilient and Energy-Efficient ScalE (REESE) computing framework to provide MPI-level redundancy for GPU applications as Figure 1. REESE extends the redundancy solutions for homogeneous systems to support GPU-accelerated systems and programs. Redundancy is implemented through MPI libraries and runtime by replicating processes and communications [56]. REESE provides resilience to various fault scenarios.

REESE supports synchronous and asynchronous execution of main and replica processes. The asynchronization mechanism enables flexible hardware resource allocation for main and replica processes, and higher performance for main work progress.

REESE supports adjustable precision and power management to reduce overhead of replications. It further utilizes acceleration technologies like CUDA-aware MPI to provide high-bandwidth and low-latency communications with NVIDIA GPUs [75].

This REESE framework has multiple features:

1. provide redundancy-based resilience for GPU-accelerated systems. It allows main and replica processes to run on either GPUs or CPUs for higher performance or other performance and power objectives.

2. support double precision and its mix with single or half precision on CPUs and GPUs to provide controllable performance and power consumption. Supporting of adjustable precision significantly reduces time and energy overhead of replication.

3. support DVFS on CPUs and GPUs to achieve flexible power management. Upon faults, main and replica processes dynamically adjust their execution rates. It also enables main and replica processes to take over the role of each other alternatively after faults.

4. reduce resilience overhead in GPU-accelerated systems by utilizing optimized MPI communications between GPUs. It results in significant time overhead reduction in MPI-level redundancy on NVIDIA GPUs.

4.1.1 Fault Scenarios

Applications in GPU-accelerated systems could be impacted by either soft errors (such as bit-flips and SDC) causing an erroneous deviation but without an interruption, or hard faults (such as processor or node failures) causing the application to crash. Applications should be protected against these fault scenarios. We focus on recovery from hard and soft faults in hardware (CPU/GPU) and assume the faults can be detected.

We assume that data in an application are classified into static data and dynamic data based on their types of storage and variation. For example, CG iteratively solves linear equations in the form of $Ax = b$, where entries of the matrix A and the column vector b are constant values, and entries of the column vector x iteratively changes in the computation. When a soft or hard fault occurs in one CPU or GPU, data in its memory is erroneous or lost. A and b are static as they don't change and can be restored from persistent storage, and x is dynamic and needs to be recovered via redundancy, checkpointing, or other schemes [87]. We focus on recovering dynamic data in application progress.

Considering different overhead of resilience between soft and hard errors, our framework provides two distinct resilience mechanisms to address them.

Fault recovery for soft errors. When a soft fault occurs in the memory of one CPU or GPU running one main MPI process, the data of this faulty process is erroneous. This soft fault does not immediately cause interruption or crash of the application. Thus, our framework recovers the faulty main process by receiving dynamic data from the corresponding replica process. Another scenario of soft faults is that an SDC has propagated to impact more than one main process. In this case, REESE recovers all faulty main processes by receiving dynamic data from all replica processes. Upon fault detection, replica processes may increase their speed and take over the role of main processes. Simultaneously, the old main processes decrease their speed and power consumption and transit to the role of replicas, as shown in Figure 2.

Redundancy for hard errors. In the hard error scenario, we assume the long-running application is interrupted or crashed by a CPU/GPU error or a node failure in a GPU-accelerated system. In these cases, requesting new resource and recovering from the crashed failure might be infeasible due to resource limitations or incur more time and energy overhead. Under this assumption, utilizing the remaining healthy resource to continue application execution is a better choice [56, 89].

Hard errors at different degrees have various impacts, and we handle them differently. If a hard fault occurs in a CPU core binding one MPI process, the core is down and data in its memory is lost. In this situation, we replace the faulty main process with the corresponding replicated process. If a hard fault occurs in a node and the node is down, we use the whole replica set to continue the execution. Upon the detection of a crashed failure in one or more main processes, our framework increases the execution rate of replicated processes to catch up with the lost progress.

4.1.2 MPI process-level redundancy for GPU

This base of REESE is transparent MPI process-level redundancy for GPU applications. This subsection introduces the design of synchronization and asynchronization of MPI process-level redundancy.

Synchronization of main and replica. Previous work like RedMPI [56] compares received messages or hashes from main and replicated processes to detect if SDC occurs in a process's communication data. This message verification mechanism requires concurrency of main and replicated MPI processes. We keep the same feature.

Asynchronization of main and replica. For the purpose of energy-efficiency, we might run replica processes on low-power devices with lower performance than main processes running on higher-power devices. Synchronization would hurt the performance of main processes. This.

To enable higher performance and energy-efficiency for applications in heterogeneous systems, this framework designs asynchronization of main and replicated processes by using two distinct MPI communicator groups. This approach excludes MPI communication between main and replicated processes for message verification. Thus, this new redundancy mechanism focuses on crashed failures detection and recovery instead of SDC.

To optimize MPI process-level redundancy in GPU-accelerated systems, REESE applies advanced techniques to GPU communication cost. This resilience overhead mainly comes from the increased amount of MPI communication due to replicated processes. The common practice of running MPI applications on multi-GPUs is to assign one MPI process to one CPU core and offload kernels to one GPU core, where CPUs serve as communication and service processors [18]. As a result, data in the source GPU's memory are first copied to the host memory, and then sent across the network to the destination host, and finally to the destination GPU memory. This data movement incurs significant cost due to additional memory copy between GPUs and CPUs. Our framework utilizes CUDA-aware MPI [75] to enable low-latency communications between NVIDIA GPUs by eliminating unnecessary memory copies. It significantly improves performance for MPI process-level redundancy with NVIDIA GPUs.

4.2 Relaxed Replication for GPU Computing

In this section, we discuss how we relax requirements of replication based resilience for GPU computing while still handling fail-stop failures and silent data corruption.

4.2.1 Relaxed Replication for MPI Process-Level Redundancy

We focus on MPI process-level redundancy, where we use the profiling interface (PMPI) to the MPI runtime to replicate and handle MPI processes, their computation, and communication [56]. When an MPI program starts running, we transparently partition its MPI processes into two sets, similar to prior work [56]. We denote the first set of MPI processes that run at user-specified settings as *main*, and the other set as *replica* or *shadow* that may run in a modified way to lower the power

consumption overhead. Furthermore, these sets are not static. As failures occur in the system leading to the loss of a main or a replica process, until its restoration, the corresponding process in the other set becomes the sole owner of that data and is associated with both sets in communication and ensures both sets have the correct number of processes.

We enable MPI-level redundancy on GPU-accelerated systems and programs as shown in Figure 4.1. To optimize MPI communication between GPUs, we leverage CUDA-aware MPI for NVIDIA GPUs [75]. Without CUDA-aware MPI, communicated data is first copied from GPU memory to the host, and then sent across the network to the destination host, and lastly copied to the destination's GPU memory. Such data movement incurs significant cost due to multiple memory copies between GPUs and CPUs [18]. By using CUDA-aware MPI [75] we eliminate unnecessary memory copies and significantly improve performance for MPI process-level redundancy with NVIDIA GPUs.

Our redundancy scheme has a key feature: replicas can run in different configurations from the main processes yet still be effective at detecting and recovering failures. We relax the requirement of exact replication and lock-step execution to achieve this. Thus, we run replicas with lower-clock frequencies or on computational elements with different capabilities. For convergent applications, we explore relaxed-precision of replica processes to further lower power consumption.

One challenge we face in our design is that the replica processes likely lag due to lower computational speed than the main processes. Should fine-grain synchronization be used like in [56], this relaxed replication results in main processes waiting at each communication operation for the slower replicas, hurting performance. To account for this execution drift and mitigate performance issues, we further relax the level of synchronization between the main and replica processes from every message to a prescribed period (detailed below) and rely on message buffering to compare corresponding messages asynchronously using a helper thread. At the synchronization point, we *rejuvenate* the state of the replica processes to be consistent with the main processes. Furthermore, implementing loose synchronization requires special care to handle failures, as discussed in Section 4.2.3. In the remainder of this subsection, we focus on the design of different synchronization modes between main and replica processes.

4.2.1.1 Synchronous Execution

Our scheme supports the common execution mode of synchronization at every message; however, this forces the corresponding processes to be at the same communication step before execution continues. In our design, we create two communicators; one for the main processes and one for the replicas. Within each communicator, the program executes as normal with processes sending messages to another in their communicator. In order to enable synchronization, we require extra communication between the main and replica processes. Figure 4.2(a) shows that for every message sent from a main to a main, there is a corresponding replica message sent from a replica to a replica in the replica set. Furthermore, each main process sends the same message to the corresponding replica receiver, and each replica process sends the same message to the corresponding receiver process in the main set (not shown in Figure 4.2(a)). Instead of the full message to the receiver in the other set, sending a hashed version dramatically decreases the communication volume, enabling better scalability and performance [56].

Because we do not require the execution rate of the main and replica processes to be equivalent, discrepancies in message arrival times between main and replica senders are possible. To account for this, we use buffers to cache messages enabling the senders to proceed without blocking and use threads to compare asynchronously with the simulation. Furthermore, if the main and replica processes compute using different precision the comparison between the two message buffers must rely on a metric — e.g., absolute error, relative error, mean-squared error – to compute the acceptability of each element and/or the full message. In this relaxed synchronization case, we exchange the use of a hash function for a checksum function to enable comparison based on a tolerance.

To control the progress gap, avoid the buffer overflowing, and reduce the cost of fault recovery, our scheme periodically *rejuvenates* the state of replica processes based on the current state of the main processes. Rejuvenation requires knowledge of key data structures for replacement. In our prototype, we require the registration of this key data using an API that records a starting address and a size[1]. This API is similar to those in widely used in application based checkpointing [18]. The period of the rejuvenation is configurable and tied to the application's execution — e.g., after a fixed number of iterations, a fixed number of communications, or a certain MPI communication.

[1]Future work will explore intercepting memory allocation calls to record metadata on the data structures similar to [29].

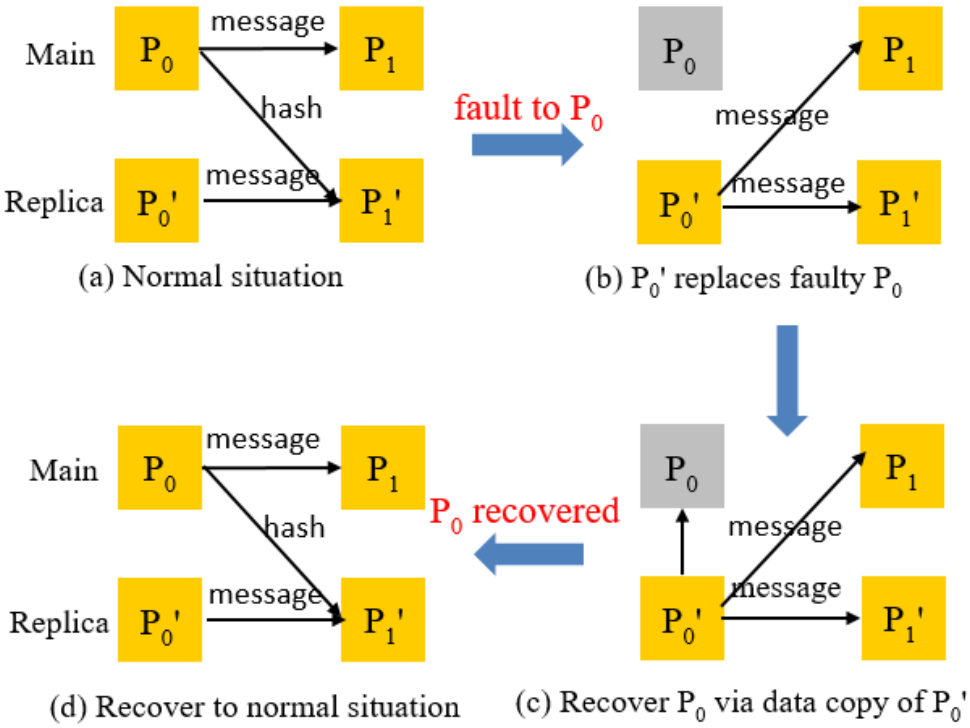


Figure 4.2: MPI process-level redundancy, the synchronous execution, and soft error recovery. (a) MPI communication between main and replica in normal situation (for simplicity, the message from $P_0{}'$ to P_1 not shown); (b) Upon the fault detection of the main process P_0, the corresponding replica $P_0{}'$ does the double duty to communicate in both main and shadow sets; (c) P_0 is recovered by data copy of replica $P_0{}'$. (d) Recovery back to normal situation as (a). Synchronous execution is similar, but without communication across main and replica in (a) and (d).

The rejuvenation is materialized through message passing — e.g., each main process sends to its corresponding replica all the required data structures; upon the receipt of the messages, the replica replaces its local values with the newly received ones.

4.2.1.2 Asynchronous Execution

In addition to synchronous execution, our redundancy scheme allows replicas to be fully out of synchronization with the main processes. As the computational speed of the replica processes decreases to save energy, the replica processes' execution begins to lag behind the main processes' execution. If synchronizing, the main processes must wait for replica processes, leading to larger overheads and increased time-to-solution. The asynchronous execution mode eliminates the needs of synchronizing communication between the main and replica processes. Only when a failure occurs or a rejuvenation occurs do the main and replica processes communicate. To establish communication,

all processes post a `MPI_Irecv` when initializing MPI to receive the fault signals. If the communication completes, then there was an error detected in the system. With the enlarged progress gap, recovering a faulty main process with its replica is more complex and may suffer a loss in computational progress.

4.2.2 Improving Energy Efficiency

4.2.2.1 Energy Efficient Techniques

Our relaxed replication scheme allows replica processes' configurations to be different from main processes. This enables us to exploit multiple techniques to reduce the resource and energy usage for replica processes. Here we introduce several techniques to apply individually or in combination to improve energy efficiency.

Adjustable Precision for Replicas. Prior work shows that mixed precision improves the performance and energy efficiency of dense and sparse linear algebra algorithms [27] *and* maintains the double precision accuracy in the resulting solution. Another prior study [13] demonstrates that computation in lower precision reduces energy consumption. For iteratively convergent applications where lower precision does not have a strong impact on accuracy, our replication scheme supports main and replica processes using different precision. For example, the main processes compute in mixed double and single or half precision, and the replica processes compute in single or half precision. Therefore, we save energy while maintaining accuracy and progress.

Presently, for a proof-of-concept, we manually configure codes to use alternative precision if the code does not natively support it. In the future, we plan to build automated support of multiple precision. Specifically, we rely on an LLVM compiler pass to generate the alternative precision version of the application's routines, similar to [29] in which an application's source code is duplicated and interleaved for redundant lock-step execution. The user can optionally set the precision of the replica processes using a command line argument that we intercept using `PMPI_Init`. The compiler lowers the precision level for the duplicated code. Moreover, it prefixes the duplicated function calls with logic that ensures the correct routine is called based on if a process is a main or a replica. If the program calls an external library, the compiler pass inserts code to marshal the data into the appropriate precision as needed.

Adjustable Performance States for Replicas. Today's CPUs [79] and GPUs [85] are

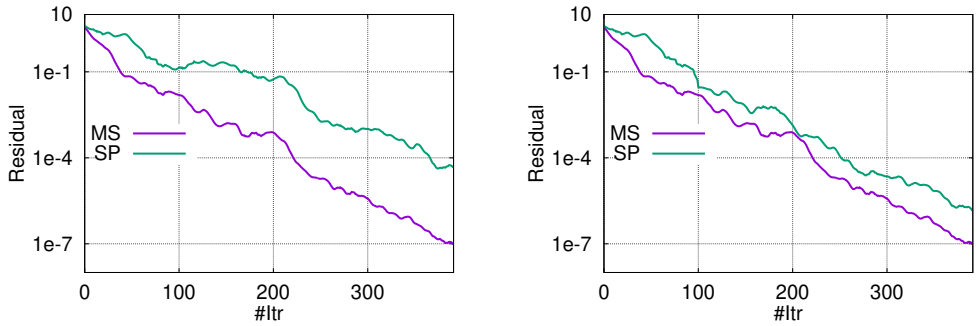
capable of transitioning between multiple performance states through dynamic voltage and frequency scaling (DVFS), selecting high-performance states to achieve high-performance and throughput or low-performance states for power savings and energy efficiency [89]. Our replication scheme supports adjustable power management to meet the user's demand of performance and energy-efficiency.

In our prototype, we bind processes to processing units and set core speed through the `cpufreq` or `nvidia-smi` utilities. The CPU and GPU speed setting is applied to each set of main and replica processes. We can set a fixed speed for the entire processes through `PMPI_Init` or certain speeds for certain kernels using an LLVM compiler pass. As we are currently exploring the energy-efficient techniques and studying the benefits, in this work we try various speeds for the two sets. In the future, we will study technologies to autotune the speed settings.

Flexible Hardware Resource Selection. There are plenty of hardware resources (high/low-speed GPUs/CPUs) in a large-scale, heterogeneous system. Besides the regular mapping where one GPU binds with one CPU core for one MPI process, our replication scheme supports arbitrary combinations for GPU applications for users' preference of performance, resilience, or energy-efficiency. For example, given a program providing both CPU and GPU versions, users may select high-speed dedicated GPUs for the main processes and low-speed or shared GPUs for the replicas. On systems where the CPUs have comparable performance as GPUs, the replicas can also run on the CPUs as indicated in command line arguments or inside the program. We have implemented this option in the prototype, but we find the CPUs have a much lower performance capability than the GPUs on our system, and thus don't provide the results.

4.2.2.2 Mitigating the Impacts on Resilience

When replicas run in a low-precision, lower performance state, or on lower performance hardware than main processes, their progress lags behind, creating a progress gap that increases over time. For example, consider the iterative linear solver conjugate gradients (CG) in which the main processes compute using mixed-precision (double with single) and the replica processes use just single precision. Figure 4.3 shows that after 200 iterations, the residual of replica processes is two orders of magnitude larger than that of the main processes. The increasing progress gap introduces two potential issues. First, it mistakenly flags SDC if message based comparison expects exact matches. Second, in the case of true soft errors for the main processes, recovering them directly with the replicas suffers significant loss of progress.



(a) No update in replica

(b) Periodic rejuvenation in replica

Figure 4.3: Residual history for: main and replica processes compute with different precision (a) without rejuvenation; (b) with periodic rejuvenation every 100 iterations. SP: single precision; MS: mixed double-single precision.

The periodic rejuvenation of replicas using main processes is effective to mitigate the impacts. At the rejuvenation point, we trigger SDC detection for the main and replica processes using application defined self-detection [47]. This periodical checking lowers the cost of checking for SDC and thus is more energy-efficient than prior work detecting at every message passing. If we detect SDC, we handle the fault and recover the corrupted process (see Section 4.2.3). Otherwise, we rejuvenate the replicas with the aid of an extra buffer, which allows precision conversion before communication and rejuvenation. Figure 4.3 shows the result of rejuvenation of lower-precision replicas for CG at a period of 100 iterations. Without periodic rejuvenation, the residual of replicas in lower precision decreases at a slower rate and subsequently diverges from that of main, as Figure 4.3(a) shows. With periodic rejuvenation, the lag in the replica's residual is bounded, as Figure 4.3(b) shows. The replica's residual is not equal to that of the main process after each rejuvenation because we update only key data in the replica. Here, we assume application SDC self-detection with a generic detection model [47]. The rejuvenation frequency should be set based on the application's sensitivity of different precision and users' requirement for SDC checking. A higher frequency reduces the divergence between main processes and the lower-precision replicas, but increases the cost of SDC checking.

4.2.3 Fault Handling

Applications on GPU-accelerated systems are impacted by both soft errors — e.g., bit-flips — that cause an erroneous deviation but without an interruption and hard faults — such

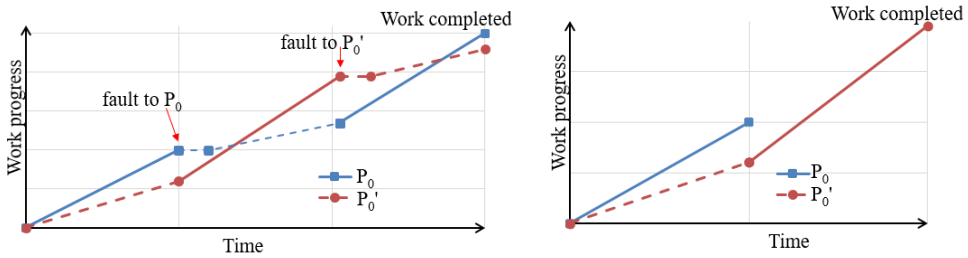
as processor or node failures — causing the application to crash. Applications must be protected against both of these scenarios. In this work, we focus on recovery from hard and soft faults in hardware (CPU/GPU) and assume the faults can be detected in software or hardware. In our design, we assume that data in an application are classified as static or dynamic based on if the data is ever written after initialized. We leverage this classification to optimize recovery.

Fault recovery from soft errors. Synchronous execution readily supports the detection of silent data corruption. Upon the receipt of a full length message, it is compared directly to the corresponding message from the other set, or the original message is hashed and then compared. Any disagreement is an indication of SDC. In the asynchronous case, we rely on application integrated detectors to notify the presence of SDC through a function call. This function will broadcast the result to all other MPI processes.

