5-2017

Extensions of Task-based Runtime for High Performance Dense Linear Algebra Applications

Chongxiao Cao
University of Tennessee, Knoxville, ccao1@vols.utk.edu

To the Graduate Council:

I am submitting herewith a dissertation written by Chongxiao Cao entitled "Extensions of Task-based Runtime for High Performance Dense Linear Algebra Applications." I have examined the final electronic copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Computer Science.

Jack Dongarra, Major Professor

We have read this dissertation and recommend its acceptance:

James Plank, Michael Berry, Yingkui Li

Accepted for the Council:
Dixie L. Thompson

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

Extensions of Task-based Runtime for High Performance Dense Linear Algebra Applications

A Dissertation Presented for the

Doctor of Philosophy

Degree

The University of Tennessee, Knoxville

Chongxiao Cao

May 2017

To my parents Zhixiang Cao and Huifen Ye, my brother Chonghan Cao and my girlfriend Ling Li for their constant love, trust, and support.

Acknowledgements

It has been an invaluable experience to spend the past six years at the University of Tennessee, from both professional, doing research in High Performance Computing, and personal, enjoying the city and the beauty of the Great Smoky Mountains, point of view. The work presented in this dissertation would not have been completed without the help of many people I would like to acknowledge here.

I would like to thank my advisor, Dr. Jack Dongarra. Six years ago, Dr. Dongarra gave me the opportunity of being a graduate research assistant at Innovative Computing Laboratory (ICL). His passion and curiosity for HPC research have always been my inspiration and motivation. His talks in conferences, weekly meetings and ICL lunch talks always helped me widen my horizons in innovative ideas in HPC. Dr. Dongarra has provided me extensive support and generous financial support throughout the whole process of this work.

In addition, I would like to thank Dr. Michael Berry, Dr. James Plank, and Dr. Yingkui Li for agreeing to serve on my graduate committee. I greatly appreciate their time and invaluable advice to this dissertation.

My appreciation goes to my co-advisor, Dr. George Bosilca, for helping me dive into the distributed computing area, teaching me coding and debugging skills for MPI and PaRSEC, and for enlightening discussions and guidance about Fault Tolerance. This dissertation could not have been written without Dr. Bosilca's insightful suggestions, mentorship, and encouragement.

I would also like to express my appreciation to many current and former ICLers. I am deeply grateful for Dr. Stanimire Tomov and Dr. Azzam Haidar. Dr. Tomov taught me lots of skills related to GPU programming. Dr Haidar shared with me a lot of ideas about optimizing linear algebra kernels on heterogeneous platform. I want to thank Dr. Thomas Herault for the help with debugging code in PaRSEC, and thank Dr. Yulu Jia for inspiring discussion related to Algorithm Based Fault Tolerance. I also want to thank Dr Fengguang Song, who has been very helpful in providing suggestions on my research and suggestions on various logistical issues. Special thank also goes to Wei Wu, who has been standing together with me during the whole PhD program, from studying graduate courses to writing dissertation, contributing lots of discussion to my research.

Finally, I am deeply indebted to my family and friends. I owe thanks to my parents, Zhixiang Cao and Huifen Ye, my brother Chonghan Cao and my girlfriend Ling Li for their love, trust, and support that are crucial to the completion of this work. I thank my friends, Sheng Hu, Xiaohu Zhang, Kang Pan, Sijia Liu for the friendship life long.

"Knowledge is power." - Sir Francis Bacon

Abstract

On the road to exascale computing, the gap between hardware peak performance and application performance is increasing as system scale, chip density and inherent complexity of modern supercomputers are expanding. Even if we put aside the difficulty to express algorithmic parallelism and to efficiently execute applications at large scale, other open questions remain. The ever-growing scale of modern supercomputers induces a fast decline of the Mean Time To Failure. A generic, low-overhead, resilient extension becomes a desired aptitude for any programming paradigm. This dissertation addresses these two critical issues, designing an efficient unified linear algebra development environment using a task-based runtime, and extending a task-based runtime with fault tolerant capabilities to build a generic framework providing both soft and hard error resilience to task-based programming paradigm.

To bridge the gap between hardware peak performance and application performance, a unified programming model is designed to take advantage of a lightweight task-based runtime to manage the resource-specific workload, and to control the dataflow and parallel execution of tasks. Under this unified development, linear algebra tasks are abstracted across different underlying heterogeneous resources, including multicore CPUs, GPUs and Intel Xeon Phi coprocessors. Performance portability is guaranteed and this programming model is adapted to a wide range of accelerators, supporting both shared and distributed-memory environments.

To solve the resilient challenges on large scale systems, fault tolerant mechanisms are designed for a task-based runtime to protect applications against both soft and hard errors. For soft errors, three additions to a task-based runtime are explored. The first recovers the application by re-executing minimum number of tasks, the second logs intermediary data between tasks to minimize the necessary re-execution, while the last one takes advantage of algorithmic properties to recover the data without re-execution. For hard errors, we propose two generic approaches, which augment the data logging mechanism for soft errors. The first utilizes non-volatile storage device to save logged data, while the second saves local logged data on a remote node to protect against node failure. Experimental results have confirmed that our soft and hard error fault tolerant mechanisms exhibit the expected correctness and efficiency.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Today's fastest supercomputers can solve problems at petascale, that is, a quadrillion (10^{15}) floating point operations each second. The number of components of supercomputers, such as CPU cores, accelerators, memory size, network bandwidth, and storage size grow exponentially. Today's most powerful supercomputer, Sunway TaihuLight Top500 (2016), from National Supercomputing Center in Wuxi, China, harnessed $10,649,600$ cores to achieve its theoretical peak performance of 125 PFlop/s to rank No.1 on November 2016 Top500 list. While these petascale supercomputers are quite powerful, the next milestone in computing achievement expected by 2018-2022, is the exascale, that is, 10^{18} floating point operations each second. As the size of supercomputer grows larger and its infrastructure becomes more complicated, it is difficult to express parallelism of applications in an efficient and reliable way on such systems.

The components of supercomputers continue to become more complicated on the road to exascale computing at different levels: multi-core CPUs with Non Uniform Memory Access (NUMA), integration of accelerators (GPU, Intel Xeon Phi), complex network interconnection and multi-level storage hierarchies. Performance portability is not guaranteed to scale in the same order with system size. It is challenging to express the algorithmic parallelism efficiently on a large scale

system, meanwhile debugging, maintaining and providing performance portability across different parallel architectures and programming environments. Task-based programming model has emerged as solution to address this challenge effectively. By using this programming model, an application is represented as a set of tasks and data dependencies between these tasks in the form of Directed Acyclic Graph (DAG). A dynamic runtime engine is designed to discover available tasks by inferring the data dependencies between the tasks and to schedule tasks to corresponding devices during execution.

The future exascale systems will be much more vulnerable to failures than current petascale systems. Two major reasons leading to this trend are: (1) the number of components required to achieve the scale is increasing; (2) an increase of Mean Time To Failure (MTTF) of each component will not be high enough to compensate the impact of the first one. Nowadays, prevailing academic thought is that the MTTF of supercomputers might drop to about one hour in the next few years Cappello (2009).

Developing a programming environment capable of delivering computation at large scale in an efficient and resilient way, will address a major challenge to fully utilize future High Performance Computing (HPC) systems to provide scientific productivity. Popular works addressing failures on large scale systems can be categorized into two types: Checkpoint/Restart (C/R) and Algorithm Based Fault Tolerance (ABFT) techniques. Both of these two methods are applicable to protect again failures on large machines, and each has its own advantages and drawbacks. C/R technique is generic in nature and highly automatic. It supports a wide range of applications but suffers relatively high checkpointing overhead as it relies on backing up data to stable storage. On the other hand, ABFT technique provides relatively low overhead but it is less generic in feature. Today, most of ABFT works are developed to protect applications in linear algebra and fast Fourier transform (FFT) domains.