When a soft error occurs in the memory of one running main MPI process, the data of this faulty process is erroneous. This soft error does not immediately cause an interruption or a crash of the application. Thus, our framework recovers the faulty main process by receiving dynamic data from the corresponding replica process. If executing with loose synchronization, the main processes wait until the replicas catch up. In this case, the main processes decrease their speed and the replica processes increase their speed to quickly close the gap (see Figure 4.4(a)). If the application iteratively coverages — e.g., linear solvers — then we replace the data from the replica process and note that the convergence rate is diminished. If the fault is on the replica process, then rejuvenation immediately occurs on all replica processes. When leveraging asynchronous execution, the processes establish communication by completing non-block communication calls initially launched in `MPI_Init()`. After recovering the faulted process, performance states return to what they were prior to initiating recovery.

Another scenario to deal with when considering soft errors is that SDC has propagated to impact multiple main processes. In this case, we recover all faulty main processes by receiving the dynamic data from all the replica processes. Again, replica processes increase their speed while main processes lower their speed. To recover from corruption in multiple replica processes, we use rejuvenation on the corrupted processes.

Fault recovery from hard faults. In the hard fault scenarios, we assume the long-running application is interrupted or crashed by a CPU/GPU error or a node failure in the system. In these cases, requesting new node(s) and recovering from crash might be infeasible due to resource



(a) Resilience to multiple soft errors

(b) Resilience to hard errors

Figure 4.4: Resilience to various errors. Solid lines show current main processes and dash lines show current replica processes.

limitations or recovery may incur high time and energy overhead. Under this assumption, utilizing the remaining healthy resources to continue application execution is a better choice [56, 89].

Hard faults occurring at different degrees have various impacts; therefore, we handle them differently. If a hard fault occurs in a CPU core binding one MPI process, the core is down and data in its memory is lost. In this situation, we replace the faulty main process with the corresponding replica process. If a hard fault occurs and impacts an entire node and several main processes, we use the replica set to continue the execution. Assuming no new processes are created to reinstantiate the faulty main processes, the replicas will take the place of the main processes (see Figure 4.4(b)). In both of these cases, if the replicas lag behind the main processes, the speed of the main processes are switched to a low-power state and the replica processes are increased in speed. If the replica runs a lower precision than the main, their data is promoted to the same precision as the main. If the crashed process is a replica, we rejuvenate the replicas and continue in synchronous mode. New processes can also be created and linked to the target communicators after computation resumes. In this case, processes will be relieved from double duty before work completes.

4.3 Experimental Results

We implement a prototype of the proposed redundancy scheme and evaluate it on a cluster with 16 GPUs. Each node consists of two 20-core Intel Xeon processors and two NVIDIA Tesla V100 GPUs. Each CPU core can vary its frequency from 1.2 GHz to 2.4 GHz, and each GPU card from 780 MHz to 1380 MHz. We only account for the power consumption of CPU and GPU processing units involved in computation and exclude unused ones. We also exclude components

such as memory, motherboard, and others that consume a relatively constant low amount of power. We use `RAPL` and `nvidia-smi` interfaces to monitor power and set frequency for CPUs and GPUs, respectively.

The application programs used for evaluation need to satisfy several conditions: parallelized with multithreading, MPI, and CUDA or another GPU programming technique, and with support for multiple precision — double, single, half, and their mixes. In addition, to evaluate the effects of disparate precision between main and replica processes, we need the applications to be iterative and converging. Implementing all these versions for an application would be tremendously time-consuming, and optimizing them typically requires teams of expertise. Limited by these conditions, we choose MiniQMC from QMCPACK [73], Convolution benchmarks in DeepBench [92], where the MiniQMC benchmark comes with all configurations, and DeepBench has all but mixed precision which we implement. We add a third application CG which iteratively solves a series of linear equations in the form of $Ax = b$. The widely used HPCG benchmark [50] is a good choice and optimized. Nevertheless, its full source is unavailable to the public to instrument and apply our energy efficient techniques. We implement our own version of CG based on routines in RAPtor [21] and CUDA libraries [104]. The solver tolerance is set at $1 \times e^{-12}$ for double precision (DP), $1 \times e^{-7}$ for single precision (SP), and $1 \times e^{-4}$ for half precision (HP).

Table 4.1 shows the performance of our implementation and its comparison with the HPCG benchmark for a 27-point stencil matrix. We run HPCG and ours on four GPUs in a fault-free environment. Our implementation achieves 173.2 GFLOPS in comparison to HPCG's 182.7 GFLOPS with DP, and 200.5 GFLOPS with mixed double and single precision on four GPUs. This performance gives us confidence that the findings using our CG implementation are valid.

	Impl.	GFLOP/s
	HPCG_GPU_DP	182.7
Ours	CG_GPU_DP	173.2
	CG_GPU_MS	200.5
	CG_GPU_MH	231.6

Table 4.1: Performances of our CG implementations. DP: double precision; MS: mixed double-single precision; MH: mixed double-half precision.

In the rest of the section, we present detailed evaluation results under various fault situations, the effects of optimization techniques, and compare our relaxed redundancy with other resilience solutions. For simplicity, all results are with dual modular redundancy (DMR) — i.e., one replica

process for each main process.

4.3.1 Resilience Support for Various Faults

We first evaluate how our redundancy scheme supports resilience under soft errors and hard failures. In all our experiments, we use dual redundancy and application-specific fault detection [66], and focus on how our redundancy scheme recovers the detected faults. Without loss of generality, we generate faults at certain times over the course of the execution instead of randomly, to remove the overhead of fault injection from the power and timing experiments. For simplicity, we consider a faulty main processes in our examples, even though our scheme recovers both faulty main and replica processes.

4.3.1.1 Soft Errors on Process Data

We first consider soft errors that corrupt data on one or more processes. Soft errors are common cases where one or more bits flip in the logic or storage of GPUs or CPUs running MPI processes. We mimic the effect of soft error by injecting faults — flipping bits — to corrupt applications' data structures. That is, the corresponding data stored in memory is corrupted.

Upon the detection of soft errors and faulty processes, our scheme recovers the corrupted data with the corresponding healthy copies so that the processes continue to perform correct computations. Prior studies show that data corruption in data structures may result in only a single wrong message that is detectable, or cascade to multiple wrong messages originating from the corrupted sender [56]. We choose to recover only the faulty process for the former case, and recover the tainted processes for the latter case. In case that it is difficult to determine between the two cases, our scheme conservatively recovers the entire set using the healthy set upon the notification of soft errors to ensure correctness. While this handling involves unnecessary message passing between the two sets, it is much less costly than schemes using fine-grain synchronization, as the communications are only performed once.

We use the execution of CG to illustrate the process of fault recovery and continued computation afterwards in Figure 4.5. In this experiment, 8 main processes run in mixed double-single precision where $f_{GPU} = 1.38$ GHz, and 8 replicas run in single precision where $f_{GPU} = 0.78$ GHz. We use the asynchronous execution mode between main and replica processes. In the beginning, the main processes progress faster, their residual value decreases faster, and the replica processes

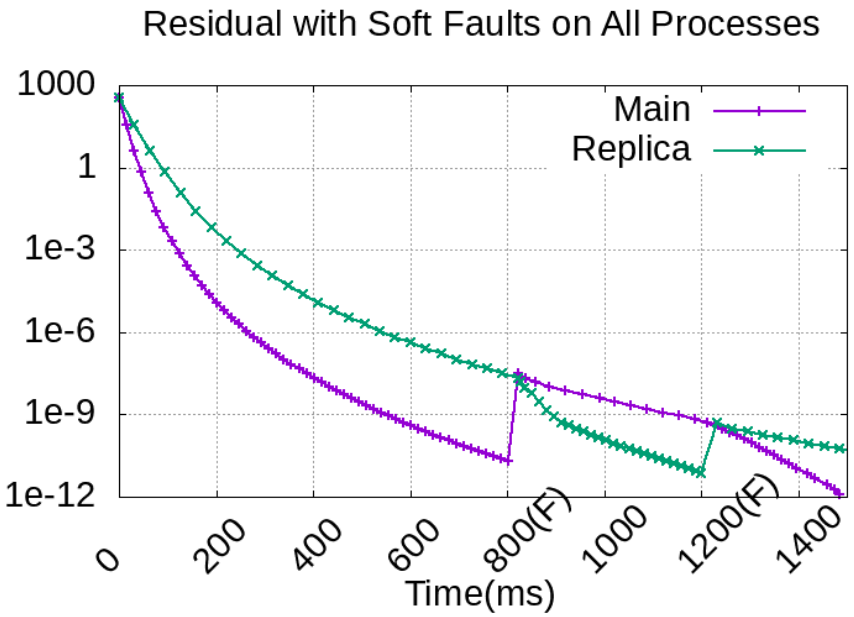


Figure 4.5: Resilience support for soft errors. CG is running on 16 GPUs with 8 main and 8 replica processes respectively. Two faults are injected at 800 ms and 1200 ms respectively.

lag behind due to a low GPU speed. We insert two errors to the data of the same main process rank, one at 800 ms and the other at 1200 ms. After each fault injection, our scheme recovers the data of the entire main set using the replica processes to highlight the worst case performance. As the recovered data lags behind the main copy, we observe an increase in the main's residual. There is also a notable delay from the fault injection to recovery completion. This delay accounts for data recovery using message passing. Once computation resumes, our scheme decides if it should maintain the roles of mains and replicas, or switch them. Figure 4.5 shows the role switch, where the initial replicas become the new mains and thus scale up the GPU speed, while the initial mains do the opposite. Upon the detection of the second error, our scheme again recovers the lost data, switches the roles of the main and its replica, and changes their GPU speeds.

4.3.1.2 Hard Failures

We now examine recovery from hard faults like process, node, or link failures. Hard failures directly terminate processes and cause data loss or disable their communications. In our experiments, we mimic the effect of hard failures by disabling a node and its link.

With our redundancy scheme, upon the detection of hard failures, a corresponding healthy process takes up the double role — its own and the faulty copy's, and resumes the computation. Once

computation resumes, we decide whether to restore the terminated processes depending on the time overhead, available resources, and the chance of more failures on the same process ranks. Restoring involves creating new processes using tools such as ULFM [23], linking it to the right communicator, and mapping it to the right ranks. Among these, the first step is most time-consuming, the second takes a similar time as a collective communication, and the rest is less costly. These newly created processes need to be assigned to healthy nodes, which may not be available. Nevertheless, should the faulted processes not be restored, an application only tolerates one hard failure for each process rank during its execution; however the probability that failures strike both a main and its corresponding replica is low, $1/p$, and decreases as the number of processes p increases.

We use the execution of CG as an example to illustrate the process of hard fault handling in Figure 4.6. Here we run CG with 16 GPUs, and use the asynchronous execution mode without rejuvenation, similar as for the soft error recovery case study. When there are no faults, main processes run on GPUs with higher speeds and progress faster. When encountering a node failure injected at 800ms, our scheme immediately makes replica processes to take over the main role and resumes computation. The residual of main increases due to the replacement with a replica, which lags in progress. The execution speed of the new main processes increases to catch up the lost work progress. For the new replica processes to continue, they need to involve some main processes with double duty to form a full set. While not shown in Figure 4.6, the execution of CG can tolerate more faults on different processes if no new processes are created, and on any processes otherwise.

4.3.2 Effects of Energy Optimization Techniques

Here, we use the GPU version of the programs to examine energy savings from individual energy-efficient techniques and their combinations. As mentioned, our reported power and energy values only account for GPU cards and the involved CPUs.

4.3.2.1 Effects of Various Precisions

Multiple prior studies have shown that iterative and convergence workloads can run in multiple precision, including mixed precision to obtain correct results [27]. Two particular questions we have for computation subject to faults are: would running replicas in a lower precision than the main processes be beneficial? and should programs always run in the lowest (mixed) correct precision? In this work, we use experimental results to provide answers. For simplicity, we run all

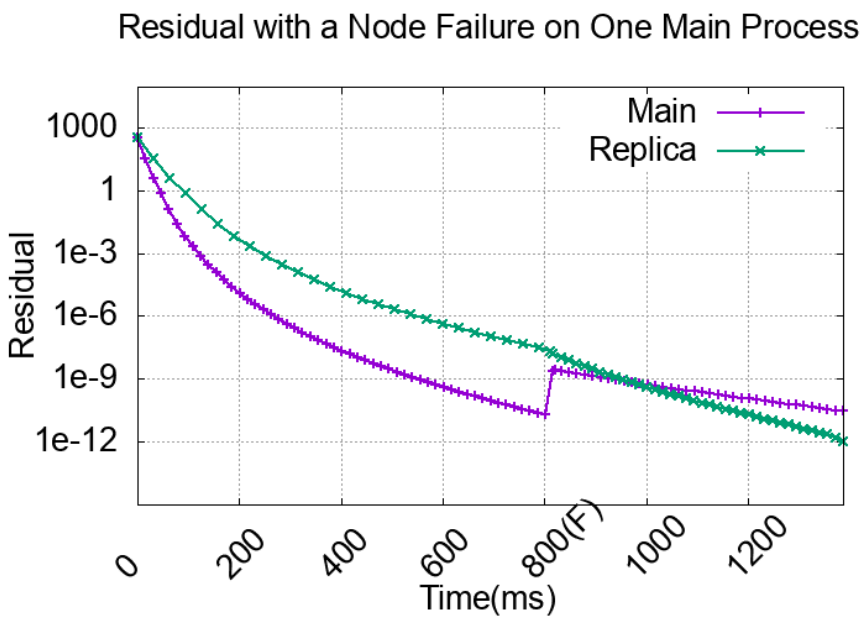


Figure 4.6: *residual∼time* under a node failure impacting one main process when CG is running on 16 GPUs with 8 main and 8 replica processes respectively. A hard fault occurs at 800 ms.

GPUs at the highest frequency, and thus energy savings come from the precision selection. We set the rejuvenation period at every 100 iterations. Figure 4.7 shows the performance, power, and energy of CG with 27-point stencil matrix with 10 soft faults on 16 GPUs. There are multiple precision settings for main and replica processes, representing various exact and relaxed replications. The tolerance is set as $1e{-}12$ for main processes with mixed double-single precision (MS), $1e{-}7$ with SP and mixed single-half precision (MH), and $1e{-}4$ with half precision (HP) for replicas in Figure 4.7.

We make several observations from Figure 4.7. First, reducing the data precision for CG lowers execution time, power and energy consumption and keep the accuracy of double precision in main processes. When main processes are recovered with less accurate data of SP, MH, and HP in replicas after faults, it takes longer time to converge. Changing the precision from MS —— MS to MS —— HP increases the execution time by about 14%. Meanwhile, such precision change reduces power by 15%. Resultantly, energy consumption reduces slightly more than execution time. Second, for a given precision for the main processes, running the replica at a lower precision leads to reduced power but increased execution time. In this CG's example, MS —— MH takes less time and energy than MH —— HP due to more accurate replicas.

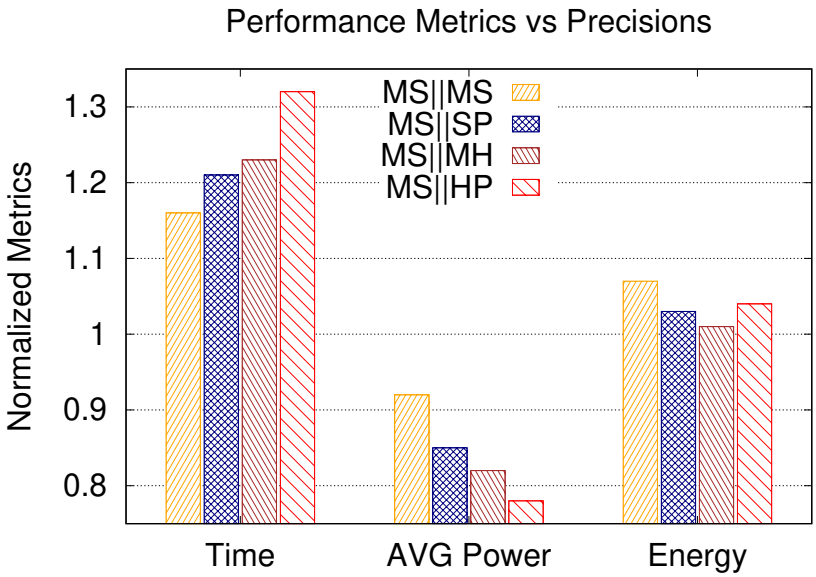


Figure 4.7: Normalized performance metrics under various precision for CG benchmark with 10 faults injected. The baseline is set as fault-free case with MS. SP —— MH represents SP for main and MH for replica. MS: mixed double-single; MH: mixed single-half; HP: half precision.

4.3.2.2 Effects of DVFS

We examine the variation of power with GPU frequency. We use the NVIDIA System Management Interface (`nvidia-smi`) to set the frequency of the streaming multiprocessors (SMs) on GPUs. For clarity, we only present power when main processes run at the maximum GPU frequency f_H and replicas are run at the minimum GPU frequency f_L. The resulting time and power values are the upper and lower bounds. Time and power of other configurations are roughly proportional in between.

In our experiments, we apply both DVFS and precision selections on the main and replica processes. Figure 4.8 illustrates how mixed precision and DVFS-based power management changes the power profile of CG. We use a 27-point stencil matrix with 10M rows under fault-free environment as a study case. By default, CG runs with DP. Both mixed double-single precision and lowest GPU frequency show a significant power saving compared to DP with f_H. Among all of these configurations, mixed double-single precision (MS) with f_H shows the best performance improvement (20%) while mixed double-single precision with f_L shows the best energy saving.

Figure 4.8 indicates that relaxed replication is able to meet the users' various requirements for applications, such as higher performance or lower power consumption, by supporting a flexible

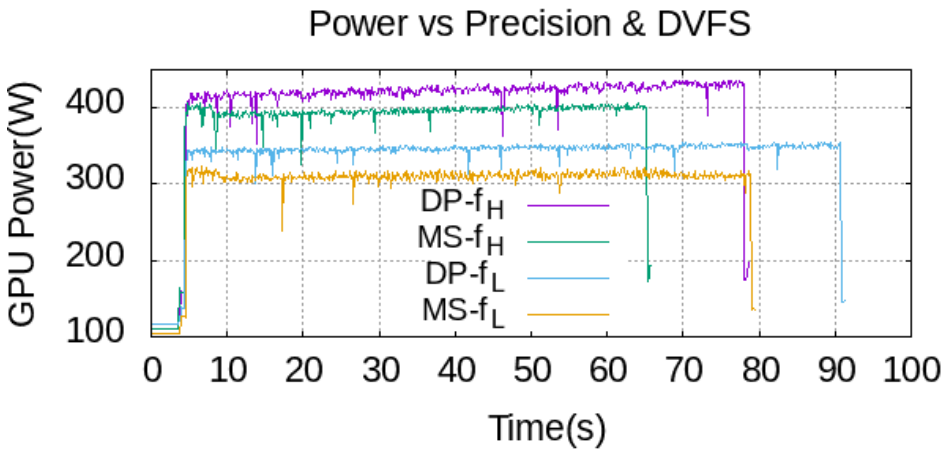


Figure 4.8: Power reduction with mixed double-single precision and f_L. MS-f_L has a similar execution time as DP-f_H, but with 30% less power. Main processes are running with f_H, and replica processes are running with f_L.

combination of configurations. One interesting observation is that a main processes with DP-f_H takes a similar time as a replica process with MS-f_L. This observation suggests that we can use the synchronization mechanism for a combination of configurations, to simultaneously support fault recovery and energy saving.

4.3.2.3 Overall Improvements from Combined Optimizations

Figure 4.9 shows the overall benefits from each individual energy-efficient techniques and in combination for the DeepBench. The base is the exact replication that runs both main and replica processes in the same precision (DP) and at the highest frequency. Existing modular redundancy schemes such as RedMPI employ this strategy. The other three case are from our relaxed replication where main processes and replicas run in DP —— MS, the acceptable lowest GPU speed, and their combination. In comparison to exact replication (Base), our scheme supports combined optimization of mix precision and DVFS, and improves performance by 8%, power by 29%, and energy by 35%. One interesting observation is that the DP —— MS precision achieves better performance and energy than DVFS, but DVFS is more effective for power reduction. While their combination achieves the best energy and power, it still underperforms mixed precision in time.

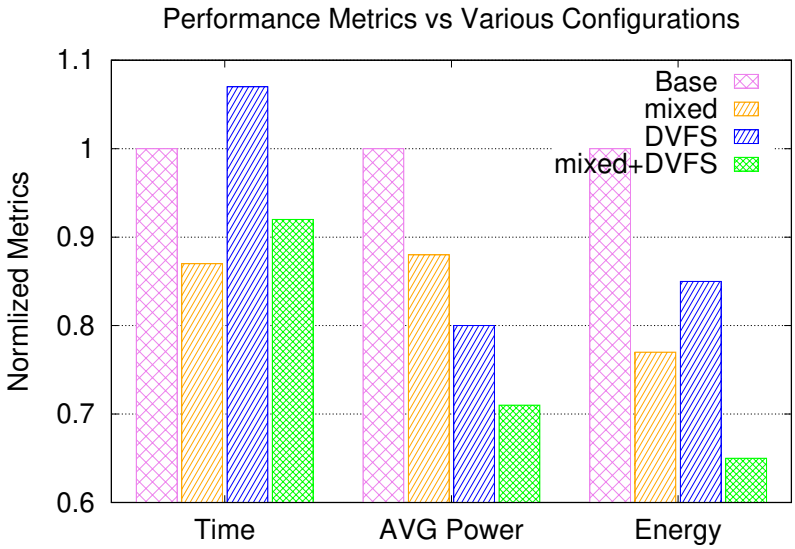


Figure 4.9: Overall improvement of performance, power, and energy for DeepBench running with relaxed replication. Base represents exact replication used in existing redundancy schemes. The others represent our relaxed replication and enabled optimizations. One fault is injected in each experiment. The rejuvenation period is set as every 1000 iterations.