In this dissertation, a unified programming model is designed to provide a light weighted environment for developing high performance dense linear algebra

applications. Applications are represented as the form of DAG, and a dynamic task-based runtime is utilized to manage device-specific workload, to manipulate dataflows and to schedule tasks in parallel. This unified programming model is enabled by taking advantage of task abstraction to support different heterogeneous devices, ranging from multi-core CPUs, NVIDIA GPUs, AMD GPUs and Intel Xeon Phi coprocessors. We provide implementation of Cholesky factorization, validating that this unified design is effective for performance portability and taking full utilization of a mix of different accelerators. Also, fault tolerant mechanisms are explored and added to a dynamic task-based runtime to build a generic framework enabling both soft-error and hard-error resilience. Dense linear applications running on such a task-based runtime obtain automatic resilient support from runtime level. We focus on protecting application data against failures. Our goal is to design generic and low-overhead solutions to handle both soft and hard errors. By combining the advantages of algorithmic properties of applications, parameterized presentation of tasks and automatic generation of minimum required execution graph in a task-based runtime, our resilient solutions guarantee that data and execution flow are secure in an error-prone environment with low cost of computational overhead and storage overhead. We design three mechanisms at two levels of granularities to handle soft errors: at the coarse level, tasks are required to re-execute to generate correct result, and at the fine level, tasks are augmented to integrate necessary algorithmic properties to handle possible data corruption. Two generic mechanisms are also designed to handle hard errors, which augment generic data logging solution for soft errors by taking advantage of reliable secondary storage and remote compute node to save intermediary dataflow. Our design is illustrated by using Cholesky factorization as a case study. We implement this resilient design in the PaRSEC Bosilca et al. (2013) framework, which uses a dynamic runtime engine to efficiently manage data dependencies and schedule tasks on distributed heterogeneous platforms.

1.1 Thesis Statement

The main objective of this dissertation is to demonstrate that dynamic task-based runtime can be extended to facilitate the development of high performance dense linear algebra applications on large scale systems toward future exascale computing. This dissertation addresses these two major challenges. The first challenge is designing a programming model which utilizes a task-based runtime to facilitate the development of high performance dense linear algebra on heterogeneous platforms. The second challenge is the fault tolerant design for a task-based runtime to handle both soft and hard errors. The proposed methodologies should guarantee low computational and storage overhead to protect application data in a failure-prone environment.

1.2 Contribution

The contribution of this dissertation consists of two parts: efficient utilization of a task-based runtime for unified linear algebra development and resilient extension to a task-based runtime.

1.2.1 Unified Linear Algebra Development

- **clMAGMA:** We design the static scheduling in clMAGMA Cao et al. (2014) library to implement high performance dense linear algebra for hybrid systems consisting of CPUs and OpenCL devices.

- **Unified Linear Algebra Development using QUARK:** We design a unified programming model of dynamic scheduling using a task-based runtime (QUARK in our implementation) to implement high performance dense linear algebra. We present how modularized methodologies are adapted to utilize a task-based runtime to guarantee that two computational goals are reached: (1)

to achieve optimal performance on the entire heterogeneous platform, (2) to facilitate the development by using a task graph programming model.

- **Supporting Distributed-memory Platform:** We propose the extensions for the unified programming model using QUARK to support distributed-memory platforms. As QUARK itself only works in a shared-memory environment, the extensions here include strategies varying from the way data is stored and moved to the way algorithms are split into tasks and scheduled for execution.

1.2.2 Resilient Design for a Task-based Runtime

Several fault tolerant techniques are developed in this dissertation such that both soft and hard errors can be tolerated, delivering a resilient execution environment for dense linear algebra applications using a task-based runtime.

Soft Errors

- **Sub-DAG Recovery Mechanism:** When a soft error strikes the output of a task during execution, a sub-DAG composed of the failed tasks and all its necessary predecessors from original input is created. This sub-DAG consists of minimum requirement to regenerate correct result for the failed task from original input, and is executed in parallel with other non-failed tasks in original DAG.

- **Data Logging Recovery Mechanism:** In order to reduce the recovery overhead, intermediary dataflow in the original execution graph is reserved periodically, by saving a copy into memory. When a soft error strikes a task, a sub-DAG consisting of only predecessors between the failed task and the reserved dataflow is created. By taking extra memory to log dataflow, the re-execution of tasks is bounded by latest version of saved data, and a relatively smaller sub-DAG is created and scheduled by runtime to recover the failure.

- **Algorithm Based Fault Tolerance Mechanism:** By exploiting the feature of ABFT, two checksum vectors are appended to every matrix tile of original data layout. It has been proved that checksum vectors keep consistent with corresponding matrix data during factorization. After every task completes, checksum vectors are utilized to validate result and correct errors. ABFT mechanism takes extra computational overhead to keep checksum vectors valid during execution, and avoids task re-execution after a failure happens. It is also able to serve as a software level failure detector.

- **Formal and experimental performance analysis:** A thorough examination of the theoretical computation complexity of the fault tolerant mechanisms is provided. We calculate the number of extra floating point operations (FLOPS) incurred by the different resilient mechanisms, and compare the extra FLOPS with the FLOPS of the original application. We also verify the theoretical analysis through experiments.

Hard Errors

- **Non-volatile Storage Mechanism:** We augment the data logging mechanism for soft errors to support hard errors. After a hard error happens, reserved dataflow in memory is also lost when the process crashes. High-speed non-volatile storage such as solid-state drive (SSD) and Non-volatile random-access memory (NVRAM) can be utilized to save intermediary dataflow. After a hard error strikes, the dataflow saved in the non-volatile storage of the crashed process, and other dataflow saved in the memory of non-crashed processes can be combined together to rebuild a restarting state for the crashed process.

- **Remote Data Logging Mechanism:** Another low-overhead mechanism to handle hard errors is designed by saving intermediary dataflow on a remote process. A remote process protects the data that the crashed process requires to restore. We augment the communication engine in PaRSEC to be adaptable

for hard error resilience, and exploit the parameterized representation of tasks in PaRSEC to reduce communication overhead introduced by saving data remotely.

1.3 Dissertation Outline

The rest of this dissertation is organized as follows: Chapter 2 introduces task graph scheduling using dynamic runtimes and the source of failures in high performance computing. We also review pioneering works done to develop high performance dense linear algebra libraries and to mitigate the impact of failures on large scale applications. Chapter 3 presents a unified programming model using a task-based runtime that mitigates the difficulty of dealing with different underling devices and programming libraries during HPC application development. Chapter 4 presents three mechanisms for soft error recovery, including sub-DAG recovery mechanism, data logging recovery mechanism and algorithm based fault tolerance mechanism. Chapter 5 presents two mechanisms to support hard error resilience, including non-volatile storage mechanism and remote data logging mechanism. Finally Chapter 6 concludes the dissertation and outlines future directions.

Chapter 2

Background

In this chapter we introduce the background of task graph scheduling using dynamic runtimes and the source of failures in high performance computing. We also review pioneering works done to develop high performance dense linear algebra libraries and to mitigate the impact of failures on applications.

2.1 Task Graph Scheduling using Dynamic Runtimes

Task Graph is a classical programming model that has been used to express task dependencies and to explore parallelism. In this dissertation, a task graph is defined as a Directed Acyclic Graph (DAG) $D = (V, E)$, where every vertex $v \in V$ represents a task (a set of sequential computations), and every edge $(v_1, v_2) \in E$ represents a data dependency between an output of task v_1 and an input of task v_2. An edge (v_1, v_2) that exists between task v_1 and task v_2 implies that task v_2 can only start after task v_1 completes and its output is received by task v_2. In this dissertation, our work is implemented on two representative task-based runtimes: QUARK and PaRSEC.

QUARK (QUeuing And Runtime for Kernels) YarKhan et al. (2011) is a lightweight runtime environment, it provides a scheduling engine that enables dynamic discovery and execution of tasks with data dependencies in a shared-memory environment. It is developed by the Innovative Computing Laboratory (ICL) from University of Tennessee. QUARK infers data dependencies and precedence constraints between tasks from the way that the data is used, and then executes the tasks in an asynchronous, dynamic fashion in order to achieve a high utilization of the available resources. It is the dynamic runtime engine used within the PLASMA linear algebra library and has been proved to deliver high productivity and performance benefits Haidar et al. (2011).

The Parallel Runtime Scheduling and Execution Controller (PaRSEC) is a generic framework for architecture-aware scheduling and management of micro-tasks on distributed many-core heterogeneous architectures. It is also developed by ICL from University of Tennessee. The core components of PaRSEC runtime are one dynamic multi-level scheduler supporting distributed-memory environment, one communication engine supporting asynchronous data transfer and one data dependencies engine parsing task availability Bosilca et al. (2011). Tasks are mapped to corresponding computing nodes by runtime based on data distribution. Both local and remote data dependencies are detected and enabled. The dynamic scheduler explores the maximum amount of parallelism by scheduling available tasks to underlying multi-core CPUs and accelerators. PaRSEC also follows the task graph programming model.

Figure 2.1 presents the detailed framework of PaRSEC. The lowest level is the support for different hardware architecture, including multi-core CPUs, memory hierarchies, cache coherence, and accelerators. The middle level is the functionalities from the parallel runtime in PaRSEC, including distributed scheduling, data movement, data collections, managing task classes and creating specialized kernels. The top level is the extensions for domain specific applications, including a concise format of representing tasks called Parameterized Task Graph (PTG) Cosnard et al. (1999), a

dynamic representation of tasks called Dynamic Task Discovery (DTD) Haidar et al. (2011), and current supported applications from dense linear algebra, sparse linear algebra to Chemistry.

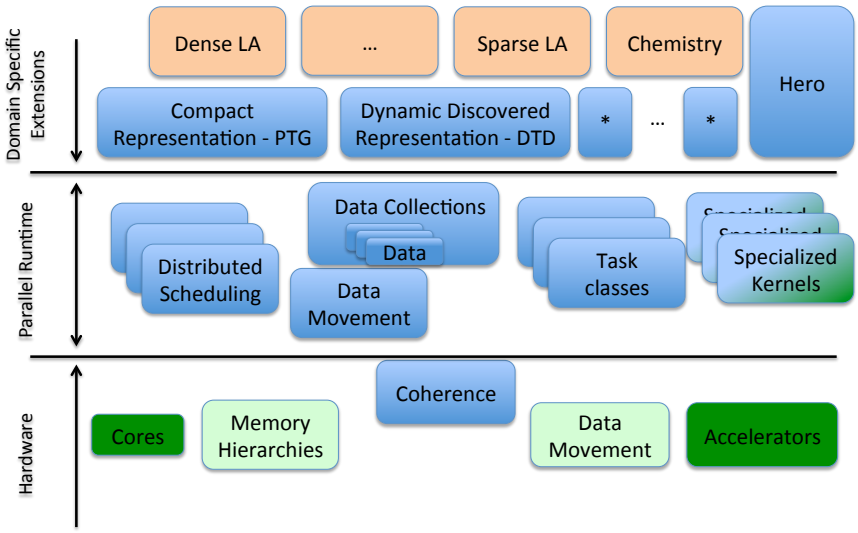

Figure 2.1: The Framework of PaRSEC.

2.2 Source of Failures

Failures have been unavoidable since the birth of computers. Resilience has become a major challenge for large scale systems over the past few years. These systems will typically gather from half a million to several millions of CPU cores running up to a billion of threads. Based on today's observations and research of statistics of failures on large scale systems, it is estimated that next generation exascale systems will be struck by multiple types of failures many times per day Cappello et al. (2009). According to failure statistic of machines at Oak Ridge National Laboratory and Los Alamos National Laboratory, three major threats for HPC systems are cosmic rays, bad solder and reducing power consumption Geist (2016):

- **Cosmic rays:** The amount of energy required to flip a bit in a transistor is decreasing as the size of the transistor gets smaller. By predicting that the size of transistors on future exascale systems will be about a third of the size it is today, there will be much more likely to introduce cosmic ray-induced errors.

- **Bad solder:** Radioactive lead may happen in the bad solder used to make the boards carrying the processors, causing bad data in the L1 cache.

- **Reducing power consumption:** Saving power is a goal of building an exaflop computer, the power savings will likely have to come from smaller transistors running at lower voltages to draw less power, which increases the probability of circuits flipping state spontaneously. Also, power cycling reduces a chip's lifetime.

Faults can be rooted in software issues as well. Parallel applications are especially difficult to develop and may have potential bugs left. If bugs are in system level and can cause the system to provide incorrect service to application codes, these bugs are considered as software faults Jia (2015).

In this dissertation we consider two different types of failures for applications running on a task-based runtime: hard errors and soft errors.

- **Hard Error:** We define hard errors as process failures, where a process is crashed after a failure happens, and the corresponding data on the failed process is lost. When a hard error happens, the application cannot continue due to lost of data.

- **Soft Error:** Soft error is defined as silent data corruption (SDC), usually manifests as bit-flips in memory, cache or processor registers. The application will not terminate when a soft error occurs, it continues executing without noticing it, and delivers wrong result at the end.

Soft errors have been highlighted as the continuous increase of memory used by applications. Compared with hard errors, soft errors are more dangerous because

of the transient feature. An application will continue to execute when a soft error shows up and deliver wrong result after completion.

2.3 Dense Linear Algebra Libraries on Distributed Heterogeneous Systems

There has been a lot of effort on enabling dense linear algebra libraries to run on heterogeneous systems. Vendors such as NVIDIA, Intel, and AMD provide their own numerical libraries, such as cuBLAS NIVIDIA (2017), MKL Intel (2016), and clBLAS AMD (2015), respectively. These libraries do not include implementations for distributed-memory systems yet.

MAGMA Agullo et al. (2009) is a linear algebra library designed for heterogeneous architectures from ICL, University of Tennessee. Linear algebra algorithms are scheduled statically in MAGMA by moving computational intensive operations to accelerators while keep communication bound ones on CPU side. Tasks are distributed equally across multiple accelerators. Most of factorizations provided by MAGMA only support shared-memory systems.

Song et al. Song et al. (2012) describe distributed-memory, multi-GPU linear algebra algorithms that use a static multi-level block-cyclic data partitioning. The static data layout allows the distributed nodes to schedule communication events without coordination. The multi-level data scheme enables CPUs and GPUs to partition work to handle the workload imbalance between the resources. This approach does not provide for GPUs of different strengths and for the addition of other resources such as Intel Xeon Phi coprocessors.

Ayguade et al. have created StarSS Ayguadé et al. (2009), a programming system that uses compiler directives to annotate code in order to allow task superscalar execution via a specialized runtime. The directives can specify that functions should be executed using specific hardware (e.g. GPU, Cell, SMP) rather than using CPUs.

The superscalar execution allows the host CPU and additional hardware to run in parallel. Many of the ideas in StarSS have been incorporated in the implementation of Task Parallelism in the OpenMP 4.0 specification Board (2013), however the OpenMP standard does not include distributed-memory execution.

The INRIA Runtime team has developed StarPU Augonnet et al. (2009), which is a dynamic scheduling runtime that uses superscalar execution methods to run sequential task-based code on parallel resources. StarPU uses a history-based scheduling mechanism to transparently schedule tasks on heterogeneous multicore and GPU resources, with extensions that allow StarPU to execute in distributed-memory environments. StarPU has been used as a runtime in MAGMA to implement the Cholesky, QR, and LU factorizations Agullo et al. (2011).

The SuperMatrix runtime system for linear algebra was extended to execute on multicore and GPUs in a shared-memory environment Chan et al. (2007). The SuperMatrix approach requires that the task-dependencies be substantially exposed before scheduling and that the GPU take the burden of the computation, not using available multicore CPUs for complex computational tasks.

2.4 Existing Fault Tolerant Techniques

Numerous methods have been proposed to protect against different types of failures at various levels of the architecture, ranging from underlying hardware level approach to the user application software level approach at the top. Several existing fault tolerant technologies have been developed and provided satisfactory results on current petascale supercomputers. To put our proposed methods into perspective, a systematic view of the related work is given in this section.

2.4.1 Hardware Duplication

A straightforward way to tolerate hardware failures is to use hardware redundancy. Hardware redundancy can be categorized into two types: passive hardware redundancy and active hardware redundancy. The basic idea of passive hardware redundancy is to execute the same program by several independent modules. After every module completes, the final result is determined by a majority-voting system to produce a single output. TMR (Triple Modular Redundancy) Lyons and Vanderkulk (1962) is a classical type of passive hardware redundancy, if any one of the three systems fails, the other two systems can correct and mask the fault. The first use of TMR in a computer was the Czechoslovak computer SAPO Howlett and Rota (1980), in the 1950s. TMR can be extended to N-modular redundancy (NMR). Active hardware redundancy, on the other hand, only keeps one redundant unit running at the same time. The STAR (Self Testing And Repair) computer constructed at the Jet Propulsion Laboratory, is an early autonomous computer system that could detect and recover failures Avizienis et al. (1971). STAR computer is designed with dynamic redundancy, consisting of replaceable components and a program rollback provision to recover transient errors. Every component in STAR has several backup units, and at any given time only one unit is powered and working. Hardware duplication is an effective way to increase the MTBF (mean time between failure) of the entire system. However, the financial cost of building a fault tolerant system increases proportionally with the number of redundant components. Modern supercomputers are composed of millions of cores, which makes this method impractical.