4.3.3 Comparisons with Existing Solutions

We compare our redundancy scheme against the checkpointing technique. We first compare them using experimental results collected from our small cluster. In our experiments, we use the same fault situations — an MTBF of 5 year per-socket — used in previous resilience studies for large-scale systems [55, 56]. It means one failure occurs during application execution. As the redundancy based resilience scheme is meant to be used on large-scale systems, we then use the direct measurement and analytical model to project their performances on large-scale systems. For checkpointing, we checkpoint one copy of protected data to remote memory to mimic level-2 of multilevel checkpointing scheme VeloC [93], which is an efficient asynchronous checkpointing scheme. Specifically, we checkpoint the data of the MPI process running on node i to CPU memory on node $(N-1-i)$ for $i \in [0, N-1]$ for N total number of nodes.

Figure 4.10 shows the results where data points are from experiments for 16 or fewer GPUs, and projection for larger numbers of GPUs. We project energy for RedMPI, CR, and the best case of relaxed replication on large-scale systems from our experimental data. The fault model is still a MTBF of 5 year per-socket [55, 56]. For each system size, we use a half for either main or replica processes for the two redundancy schemes, and all for computation for checkpoint-restart (CR). We

use CG's weak scaling performance for matrices with 50K non-zeros per processor and estimate the time-to-solution of relaxed replica, RedMPI and CR with the method in studies [87, 55]. We always use double precision for RedMPI and CR, and MS —— SP for relaxed replica. Therefore, the power per process can be assumed roughly constant for each scheme.

We observe that while energy overhead increases with system size under each scheme, it increases only slightly with redundancy based schemes but significantly with CR. While our data may not accurately reflect the actual overheads on specific systems, it captures the changes in their comparative relation: even though redundancy based schemes incur a higher overhead on small systems, they become more energy efficient on large scale systems. Relaxed replication has a better scalability compared to CR. As system size increases to exascale systems, CR could incur prohibitively large time overhead and consume more time to solution than dual redundancy mechanisms. The energy gap between relaxed replication and RedMPI is roughly stable. As this gap is normalized, the actual difference in energy amounts is significant as the base power with hundreds of thousands of processors could be megawatts, e.g., assuming 10 Watts/processor.

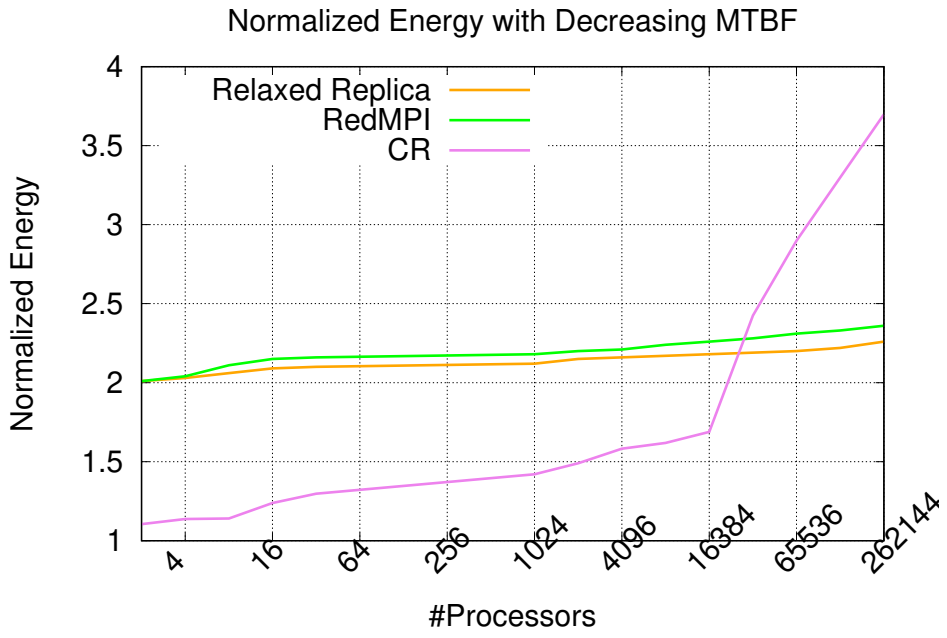


Figure 4.10: Normalized energy in weak scaling under our relaxed replication, RedMPI, and CR. Data points on each #processes are normalized to the fault-free execution with the same #processes. The illusive linear change at large #processes is due to the logarithmic X-axis.

4.4 Summary

This chapter proposes a novel redundancy technique, relaxed replication, for resilient and energy-efficient HPC applications in GPU-accelerated systems. Relaxed replication provides MPI-process level redundancy, enabling optimizations for GPU applications. It supports multiple technologies including heterogeneous processors, mixed precision computation, and power management to reduce time and energy overhead of replication. Experimental results show that our approach significantly reduces the resilience overhead while maintaining energy-efficiency compared to previous resilience mechanisms. Overall, this work makes the following contributions:

- We present a novel redundancy mechanism that relaxes the requirement of synchronization and identicalness for replicas for MPI programs running on GPU-accelerated systems. Our mechanism supports replicas to run in lower-precision and at lower power/performance states with periodical rejuvenation or asynchronization.
- We explore multiple energy-efficient techniques and study their applicability to HPC applications, their impact on energy efficiency, resilience overhead, and their trade-offs. These techniques leverage readily available GPU architectures and technologies. Results can benefit real applications and GPU-accelerated systems.
- We prototype the relaxed replication mechanism and evaluate it on a cluster. Our results show up to a 19% reduction in energy for unmodified programs in various fault situations compared to exact replication techniques.

Chapter 5

Scalable Algorithms for Large-Scale Sparse Tensor Decomposition

Tensors are multidimensional arrays, and tensor decomposition generates lower-rank representation of the data for a wide range of applications, including machine learning and social networks. The extremely large size of tensors for real-world applications requires parallel tensor decomposition algorithms for distributed memory systems. Load imbalance and communication cost are two major bottlenecks for performance and scalability. Nevertheless, improving them is challenging due to various complex tradeoffs between computation and communication within and among the kernels. Existing work in CPD focuses on computation balance in tensor-related kernels, leading to imbalanced computation and communication for other kernels.

In this chapter, we present a performant scalable distributed algorithm BA-CPD, including irregularity-aware algorithm for workload partition and asynchronous CPD implementation. Our BA-CPD improves workload balance and reduces communication cost in comparison to existing work. Our irregularity-aware algorithm employs a workload partition scheme that co-optimizes load imbalance and communication. Unlike previous works those focus on balancing tensor nonzero values, our scheme finds a partition with the smallest tensor nonzero imbalance from a mode-balanced base that leads to less commutation volume. Based on our irregularity-aware algorithm

for workload partition, we further present asynchronous algorithms to reduce the communication overhead of collective communication operations in traditional bulk-synchronous CPD. We perform on fine-grain decoupling of computation and communication tailored for different kernels in CPD. The decoupling enables our asynchronous algorithms to leverage one-sided communication to best hide latency.

In this chapter, we present the detailed design of irregularity-aware algorithm for workload partition which best balances across the kernels and the tailored decoupling of computation and communication to leverage asynchronous communication. The workload decomposition achieves balances for all but computation in tensor-related kernel and co-optimizes computation and communication imbalance. We present two strategies for asynchronous algorithm: $\mathcal{X}$-Stationary 1D and $\mathcal{X}$-Stationary 2D. In each strategy, we decouple computation and communication for different kernels to take the advantage of asynchronous communication and hide latency. Our result shows that BA-CPD is scalable and outperforms the state-of-the-art distributed implementations.

5.1 Learning the Performance of Distributed Tensor Decompositions

This section illustrates the general medium-grained, bulk-synchronous distributed CPD algorithm and its performance problem abstraction and analysis along with our findings. We highlight six findings distilled from an extensive evaluation of the state-of-the-art medium-grained CPD implementation in SPLATT [115]. The experimental hardware and software configurations are described in Section 5.4. This is the first work that studies the distributed CPD in depth by carefully profiling all stages of the algorithm on diverse datasets with different process settings.

5.1.1 Problem Statement and Analysis

We first present general models to capture the execution time of medium-grained distributed CPD in Algorithm 2. Our target is to find the optimal data distribution by designing a grid configuration and distribution policy, to obtain the best CPD performance, expressed in Equation (5.1). The optimal grid configuration $\mathcal{G}_{opt}$ and distribution policy $\mathcal{D}_{opt}$ have the minimum overall execution time. The execution time of CPD is dominated by the iterations (Line 7-16) in Algorithm 2. We use

the time of one iteration to represent the CPD execution time, noted by T_{cpd}, which aligns with our experiments.

$$\mathcal{G}_{opt}, \mathcal{D}_{opt} = argmin_{\mathcal{G},\mathcal{D}} T_{cpd} \tag{5.1}$$

5.1.1.1 Execution time analysis

T_{cpd} consists of the aforementioned five steps: MTTKRP, MAT SOLVE, MAT NORM, MAT $A^T A$, and MTTKRP COMM. Due to the bulk synchronous feature of MGBS-CPD, T_{cpd} is expressed in Equation (5.2).

$$\begin{aligned} T_{cpd} = T_{mttkrp}(c_N, R, M_p) + T_{mcomm}(P, I_l, I_p) + \\ (T_{solve}(R, I_p) + T_{norm}(P, R, I_p) + T_{ata}(P, R, I_p)) \end{aligned} \tag{5.2}$$

The time complexity of each step is listed in Table 5.1. Two collective communications are employed to synchronize and update local data, *MPI_Alltoall* in MTTKRP COMM and *MPI_Allreduce* in MAT NORM and MAT $A^T A$. The communication time is modeled as $\alpha + \beta n$, where α and β are the memory latency and bandwidth respectively, and n is the number of bytes to be transferred [123]. We assume the tensor rank R (usually a small value < 100) and $c_N < N$ are constants [1]. T_{cpd} is mainly determined by the number of nonzeros of a local sparse tensor M_p, layer size I_l and local matrix size I_p though computation and communication are different functions of these variables. M_p dominates T_{mttkrp}; I_p affects the time complexity of all matrix steps, $T_{solve}, T_{norm}, T_{ata}$; I_l and I_p both influence the other communications T_{mcomm}.

Comparing these steps, we see that, in general, M_p is several orders of magnitude larger than I_l and I_p for relatively small or mildly sparse tensors, where T_{mttkrp} might take a larger percentage in T_{cpd}. However, M_p could be in the similar order-of-magnitude as I_l and I_p for relatively sparse tensors or tensors with irregular shapes, where matrix computations and communication might have non-negligible costs. Besides, we also observe that some configurations of $\mathcal{G}, \mathcal{D}$ could decrease the execution time of one step but increase that of other step(s). (Experiments in Section 5.1.2 verify this analysis.) Thus, it is non-trivial to infer the optimal settings for $\mathcal{G}, \mathcal{D}$ to gain the highest distributed performance only relying on theoretical analysis even with c_N, P, R all fixed, plus the analysis is

[1] c_N is a constant for a given tensor in an MTTKRP algorithm [114, 81].

closely related to the features of input sparse tensors.

5.1.1.2 Load imbalance ratios

Thus far, we consider M_p, I_l, and I_p as the average values on each process, which is the ideally balanced data distribution. However, in reality, especially for irregular sparse tensors, the data distribution could be very skewed. We present three imbalance ratios as metrics to measure this effect.

We use a more accurate *imbalance ratio* r, adopted from the one used in [114] [2] , to represent the imbalance of sparse tensor computation, matrix computation, and communication. From Table 5.1, sparse tensor computation, MTTKRP, is influenced by M_p. Nonzero imbalance ratio $r_{nnz} = (max\{M_p\} - min\{M_p\})/max\{M_p\}$ represents the gap between the maximal and minimal number of nonzeros assigned to a process among P processes. Our imbalance ratio r, always less than 1.0, better evaluates long and short jobs per process. A ratio close to 0.0 means an ideal, even nonzero distribution; while a ratio close to 1.0 means extreme imbalance indicating that the gap between the longest and shortest MTTKRP execution time is huge. Analogously, r_{I_p} represents the imbalance ratio of I_p thus for matrix computation; r_{vol} is the imbalance ratio of communication volume. We use the imbalance ratio for communication volume rather than I_l because the communication volume is influenced by both I_l and I_p; therefore, r_{vol} better represents the communication. The three imbalance metrics help determine the $\mathcal{G}_{opt}, \mathcal{D}_{opt}$ by reflecting features of real sparse tensors from three distribution-related perspectives.

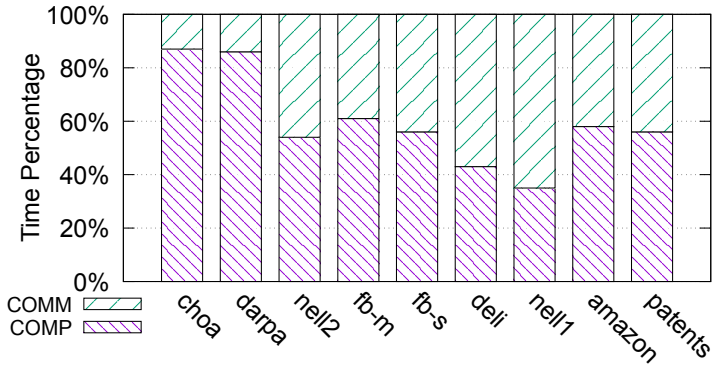


Figure 5.1: Computation and communication percentage of CPD.

[2]The nonzero imbalance in the work [114] represents the gap between the maximal and average number of nonzeros assigned to a process, which cannot measure the imbalance from the short tasks well.

5.1.2 Findings

Based on our theoretical analysis and the proposed imbalance ratios, we discuss performance findings on MGBS-CPD. The tests are run on the open-source SPLATT MPI library [116], representing a fast state-of-the-art MGBS implementation from studies [114, 17, 101, 6].

Finding 1: *Both computation and communication have non-negligible costs, and the dominance varies with tensors.*

We only use T_{mttkrp} and T_{ocomm} in Equation (5.2) as representatives to computation and communication to enlighten this finding theoretically. Comparing the dominant parameters: M_p and I_l, either one could be larger for different sparse tensors. For example, tensor *choa* has a maximum $M_p = 400K$, $I_l = 15K$ while tensor *deli* has $M_p = 2M$, $I_l = 4M$ on 768 processors. Thus, it is hard to tell whether computation or communication is dominant. We further study the overall performance of the SPLATT CPD implementation running on 768 processors. Figure 5.1 depicts the percentage of the execution time taken by computation and all types of communication operations in Algorithm 2 respectively on nine sparse tensors from real applications (refer to Section 5.4 for tensor descriptions). Computation takes 35-81% while communication takes 19-65% of the total execution time. Computation largely dominates the CPD execution on two tensors: *choa* and *darpa*; communication largely dominates on tensors *nell1* and *deli*. This matches the M_p and I_l examples given above. On the rest of five tensors, computation and communication take a similar amount of time, with a percentage difference less than 10%. The shifting of dominance between computation and communication among tensors raises the difficulty of performance optimization. Taking tensor dimension sizes into consideration, *fb-m*, *fb-s*, *choa*, and *patents* are more irregular tensors in Table 5.3, and tend to be computation dominated, while the other tensors are more communication dominated or without significant dominance.

Finding 2: *Computation cost is not always dominated by sparse tensor computation, but also dense matrix computations.*

Compare the computation complexity of matrix operations, MAT SOLVE, NORM, $A^T A$, versus the MTTKRP complexity in Table 5.1 shows $I_p < c_N \times M_p$ is generally true if there are not many empty slices in dimension-n. However, $R \times I_p < c_N \times M_p$ is not necessarily true and depends on the values of R, the constant c_N ($R > c_N$ usually), the distribution policy which determines the sparsity pattern of the local tensor $\mathcal{X}_p$ and influence value I_p in the next process-distribution

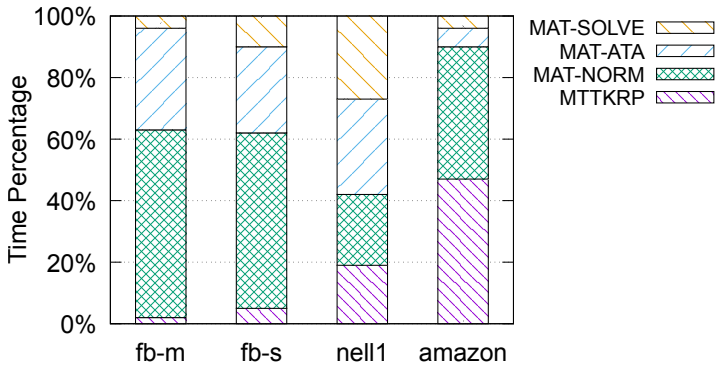


Figure 5.2: Time percentage of computational kernels of CPD.

for matrices. This is especially prudent for irregular tensors with $I = \Theta(M)$ on one dimension. If $R \times I_p > c_N \times M_p$, then the complexity of MAT $A^T A$ and SOLVE steps could take more time than MTTKRP. While these matrix operations are all dense and generally perform more efficiently than the sparse MTTKRP, dense matrix computation can influence computational performance. We conclude Finding 2 that MTTKRP is not always the dominant computational kernel in CPD, the matrix computation kernels are also expensive as tensor rank grows and for tensors with preferable sparse patterns (e.g. irregular tensors). Therefore, the state-of-the-art work [114, 39, 6] that focuses on minimizing the computational cost of MTTKRP may not gain much performance improvement for all types of tensors.

Figure 5.2 shows the time percentage of the four computational steps on four representative tensors: *fb-m*, *fb-s*, *nell1*, and *amazon*, verifying our theoretical analysis above. For the four tensors, MTTKRP, MAT NORM, MAT $A^T A$, and MAT SOLVE take 2-47%, 23-61%, 6-33%, and 4-27% of the CPD computation time respectively. The other three computations easily takes more execution time than MTTKRP, which needs to be optimized as well for better performance. These insights about dominating costs of Findings 1 and 2 could guide our following optimization for distribution policy.

Finding 3: *Different load imbalance factors influence computation and communication overhead.*

Figure 5.3 shows these three ratios r_{nnz}, r_{vol}, and r_{Ip} for sparse tensors as the increasing order of r_{nnz}, where r_{nnz} and r_{Ip} reflect computation imbalance and r_{vol} reflects communication imbalance. The nonzero imbalance is less than 0.2 for the left six tensors, while tensors *patents*, *fb-m*, and *fb-s* have a much higher nonzero imbalance, all of which are very irregular in dimension sizes.

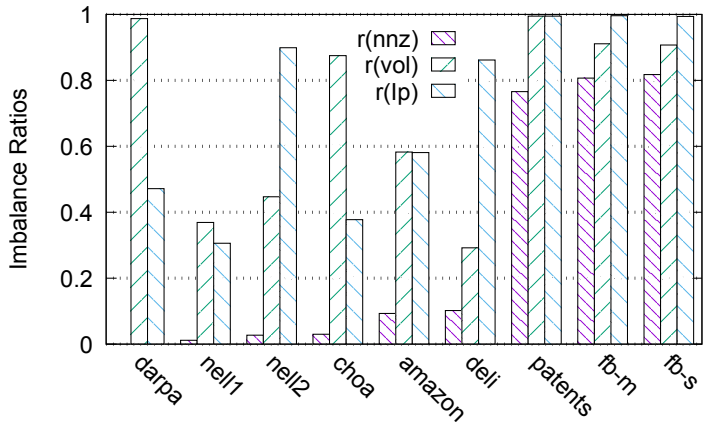


Figure 5.3: Load imbalance ratios (r_{nnz}, r_{vol}, and r_{Ip}) for sparse tensors.

All the tensors have much higher volume and I_p imbalance ratios than nonzero imbalance ratios. Different from the dominance perspectives in Findings 1 and 2, the imbalance ratios expose the load imbalance issues which influence the overhead of all the key steps in Table 5.1 correspondingly. Almost all tensors have at least one imbalance ratio with the value higher than 0.8, which indicates the difficulty to do a good tradeoff among the three imbalance ratios. The state-of-the-art work puts efforts on optimizing the nonzero imbalance [114, 31], which only influences sparse tensor computation. Therefore, they only target minimizing the tensor computation imbalance not communication or the other matrix computation imbalances.

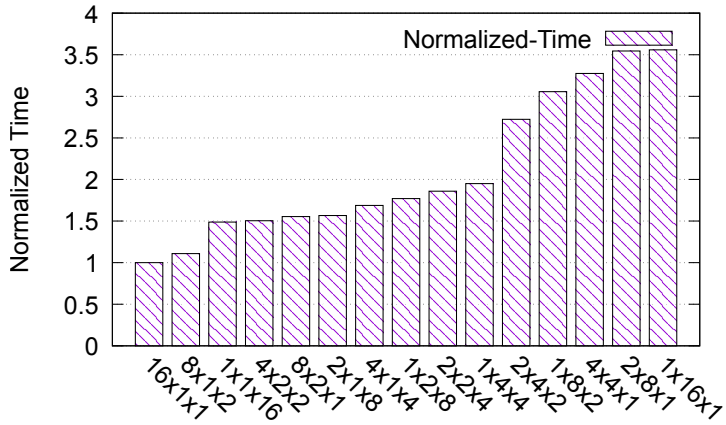


Figure 5.4: Normalized time of all possible grid configurations to the one performs the slowest for tensor *amazon* on 16 MPI processes.