2.4.2 Error Correcting Code

Cosmic rays can strike transistors of dynamic random-access memory (DRAM) to cause a single bit flip, resulting in an opposite value. As the density of DRAM is increasing to reach higher memory size, and the size of transistors on chips gets smaller, meanwhile the energy required to spontaneously flip the bits on DRAM

is decreasing. Future DRAMs will be much more prone to cosmic ray-induced errors Mittal and Vetter (2016). A hardware level solution to solve this challenge is to use extra memory bits and memory controllers to protect original bits. These extra bits are used to record parity or to use an error-correcting code (ECC). Parity allows the detection of all single-bit errors (to be more precise, any odd number of wrong bits). The single-error correction and double-error detection (SECDED) Hamming code is the most widely used error correcting code. It provides single-bit error correction and double-bit errors detection in every 64-bit memory word. There is a trade-off between capabilities of handling bit flips and a higher commercial cost when using ECC memory. Compared with non-ECC memory, the price of EEC memory is higher, as extra hardware components are added to implement ECC functionalities. Also, ECC may lower memory bandwidth by 2-3 percent on some systems, as ECC memory controllers require extra time to perform error detection and correction Wikipedia (2017a). Many CPUs have equipped ECC in its on-chip cache, for example, Intel Itanium processor, AMD Athlon and Opteron processors, and the DEC Alpha 21264 Yoon and Erez (2009).

2.4.3 Rollback Recovery based on Checkpoint and Message Logging

Rollback recovery scheme considers a distributed system as a collection of application processes that communicate through a network. Each process is equipped with a stable storage device that is robust under failures. During execution, intermediary data is stored to stable storage periodically. When a failure happens, the failed process uses the newest version of saved data to restart the execution. The saved data serving as the starting state for the recovery, is called checkpoints. Message logging-based rollback recovery combines checkpointing scheme with logging of nondeterministic events. It follows the the piecewise deterministic (PWD) assumption Strom and Yemini (1985) that all nondeterministic events that a process executes can be retrieved

later after storing necessary information safely during original execution. By using stored information to replay the nondeterministic events in their original order, a failed process can be recovered.

Checkpoint-based rollback-recovery techniques can be categorized into three types: *uncoordinated checkpointing*, *coordinated checkpointing*, and *communication-induced checkpointing*. By using *uncoordinated checkpointing*, every process makes its own decision about when to checkpoint necessary data. *Uncoordinated checkpointing* is simple to implement, however, under some extreme conditions, it may lead to the domino effect Randell (1975). The domino effect means that rollback is propagated to the beginning of the application, repeating all the computation completed before a failure happens. Several techniques have been designed to avoid the domino effect. One well-known technique is named *coordinated checkpointing* in which all the processes coordinate their checkpoints in order to create a consistent snapshot for the whole application Chandy and Lamport (1985). Rollback propagation is limited by such a consistent snapshot. The other technique is called *communication-induced checkpointing*, which allows every process to take checkpoints based on the information induced by communication with remote process Russell (1980).As checkpoints are stored on stable storage, a system-wide consistent state always exists and the domino effect is avoided. Checkpoint-based rollback recovery does not rely on the PWD assumption, and so does not need to detect, log, or replay nondeterministic events. Compared with log-based rollback recovery, checkpoint-based rollback recovery has less restrictions and is simpler to implement. However, checkpoint based rollback recovery does not guarantee that execution before a failure can be deterministically regenerated after a rollback Elnozahy et al. (2002), thus it is not suitable for applications that require frequent interactions with the outside world.

As opposed to checkpoint-based rollback recovery, log-based rollback recovery makes explicit use of the fact that a process execution can be modeled as a sequence of deterministic state intervals, each starting with the execution of a nondeterministic event Strom and Yemini (1985). Log-based rollback-recovery techniques guarantee

that during the recovery of failures, there is no orphan process (i.e., a process whose state depends on a nondeterministic event that cannot be reproduced during recovery) left in the system. Specific message logging implementation delivers different failure-free performance overhead and rolling back recovery overhead. Pessimistic message logging-based rollback-recovery technique guarantees that orphan processes are never created after a failure happens. It simplifies the recovery while introducing higher failure-free performance overhead. Optimistic message logging-based rollback-recovery technique reduces the failure-free performance overhead, but allows orphan processes to be created after a failure happens. Leaving orphan processes in the system complicates the recovery. Causal message logging-based rollback-recovery technique is a combination of the abovementioned two techniques, which attempts to get low performance overhead but requires more complicated implementation.

2.4.4 Algorithm Based Fault Tolerance

Algorithm Based Fault Tolerance (ABFT), which initially stemmed from the effort of detecting and correcting errors caused by permanent or transient failures in the hardware Huang and Abraham (1984), and is now widely adapted in dense linear algebra, sparse linear algebra and fast Fourier transform (FFT) to recover failures. The basic idea of ABFT is to add redundant data in the form of checksum to original compute data. ABFT maintains consistency between the checksum and compute data by applying appropriate mathematical operations to both parties. Typically, in dense linear algebra, original matrix is extended by row-checksum vectors or column-checksum vectors. This encoding happens only once before the computation, and original matrix algorithm is modified to update checksums with appropriate mathematical operations during computation, which enables checksums to keep consistent relationship with compute data. Whenever a failure strikes the compute data, checksums are inverted to recreate missing data. When applicable, ABFT provides resilience with very low overhead as there is no periodical checkpoint

or rollback recovery involved, and the extra computational operations updating checksums is in a lower order compared with original algorithm. ABFT techniques are well developed in dense linear algebra to handle hard errors and soft errors. For example, Du et al. have shown that by combining ABFT and diskless checkpoint, a full matrix protection solution with low space and time overhead is implemented for LU and QR factorizations against hard errors Du et al. (2012). Jia et al. extend this method a step further to support Parallel Reduction to Hessenberg Form in two-sided factorizations Jia et al. (2013). Toward soft errors, Du et al. Du et al. (2011a) apply Sherman-Morrison formula to recover soft errors happening on dense linear system solver. FT-ScaLAPACK library Wu and Chen (2014) integates ABFT functionalities to into LU, QR and Cholesky factorizations in ScaLAPACK Blackford et al. (1996), by making every update on a matrix block robust against soft errors. ABFT enabled soft error protection methods have also been explored to support one-sided and two-sided factorizations on CPU-GPU heterogeneous computing platforms. In a series of work related to resilient QR factorization Du et al. (2011b) on hybrid system and resilient Hessenberg reduction on hybrid system Jia et al. (2016), it has been shown that in the presence of round-off error on heterogeneous system, soft errors in both the left and right factors in dense linear algebra operations can be detected and corrected.

2.4.5 Application Driven Fault Mitigation

Fault tolerance can be implemented in software level. Developers can add extra codes in original application explicitly to recover possible failures during execution. In order to inject recovery codes into applications, underlying libraries should export necessary functionalities and interfaces to support resilience. The idea of application driven fault mitigation is to provide resilient support in programming model, which enables the application to use the programming model to handle failures. User Level Failure Mitigation (ULFM) Herault et al. (2015) is a set of new interfaces for MPI that enables Message Passing programs to restore MPI functionality affected by process failures.

Using the fault tolerant interfaces defined by ULFM, applications and libraries handle the recovery of the MPI state by themselves. Consistency issues resulting from failures are addressed according to an application's needs and the recovery actions are limited to minimum MPI communication objects. Therefore, the recovery scheme is more efficient than a generic, automatic recovery technique, and can achieve both goals of enabling applications to resume communication after failure and maintaining extreme communication performance outside of recovery periods.

2.4.6 Fault-tolerant Task-based Systems

Fault tolerance has been implemented into many task-based runtimes, providing a resilient running environment for applications. Task scheduling can be static or dynamic, depending on whether an application's task graph is known before computation starts Johnson (1993). In static scheduling systems, tasks are allocated to processes or computing nodes ahead of time. In order to handle hard errors in static task-based systems, tasks are duplicated and distributed to different processes in case process failure happens. Such task duplication strategy has been applied in grids Fechner et al. (2008) and in real-time systems Qin and Jiang (2006). However, task duplication repeats the execution for protected task and introduces high performance penalty in failure-free execution. On the other hand, dynamic task-based systems distribute tasks to processors during the execution. This requires the runtime to recover the failure efficiently during computation without introducing significant performance penalty. Tremendous works have been proposed to implement resilience in dynamic task-based systems. Kepler scientific workflow system provides checkpointing and re-execution mechanisms to handle both hard and soft errors Mouallem et al. (2010). However, the goal of fault tolerance framework in a workflow system is to provide an appropriate end-to-end support for handling failures, not to focus on a low-overhead solution. The NABBIT system Agrawal et al.

(2010) implements a fault tolerant work stealing algorithm Kurt et al. (2014) to re-execute minimum number of tasks for known data corruption reported by underlying hardware (i.e., no soft error detectors) in a shared-memory environment. An improved coordinated checkpoint and rollback recovery mechanism is implemented in KAAPI framework Gautier et al. (2007) to handle hard errors, it reduces the number of processes that are required to rollback by exploring the communication dependencies Besseron et al. (2006).

Chapter 3

Unified Linear Algebra Development using a Task-based Runtime

3.1 Introduction

Both academia and industry have been enjoying the performance benefits provided by GPU since it was released. The ever expanding capabilities of the hardware accelerators allowed GPUs to deal with more demanding kinds of workloads and there was very little need to mix different GPUs in the same machine. Many Integrated Cores (MIC) known as Xeon Phi, is a from of coprocessor in the realm of hardware acceleration provided by Intel. Considering computational capabilities, Xeon Phi delivers similar performance to GPU but on the other hand, Xeon Phi handles specific size of workloads that is different with GPU. For any application running on a heterogeneous platform, the optimal performance can be obtained by combining CPUs, GPUs and coprocessors together, splitting size of workload into appropriate one for each device in order to leverage their maximum strength. This scenario is called multi-way heterogeneity. Programming on different devices

requires dealing with different software libraries, which increases the complexity of developing high performance applications on heterogeneous platforms. In this chapter, we present a unified programming model that alleviates the complexity of dealing with multiple software stacks for computing, communication, and software libraries. This programming model addresses the design of high-performance dense linear algebra (DLA) in heterogeneous environments, consisting of a mix of multi-core CPUs, GPUs, and Intel Xeon Phi coprocessors (MICs). This mix can consist of two levels: (1) combination of different accelerators; (2) combination of same accelerator with various capabilities, e.g., GPUs from different vendors and GPUs from the same vendor under different device generations. While the main goal is to reach as high fraction of the peak performance as possible for an entire heterogeneous system, a competing secondary goal is to propose a programming model that would alleviate the burden from development. To achieve these two goals, a generic lightweight environment is designed by utilizing a task-based runtime, and several popular dense linear algebra routines are implemented in this environment. We demonstrate the new algorithms, their performance, and the programming model design using the Cholesky factorization.

The rest of the chapter is organized as follows: Section 3.2 introduces and the background of one-sided factorizations in dense linear algebra, including programming model for heterogeneous platforms and looking ahead technique Strazdins (1998) to overlap imbalance. Section 3.3 presents the design of static scheduling in clMAGMA Cao et al. (2014) to implement high performance dense linear algebra for hybrid systems consisting of CPUs and OpenCL devices. Section 3.4 presents the design of dynamic scheduling using a task-based runtime (QUARK in our implementation) and several optimization schemes. Section 3.5 describes the extension to support distributed-memory environment. Section 3.6 shows the experimental results and Section 3.7 concludes this chapter.

The portion of this chapter has been published in the following publications of mine:

- Chongxiao Cao, Jack Dongarra, Peng Du, Mark Gates, Piotr Luszczek, Stanimire Tomov, "clMAGMA: High performance dense linear algebra with OpenCL", 1st International Workshop on OpenCL (IWOCL)

- Azzam Haidar, Chongxiao Cao, Asim YarKhan, Piotr Luszczek, Stanimire Tomov, Khairul Kabir, Jack Dongarra, "Unified Development for Mixed Multi-GPU and Multi-coprocessor Environments Using a Lightweight Runtime Environment", Parallel and Distributed Processing Symposium, 2014 IEEE 28th International

- Azzam Haidar, Asim YarKhan, Chongxiao Cao, Piotr Luszczek, Stanimire Tomov, Jack Dongarra, "Flexible linear algebra development and scheduling with cholesky factorization", 2015 IEEE 17th International Conference on High Performance Computing and Communications

3.2 One-sided Factorizations in Dense Linear Algebra

In this section, we present the linear algebra aspects of our generic solution for development of either Cholesky, Gauss (LU), and Householder (QR) factorizations based on block outer-product updates of the trailing matrix. Conceptually, one-sided factorization maps a matrix A into a product of matrices X and Y:

$$\mathcal{F}: \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \mapsto \begin{bmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{bmatrix} \times \begin{bmatrix} Y_{11} & Y_{12} \\ Y_{21} & Y_{22} \end{bmatrix}$$

Algorithmically, this corresponds to a sequence of in-place transformations of A, whose storage is overwritten with the entries of matrices X and Y (P_{ij} indicates

currently factorized panels):

$$\begin{bmatrix} A_{11}^{(0)} & A_{12}^{(0)} & A_{13}^{(0)} \\ A_{21}^{(0)} & A_{22}^{(0)} & A_{23}^{(0)} \\ A_{31}^{(0)} & A_{32}^{(0)} & A_{33}^{(0)} \end{bmatrix} \rightarrow \begin{bmatrix} P_{11} & A_{12}^{(0)} & A_{13}^{(0)} \\ P_{21} & A_{22}^{(0)} & A_{23}^{(0)} \\ P_{31} & A_{32}^{(0)} & A_{33}^{(0)} \end{bmatrix} \rightarrow$$

$$\rightarrow \begin{bmatrix} XY_{11} & Y_{12} & Y_{13} \\ X_{21} & A_{22}^{(1)} & A_{23}^{(1)} \\ X_{31} & A_{32}^{(1)} & A_{33}^{(1)} \end{bmatrix} \rightarrow \begin{bmatrix} XY_{11} & Y_{12} & Y_{13} \\ X_{21} & P_{22} & A_{23}^{(1)} \\ X_{31} & P_{32} & A_{33}^{(1)} \end{bmatrix} \rightarrow$$

$$\rightarrow \begin{bmatrix} XY_{11} & Y_{12} & Y_{13} \\ X_{21} & XY_{22} & Y_{23} \\ X_{31} & X_{32} & A_{33}^{(2)} \end{bmatrix} \rightarrow \begin{bmatrix} XY_{11} & Y_{12} & Y_{13} \\ X_{21} & X_{22} & Y_{23} \\ X_{31} & X_{32} & P_{33} \end{bmatrix} \rightarrow$$

$$\rightarrow \begin{bmatrix} XY_{11} & Y_{12} & Y_{13} \\ X_{21} & XY_{22} & Y_{23} \\ X_{31} & X_{32} & XY_{33} \end{bmatrix} \rightarrow \begin{bmatrix} XY \end{bmatrix},$$

where XY_{ij} is a compact representation of both X_{ij} and Y_{ij} in the space originally occupied by A_{ij}.

Table 3.1: BLAS and LAPACK routines for three one-sided factorizations.

	Cholesky	Householder	Gauss
PanelFactorize	xPOTF2 xTRSM	xGEQF2	xGETF2
TrailingMatrixUpdate	xSYRK2 xGEMM	xLARFB	xLASWP xTRSM xGEMM

Observe two distinct phases in each step of the transformation from $[A]$ to $[XY]$: *panel factorization* (P) and trailing matrix update: $A^{(i)} \rightarrow A^{(i+1)}$. Implementation of these two phases leads to a straightforward iterative scheme shown in Algorithm 1. Table 3.1 shows (Basic Linear Algebra Subprograms)) BLAS and LAPACK Anderson

et al. (1999) routines that should be substituted for the generic routines named in the algorithm.