Finding 4: *Different grid configurations could lead to very different distributed* CPD *per-*

formance.

For a given tensor, the process grid on which the tensor is mapped determines the computation and communication costs from the first sight, even before the distribution policy takes effect. Figure 2.4 shows the tensor and matrix decomposition on 12 processes as a $2 \times 3 \times 2$ grid. Given 12 processes, there are 18 unique configurations on which the tensor can be mapped to the processes. Configurations $12 \times 1 \times 1$, $1 \times 12 \times 1$, and $1 \times 1 \times 12$ are considered as different ones due to partitioning the first, second, and third dimensions correspondingly. A cluster with hundreds of nodes will have thousands of configuration, or more. Figure 5.4 shows all grid configurations for 16 MPI processes, with the execution time varies up to 3.5×. Thus, finding the optimal process grid is critical to choosing the distribution policy and overall performance, which also requires an intelligent approach over the costly brute-force method.

Finding 5: *Workload partition can not simultaneously balance computation and communication for all kernels in irregular tensors.*

There are four computation kernels in the iterative CPD algorithm. Below, we analyze the distribution in each kernel and illustrate that no data composition can simultaneously balance computation and communication for all kernels for irregular tensors.

Workload Partition and Inherent Communications. For each kernel, a processor only performs a portion of the computation using its local and remote data. The computation distribution and necessary communication among the processors is determined by the data decomposition. Table 5.1 shows local computational complexity and communication volume, where I_p is the number of local matrix rows and I_l is the layer dimension size in current mode. I_l and I_p are different across processes depending on how the workload is partitioned.

Key Steps	Local Comp	Local Comm Volume
MTTKRP	$\mathcal{O}(c_N \times R \times M_p)$	$2(I_l - I_p) \times R + 2I_p \times R$
MAT SOLVE	$\Theta(R^2 \times I_p)$	0
MAT NORM	$\Theta(R \times I_p)$	R
MAT $A^T A$	$\Theta(R^2 \times I_p)$	R^2

Table 5.1: Time complexity of the key steps in MGBS-CPD.

Mttkrp. Each process first performs its local Khatri-Rao product (Line 11), and then send this local results to all other processes in the same layer as partial MTTKRP results (Line 12). Each process then sums all partial dense matrices to obtain the final MTTKRP result in submatrix

$\tilde{\mathbf{A}}_p^{(n)}$. The local computation is usually imbalanced because M_p is different for a sparse tensor from real applications, as Table 5.1 shows. The communication of MTTKRP has a complexity of $\mathcal{O}(\alpha P + \beta R I_p) + \mathcal{O}(\alpha P + \beta R(I_l - I_p))$ via MPI collective operations.

MAT SOLVE. Each process updates $\tilde{\mathbf{A}}_p^{(n)}$ using the Cholesky method based on the temporary results from MTTKRP (Line 13). The local computation is imbalanced if I_p is different. I_p of each process can be different or not depending on how the workload is partitioned. There is no communication in MAT SOLVE for bulk-synchronous algorithm.

MAT NORM. Each process normalizes $\tilde{\mathbf{A}}_p^{(n)}$ locally and then performs a parallel reduction to obtain $\tilde{\boldsymbol{\lambda}}$ (Line 14). The local computation is imbalanced if I_p is different. The communication has a complexity of $\mathcal{O}(\alpha \log P + \beta R \cdot \log P)$.

MAT ATA. Each process uses symmetric matrix multiplication locally and then performs a reduction to form the new $\tilde{\mathbf{U}}_n$ for the next iteration (Line 15). The local computation is imbalanced if I_p is different. The communication has a complexity of $\mathcal{O}(\alpha \log P + \beta R^2 \cdot \log P)$.

Complex Computation-Communication Tradeoffs within Kernels It is impossible to simultaneously have balanced computation and balanced communication for an irregular tensor with non-uniform nonzero distribution in MTTKRP. Focusing on balancing M_p must leads to imbalanced I_l and vice versa. Figure 5.7 shows workload partitions for an irregular tensor on $2 \times 3 \times 2$ processes. Balancing nonzero computation by evenly partitioning tensor nonzeros among the processes leads to a partition in Figure 5.7(a). While balancing communication by evenly partitioning matrix size leads a partition in Figure 5.7(b).

Complex Tradeoffs among Kernels. Either tensor- or matrix-related kernels can dominate in CPD depends on the size, sparsity, and nonzero distribution of the tensor. The execution time of tensor-related kernels is a function of M, I_l, and I_p while the time of matrix-related kernels is a function of R and I_p from Table 5.1. MTTKRP dominates for a relatively small and dense matrix where $I_p \times R < M/P$. However, either types of kernels could dominate, depending on the nonzero distribution of a large and sparse matrix. Matrix-related kernels are more likely to dominate for very sparse large tensors.

Finding 6: *There is no execution overlap between computation and communication.* In Algorithm 2, the steps of computation and communication are executed sequentially such that every step waits for the complete results from the prior step before executing. However, this is not necessary for some steps like MTTKRP and the *AlltoAll* used to update $\tilde{\mathbf{A}}_p^{(n)}$ and MAT $A^T A$.

By using point-to-point non-blocking communication, it is possible to transfer partial results from MTTKRP and matrix multiplication in pipeline. Thus, some computation is overlapped with the communication to reduce the overall execution time of CPD.

5.2 Irregularity-Aware Algorithm for Workload Partition

The findings above motivate our optimizations in considering different tensor irregularities and finding the optimal grid configuration $\mathcal{G}$ and distribution policies $\mathcal{D}$ to improve runtime performance. This subsection presents our proposed irregularity-aware CPD. We propose new methods for grid configuration and distribution policy, and the implementations of them are detailed in Algorithms 4 and 5.

5.2.1 Prediction-Based Grid Configuration

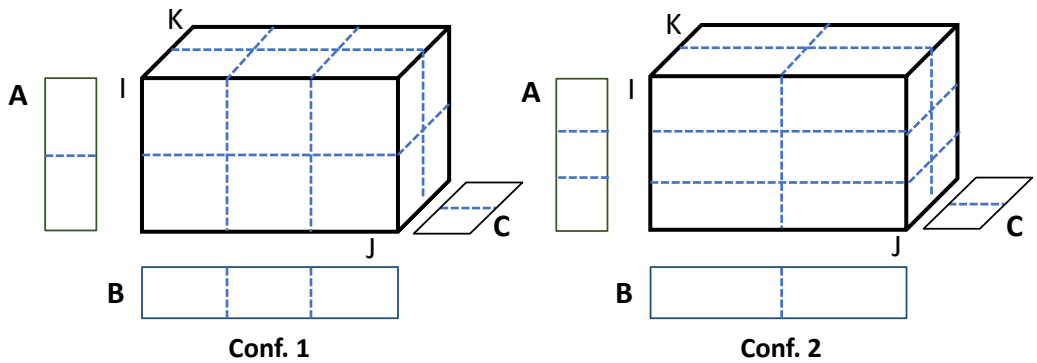


Figure 5.5: Two example grid configurations for 12 processes.

It is important to find the optimal process grid because the performance varies a lot between different grid configurations, based on our Finding 4 in Section 5.1. Figure 5.5 compares two example grid configurations: $2 \times 3 \times 2$ and $2 \times 2 \times 3$. In Conf. 1, tensor $\mathcal{X}$ is split to two pieces in mode-I and three pieces in mode-J; Conf. 2 is the opposite. Distribution on mode-K is the same. Assume $J > K$, ostensibly, Conf. 1 should be more reasonable than Conf. 2 by splitting the larger dimension. For a dense tensor $\mathcal{X}$, this is true. The different matrix distribution on $\mathbf{A}$ and $\mathbf{B}$ could lead to uneven matrix communications, thus influence overall CPD performance. We prove this using a dense, cubical third-order tensor $\mathcal{X} \in \mathbb{R}^{I \times I \times I}$ along with three matrices $\mathbf{A}^{(n)} \in \mathbb{R}^{I \times R}, n = 1, 2, 3,$

distributed on $P = P_1 \times P_2 \times P_3$. From Algorithm 2, the data to be communicated is dominated by $\tilde{\mathbf{A}}_l^{(n)} - \tilde{\mathbf{A}}_p^{(n)}$ and $\tilde{\mathbf{A}}_p^{(n)}$ to communicate in its own layer. For each inside loop, its communication volume in the first dimension is $P_2 P_3 (\frac{I}{P} + (\frac{I}{P_1} - \frac{I}{P})) = P \frac{I}{P_1^2}$. Thus, the total volume of CPD in all dimensions is

$$VOL_{comm} = I \times P \times (\frac{1}{P_1^2} + \frac{1}{P_2^2} + \frac{1}{P_3^2}). \tag{5.3}$$

According to Cauchy-Schwarz inequality, the minimum of the total volume is obtained when $P_1 = P_2 = P_3$. For a cubical dense tensor, equally split the dimension sizes obtains the minimum communication cost. For a tensor with an irregular shape, we proportionally assign more processes to a longer dimension to maintain the minimum communication. The state-of-the-art work [114, 71] developed an easy-to-use prediction algorithm based on the above idea. It assigns the number of processes based on the tensor dimension sizes. However, for irregular sparse tensors with a non-uniform nonzero distribution, their method leads to a severe imbalance for computation and communication.

To solve their problem, we propose a new online prediction algorithm that simultaneously considers communication volume and nonzero balance when deciding the process grid. Our key idea is to find a process grid with the smallest nonzero imbalance from a mode-balanced base. We have two steps to achieve the above goal. First, we build an intermediate process grid that leads to balanced communication and matrix computations based on the existing work [114, 71]. This intermediate grid uses most but not all the processes. Second, we construct the grid candidates by adjusting the intermediate grid with the remaining process(es) and predict the optimal grid among them. Prediction on a virtual data distribution is leveraged to make a balance among the imbalance ratios in Section 5.1.1.2.

Algorithm 4 illustrates our method. Our goal in the first step is to form an intermediate grid as a base of all candidates. The brute-force results indicate those girds with better performance are more likely to share the same base. For example, 4 of the top 5 grids has the base of $4 \times 1 \times 1$ in Figure 5.4. Therefore, we need to build this balanced base first. To form this intermediate grid, we first find all the prime factors of the total process count and sort them in descending order in prs_o. Using all but the last n_p factors, we form an intermediate grid $\mathcal{G}_{int}$ (Line 8). For example, $n_p = 1$ indicates the smallest prime factor is unused in the intermediate grid. Specifically, it repeatedly assigns the largest prime factor to the current longest tensor dimension, which dynamically changes after each loop iteration. After the loop ends, the intermediate grid $\mathcal{G}_{int}$ has the best effort in

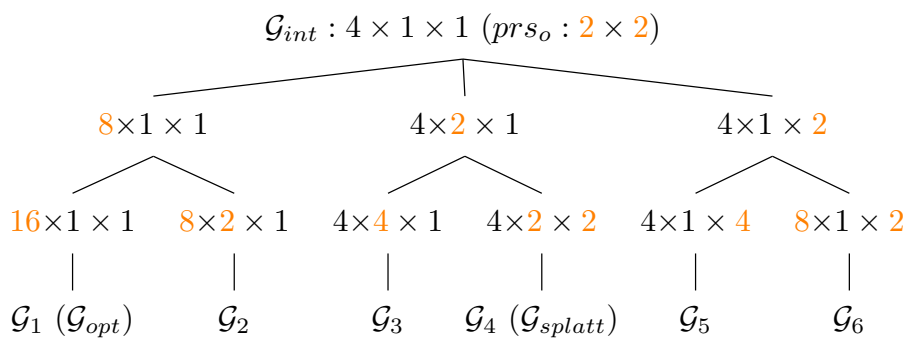


Figure 5.6: Six grid candidates on 16 (=$2 \times 2 \times 2 \times 2$) processes for tensor *amazon* with $n_p = 2$. We assign two smallest primes (2×2) to $\mathcal{G}_{int}$ and obtain six grid candidates.

balancing communication and matrix computations. We assign the remaining n_p primes to form a complete process gird in the following step.

The key idea in the second step is to build all possible candidates and identify the optimal grid among them by predicting their nonzero imbalance. We form six grid candidates from $\mathcal{G}_1$ to $\mathcal{G}_6$ with $\mathcal{G}_{int}$ by assigning two smallest primes to each dimension. Figure 5.6 displays how we form all candidates from $\mathcal{G}_{int}$ for tensor *amazon* with 16 MPI processes. The first step, build $\mathcal{G}_{int}$ as $4 \times 1 \times 1$ based on *amazon*'s dimension size as $4.8M \times 1.8M \times 1.8M$. We build six candidates after assigning the remaining n_p primes 2×2. These six candidates are considered having an equal chance to obtain the optimal performance from the first step with tensor dimension size and implied communication information. To identify the optimal grid among them, we need to predict the nonzero imbalance ratio r_{nnz} for each candidate on a virtual data distribution. If we want to compute the actual r_{nnz} with the nonzeros of each process (M_p) as stated in Section 5.1.1.2, we need to take the tensor slice information to determine the index range of each process. However, the above computation of r_{nnz} in a virtual distribution has a complexity of $\mathcal{O}(c_N \times P \times M_p)$ for each candidate. This is expensive for tensors with large amounts of nonzeros. We present a new metric r_{layer_nnz} as the imbalance radio of nonzeros among different layers to predict r_{nnz}. Figure 2.4 displays the layers affiliated with the tensor partition. In mode I_2 there are three layers each with 4 subtensors. Particularly, we take the tensor slice information to compute the nonzeros of each layer Ln_p in each mode. In mode I_n, $r_{layer\_nnz}(n) = (max\{Ln_p\} - min\{Ln_p\})/max\{Ln_p\}$. We then compute r_{layer_nnz} as the average $r_{layer\_nnz}(n)$ for all modes. The total complexity is $\mathcal{O}(c_N \times I_n)$. $r_{layer\_nnz} = r_{nnz} = 0$ in a dense

Algorithm 4: Prediction-based grid configuration with $n_p = 2$.

Require: Number of processes P, tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times I_3}$;
Ensure: Grid configuration $\mathcal{G}_{opt} = \{P_1, P_2, P_3\}$, $P_1 \times P_2 \times P_3 = P$;

```
Initialize intermediate grid G_int = {1, 1, 1}
// Step 1: intermediate grid generation
prs_o = getPrimes(P);                                  ▷ Ordered from large to small
I_avg = (I_1 + I_2 + I_3)/3
for pr in prs_o[1 : −1] do
    G_int[n]* = pr, s.t. I_n = max{I_1, I_2, I_3}
    I_n − = I_avg
end for
// Step 2: sparsity-aware grid trimming
Initialize six grid candidates G_1, ..., G_6 = G_int
G_i* = (prs_o[−2] * prs_o[−1]), i = {1, 2, 3, 4, 5, 6}   ▷ Assign two smallest primes to six candidates
Compute r_layer_nnz to predict r_nnz of G_1, ..., G_6 with virtual data distribution
G_opt = G_i, s.t. min_{r_nnz}{G_1, ..., G_6}
Return G_opt;
```

tensor or a sparse tensor with an even nonzero distribution. In a sparse tensor with an imbalanced nonzero distribution, r_{layer_nnz} is able to predict r_{nnz} by considering several subtensors as a group. Therefore, compared to r_{nnz}, r_{layer_nnz} can capture the imbalance of nonzero distribution by a low-cost estimation. Finally, we select the grid candidates with the best nonzero balance as the optimal grid $\mathcal{G}_{opt}$. Figure 5.6 shows that Algorithm 4 predict the optimal grid as $16 \times 1 \times 1$ as with smallest r_{layer_nnz} for tensor *amazon* on 16 MPI processes. And Figure 5.4 indicates that our $\mathcal{G}_{opt}$ has a better performance than $\mathcal{G}_{splatt}$ built from SPLATT's grid configuration. The selected grid configuration is used for the following distribution policy and CPD computation.

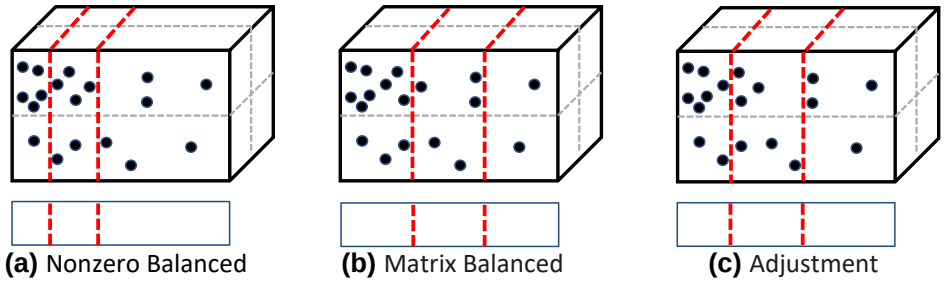


Figure 5.7: Distribution policy on 12(=2 × 3 × 2) processes. Layer boundaries in red are adjusted in the 2nd dimension; boundaries in gray are fixed and in the 1st and 3rd dimensions.

Algorithm 5: Matrix-oriented distribution policy generation in tensor dimension n.

Require: Sparse tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times I_3}$, number of processes P_n in dimension-n;
Ensure: Distribution policy $\mathcal{D}$ (a.k.a. layer configuration $\{I_L\}$);

```
1: // Matrix-balancing strategy: set
2: for i in P_n do
3:     I_{L_i} = I_n / P_n;                                   ▷ Initial layer size
4: end for
5: if Ordered adjustment then
6:     // Ordered adjustment strategy: ordered-c
7:     for i in P_n do
8:         m_i = #nonzeros in layer L_i                       ▷ c is a user-given parameter
9:         I_{L_i} -= (m_i − M/P_n)/(c . S_{L_i})
10:    end for
11: else if Max-to-min adjustment then
12:    // Max-min adjustment strategy: max-min
13:    I'_L: Sorted {I_{L_i}, i = 1, ..., P_n} by #nonzeros in a descending order
14:    for i in P_n/2 do
15:        I'_L[i] −= (I'_L[i] − I'_L[P_n − i])/S_n
16:        I'_L[P_n − i] += (I'_L[i] − I'_L[P_n − i])/S_n
17:    end for
18:    I'_L = I_L
19: end if
20: Return D = {I_L};
```

5.2.2 Matrix-Oriented Distribution Policy

Once we decide on a process grid, the next challenge is to choose a distribution policy which leads to an optimal partitioning of the tensor and matrices, and balanced computation and communication and their trade-offs among the processes. Thus, three parameters M_p, I_p, and I_l in Table 5.1 are influenced by a distribution policy $\mathcal{D}$. The optimal strategies effectively eliminate performance bottlenecks, resulting in balanced computation and communication and their trade-offs.

The state-of-the-art work [114] takes a strategy that balances nonzero computation by evenly partitioning tensor nonzeros among the processes, shown in Figure 5.7(a). It only considers M_p and targets to minimize r_{nnz}. Thus, it is advantageous for CPD dominated by the sparse tensor computation kernel MTTKRP. In general, such tensors have moderate sparsity and uniform nonzero distributions along the dimensions. Nevertheless, this strategy may not be beneficial for irregular tensors. For example, tensor *fb-m* has one dimension size multiple orders-of-magnitude smaller than the others, and its nonzeros mainly reside along a diagonal with increasing density while most nonzeros concentrate at a bottom corner. Applying the nonzero balancing strategy to such tensors results in severe imbalances, in all aspects including nonzero computation, matrix computations,

and communication (See Figure 5.16). Furthermore, CPD on some tensors under study do not benefit from balanced nonzero computation as the execution is dominated by communication or matrix computations in Figures 5.1 and 5.2. We leverage the sparsity and irregularity information that reflects in matrix computation imbalance. We identify the dominant imbalance ratio as matrix computation imbalance for irregular tensors. Therefore, all our strategies are based on balancing matrix computations and then achieve the best trade-offs between different imbalances.

To balance matrix computations, we first propose an easy-to-use *set* strategy that balances I_p by evenly partitioning matrices among the processes in every dimension, shown as Figure 5.7(b). This results in minimal r_{I_p} and balanced matrix computation, but could exacerbate the imbalance for nonzero computation. *Set* strategy is advantageous for CPD dominated by matrix computations, typically very sparse tensors with a uniformed distribution of nonzeros, and could tolerate irregular tensor dimension sizes. On the other hand, applying the matrix balancing strategy improves the balance for matrix computations and communication but exacerbates the imbalance for nonzero computation. Yet, neither of these two strategies works well for irregular sparse tensors like *fb-m*, because they target to minimize only one imbalance ratio, either r_{nnz} or r_{I_p}, without considering the trade-offs among the three ratios counting r_{vol} for communication.

The challenge for irregular sparse tensors is extremely high imbalance in both computation and communication, as our Finding 3 in Section 5.1 shows. Focusing only on optimizing one of the imbalance ratios might cause the other two ratios higher. To support irregular sparse tensors, we propose new distribution policies to achieve better trade-offs between these imbalance ratios. *Our proposed distribution policies begin with the matrix-balancing strategy, but adjust according to the nonzero-balancing strategy*, illustrated in Figure 5.7(c) where red lines are shifted based on (b) but not as skewed as (a). Algorithm 5 shows the three generation strategies of distribution policies: *set*, *ordered-c*, and *max-min*. From Figure 5.7, a distribution policy is a layer configuration and represented by $\{I_L\}$, an array of dimension sizes distributed to each process which sum up to the dimension size in dimension-n. Assume the processor grid is $P = P_1 \times P_2 \times \cdots \times P_N$. We first employ *set* strategy by partitioning I_n/P_n consecutive slices of $\mathcal{X}$ to each process in dimension n, yielding balanced matrix computations but potentially skewed nonzeros among processes. We then adjust layer boundaries to mitigate nonzero imbalance using either *ordered-c* or *max-min* strategies.