Algorithm 1: Two-phase implementation of a one-sided factorization.

for $P_i \in \{P_1, P_2, \ldots, P_n\}$ **do**
    PanelFactorize(P_i)
    TrailingMatrixUpdate($A^{(i)}$)

Algorithm 2: Two-phase implementation with a split update.

for $P_i \in \{P_1, P_2, \ldots\}$ **do**
    PanelFactorize(P_i)
    TrailingMatrixUpdate$_{\text{Kepler}}$($A^{(i)}$)
    TrailingMatrixUpdate$_{\text{Phi}}$($A^{(i)}$)

The utilization of multiple accelerators for the computations complicates the simple loop from Algorithm 1: we have to split the update operation into multiple instances for each of the accelerators. This was done in Algorithm 2. Notice that PanelFactorize() is not split for execution on accelerators because it is considered a memory-bound workload which faces a number of inefficiencies on throughput-oriented devices. Considering the fact that the trailing matrix update requires the majority of floating point operations and accelerators provide high performance to carry out these operations, the trailing matrix update is split and assigned to corresponding accelerator. Computational activities for the same matrix block are moved between the main memory of CPU and device memory of accelerator. The distinction of different address spaces requires data transfer and synchronization between CPU and accelerator, and is included in the implementation shown in Algorithm 3.

This algorithm is also required to be modified further to achieve closer to hardware peak performance, as the current performance is bounded by imbalance. The imbalance comes from the fact that the bandwidth between the CPU and the devices

Algorithm 3: Two-phase implementation with a split update and explicit communication.

for $P_i \in \{P_1, P_2, \ldots\}$ **do**
  PanelFactorize(P_i)
  PanelSend$_{\text{Kepler}}$(P_i)
  TrailingMatrixUpdate$_{\text{Kepler}}$($A^{(i)}$)
  PanelSend$_{\text{Phi}}$(P_i)
  TrailingMatrixUpdate$_{\text{Phi}}$($A^{(i)}$)

is orders of magnitude too slow to sustain computational rates of accelerators*. The classical technique to alleviate this imbalance is to use *lookahead* Strazdins (1998).

Algorithm 4: Lookahead of depth 1 for the two-phase factorization.

PanelFactorize(P_1)
PanelSend(P_1)
TrailingMatrixUpdate$_{\{\text{Kepler,Phi}\}}$(P_2)
PanelStartReceiving(P_2)
TrailingMatrixUpdate$_{\{\text{Kepler,Phi}\}}$($R^{(1)}$)
for $P_i \in \{P_2, P_3, \ldots\}$ **do**
  PanelReceive(P_i)
  PanelFactorize(P_i)
  PanelSend(P_i)
  TrailingMatrixUpdate$_{\{\text{Kepler,Phi}\}}$(P_{i+1})
  PanelStartReceiving($P_{(i+1)}$)
  TrailingMatrixUpdate$_{\{\text{Kepler,Phi}\}}$($R^{(i)}$)
PanelReceive(P_n)
PanelFactor(P_n)

An example of setting lookahead depth as 1 is shown in Algorithm 4. In this example, trailing matrix update is divided into three steps: (1) update of next panel on accelerator side; (2) transfer next panel from accelerator to CPU; (3) update of remaining trailing matrix R on accelerator side. By using this division, the communication of panel is overlapped with the update operation. Due to different communication bandwidth and accelerator peak performance, a

*The bandwidth for current generation PCI Express is at most 16 GB/s and the devices achieve over 1000 Gflop/s performance.

different lookahead depth might be required for optimal performance on different heterogeneous platforms. In fact, the optimal value of lookahead depth is decided by a thorough research of targeted factorization, in order to fully overlap massive trailing matrix update on accelerator side with sequential panel factorization and panel communication on CPU side.

3.3 Static Scheduling in clMAGMA

In this chapter, we introduce the design of the clMAGMA library Cao et al. (2014), an open source, high performance OpenCL Khronos OpenCL Working Group (2009) library that incorporates various methods of optimization, and in general provides the dense linear algebra functionality of the popular LAPACK library on heterogeneous architectures. We consider a redesign of the LAPACK algorithms to facilitate their OpenCL implementation, and to add efficient support for heterogeneous systems of multi-core processors with GPU accelerators and coprocessors.

The hybridization methodology used in MAGMA Agullo et al. (2009) library is now used in clMAGMA. It is an extension of the task-based approach for parallelism and developing dense linear algebra on homogeneous multi-core systems. The hybridization methodology is described as below:

- The factorization is split into BLAS-based tasks of various granularities, with their data dependencies.
- Small, latency-bound tasks with significant control-flow are executed on the CPUs.
- Large, compute-bound tasks are executed on GPUs.

The difference between multi-core algorithms and hybridization is the task splitting, which are of various granularities to make different tasks suitable for particular hardware. An example of static scheduling of Cholesky factorization, demonstrating

how GPU part computation and CPU part computation is overlapped, is shown in Figure 3.1. In line 4, next available panel (next diagonal block in Cholesky factorization) is transferred to CPU from GPU. This data transfer is asynchronous, meaning that GPU starts to update remaining part of trailing matrix while data is being transferred. Line 7 is a synchronization barrier to guarantee that communication has completed. Also, line 8 is panel factorization on CPU side, which is overlapped with GPU computation in line 6. Line 6 is an asynchronous request from CPU side to start the ZGEMM operation on GPU side. After panel factorization is finished on CPU side, the resulting panel is sent to GPU by using asynchronous communication. After the communication is synchronized in line 13, GPU starts to perform ZTRSM operation.

```
1   for ( j=0; j<n; j += nb)  {
2       jb = min(nb, n – j);
3       magma_zherk( MagmaUpper, MagmaConjTrans,
                     jb, j, m_one, dA(0, j), ldda, one, dA(j, j), ldda, queue );
4       magma_zgetmatrix_async( jb, jb, dA(j,j), ldda, work, 0, jb, queue, &event );
5       if ( j+jb < n )
6           magma_zgemm( MagmaConjTrans, MagmaNoTrans, jb, n-j-jb, j, mz_one,
                         dA(0, j ), ldda, dA(0, j+jb), ldda, z_one,  dA(j, j+jb), ldda, queue );
7       magma_event_sync( event );
8       lapackf77_zpotrf( MagmaUpperStr, &jb, work, &jb, info );
9       if ( *info != 0 )
10          *info += j;
11      magma_zsetmatrix_async( jb, jb, work, 0, jb, dA(j,j), ldda, queue, &event );
12      if ( j+jb < n ) {
13          magma_event_sync( event );
14          magma_ztrsm( MagmaLeft, MagmaUpper, MagmaConjTrans, MagmaNonUnit,
                         jb, n-j-jb, z_one, dA(j, j), ldda, dA(j, j+jb), ldda, queue );
        }
    }
```

Figure 3.1: Cholesky factorization in clMAGMA.

In OpenCL, performing work on a device, such as executing kernels or moving data to and from the device's local memory, is done using a corresponding command-queue Khronos OpenCL Working Group (2009). A command-queue is an interface for a specific device to launch its associated work. The host (usually, a CPU) places device kernels into a command-queue and then submits it to the device. For example, in Figure 3.1, line 6 puts a ZGEMM kernel in a command-queue `queue`. The host still must submit the ZGEMM to the device for execution, due to the standard implementation in OpenCL 1.1, the kernel may start on device side immediately.

As a result, it is possible that when CPU starts the panel factorization at line 8, the ZGEMM on device side hasn't started. Thus, although our high-level algorithm is designed to overlap CPU and GPU work, overlap may not happen in practice. From OpenCL standard 1.1, in order to force the command-queue to immediately submit the command queued to the appropriate device, host must call `clFlush(queue)` Khronos OpenCL Working Group (2009) after launching the command-queue. Therefore, all BLAS wrappers in clMAGMA are implemented in two steps: the corresponding OpenCL BLAS is firstly queued and a `clFlush` is called to force the kernel to start immediately on device side.