The key idea of the *ordered-c* strategy is to reduce nonzero imbalance of each partition independently. It adjusts layer boundaries along with the index in each tensor dimension. To

achieve this, we first calculate M/P_n as the target nonzero size of each partition in dimension-n, and add/remove slices if the nonzeros in a partition are greater/less than the target size. Second, we need to move layer boundaries to make nonzeros in each partition closer to the target nonzero size M/P_n. Assume the current nonzeros in the ith partition is m_i and the average number of nonzeros for one slice in this partition is S_{L_i}, then the number of slices to be adjusted is given by $(m_i - M/P_n)/(c \cdot S_{L_i})$, where c is a user-given integer. The larger the c value, the finer the adjustment granularity. Partitioning with $c = 1$ is the same as SPLATT for dense tensors or sparse tensors with uniformed nonzero distribution. With larger c, our partitioning keeps more balanced I_P rather than nonzeros for irregular sparse tensor. When c is extremely large, the *ordered-c* strategy has little difference with the *set* strategy, as it has little adjustment. Therefore, we set c as 1 or 2 to achieve better trade-offs between nonzero and I_P imbalance, and distinguish with the *set* strategy.

Instead of adjusting each partition independently, the key idea of the *Max-min* method is to balance nonzeros in partitions based on the differences between them. There is no target nonzero size in this strategy. It moves slices from partitions with the maximal nonzeros to the ones with the minimal nonzeros. We first sort the layer configuration I_L in descending order and save it as I'_L. By looping the first half of I'_L, the max-min pair is $I'_L[i]$, $I'_L[P_n - i]$ respectively. Second, we adjust the layer boundaries of each max-min pair. Let S_n be the average number of nonzeros per slice for all partitions of dimension-n. The number of slices to be adjusted is $(I'_L[i] - I'_L[P_n - i])/S_n$. *Max-min* adjusts only the maximal and minimal nonzero partitions, but might be less accurate in partitioning nonzeros by considering the global slice information with S_n among partitions rather than the local S_{L_i} within a partition. As each partition must contain continuous slices, this method might involve adjusting the boundaries of all partitions. Therefore, we expect lower performance than the first method, but it still outperforms the nonzero-balancing strategy for irregular tensors.

Our proposed prediction-based grid configuration and matrix-oriented distribution policy are directly applied to medium-grained, bulk-synchronous distributed CPD (Algorithm 2) as Lines 2 and 3 separately, to gain performance improvement and better scalability.

5.3 The BA-CPD Algorithm

In this section, we describe our BA-CPD distributed tensor decomposition algorithm with asynchronous CPD implementation. The key idea of our asynchronous algorithms is to take ad-

vantages of RDMA-based one-sided communication in CPD. RDMA allows processes to arbitrarily access data from shared memory of other processes. However, simply changing collective communication to RDMA operations will not make many benefits in CPD because computation and communication are in a sequential order in each kernel, where every step waits for the complete results from the prior step before executing. We need to overlap some computation with communication to obtain better performance of RDMA operations. Thus, we perform on fine-grain decoupling of computation and communication and optimization for different kernels. The main challenge of this optimization is that each kernel has different computation and communication patterns. Some kernels worth fine-grain decoupling, but others do not. For example, there is no need to optimize MAT SOLVE for zero communication. In this decoupling, we also need to make sure that none of communication operations depends on explicit synchronization with any other process. We design two strategies as BA-1D and BA-2D.

5.3.1 BA-1D Strategy

Algorithm 6: Balanced Asynchronous CPD algorithm.

```
   ...
   // CPD Computation
1: U_n = Async (A_p^{(n)T} A_p^{(n)})
2: do
3:     for n = 1, ..., N do
4:         Ã_p^{(n)} = MTTKRP(𝒳_p, A_p^{(1)}, ..., A_p^{(n-1)}, A_p^{(n+1)}, ..., A_p^{(N)})   ▷ MTTKRP
5:         Ã_p^{(n)} = Ã_p^{(n)} (U_1 * ··· * U_N)^†                                          ▷ MAT SOLVE
6:         λ̃ = Normalize (Ã_p^{(n)})                                                          ▷ MAT NORM
7:         Ũ_n = Async(Ã_p^{(n)T} Ã_p^{(n)})                                                   ▷ MAT A^T A
8:     end for
9: while fit not change or maximum iterations exhausted
```

Partition. In BA-1D algorithm, we use N-D partitionings for tensor distribution while 1D partitionings for matrix distribution, as Figure 5.8 shows. There are two major changes of BA-CPD algorithm in Algorithm 6 compared to Algorithm 2: First, all collective communication is now implemented with asynchronous communication. Second, we do not store $\mathbf{A}_l$ for MTTKRP and thus there is no MAT UPDATE kernel. The original communication for MTTKRP in Algorithm 2 (Line 12 and 16) are now implicitly in the MTTKRP asynchronous implementation. The computation and communication volume for each process is similar to those in Table 5.1. While I_l and I_p are balanced

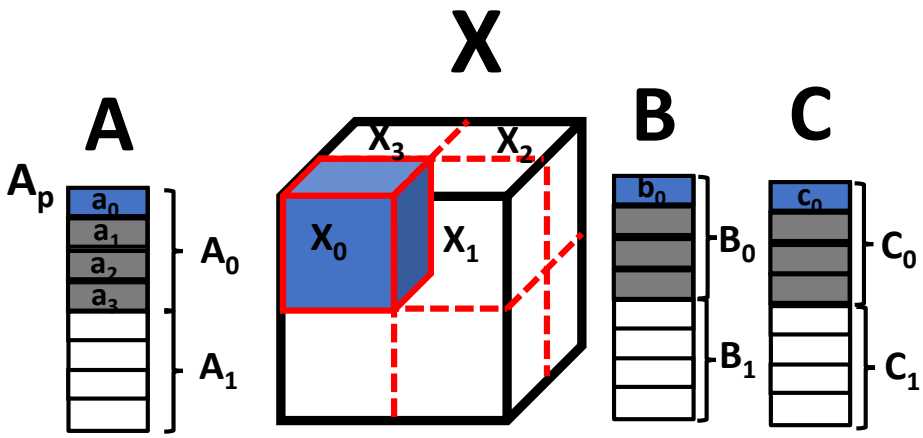


Figure 5.8: Tensor and matrix partitions in BA-1D algorithm, where P(0,0,0) locally stores subtensor and submatrices in blue and needs communication with submatrices in gray.

and, so are local computation of matrix-related kernels.

Listing 5.1: Optimized MTTKRP in BA-1D

```
for i in 0...I-1:
 for j in 0...J-1:
  for k in 0...K-1:
   for q in (0...layer_size-1 and q!=rank()):
     local_b = B.get_tile();
     local_c = C.get_tile(); //retrieves matrix tiles on demand
     buff_a = local_x * local_b * local_c; //partial results for comm
     buff_a.put();
   local_b = B.get_tile();
   local_c = C.get_tile();
   local_a = local_x * local_b * local_c; //result in local A
   for q in (0...layer_size-1 and q!=rank()):
     local_a += buff_a;
```

Mttkrp. We decouple computation and communication in q (q is the number of processes in the same layer) stages for each process. In each stage, each process first retrieves tiles of the matrix on demand, and then performs on partial local MTTKRP followed by sending it. In the final stage, each process performs it own local computation and then sums up all partial results received from other

processes. From Stage 0 to $q-2$, $P(0)$ in Figure 5.8 computes a portion of its local nonzeros in MTTKRP computation. This portion of nonzero corresponds to the matrix $\mathbf{A}$ update, which needs to be sent to one of the other processes in the same layer ($P(1)$ to $P(3)$). After the local computation $P(0)$ sends the partial results to one of the other processes via one-sided asynchronous operations in each stage. In the last Stage $q-1$, the rest computation is completed to update the local matrix portion of $\mathbf{A}$. And $P(0)$ then sums up all partial results received from other processes. In this procedure, We overlap the local $\mathbf{A}$ computation and communication for each process in the same layer. Its effectiveness depends on the number of nonzeros involved in MTTKRP computation and the transferred matrix volume.

MAT NORM. The computation in MAT NORM is column normalization of dense-matrix locally stored in each process. We do not decouple computation here as it will increase the communication volume. After local computation, each process asynchronous sends the normalized vector to all other processes. Each process then sums up the local normalized vector with those from all other processes.

MAT ATA. In ATA, each process performs symmetric matrix multiplication locally and then asynchronous sends this local result to all other processes. Each process then sums up partial results from all other processes.

Strengths and Limitations BA-1D is a natural extension from the medium-grained decomposition. It keeps the advantages from medium-grained decomposition and provides opportunities to further optimize MTTKRP. It can achieve better performance for tensors that MTTKRP has dominant overhead in CPD. However, it still has a limitation that matrix-related kernels are not best optimized.

5.3.2 BA-2D Strategy

Partition. In BA-2D algorithms, we use N-D partitionings for tensor distribution while 2D partitionings for matrix distribution, as Figure 5.9 shows. The main stages of BA-2D algorithm are similar to those in Algorithm 6. The difference is that $\mathbf{A}_p$ is column-wise distributed in each layer now. 2D partitionings for matrix also lead to different local computation and communication, as Table 5.2 shows, where $layer\_size$ is the num of processes in one layer.

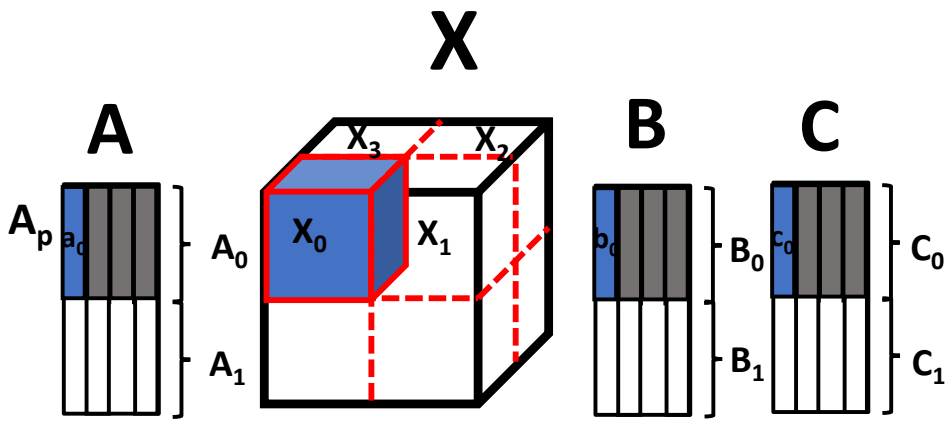


Figure 5.9: Tensor and matrix partitions in BA-2D algorithm, where P(0,0,0) locally stores subtensor and submatrices in blue and needs communication with submatrices in gray.

Key Steps	Local Comp	Local Comm Volume
MAT NORM	$\Theta(R/layer\_size \times I_l)$	$R/layer\_size$
MAT A^TA	$\Theta\big(R^2/layer\_size \times I_l\big)$	R^2

Table 5.2: Time complexity of steps changed in BA-2D.

Mttkrp. In MTTKRP each process performs on local computation and then decouples communication in q (q is the number of processes in the same layer) stages. In each stage, each process sends partial result local MTTKRP to other processes in the same layer, and then sums up all partial results received from other processes. After local MTTKRP, from Stage 0 to $q-2$, $P(0)$ in Figure 5.9 prepares partial results needing to be sent to one of the other processes in the same layer ($P(1)$,$P(2)$,$P(3)$), and then sends the partial results to one of the other processes via one-sided asynchronous operations in each stage. $P(0)$ then sums up all partial results received from other processes.

MAT NORM. In MAT NORM, we do not decouple computation for the same reason in BA-1D strategy. Each process computes a local normalization and then asynchronous sends the result to processes in the same column on other layers. Each process then sums up all received local normalized vectors. $P(0)$ in Figure 5.9 computes a local normalization and then asynchronous sends the result to the process in the column ($P(4)$). $P(0)$ then sums up the local normalized vector with those from $P(4)$.

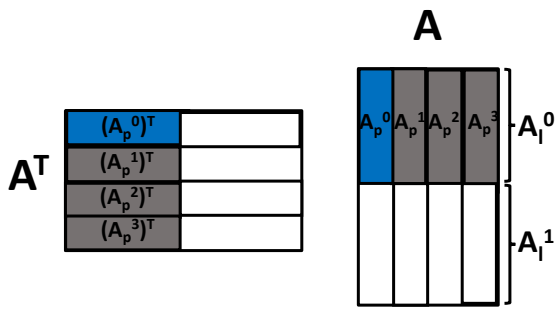


Figure 5.10: ATA in SX-2D algorithm.

ATA. We decouple computation and communication in q (q is the number of processes in the same layer) stages. In each stage, each process first retrieves tiles of the matrix on demand, and then performs on partial local ATA followed by sending it to processes in other layers. Each process then sums up all received partial results. In the ith stage $P(0)$ computes R/q rows of matrix multiplication locally in the form of $\tilde{\mathbf{A}}_p^{(i)T}\tilde{\mathbf{A}}_l^{(0)}$ as Figure 5.10 shows. And $P(0)$ then asynchronous sends the result to all processes in other layers. Finally, $P(0)$ sums up partial results from all processes in other layers to obtain the result of ATA.

Listing 5.2: Optimized ATA in BA-2D

```
for i in 0...P-1:
    Block_A = A.get_block(); //retrieves matrix tiles on demand
    for q in 0...layer_size-1:
        local_at = A.get();
        local_ATA(q,:) = ATA(Block_A,local_at); //partial ATA
        buff_a = local_ATA(q,:);
        buff_a.put();
        local_ATA += buff_a;
```

Strengths and Limitations BA-2D uses a 2D partitioning of matrix and provides opportunities for fine-grain decoupling of computation and communication for matrix-related kernels. It can achieve better performance for tensors, those matrix-related kernels have dominant overhead in CPD. However, it has a limitation of coarser optimization on MTTKRP compared to BA-1D algorithm.

Tensors	Dimensions	#Nonzeros	Density
stac	$545K \times 96K \times 1.2K$	1.3M	2.1×10^{-8}
choa	$712K \times 10K \times 767$	27M	5.0×10^{-6}
darpa	$22K \times 22K \times 24M$	28M	2.4×10^{-9}
nell2	$12K \times 9K \times 29K$	77M	2.4×10^{-5}
fb-m	$23M \times 23M \times 166$	100M	1.1×10^{-9}
flic	$319K \times 28M \times 1.6M$	113M	7.8×10^{-12}
fb-s	$39M \times 39M \times 532$	140M	1.7×10^{-10}
deli	$533K \times 17M \times 2.5M$	140M	6.1×10^{-12}
nell1	$2.9M \times 2.1M \times 25M$	144M	9.1×10^{-13}
amazon	$4.8M \times 1.8M \times 1.8M$	1742M	1.1×10^{-10}
patents	$46 \times 239K \times 239K$	3597M	1.4×10^{-3}

Table 5.3: Description of sparse tensors.

5.4 Experimental Results

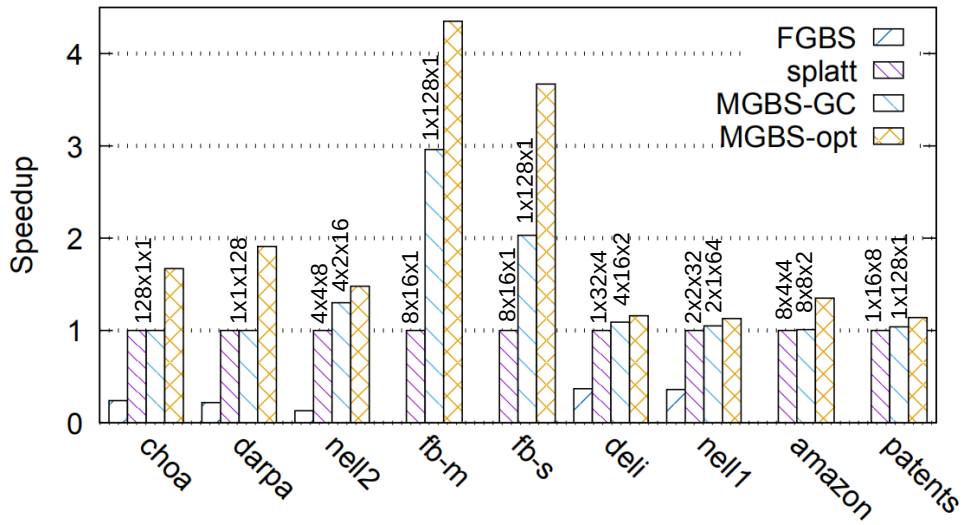
Platform. We perform experiments on the Constance cluster at the Pacific Northwest National Laboratory; each node has 2×12-core Intel Xeon CPU E5-2670 v3 CPUs. The Constance system has 520 2×12-core nodes (totaling 12480 cores), 64GB DDR4 memory per node on a 56Gb/s FDR Infiniband interconnect. We use up to a total number of 1536 cores, with 128 nodes and 12 cores/node, GCC 7.3.0 and OpenMPI 4.0.1 as compilers. Our experiments consume 25% of the whole system. The default BLAS and LAPACK libraries v3.2.1 on Linux are used for the dense matrix routines.

Dataset. We evaluate sparse tensors from real-world applications in Table 5.3, ordered by increasing number of nonzeros. Most of these tensors are from the Formidable Repository of Open Sparse Tensors and Tools (FROSTT) [113]. The *stac* is from Koblenz Network Collection [76]. The *darpa* (source IP-destination IP-time triples), *fb-m*, and *fb-s* (entity-entity-relation triples) are from HaTen2 [68], and *choa* (patient-visit-time triples) is built from electronic health records (EHRs) [99].

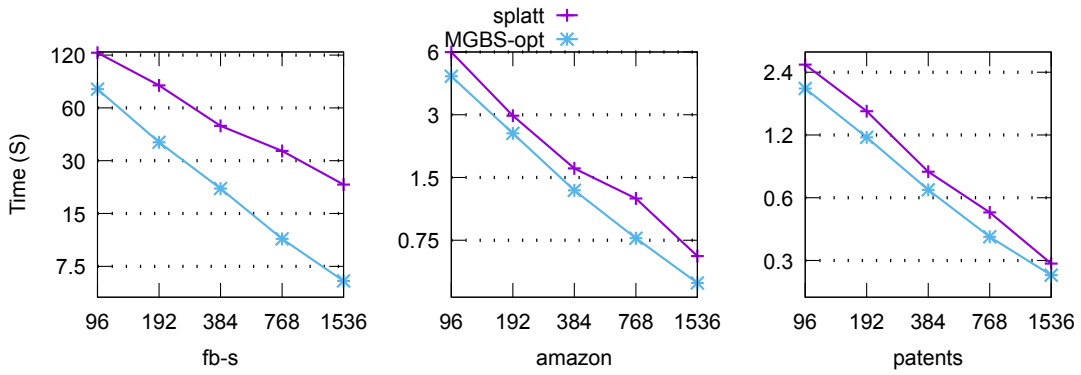
Baseline. We use Splatt as our baseline, representing a medium-grained, bulk-synchronous distributed Cpd [114][3], which is generally considered faster than MapReduce implementations [70, 68]. We also compare to the fine-grained distributed Cpd algorithm (represented as FGBS) from HyperTensor [72] [4]. We implement our irregularity-aware algorithm for workload partition, represented as MGBS-opt or BS-opt. We then implement and evaluate our asynchronous algorithms as BA-1D and BA-2D. Both medium- and fine-grained Cpd are hybrid MPI+OpenMP parallelized. We use 12

[3]ENSIGN [77] is a closed-sourced, commercial library and CarHP [6] is not open-sourced.
[4]Implemented in Splatt as its open-source version.

threads (referred to as processors uniformly) for each CPU for all experiments and set $R = 32$ as using a different R has no impact on our evaluation. All experiments use single-precision floating point values, and the average execution time of 5 iterations is reported. Due to the CPD execution time variance on different tensors, we normalize the time of other implementations to medium-grained SPLATT.



(a) Overall performance speedup for CPD on 1536 processors.



(b) Strong scalability from 96 to 1536 processors.

Figure 5.11: Overall performance comparison and scalability.