While CPU computation and GPU computation is overlapped, communication and computation on GPU side are executed sequentially as they are submitted to the same command-queue and OpenCL only support in order execution inside a queue. One way to overlap CPU-GPU communication and GPU computation is using multiple command-queues. Here, two queues are created, one queue is used for executing communication and the other is used for executing kernel computation. Figure 3.2 shows a part of the trace of double precision LU factorization based on the optimization of using two queues. The first row is the execution trace of CPU, where the black panel represents panel factorization; the second row is the execution trace of queue 1 on GPU focusing on trailing matrix update, where the red panel represents DGEMM operations and green panel represents DTRSM operations; and the third row is the execution trace of queue 2 on GPU focusing on communication, where yellow panel represents the data movement from GPU to CPU and the grey panel is data movement from CPU to GPU. It is important to note that CPU-GPU communication is overlapped with GPU computation by splitting corresponding work into separate command-queues.

The importance of overlapping CPU and GPU work is quantified in Figure 3.3 for the example of LU factorization in double precision (the DGETRF routine). The heterogeneous platform is composed of one six-core AMD Phenom CPU and one AMD Radeon 7970 GPU. The blue curve is the performance of DGETRF(LU factorization)

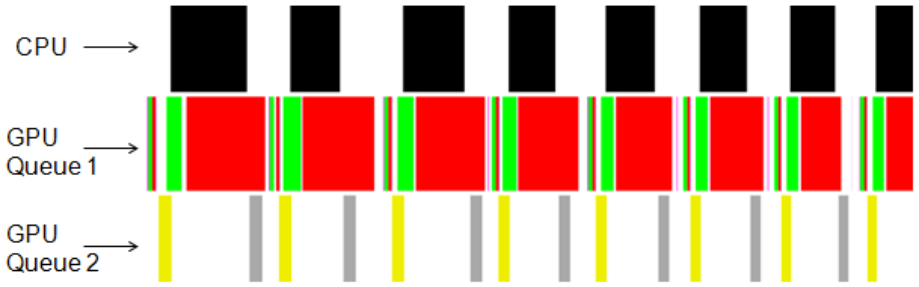

Figure 3.2: Partial CPU-GPU execution trace of a hybrid LU factorization in clMAGMA based on the two command-queues' optimization.

without overlapping CPU computation and GPU computation. It achieves up to 195 Gflop/s. After using `clFlush` command to overlap CPU computation and GPU computation, the red curve shows the corresponding performance of DGETRF. It achieves up to 280 Gflop/s, i.e., gaining about 1.4× speedup. Other further optimizations are also shown in Figure 3.3. The overlap of CPU-GPU communications with GPU computation is achieved by using two command queues. Device memory is also pinned on GPU side in order to reach high bandwidth between CPU-GPU communication. By combing all these optimizations together, the best performance of LU factorization is shown with the purple curve. It achieves up to 326 Gflop/s, which is almost a 60% speedup compared to the original version without any optimization.

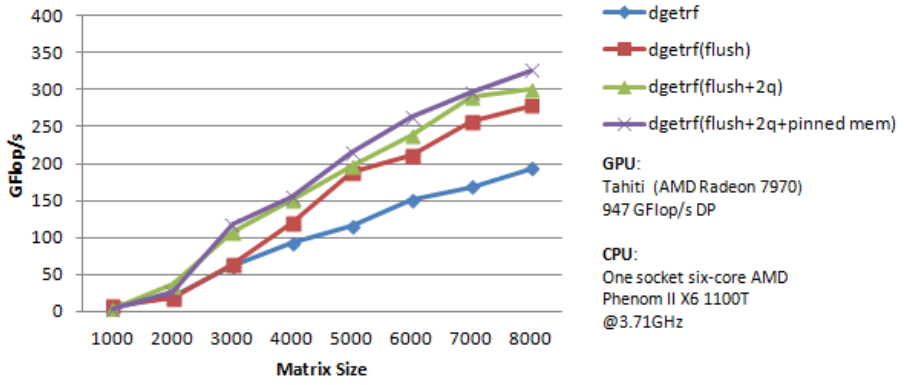

Figure 3.3: Advanced performance optimizations of LU factorization in clMAGMA.

3.4 Dynamic Scheduling using a Task-based Runtime

Since the management of performance optimizations for static scheduling algorithms is tedious, it is desirable to utilize a dynamic task-based runtime to schedule algorithms efficiently and to maintain performance portability. However, due to the homogeneity inherent in most of the existing runtime systems, it is difficult to manage different types of computing devices on a mixed platform. Also, common scheduling techniques, such as task stealing, are not applicable here due to the disjoint address spaces from different devices and the associated large overhead of moving tasks. These challenges are dealt with comprehensively in the remainder of this section.

3.4.1 Task Superscalar Scheduling

Task-superscalar execution is an abstraction of instruction-level out-of-order pipeline that operates at the task level. Like instruction-level parallelism pipelines, which uncover parallelism in a sequential instruction stream, task-superscalar execution takes a serial sequence of tasks as input and schedules them for execution in parallel, by discovering data dependencies between tasks during runtime. The dependencies between the tasks are inferred through the resolution of three data hazards: *Read after Write* (RaW), *Write after Read* (WaR) and *Write after Write* (WaW). The dependencies between tasks are annotated in original serial code by application developers using data definition interfaces provided by runtime system, indicating the data property to be Read and/or Written. The RaW hazard, often referred to as the *true dependency*, is the most common one when exploring parallelism. It shows the race condition that one task is writing some data and another tasks is reading that data. In order to avoid race condition, the reading task has to wait until the writing task finishes. Also, in another scenario, if multiple tasks request to read the same data, there is no need for them to execute in sequential as race condition

doesn't exist. Multiple read requests to be same data can be executed in parallel. Task-superscalar execution is a kind of asynchronous and data-driven execution, which can be represented by the form of DAG, where the tasks are the vertices in the graph and the edges correspond to data movement between the tasks. Task-superscalar execution is a powerful tool for exploring parallelism and maintaining performance portability. The effort of using task-superscalar execution is to annotate data dependencies in serial code correctly. The runtime system takes the serial code as input, explores concurrent execution of multiple available tasks by avoid data hazards and guarantees correct result.

By implementing task superscalar execution, the runtime can achieve parallelism by executing tasks with non-conflicting data dependencies (e.g., simultaneous reads of data by multiple tasks). Superscalar execution also enables lookahead technique in the serial code, as future tasks in sequential execution can be executed as soon as their data dependencies are fulfilled. In this section, we implement our unified design by using QUARK runtime, as QUARK provides lower level control support on task location and binding that would be harder to utilize when using the other superscalar runtime systems. However, our conceptual design can be incorporated into any existing task-based runtime system, so here QUARK is just treated as a simple representation of a lightweight, task-superscalar runtime environment.

3.4.2 Efficient and Scalable Programming Model Across Multiple Devices

GPU accelerators and coprocessors have much higher peak performance compared with CPUs. For simplicity in the design here, we refer to both GPUs and coprocessors as accelerators. Also, different types of accelerators provide different computing capabilities, which makes it challenging to develop an algorithm that can achieve high performance and keep good scalability. From the hardware point of view, an accelerator communicates with the CPU using I/O commands and (Direct memory

access) DMA memory transfers, whereas from the software standpoint, the accelerator is a computing platform interacted with CPU through a programming interface. The key features considered in our design are the device's computing capability (CPUs, GPUs, Xeon Phi), the memory access cost, and the communication cost. From the CPU's side of serving as a host, the access cost to the device memory for accelerators is much more expensive comparing with benefit from accelerators' peak performance. Hierarchical caches have been designed to improve the long memory access latency and bandwidth issues on CPU side. This does not solve the slow memory access problem completely but is often effective. On the other hand, accelerators use multithreading operations that access large data sets that would hide the memory access latency. The reason of hiding memory access latency is to take advantage of accelerator's massive lightweight threads. When one of the accelerator's threads issues an instruction to access device memory, that thread stalls until memory access completes. Meanwhile, the accelerator's scheduler switches to another hardware thread, and continues to execute instructions on that thread. By keeping switching to active hardware thread, an accelerator is able to exploit program parallelism to keep functional units busy while waiting the memory to fulfill past requests. By comparison with CPUs, the device memory delivers higher absolute bandwidth (around 180 GB/s for Xeon Phi and 160 GB/s for Kepler K20c). To solve memory access issues, a strategy is developed in this section to prioritize the data-intensive operations to be executed by the accelerator, and to keep the memory-bound ones for the CPUs since the hierarchical caches with out-of-order superscalar scheduling are more appropriate to handle it. Moreover, in order to utilize accelerators more efficiently, a hardware guided data distribution strategy is designed to distribute optimal size of workloads to different accelerators to keep them busy and achieve optimal performance.