5.4.1 Overall Performance

Figure 5.11(a) shows the speedup of our distributed CPD (MGBS-opt) compared to medium-grained (SPLATT) and fine-grained (FGBS) CPD when using 1536 processors. The speedup over

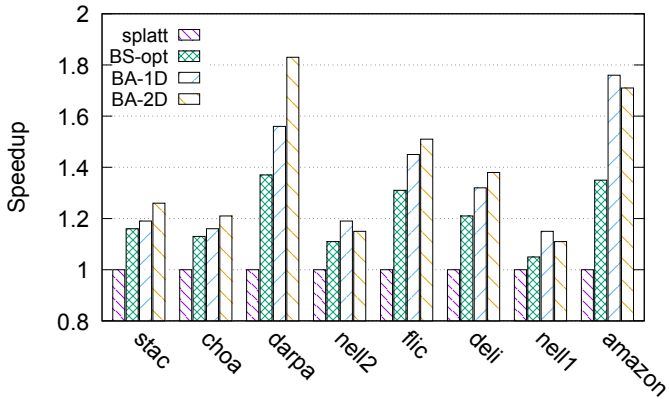
SPLATT ranges from 1.2× to 4.4× for all nine tensors. The two irregular tensors, *fb-m* and *fb-s*, benefit the most from our methods because they suffer severe r_{nnz}, r_{vol}, and r_{Ip} imbalance in prior implementations (see Figure 5.16). Relatively small sparse tensors like *choa*, *darpa*, and *nell2* have a speedup from 1.5× to 1.7×. Other tensors such as *deli* and *amazon* gain a speedup from 1.2× to 1.4× from our methods, even though they have decent balances with SPLATT.

Comparing to fine-grained distributed CPD (FGBS) with hypergraph partitioning generated by Zoltan [25], MGBS-opt always performs better by $3.1 - 11.4\times$. The missing bars on large and/or irregular tensors, *amazon*, *patents*, *fb-m*, and *fb-s* are due to failures of generating hypergraph partitions by Zoltan on 1536 processors. We observe that SPLATT achieves higher performance than FGBS on all cases, aligned with the work [114].

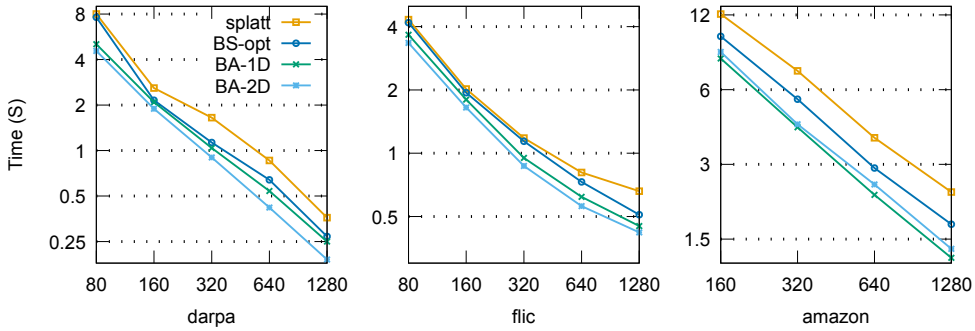
Figure 5.11(a) also presents the performance effect of our prediction-based grid configuration (Algorithm 4) as MGBS-GC. By comparing SPLATT, MGBS-GC, and MGBS-opt, we see the incremental performance from our optimizations. The prediction-based grid configuration and matrix-oriented distribution policy increase the performance by $0 - 296\%$ and $7 - 91\%$ separately. The labels on top of SPLATT and MGBS-GC bars show their chosen process grids. MGBS-GC and SPLATT obtain the same grid thus lead to the same performance on *choa* and *darpa*. Our prediction-based grid configuration accelerates performance for 7 out of 9 tensors. Tensor *fb-m* gets the highest gain at 2.96× with a better grid configuration. These results verify that irregularity-aware grid configuration is critical to CPD performance.

Figure 5.11(b) demonstrates that MGBS-opt obtains better strong scalability than SPLATT on three large tensors from 96 to 1536 processors. MGBS-opt shows significantly better scalability than SPLATT on irregular yet sparse tensor *fb-s*. This is because r_{Ip} that impacts matrix computation and communication time reduces significantly in MGBS-opt. Detailed profiling shows that both communication and computation time are closed to be halved as the number of processors doubles in MGBS-opt. MGBS-opt scales slightly better for matrix computation and communication on tensors *amazon* and *patents*, where MTTKRP occupies a larger time percentage. For other tensors: *fb-m* shows similar scalability to *fb-s*; *deli* and *nell1* are similar to *patents*; both SPLATT and MGBS-opt show good scalability on small tensors *choa*, *darpa*, and *nell2*.

Figure 5.12(a) shows the speedup of our BS-opt and BA methods compared to SPLATT when using 1280 processors. The best speedup of our methods over SPLATT ranges from 1.2× to 1.8× for all eight tensors. BA-1D gains the best speedup in 3 of 8 tensors and BA-2D gains the best speedup



(a) Overall performance speedup for CPD on 1280 processors.



(b) Strong scalability from 80 to 1280 processors.

Figure 5.12: Overall performance comparison and scalability.

on 5 of 8 tensors. BA-1D performs better in tensors those have dominant overhead in MTTKRP as it focuses more on optimizing it. Figure 5.12(a) also indicates the impact of our workload partitioning on bulk-synchronous CPD as BS-opt. The speedup of BS-opt over SPLATT ranges from 1.1× to 1.4× for all eight tensors.

Figure 5.12(b) shows that our methods obtain better strong scalability than SPLATT on three different tensors from 80 to 1280 processors. Our BA-CPD shows significantly better scalability than SPLATT on irregular yet sparse tensor like *darpa* and *flic*. This is because BA-CPD balances I_p much better and reduces more communication cost than SPLATT in matrix-related kernels, those are dominant in irregular tensors. For relatively regular tensor *amazon*, both SPLATT and BA-CPD gain good scalability. But BA-CPD always shows better performance as it achieves a better balance of computation and communication in MTTKRP that is dominant in *amazon*.

5.4.2 Detailed Analysis

Tensors	Splatt imb(nnz)	Splatt imb(comm)	Irregularity-aware Cpd imb(nnz)	Irregularity-aware Cpd imb(comm)
stac	97.9%	93.3%	**88.6%**	**80.4%**
choa	3.0%	87.5%	23.2%	**2.9%**
darpa	0.0%	98.7%	20.4%	**69.0%**
nell2	2.7%	44.7%	23.6%	**6.5%**
flic	24.9%	84.2%	**23.1**%	**47.5%**
deli	10.2%	29.2%	**9.6%**	**1.3%**
nell1	1.2%	36.9%	9.3%	**2.4%**
amazon	12.1%	58.3%	**5.5%**	**11.2%**

Table 5.4: Imbalance of tensor nonzero and communication volume.

Workload Partitioning Analysis. Table 5.4 shows the imbalance of tensor nonzero and communication volume among processes with workload partition in BA-CPD compared to Splatt. As we discussed in Section 3.1, Splatt focuses only on balancing tensor nonzero for all tensors. However, for irregular tensor the imbalance of communication volume is much more severe than that of tensor nonzero, such as *darpa* and *flic*. BA-CPD optimizes both tensor nonzero and communication volume and significantly reduces the imbalance of communication. BA-CPD gains more balanced nonzero in 4 of 8 tensors and more balanced communication in all tensors. The results of Table 5.4 and Figure 5.12(a) demonstrate that balancing only tensor nonzero leads to suboptimal performance, and trade-offs are required among nonzero and communication volume.

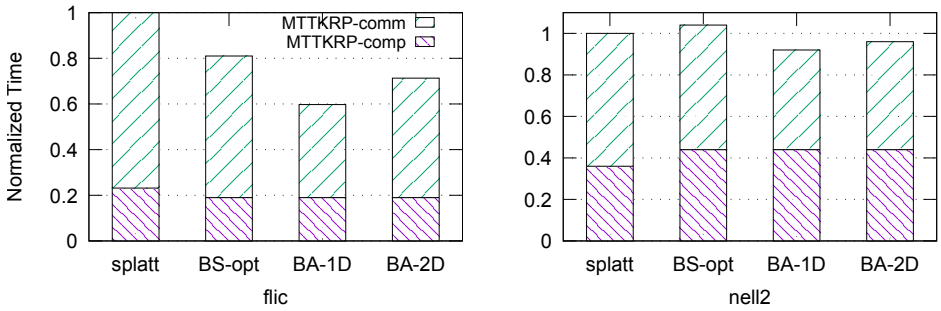


Figure 5.13: Time percentage of MTTKRP for flic and deli on 1280 processors.

Mttkrp Kernel Analysis. Figure 5.13 shows detailed time percentages of communication and computation in Mttkrp. BS-opt optimizes both computation and communication in *flic* by our workload partitioning in *flic*. While it sightly increases computation overhead in *nell*2 by larger imbalance of tensor nonzero as Table 5.4 shows. Among our three methods, BA-1D always performs

the best, as it enables better computation and communication overlap in MTTKRP by fine-grained decoupling. BA-2D performs better than BS-opt by utilizing asynchronous communication. BA-2D performs worse than BA-1D in MTTKRP as its 2D partition focuses more on optimizing matrix-related kernels.

Detailed Performance Analysis. We show how our methods optimize each kernel of CPD and gain a better trade-off among them for tensors *darpa* and *amazon* in Figure 5.14. For *darpa*, BS-opt slightly increases MTTKRP-comp but optimizes MTTKRP-comm and matrix-related kernels by our workload partition. Based on the partition, BA-1D further optimizes MTTKRP-comm by fine-grained decoupling of computation and communication in MTTKRP. BA-2D performs the best as its 2D partition focuses more on optimizing matrix-related kernels, those are more dominant for *darpa*. For *amazon*, our three methods show similar behaviors as *darpa* except that BA-1D performs the best. This is because it focuses more on optimizing MTTKRP that is more dominant for *amazon*.

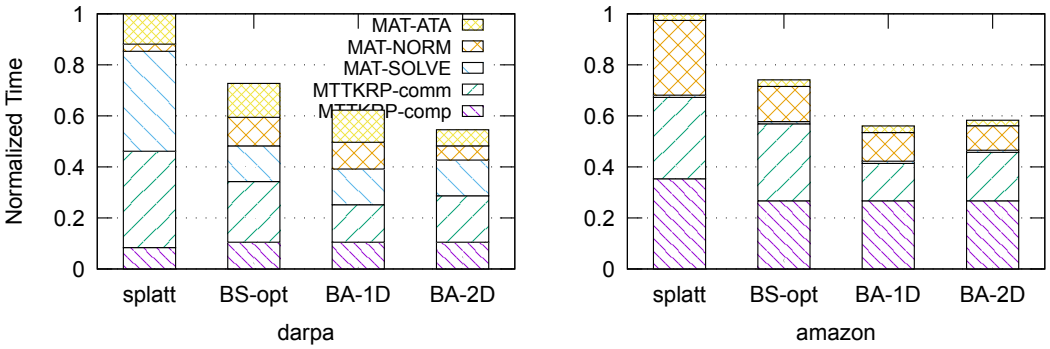


Figure 5.14: Time percentage of main kernels for darpa and amazon on 1280 processors.

5.4.3 Balanced Distribution Policy Analysis

Figure 5.15 shows the speedup of CPD from our four matrix-oriented distribution policies against SPLATT on 1536 processors. *Set*, *ordered-1*, *ordered-2*, and *max-min* represent the strategies of matrix-balancing, two types of ordered adjustment, and max-min adjustment separately. *Ordered-1* and *ordered-2* incline the adjustment to nonzero and I_p balance respectively. Overall, our strategies obtain speedup on all tensors. The *set* strategy performs the best on three, *ordered-1* on one, *ordered-2* on three, and *max-min* on one tensor respectively. All the four strategies achieve significant speedups on the two most-irregular tensors *fb-m* and *fb-s*, with *ordered-2* the most advantageous.

An interesting observation is that simple strategies (*set* and *max-min*) could perform the best. These results verify our findings that balancing only nonzeros results in suboptimal performance, and trade-offs are required among nonzero, matrix computation and communication volume.

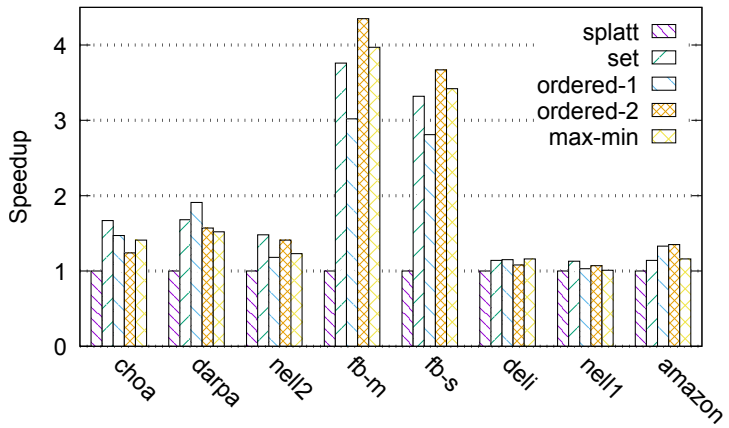


Figure 5.15: The effect of different distribution policies: matrix-balancing (*set*), two ordered adjustments (*ordered*-1 and *ordered*-2), and max-to-min adjustment (*max-min*).

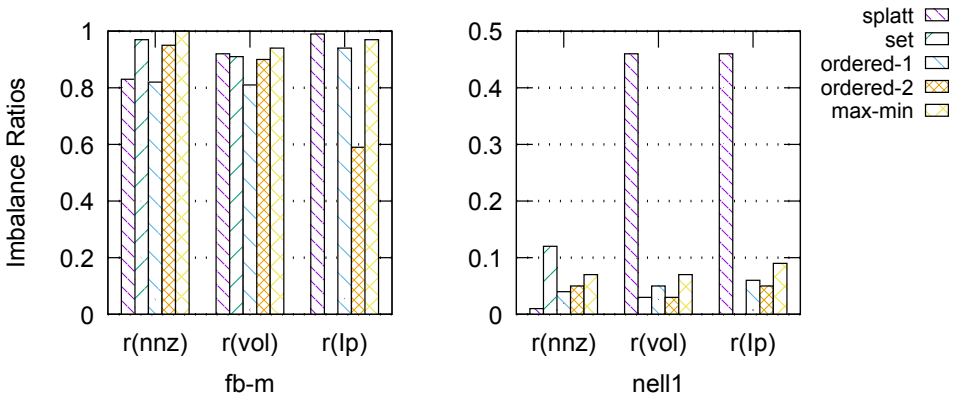


Figure 5.16: Load imbalance ratios (r_{nnz}, r_{vol}, and r_{I_P}).

To further understand why some tensors benefit more from our strategies than others, we look into how their imbalance ratios change. We explore two representative tensors in Figure 5.16. Two general observations are obtained: First, no strategy simultaneously obtains the lowest imbalance ratios from all the three aspects: nonzero, matrix computation, and communication. Second, all strategies trade higher r_{nnz} for lower r_{vol} and r_{I_P} to gain performance improvement. The irregular tensor *fb-m* suffers very high imbalance ratios for all strategies in all three aspects. SPLATT has the

smallest r_{nnz} balance, *set* has nearly perfect r_{I_p} balance (around 0, invisible in bars), while *ordered-1* gets the best r_{vol} balance. However, *ordered-2* obtains the best performance in Figure 5.15, since none of SPLATT, *set*, and *ordered-1* obtains a good tradeoff among the three ratios. Different from irregular tensors, regular tensors like *nell1* have much lower imbalance ratios in each category. Its r_{nnz} imbalance ratio is actually under control at 1% with SPLATT. *Set*, which gets the highest performance gain, has the worst r_{nnz} imbalance but the best r_{I_p} and r_{vol} balance. Regular tensors tend to be easier to get balanced in all categories and the differences among them are small. These results demonstrate that the trade-off among different load balances is complex and the optimal solution is determined by tensor properties, i.e., sparsity, shape, and distribution of nonzeros among the modes. We identify the dominant imbalance ratio as r_{I_p} for irregular tensors because of its impact on matrix computation. However, the best performance of CPD is usually not achieved by the optimal r_{I_p} because other imbalance ratios are also important. It is still very difficult or impossible to obtain the optimal balance simultaneously among all categories, thus a careful trade-off is required for the best performance.

Guideline for choosing strategies. We provide general guidelines for users to easily pick from the strategies for their own tensors. Our strategies try to find the best trade-off among three imbalance ratios though it is difficult to match each strategy for one certain type of tensors. If r_{I_p} is the dominant imbalance factor in CPD and we need to control it as small as possible, the ascending order of r_{I_p} in our strategies is *set* < *ordered-2* < *ordered-1*. Generally, users could safely choose *set* if lacking of statistical information on a sparse tensor because it always performs better than SPLATT on a large cluster as Figure 5.15 shows. Our recommendations are as follows: 1) Use *set* for relatively small or regular tensors as it obtains the smallest r_{I_p} while the other two imbalance ratios have little increase in those tensors like *choa* and *nell2*; 2) Use *ordered-2* for relatively large and irregular tensors as it optimizes both r_{vol} and r_{I_P} well on tensors like *fb-m* and *fb-s*.

5.4.4 Bottleneck Shifting

We show how MGBS-opt influences the performance bottleneck of major computation and communication kernels of CPD for tensors *choa* and *fb-m* in Figure 5.17. For *choa*, MGBS-opt shifts the performance bottleneck from communication in SPLATT to *MAT-SOLVE* as a result of communication time reduction, while also decreasing the time of *MAT NORM*. For *fb-m*, the MGBS-opt performance is still dominated by *COMM* as in SPLATT, but largely reduced. Since SPLATT

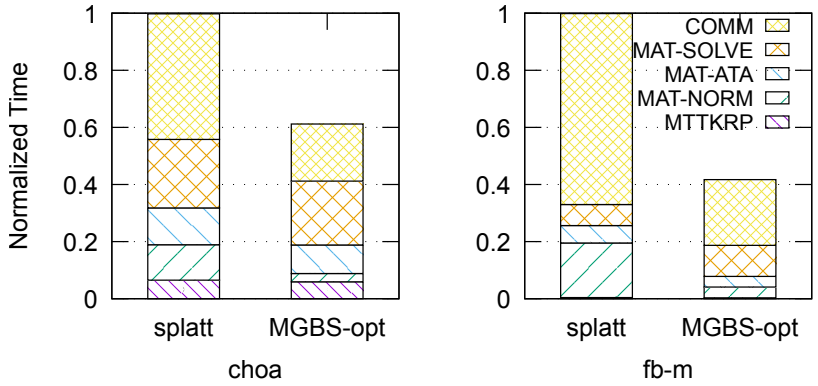


Figure 5.17: Time percentage of main kernels on 1536 processors.

focuses on optimizing the nonzero imbalance for MTTKRP which only accounts for a negligible portion (invisible in Figure 5.17), MGBS-opt correctly identifies bottlenecks and significantly improves their execution.

5.4.5 Partitioning Strategies Comparison

Several previous works have compared MGBS with coarse-grained CPD [39]. It has been proved that SPLATT is 41× to 76× faster than DFacTo on 1024 cores [114]. Therefore, we no longer compare MGBS-opt with coarse-grained CPD in this work. We examine the fine-grained distribution with hypergraph partitioning of each tensor generated by Zoltan [25]. Large tensors such as *amazon* and *patents* are unable to compute a hypergraph partitioning due to their memory requirements. Figure 5.11(a) already shows FGBS achieves lower performance than both SPLATT and MGBS-opt for 5 tensors on 1536 processors. The hypergraph partitions of *fb-m* and *fb-s* can be generated on 768 processors. SPLATT achieves higher performance than fine-grained distribution in 5 out of 7 tensors on 768 processors except for *fb-m* and *fb-s*. Figure 5.18 displays the normalized time of major computation and communication kernels in FGBS, SPLATT and MGBS-opt on 768 processors. We first disclose that FGBS performs faster than SPLATT on tensors *fb-m* and *fb-s* by 3.2× and 1.3×, but only achieves 70% and 30% of the performance of MGBS-opt, which further strengthens our motivation of study on irregular tensors. Compared to SPLATT, both FGBS and MGBS-opt significantly improve the performance of matrix-related computations on *fb-m* and achieve similar speedups. While on *fb-s* FGBS only gains a small improvement over SPLATT. This demonstrates the performance improvement of MGBS-opt is more stable than FGBS on different irregular tensors.

To purely compare with a hypergraph-partitioning model based on medium-grain distributed CPD computation, we also compare MGBS-opt with CartHP [6] built upon MGBS algorithm with SPLATT. Due to the lack of access of CartHP code and dataset, we are unable to compare CartHP and MGBS-opt on the same dataset. CartHP achieves an average of 1.32× and up to 2.34× speedup over SPLATT on their dataset in the paper [6], inferring that MGBS-opt generally gains higher speedup by comparing these numbers.

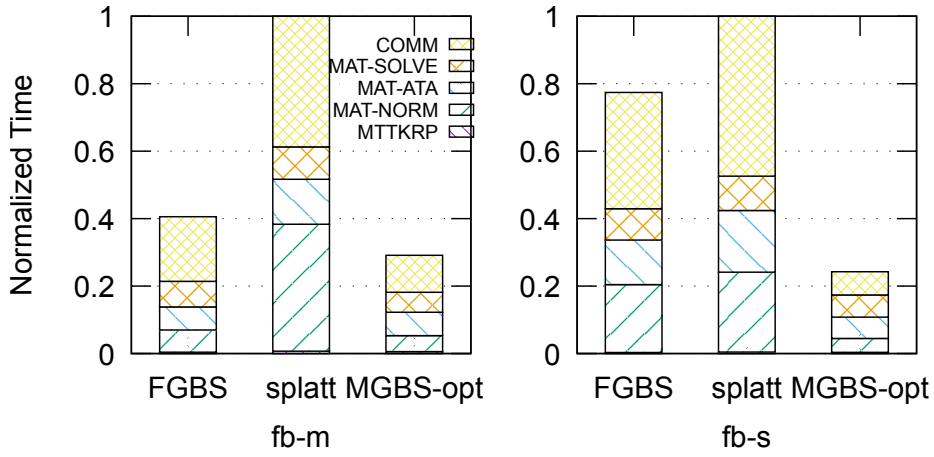


Figure 5.18: Time percentage of main kernels for $fb-m$ and $fb-s$ on 768 processors.

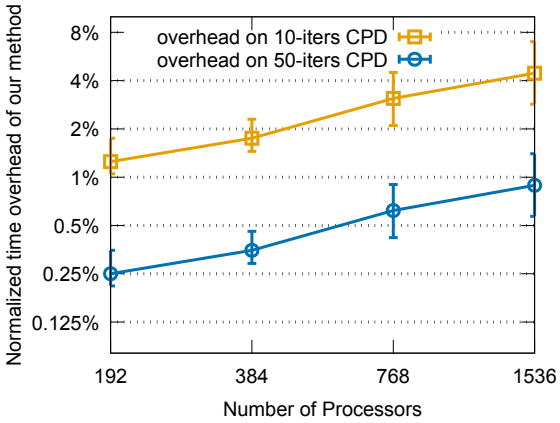


Figure 5.19: Time overhead of our method (Algorithm 2 and 3). The time of our method is normalized to CPD time.

5.4.6 Time overhead of irregularity-aware method.

We evaluate the time overhead of our irregularity-aware method and compare it with CPD time. Our proposed prediction-based grid configuration incurs trivial time cost in the virtual distribution as it needs to compute the nonzero imbalance ratio r_{layer_nnz} for each candidate. The cost of our matrix-oriented distribution policy is negligible because its complexity is $\mathcal{O}(P)$. The time cost of irregularity-aware method is mainly determined by the total dimension sizes of the tensor c_N in the complexity of $\mathcal{O}(c_N \times I_n)$ in computation of r_{layer_nnz}. Our method does not incur expensive data redistribution because we only do data distribution once as SPLATT. The number of CPD iterations is determined comprehensively by the size, nonzero distribution, and sparsity of a tensor. We set 10 and 50 as the minimum and maximum iterations because we observe tensors in our dataset converge for CPD in this range of iterations. Figure 5.19 displays the average, maximum and minimum time overhead of irregularity-aware method normalized to 10 and 50 CPD iterations all tensors in our dataset. As the system size increases, the normalized time overhead increases for both cases. This is because our proposed irregularity-aware method is sequential with relatively stable time on different system sizes. The average overhead is 4.5% to 10 CPD iterations and 0.9% to 50 iterations on 1536 processors. Overall, the time cost of irregularity-aware method is low and acceptable compared to CPD time. And its time overhead is negligible compared to hypergraph partitioning in both fine-grained [72, 71] and medium-grained CPD [6].

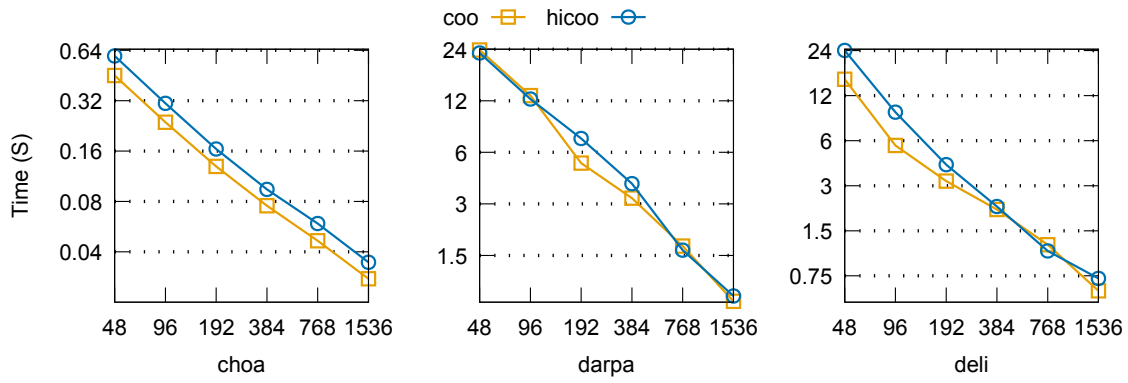


Figure 5.20: Scalability of MGBS-opt applied on ParTI for COO and HiCOO formats.

5.4.7 Application of irregularity-aware method to Other Formats

We extend MGBS-opt to support other sparse tensor formats such as the coordinate (COO) and Hierarchical Coordinate (HICOO) [81] by extending the ParTI library [80]. COO, the simplest yet arguably most popular format by far, stores each nonzero value along with all of its position indices. Hierarchical Coordinate (HICOO) [81] format improves upon COO by compressing the indices in units of sparse tensor blocks. Figure 5.20 plots strong scalability of MGBS-opt applied to COO and HiCOO formats for three tensors on 48 to 1536 processors. MGBS-opt obtains near-linear scalability for HiCOO on these tensors. With COO format *darpa* on 96 to 192 and *deli* on 48 to 96 processors show super-linear speedup. Detailed profiling shows that computation time for matrix-related kernels reduces more than halve in both cases because of much better matrix-balance. MGBS-opt is flexible to support to other variant formats in CSF or COO families [94, 84].

5.4.8 Prototype of Asynchronous Cpd on GPUs.

We implement the prototype of asynchronous CPD on GPUs to prove that our BA-CPD algorithm works in heterogeneous systems with GPUs. We implement MTTKRP and matrix-related kernels with asynchronous algorithms on GPUs. We utilize cuSPARSE routines in CUDA to implement computation and use NVSHMEM to implement communication for these kernels. Figure 5.21 shows our prototype of asynchronous CPD on GPUs achieves 4.23× speedup over SPLATT and 3.27× speedup over BA-CPD on CPUs for MTTKRP, and achieves 2.44× speedup over SPLATT and 1.97× speedup over BA-CPD on CPUs for MAT NORM. The result shows that our prototype of asynchronous CPD on GPUs mainly benefits from the high performance of computation on GPUs. The difference between asynchronous communication via OpenSHMEM and NVSHMEM is not obvious on a small scale of distribution.

There are two main challenges to implementing asynchronous CPD on multiple GPUs. First, it is necessary to design a better workload partition to exploit the storage format of sparse tensors and dense matrices in the GPU memory to save computation and memory space. The current workload partition works well on CPUs while it has space to optimize on GPUs. Second, the computation and communication percentages on GPUs are vastly different from those on CPUs. Thus we need to employ new optimization techniques to distributed CPD on GPUs.

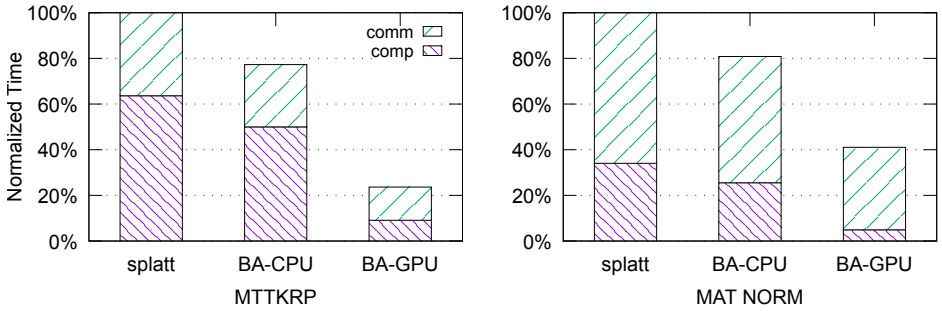


Figure 5.21: Time percentage of MTTKRP and MAT NORM for *nell*2 of our prototype on 4 GPU nodes compared to BA-CPD and splatt on 4 CPU nodes.

5.5 Summary

Distributed CANDECOMP/PARAFAC decomposition is well-studied due to the increasing needs of processing large-scale data. This work presents a sparsity-aware tensor decomposition on a distributed memory system. We thoroughly investigate the distributed CPD performance behavior using a state-of-the-art implementation and find three performance factors, grid configuration, load imbalance, and communication/computation overlap along with five observations. Based on these observations, we propose three optimization strategies: prediction-based grid configuration, tensor dimension-oriented data distribution, and overlap of computation and communication. Our proposed sparsity-aware distributed CANDECOMP/PARAFAC decomposition, outperforms the state-of-the-art distributed SPLATT library by up to 4.41× on 768 processors and 4.36× on 1,536 processors. Overall, the major findings and contributions of this work include:

- Our work investigates the common algorithm structure of state-of-the-art distributed implementations from theoretical and experimental analysis and observes four findings to guide performance optimization.

- We demonstrate that the imbalance of computation and communication, and their trade-offs, are critical to the overall CPD performance and scalability. We identify the dominant imbalance ratio as matrix computation imbalance for irregular tensors. We propose irregularity-aware CPD that co-optimizes these imbalances with high priority in matrix computation imbalance in grid configuration and distribution policy with a low time overhead.

- Our irregularity-aware method supports different sparse tensor formats like compressed sparse fiber (CSF), coordinate (COO), and Hierarchical Coordinate (HiCOO), and gain good scala-

bility for all of them.

- We present a performant scalable algorithm, BA-CPD. It advances the state-of-the-art by best balancing computation and communication within and across the kernels and hiding communication costs. We propose techniques to enable fine-grained overlap for computation and communication tailored for each kernel.

- We demonstrate that our method scales well when using up to 1536 processors and obtains up to 4.4× and 11.4× performance improvement over the distributed medium- and fine-grained CPD libraries [114, 72] respectively.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

Emergent HPC systems must address challenges from both performance-scalability and power-scalability in the presence of failures. Resilience allows programs to progress when subjected to faults and is an integral component of large-scale systems, but incurs significant time and energy overhead. Sparse data computation is the fundamental kernel in many scientific applications. It is suitable for the studies of scalability and resilience on heterogeneous systems due to its computational characteristics. There is an urgent need for efficient, reliable and scalable sparse data computing to maximize utilization of HPC systems under constraints of failures. This thesis presents various algorithms and optimizations for enabling reliable and scalable sparse data computing on large-scale systems.

In Chapter 3, we present a novel approach to analyze and optimize the cost of resilience techniques for sparse linear solvers. We present a set of models to better understand the resilience and energy overhead of applications in a faulty environment, and we perform power optimizations to reduce the overhead of forward recovery. Our experiments show that our optimized forward-recovery algorithm significantly reduces the resilience overhead and provides insights for selecting recovery schemes for certain workloads. Our projection result reveals trends of resilience cost on large systems and provides direction for optimizing resilience schemes. We demonstrate the importance of the development of efficient resilience mechanisms, as current resilience schemes do not meet the requirements of future larger and more faulty systems. We need more optimizations to further

reduce time overhead in checkpointing, recovery or reconstruction phases. Decreasing them can significantly improve the full application's performance.

In Chapter 4, we propose a novel redundancy technique, relaxed replication, for resilient and energy-efficient HPC applications in GPU-accelerated systems. Relaxed replication provides MPI-process level redundancy, enabling optimizations for GPU applications. It supports multiple technologies including heterogeneous processors, mixed precision computation, and power management to reduce the time and energy overhead of replication. Experimental results show that our approach significantly reduces the resilience overhead while maintaining energy-efficiency compared to previous resilience mechanisms. We demonstrate that relaxed replication can address the challenge of high replication cost, and achieve energy-efficiency in GPU-accelerated systems with multiple computation and power optimizations.

In Chapter 5, we present an irregularity-aware algorithm for workload partition and a performant scalable distributed algorithm, BA-CPD. It improves workload balance and reduces communication cost in comparison to existing work. Our workload partition achieves balances for all but computation in tensor-related kernel and co-optimizes computation and communication imbalance. Based on our workload partition, we further present asynchronous algorithms to reduce the communication overhead of collective communication operations in traditional bulk-synchronous CPD. We prove that the prototype of our algorithms works well in heterogeneous systems. We demonstrate that understanding different bottlenecks for various types of tensors plays critical roles in improving the scalability of distributed tensor decomposition.

6.2 Future Work

This dissertation has laid the groundwork for research on the scalability and resilience of sparse data computing on emergent HPC systems. Long term future directions can seek to develop the ideas around scalability, resilience, and their trade-off for various resilience mechanisms, emerging hardware, and different applications of sparse data computing. In each of the following future directions, optimization for resilience and scalability of future HPC systems will be the focus. And each direction has significant, very interesting and challenging unsolved problems.

6.2.1 Optimization for Various Resilience Mechanisms

We focus on two types of resilience mechanisms: forward recovery and redundancy in this dissertation due to their opportunities in optimization for sparse linear solvers. We believe it is worth to explore improving scalability and reducing resilience overhead for other fault-tolerance mechanisms.

One future direction will be exploring how to reduce overhead of checkpoints in checkpointing/restart mechanism when there is no fault. One of the research ideas is to combine checkpointing/restart with fault prevention technologies like failure prediction. Failure prediction aims to predict faults by certain patterns of behaviors in systems. The main challenge for failure prediction is the relatively low accuracy of prediction. Another important challenge for this direction is still achieving a good trade-off between scalability and resilience.

6.2.2 Optimization for Emerging Hardwares

The main computational components have evolved from CPU to various processor components including GPU, Field-Programmable Gate Array(FPGA), etc. FPGA provides opportunities for better energy-efficiency than CPU and GPU. But it also incurs challenges like resilience and power management. Further work can extend our algorithms and optimizations in Chapter 3 and 4 to deal with heterogeneous systems with FPGAs. We present scalable algorithms for sparse tensor decomposition in homogeneous systems and prove that our prototype works well in heterogeneous systems with GPUs. Future work can extend our prototype with more optimizations in GPU computing to support tensor decomposition in heterogeneous systems with GPUs.

6.2.3 Optimization for Other Sparse Data Computing

We focus on optimizing scalability and resilience for sparse linear solvers in Chapter 3 and 4. One future direction will be exploring optimization for scalability and resilience of other computations like sparse-sparse matrix-matrix multiplication or sparse-dense matrix-matrix multiplication. These directions will provide both new opportunities and new challenges for optimization. Chapter 5 focuses on the optimization of CANDECOMP/PARAFAC decomposition for sparse tensors. Further work can extend our algorithms or ideas in Chapter 5 to deal with other tensor decompositions like Tucker decomposition.

Bibliography

[1] El capitan exascale supercomputer. `https://www.cray.com/company/customers/lawrence-livermore-national-lab`.

[2] Frontier exascale supercomputer. `https://www.olcf.ornl.gov/frontier`.

[3] Top 500 supercomputers. `https://top500.org`.

[4] University of florida sparse matrix collection. `https://sparse.tamu.edu/`.

[5] Martín Abadi et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015.

[6] Seher Acer, Tugba Torun, and Cevdet Aykanat. Improving medium-grain partitioning for scalable sparse tensor decomposition. *IEEE Transactions on Parallel and Distributed Systems*, 29(12):2814–2825, 2018.

[7] Emmanuel Agullo, Luc Giraud, Abdou Guermouche, Jean Roman, and Mawussi Zounon. Numerical recovery strategies for parallel resilient krylov linear solvers. *Numerical Linear Algebra with Applications*, 23(5):888–905, 2016.

[8] Emmanuel Agullo, Luc Giraud, and Mawussi Zounon. On the resilience of parallel sparse hybrid solvers. In *HiPC*, pages 75–84. IEEE, 2015.

[9] Rob Aitken, Ethan H Cannon, Mondira Pant, and Mehdi B Tahoori. Resiliency challenges in sub-10nm technologies. In *VLSI Test Symposium (VTS), 2015 IEEE 33rd*, pages 1–4. IEEE, 2015.

[10] Phillip Alpatov, Greg Baker, H Carter Edwards, John Gunnels, Greg Morrow, James Overfelt, and Robert Van de GEijn. PLAPACK parallel linear algebra package design overview. In *SC'97: Proceedings of the 1997 ACM/IEEE Conference on Supercomputing*, pages 29–29. IEEE, 1997.

[11] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485, 1967.

[12] Animashree Anandkumar, Rong Ge, Daniel Hsu, Sham M. Kakade, and Matus Telgarsky. Tensor decompositions for learning latent variable models. *J. Mach. Learn. Res.*, 15(1):2773–2832, January 2014.

[13] Hartwig Anzt, Björn Rocker, and Vincent Heuveline. Energy efficiency of mixed precision iterative refinement methods using hybrid hardware platforms. *Computer Science-Research and Development*, 25(3-4):141–148, 2010.

[14] Steve Ashby, Pete Beckman, Jackie Chen, Phil Colella, Bill Collins, Dona Crawford, Jack Dongarra, Doug Kothe, Rusty Lusk, Paul Messina, et al. The opportunities and challenges of exascale computing. *ASCAC*, pages 1–77, 2010.

[15] Guillaume Aupy, Anne Benoit, Thomas Hérault, Yves Robert, and Jack Dongarra. Optimal checkpointing period: Time vs. energy. In *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*, pages 203–214. Springer, 2013.

[16] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Secur. Comput.*, 1(1):11–33, January 2004.

[17] Muthu Baskaran, Thomas Henretty, and James Ezick. Fast and scalable distributed tensor decompositions. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2019.

[18] Leonardo Bautista-Gomez, Seiji Tsuboi, Dimitri Komatitsch, Franck Cappello, Naoya Maruyama, and Satoshi Matsuoka. Fti: High performance fault tolerance interface for hybrid systems. In *SC'11*, pages 1–12. IEEE, 2011.

[19] Eduardo Berrocal, Leonardo Bautista-Gomez, Sheng Di, Zhiling Lan, and Franck Cappello. Exploring partial replication to improve lightweight silent data corruption detection for hpc applications. In *European Conference on Parallel Processing*, pages 419–430. Springer, 2016.

[20] Amanda Bienz, William D Gropp, and Luke N Olson. Node aware sparse matrix-vector multiplication. *Urbana*, 51:61801, 2016.

[21] Amanda Bienz and Luke N. Olson. RAPtor: parallel algebraic multigrid v0.1, 2017. Release 0.1.

[22] Zachary Blanco, Bangtian Liu, and Maryam Mehri Dehnavi. CSTF: Large-scale sparse tensor factorizations on distributed platforms. In *Proceedings of the 47th International Conference on Parallel Processing*, ICPP 2018, pages 21:1–21:10, New York, NY, USA, 2018. ACM.

[23] Wesley Bland. User level failure mitigation in mpi. In *European Conference on Parallel Processing*, pages 499–504. Springer, 2012.

[24] Wesley Bland, Peng Du, Aurelien Bouteiller, Thomas Herault, George Bosilca, and Jack J Dongarra. Extending the scope of the checkpoint-on-failure protocol for forward recovery in standard mpi. *Concurrency and computation: Practice and experience*, 25(17):2381–2393, 2013.

[25] E. G. Boman, U. V. Catalyurek, C. Chevalier, and K. D. Devine. The Zoltan and Isorropia parallel toolkits for combinatorial scientific computing: Partitioning, ordering, and coloring. *Scientific Programming*, 20(2):129–150, 2012.

[26] William L Briggs, Van Emden Henson, and Steve F McCormick. *A multigrid tutorial*. SIAM, 2000.

[27] Alfredo Buttari, Jack Dongarra, Jakub Kurzak, Piotr Luszczek, and Stanimir Tomov. Using mixed precision for sparse matrix computations to enhance the performance while achieving 64-bit accuracy. *TOMS*, 34(4):17, 2008.

[28] Jon Calhoun, Franck Cappello, Luke N Olson, Marc Snir, and William D Gropp. Exploring the feasibility of lossy compression for pde simulations. *The International Journal of High Performance Computing Applications*, 33(2):397–410, 2019.

[29] Jon Calhoun, Marc Snir, Luke N. Olson, and William D. Gropp. Towards a more complete understanding of sdc propagation. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '17, pages 131–142, New York, NY, USA, 2017. ACM.

[30] Franck Cappello, Al Geist, William Gropp, Sanjay Kale, Bill Kramer, and Marc Snir. Toward exascale resilience: 2014 update. *Supercomputing frontiers and innovations*, 1(1):5–28, 2014.

[31] Venkatesan T Chakaravarthy, Jee W Choi, Douglas J Joseph, Prakash Murali, Shivmaran S Pandian, Yogish Sabharwal, and Dheeraj Sreedhar. On optimizing distributed tucker decomposition for sparse tensors. In *Proceedings of the 2018 International Conference on Supercomputing*, pages 374–384, 2018.

[32] Raghunath Raja Chandrasekar, Akshay Venkatesh, Khaled Hamidouche, and Dhabaleswar K Panda. Power-check: An energy-efficient checkpointing framework for hpc clusters. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 261–270. IEEE, 2015.

[33] Jieyang Chen, Sihuan Li, and Zizhong Chen. Gpu-abft: Optimizing algorithm-based fault tolerance for heterogeneous systems with gpus. In *NAS*, pages 1–2. IEEE, 2016.

[34] Zizhong Chen. Algorithm-based recovery for iterative methods without checkpointing. In *Proceedings of the 20th international symposium on High performance distributed computing*, pages 73–84. ACM, 2011.

[35] Zizhong Chen. Online-abft: An online algorithm based fault tolerance scheme for soft error detection in iterative methods. In *ACM SIGPLAN Notices*, volume 48, pages 167–176. ACM, 2013.

[36] Zizhong Chen and Jack Dongarra. Algorithm-based fault tolerance for fail-stop failures. *IEEE Transactions on Parallel and Distributed Systems*, 19(12):1628–1641, 2008.

[37] Zizhong Chen, Graham E Fagg, Edgar Gabriel, Julien Langou, Thara Angskun, George Bosilca, and Jack Dongarra. Fault tolerant high performance computing by a coding approach. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 213–223. ACM, 2005.

[38] Jaeyoung Choi, James Demmel, Inderjiit Dhillon, Jack Dongarra, Susan Ostrouchov, Antoine Petitet, Ken Stanley, David Walker, and R Clinton Whaley. ScaLAPACK: A portable linear algebra library for distributed memory computers—design issues and performance. *Computer Physics Communications*, 97(1-2):1–15, 1996.

[39] Joon Hee Choi and S. Vishwanathan. DFacTo: Distributed factorization of tensors. In Z. Ghahramani, M. Welling, C. Cortes, N.D. Lawrence, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 1296–1304. Curran Associates, Inc., 2014.

[40] Andrzej Cichocki. Era of big data processing: A new approach via tensor networks and tensor decompositions. *CoRR*, abs/1403.2048, 2014.

[41] Tao Cui, Jinchao Xu, and Chen-Song Zhang. An error-resilient redundant subspace correction method. *Computing and Visualization in Science*, 18(2-3):65–77, 2017.

[42] Xiaolong Cui, Taieb Znati, and Rami Melhem. Adaptive and power-aware resilience for extreme-scale computing. In *2016 Intl IEEE Conferences on Ubiquitous Intelligence & Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud*

and Big Data Computing, Internet of People, and Smart World Congress (UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld), pages 671–679. IEEE, 2016.

[43] John T Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future generation computer systems*, 22(3):303–312, 2006.

[44] Teresa Davies and Zizhong Chen. Correcting soft errors online in lu factorization. In *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*, pages 167–178. ACM, 2013.

[45] Eduardo D'Azevedo and Jack Dongarra. The design and implementation of the parallel out-of-core scalapack lu, qr, and cholesky factorization routines. *Concurrency: Practice and Experience*, 12(15):1481–1493, 2000.

[46] Sheng Di, Mohamed Slim Bouguerra, Leonardo Bautista-Gomez, and Franck Cappello. Optimization of multi-level checkpoint model for large scale hpc applications. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 1181–1190. IEEE, 2014.

[47] Sheng Di and Franck Cappello. Adaptive impact-driven detection of silent data corruption for hpc applications. *IEEE Trans. Parallel Distrib. Syst.*, 27(10):2809–2823, October 2016.

[48] Catello Di Martino, Zbigniew Kalbarczyk, Ravishankar K Iyer, Fabio Baccanico, Joseph Fullop, and William Kramer. Lessons learned from the analysis of system failures at petascale: The case of blue waters. In *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*, pages 610–621. IEEE, 2014.

[49] Jack Dongarra, George Bosilca, Zizhong Chen, Victor Eijkhout, Graham E Fagg, Erika Fuentes, Julien Langou, Piotr Luszczek, Jelena Pjesivac-Grbovic, Keith Seymour, et al. Self-adapting numerical software (sans) effort. *IBM Journal of Research and Development*, 50(2.3):223–238, 2006.

[50] Jack Dongarra, Michael A Heroux, and Piotr Luszczek. High-performance conjugate-gradient benchmark: A new metric for ranking high-performance computing systems. *The International Journal of High Performance Computing Applications*, 30(1):3–10, 2016.

[51] Mohammed el Mehdi Diouri, Olivier Glück, Laurent Lefevre, and Franck Cappello. Energy considerations in checkpointing and fault tolerance protocols. In *DSN Workshops*, pages 1–6. IEEE, 2012.

[52] James Elliott, Mark Hoemmen, and Frank Mueller. Evaluating the impact of sdc on the gmres iterative solver. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 1193–1202. IEEE, 2014.

[53] Christian Engelmann and Swen Böhm. Redundant execution of hpc applications with mr-mpi. In *PDCN*, pages 15–17, 2011.

[54] Kurt Ferreira, Rolf Riesen, Ron Oldfield, Jon Stearley, James Laros, Kevin Pedretti, and Ron Brightwell. rmpi: increasing fault resiliency in a message-passing environment. *Sandia National Laboratories, Albuquerque, NM, Tech. Rep. SAND2011-2488*, 2011.

[55] Kurt Ferreira, Jon Stearley, James H Laros III, Ron Oldfield, Kevin Pedretti, Ron Brightwell, Rolf Riesen, Patrick G Bridges, and Dorian Arnold. Evaluating the viability of process replication reliability for exascale systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 44. ACM, 2011.

[56] David Fiala, Frank Mueller, Christian Engelmann, Rolf Riesen, Kurt Ferreira, and Ron Brightwell. Detection and correction of silent data corruption for large-scale high-performance computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, Washington, DC, USA, 2012. IEEE Computer Society Press.

[57] Cijo George and Sathish Vadhiyar. Fault tolerance on large scale systems using adaptive process replication. *IEEE Transactions on Computers*, 64(8):2213–2225, 2014.

[58] Amit Golander, Shlomo Weiss, and Ronny Ronen. Ddmr: Dynamic and scalable dual modular redundancy with short validation intervals. *IEEE Computer Architecture Letters*, 7(2):65–68, 2008.

[59] Ryan E Grant, Stephen L Olivier, James H Laros, Ron Brightwell, and Allan K Porterfield. Metrics for evaluating energy saving techniques for resilient hpc systems. In *2014 IEEE International Parallel & Distributed Processing Symposium Workshops*, pages 790–797. IEEE, 2014.

[60] John L Gustafson. Reevaluating amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988.

[61] Paul H Hargrove and Jason C Duell. Berkeley lab checkpoint/restart (blcr) for linux clusters. In *Journal of Physics: Conference Series*, volume 46, page 494. IOP Publishing, 2006.

[62] Michael T Heath. *Scientific computing*. McGraw-Hill New York, 2002.

[63] Joyce C. Ho, Joydeep Ghosh, and Jimeng Sun. Marble: High-throughput phenotyping from electronic health records via sparse nonnegative tensor factorization. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '14, pages 115–124, New York, NY, USA, 2014. ACM.

[64] Kuang-Hua Huang et al. Algorithm-based fault tolerance for matrix operations. *IEEE transactions on computers*, 100(6):518–528, 1984.

[65] Markus Huber, Bjorn Gmeiner, Ulrich Rude, and Barbara Wohlmuth. Resilience for massively parallel multigrid solvers. *SIAM Journal on Scientific Computing*, 38(5):S217–S239, 2016.

[66] Saurabh Hukerikar, Pedro C Diniz, Robert F Lucas, and Keita Teranishi. Opportunistic application-level fault detection through adaptive redundant multithreading. In *2014 International Conference on High Performance Computing & Simulation (HPCS)*, pages 243–250. IEEE, 2014.

[67] Luc Jaulmes, Marc Casas, Miquel Moretó, Eduard Ayguadé, Jesús Labarta, and Mateo Valero. Exploiting asynchrony from exact forward recovery for due in iterative solvers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 53. ACM, 2015.

[68] Inah Jeon, Evangelos E. Papalexakis, U Kang, and Christos Faloutsos. HaTen2: Billion-scale tensor decompositions. In *IEEE International Conference on Data Engineering (ICDE)*, 2015.

[69] David B Johnson and Willy Zwaenepoel. *Sender-based message logging*. Rice University, Department of Computer Science, 1987.

[70] U. Kang, Evangelos Papalexakis, Abhay Harpale, and Christos Faloutsos. GigaTensor: Scaling tensor analysis up by 100 times - algorithms and discoveries. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '12, pages 316–324, New York, NY, USA, 2012. ACM.

[71] O. Kaya and B. Uçar. Parallel Candecomp/Parafac decomposition of sparse tensors using dimension trees. *SIAM Journal on Scientific Computing*, 40(1):C99–C130, 2018.

[72] Oguz Kaya and Bora Uçar. Scalable sparse tensor decompositions in distributed memory systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 77:1–77:11, New York, NY, USA, 2015. ACM.

[73] Jeongnim Kim, Andrew D Baczewski, Todd D Beaudet, Anouar Benali, M Chandler Bennett, Mark A Berrill, Nick S Blunt, Edgar Josué Landinez Borda, Michele Casula, David M Ceperley, et al. Qmcpack: an open source ab initio quantum monte carlo package for the electronic structure of atoms, molecules and solids. *Journal of Physics: Condensed Matter*, 30(19):195901, 2018.

[74] T. Kolda and B. Bader. Tensor decompositions and applications. *SIAM Review*, 51(3):455–500, 2009.

[75] Jiri Kraus. An introduction to cuda-aware mpi. *Weblog entry]. PARALLEL FORALL*, 2013.

[76] Jérôme Kunegis. Konect: the koblenz network collection. In *Proceedings of the 22nd international conference on world wide web*, pages 1343–1350, 2013.

[77] Reservoir Labs. ENSIGN: Multi-domain analytics. Available from `https://www.reservoir.com/ensign/`.

[78] Julien Langou, Zizhong Chen, George Bosilca, and Jack Dongarra. Recovery patterns for iterative methods in a parallel unstable environment. *SIAM Journal on Scientific Computing*, 30(1):102–116, 2007.

[79] Jungseob Lee and Nam Sung Kim. Optimizing throughput of power-and thermal-constrained multicore processors using dvfs and per-core power-gating. In *Design Automation Conference, 2009. DAC'09. 46th ACM/IEEE*, pages 47–50. IEEE, 2009.

[80] Jiajia Li, Yuchen Ma, and Richard Vuduc. ParTI! : A parallel tensor infrastructure for multicore CPUs and GPUs (Version 1.0.0), Oct 2018.

[81] Jiajia Li, Jimeng Sun, and Richard Vuduc. HiCOO: Hierarchical storage of sparse tensors. In *Proceedings of the ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, Dallas, TX, USA, November 2018.

[82] Jian Li and José F Martínez. Power-performance considerations of parallel computing on chip multiprocessors. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2(4):397–422, 2005.

[83] Xin Liang, Jieyang Chen, Dingwen Tao, Sihuan Li, Panruo Wu, Hongbo Li, Kaiming Ouyang, Yuanlai Liu, Fengguang Song, and Zizhong Chen. Correcting soft errors online in fast fourier transform. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, pages 30:1–30:12, New York, NY, USA, 2017. ACM.

[84] B. Liu, C. Wen, A. D. Sarwate, and M. M. Dehnavi. A unified optimization approach for sparse tensor operations on GPUs. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 47–57, Sept 2017.

[85] Xinxin Mei, Qiang Wang, and Xiaowen Chu. A survey and measurement study of gpu dvfs on energy conservation. *Digital Communications and Networks*, 3(2):89–100, 2017.

[86] Esteban Meneses, Osman Sarood, and Laxmikant V Kalé. Energy profile of rollback-recovery strategies in high performance computing. *Parallel Computing*, 40(9):536–547, 2014.

[87] Zheng Miao, Jon Calhoun, and Rong Ge. Energy analysis and optimization for resilient scalable linear systems. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 24–34. IEEE, 2018.

[88] Bryan Mills, Ryan E Grant, Kurt B Ferreira, and Rolf Riesen. Evaluating energy savings for checkpoint/restart. In *Proceedings of the 1st International Workshop on Energy Efficient Supercomputing*, page 6. ACM, 2013.

[89] Bryan Mills, Taieb Znati, and Rami Melhem. Shadow computing: An energy-aware fault tolerant computing model. In *ICNC*, pages 73–77. IEEE, 2014.

[90] Amitabh Mishra and Prithviraj Banerjee. An algorithm-based error detection scheme for the multigrid method. *IEEE Transactions on Computers*, 52(9):1089–1099, 2003.

[91] Adam Moody, Greg Bronevetsky, Kathryn Mohror, and Bronis R de Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *Proceedings of the 2010 ACM/IEEE international conference for high performance computing, networking, storage and analysis*, pages 1–11. IEEE Computer Society, 2010.

[92] S Narang and G Diamos. Baidu deepbench, 2017.

[93] Bogdan Nicolae, Adam Moody, Elsa Gonsiorowski, Kathryn Mohror, and Franck Cappello. Veloc: Towards high performance adaptive asynchronous checkpointing at large scale. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 911–920. IEEE, 2019.

[94] Israt Nisa, Jiajia Li, Aravind Sukumaran-Rajam, Prasant Singh Rawat, Sriram Krishnamoorthy, and Ponnuswamy Sadayappan. An efficient mixed-mode representation of sparse tensors. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–25, 2019.

[95] Alexander Novikov, Dmitry Podoprikhin, Anton Osokin, and Dmitry Vetrov. Tensorizing neural networks. *CoRR*, abs/1509.06569, 2015.

[96] Chris C Paige, Beresford N Parlett, and Henk A Van der Vorst. Approximate solutions and eigenvalue bounds from krylov subspaces. *Numerical linear algebra with applications*, 2(2):115–133, 1995.

[97] Evangelos E. Papalexakis, Christos Faloutsos, and Nicholas D. Sidiropoulos. ParCube: Sparse parallelizable tensor decompositions. In *Proceedings of the 2012 European Conference on Machine Learning and Knowledge Discovery in Databases - Volume Part I*, ECML PKDD'12, pages 521–536, Berlin, Heidelberg, 2012. Springer-Verlag.

[98] Ioakeim Perros, Robert Chen, Richard Vuduc, and Jimeng Sun. Sparse hierarchical Tucker factorization and its application to healthcare. In *Proceedings of the 2015 IEEE International Conference on Data Mining (ICDM)*, ICDM '15, pages 943–948, Washington, DC, USA, 2015. IEEE Computer Society.

[99] Ioakeim Perros, Evangelos E. Papalexakis, Fei Wang, Richard Vuduc, Elizabeth Searles, Michael Thompson, and Jimeng Sun. SPARTan: Scalable PARAFAC2 for large & sparse data. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '17, pages 375–384, New York, NY, USA, 2017. ACM.

[100] James S Plank, Kai Li, and Michael A Puening. Diskless checkpointing. *IEEE Transactions on parallel and Distributed Systems*, 9(10):972–986, 1998.

[101] Thomas B Rolinger, Tyler A Simon, and Christopher D Krieger. Performance considerations for scalable parallel tensor decomposition. *Journal of Parallel and Distributed Computing*, 129:83–98, 2019.

[102] Thomas Ropars, Arnaud Lefray, Dohyun Kim, and André Schiper. Efficient process replication for mpi applications: sharing work between replicas. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 645–654. IEEE, 2015.

[103] Yousef Saad. *Iterative Methods for Sparse Linear Systems - Second Edition*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 2003.

[104] Jason Sanders and Edward Kandrot. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[105] Naoto Sasaki, Kento Sato, Toshio Endo, and Satoshi Matsuoka. Exploration of lossy compression for application-level checkpoint/restart. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 914–922. IEEE, 2015.

[106] Martin D Schatz, Robert A Van de Geijn, and Jack Poulson. Parallel matrix multiplication: A systematic journey. *SIAM Journal on Scientific Computing*, 38(6):C748–C781, 2016.

[107] Alexander Schöll, Claus Braun, Michael A Kochte, and Hans-Joachim Wunderlich. Efficient algorithm-based fault tolerance for sparse matrix operations. In *Dependable Systems and Networks (DSN), 2016 46th Annual IEEE/IFIP International Conference on*, pages 251–262. IEEE, 2016.

[108] Alexander Schöll, Claus Braun, and Hans-Joachim Wunderlich. Energy-efficient and error-resilient iterative solvers for approximate computing. In *On-Line Testing and Robust System Design (IOLTS), 2017 IEEE 23rd International Symposium on*, pages 237–239. IEEE, 2017.

[109] Lin Shi, Hao Chen, and Ting Li. Hybrid cpu/gpu checkpoint for gpu-based heterogeneous systems. In *International Conference on Parallel Computing in Fluid Dynamics*, pages 470–481. Springer, 2013.

[110] Alex Shye, Joseph Blomstedt, Tipp Moseley, Vijay Janapa Reddi, and Daniel A Connors. Plr: A software approach to transient fault tolerance for multicore architectures. *IEEE Transactions on Dependable and Secure Computing*, 6(2):135–148, 2009.

[111] Jaswinder Pal Singh, John L Hennessy, and Anoop Gupta. Scaling parallel programs for multiprocessors: Methodology and examples. *Computer*, 26(7):42–50, 1993.

[112] Joseph Sloan, Rakesh Kumar, and Greg Bronevetsky. An algorithmic approach to error localization and partial recomputation for low-overhead fault tolerance. In *Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on*, pages 1–12. IEEE, 2013.

[113] Shaden Smith, Jee W. Choi, Jiajia Li, Richard Vuduc, Jongsoo Park, Xing Liu, and George Karypis. FROSTT: The Formidable Repository of Open Sparse Tensors and Tools, 2017.

[114] Shaden Smith and George Karypis. A medium-grained algorithm for distributed sparse tensor factorization. In *Parallel and Distributed Processing Symposium (IPDPS), 2016 IEEE International*. IEEE, 2016.

[115] Shaden Smith, Niranjay Ravindran, Nicholas Sidiropoulos, and George Karypis. SPLATT: Efficient and parallel sparse tensor-matrix multiplication. In *Proceedings of the 29th IEEE International Parallel & Distributed Processing Symposium*, IPDPS, 2015.

[116] Shaden Smith, Niranjay Ravindran, Nicholas Sidiropoulos, and George Karypis. SPLATT: The Surprisingly ParalleL spArse Tensor Toolkit (Version 1.1.1). Available from `https://github.com/ShadenSmith/splatt`, 2016.

[117] Marc Snir, Robert W Wisniewski, Jacob A Abraham, Sarita V Adve, Saurabh Bagchi, Pavan Balaji, Jim Belak, Pradip Bose, Franck Cappello, Bill Carlson, et al. Addressing failures in exascale computing. *The International Journal of High Performance Computing Applications*, 28(2):129–173, 2014.

[118] Paulo Sousa, Nuno Ferreira Neves, and Paulo Veríssimo. Resilient state machine replication. In *11th Pacific Rim International Symposium on Dependable Computing (PRDC'05)*, pages 5–pp. IEEE, 2005.

[119] Omer Subasi, Osman Unsal, and Sriram Krishnamoorthy. Automatic risk-based selective redundancy for fault-tolerant task-parallel hpc applications. In *Proceedings of the Third International Workshop on Extreme Scale Programming Models and Middleware*, pages 1–8, 2017.

[120] Omer Subasi, Gulay Yalcin, Ferad Zyulkyarov, Osman Unsal, and Jesus Labarta. Designing and modelling selective replication for fault-tolerant hpc applications. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 452–457. IEEE, 2017.

[121] Li Tan. *The interplay between energy efficiency and resilience for scalable high performance computing systems*. University of California, Riverside, 2015.

[122] Dingwen Tao, Shuaiwen Leon Song, Sriram Krishnamoorthy, Panruo Wu, Xin Liang, Eddy Z. Zhang, Darren Kerbyson, and Zizhong Chen. New-sum: A novel online abft scheme for general iterative methods. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '16, pages 43–55, New York, NY, USA, 2016. ACM.

[123] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of collective communication operations in mpich. *The International Journal of High Performance Computing Applications*, 19(1):49–66, 2005.

[124] Devesh Tiwari, Saurabh Gupta, James Rogers, Don Maxwell, Paolo Rech, Sudharshan Vazhkudai, Daniel Oliveira, Dave Londo, Nathan DeBardeleben, Philippe Navaux, et al. Understanding gpu errors on large-scale hpc systems and the implications for system design and operation. In *HPCA*, pages 331–342. IEEE, 2015.

[125] Robert A Van De Geijn and Jerrell Watts. SUMMA: Scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience*, 9(4):255–274, 1997.

[126] Long Wang, Karthik Pattabiraman, Zbigniew Kalbarczyk, Ravishankar K Iyer, Lawrence Votta, Christopher Vick, and Alan Wood. Modeling coordinated checkpointing for large-scale supercomputers. In *2005 International Conference on Dependable Systems and Networks (DSN'05)*, pages 812–821. IEEE, 2005.

[127] Eric W Weisstein. Gershgorin circle theorem. *https://mathworld. wolfram. com/*, 2003.

[128] Panruo Wu, Nathan DeBardeleben, Qiang Guan, Sean Blanchard, Jieyang Chen, Dingwen Tao, Xin Liang, Kaiming Ouyang, and Zizhong Chen. Silent data corruption resilient two-sided matrix factorizations. *SIGPLAN Not.*, 52(8):415–427, January 2017.

[129] Panruo Wu, Qiang Guan, Nathan DeBardeleben, Sean Blanchard, Dingwen Tao, Xin Liang, Jieyang Chen, and Zizhong Chen. Towards practical algorithm based fault tolerance in dense linear algebra. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, pages 31–42. ACM, 2016.

[130] Zhiwei Xu and Kai Hwang. Modeling communication overhead: Mpi and mpl performance on the ibm sp2. *IEEE Parallel & Distributed Technology: Systems & Applications*, 4(1):9–24, 1996.

[131] Yavuz Yetim, Sharad Malik, and Margaret Martonosi. Eprof: An energy/performance/reliability optimization framework for streaming applications. In *17th Asia and South Pacific Design Automation Conference*, pages 769–774. IEEE, 2012.

[132] John W Young. A first order approximation to the optimum checkpoint interval. *Communications of the ACM*, 17(9):530–531, 1974.

[133] Ziming Zheng and Zhiling Lan. Reliability-aware scalability models for high performance computing. In *2009 IEEE International Conference on Cluster Computing and Workshops*, pages 1–9. IEEE, 2009.