From a programming model point of view, it is not possible to hide the distinction when applying two different levels of parallelism. In order to solve the distinction, linear algebra algorithms are redesigned and divided into a host part and an accelerator part. Every computational routine running on accelerator side

Algorithm 5: Cholesky implementation for multiple devices.

```
Task_Flags panel_flags = Task_Flags_Initializer
Task_Flag_Set(&panel_flags, PRIORITY, 10000)
memory-bound → locked to CPU
Task_Flag_Set(&panel_flags, BLAS2, 0)
for k ∈ {0, nb, 2 × nb, ..., n} do
    Factorization of the panel dA(k:n,k)
    Cholesky on the tile dA(k,k)
    TRSM on the remaining of the panel dA(k+nb:n,k)
    DO THE UPDATE: SYRK task has been split into a set of parallel
    compute intensive GEMM to increase parallelism and enhance the
    performance. Note that the first GEMM consists of the update of the
    next panel, thus the scheduler check the dependency and once finished
    it can start the panel factorisation of the next loop on the CPU.
    if panel_m > panel_n then
        SYRK with trailing matrix
        for j ∈ {k + nb, k + 2nb, ..., n} do
            GEMM dA(j:n,k) × dA(j,k)^T = dA(j:n,j)
```

is extracted into specific kernel function targeted for different hardware. It is also necessary to optimize kernel functions on the accelerator, including instruction level optimization and algorithmic level optimization. Several optimization schemes have been applied to optimize kernels for specific device, e.g., loop unrolling, trading slower memory-bound operations with compute-intensive ones with introducing extra marginal computational cost, and reordering task sequence to utilize device memory more efficiently. The host part code manages device memory allocation, CPU-device data transfer and launching device kernel. The runtime engine from QUARK is also redesigned to provide easier programming interfaces for simplifying scheduling. This simplified support to able to alleviate users' effort of maintaining a single version of serial code and to utilize runtime to provide performance portable execution for different devices. The intention in this work is to simplify most of the hardware details, while giving application developers finer levels of control. Algorithm 5 shows the pseudo code for the Cholesky factorization from an algorithm designer's point of

view. It consists of a sequential code that is simple to comprehend and independent of the architecture. Each of these calls represents a task that is inserted into the scheduler, which stores it to be executed when all of its dependencies are satisfied. Each task by itself consists of a call to a kernel function that could either be a CPU or an accelerator function. After wrapping kernel functions for different devices into a generic interface, the differences between hardware is hidden and the runtime scheduler is able to handle data movement automatically. Also, low-level optimization schemes are designed for accelerators to accommodate hardware- and library-specific tuning and requirements. Furthermore, we implemented a set of directives that are evaluated at runtime in order to fully map the algorithm to the hardware and run close to the peak performance of the system. Using these strategies, application developers are able to design simple serial code in a lightweight environment. The efforts related to performance improvement and portability is transferred to the runtime system.

3.4.3 Optimizations for Performance Improvement

Since there is no simple way to express the difference in the workload-capabilities between the CPUs and accelerators. Clearly, we cannot balance the load, if we treat them as peers and assign them equivalent amount of work. Such a naive strategy would cause the accelerator to be substantially idle. As described above, in our programming model we propose to assign the latency-bound operations to the CPUs and the compute-intensive ones to accelerators. In order to support multi-way heterogeneous hardware, QUARK is extended with a mechanism for distributing tasks based on the computing capabilities of each device. For each device i and each kernel type k, QUARK maintains an α_{ik} parameter which corresponds to the effective performance rate that can be achieved on that device. In the context of linear algebra algorithms, this means that we need an estimation of performance for Level 1, 2, and 3 BLAS operations. This can be done either by the developer during the implementation where the user gives a directive to QUARK that this kernel is

either bandwidth-bound or compute-bound function (as shown in Algorithm 5 with a call to Task_Flag_Set with BLAS2 argument) or estimated according to the volume of data and the elapsed time of a kernel by the QUARK engine at runtime.

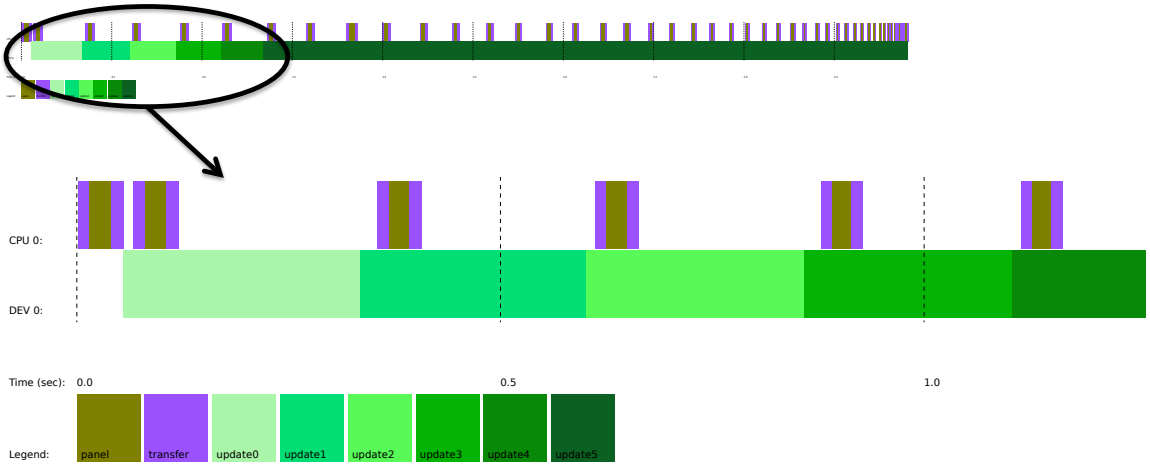

Figure 3.4: A trace of the Cholesky factorization on one 16-core Sandy Bridge CPU and one K20c GPU.

Figure 3.4 shows the execution trace of the Cholesky factorization on a system consisting of one multi-core CPU and one NVIDIA K20c GPU. It is observed that the memory-bound operations (e.g., the panel factorization for the Cholesky algorithm) have been assigned to the CPU while the compute-bound ones (e.g., the update performed by DSYRK) have been assigned to the accelerator. The initial data is assumed to be on the device, and when the CPU is executing a task, data is required to transfer from device to CPU. Also when CPU completes the panel factorization, the panel is transferred back to device to update the trailing matrix. The data transfer is represented by the *purple* color in the trace. The CPU panel computation is represented by the *gold* color. The trailing matrix update is represented in *green* color. For clarity, we varied the intensity of the green color representing the update from light to dark for the first 5 steps of the factorization. From this trace, we can see that the GPU is kept busy all the way until the end of execution. The use of the lookahead technique described in Algorithm 4, does not require any extra effort since it is automatically handled by the QUARK runtime engine through the resolution of

data dependencies. As defined by the data dependencies, the next panel (panel of step $k+1$) is updated on GPU side as soon as possible, and transferred to CPU side to be factorized. Meanwhile on GPU side, the remaining part of the trailing matrix of step k continues to be updated. Also, the QUARK engine manages the data transfer to and from the CPU automatically. The advantage of such strategy is not only to hide the data transfer cost between the CPU and GPU (since it is overlapped with the GPU computation), but also to keep the GPU's computing queues (i.e., CUDA streams for NVIDIA GPU) busy by providing enough tasks to execute. As shown in Figure 3.4, we can see that the panel of step 1 is quickly updated by the GPU and sent to the CPU to be factorized and sent back to the GPU, which is a perquisite to perform the trailing matrix update of step 1, before the GPU has already finished the update of trailing matrix of step 0, and so on.

Improved Task Priorities: In order to highlight the importance of task priority, we recall, that the panel factorization tasks of most of the one-sided factorizations (e.g., the Cholesky, QR and LU algorithms) are on the critical path of execution. In other words, only if a panel computation is done in its entirety, its corresponding update computation (compute-bound operation) can proceed. In the traces in Figure 3.4, it can be observed that the panel factorization on the CPU occurs at regular intervals (e.g., the lookahead depth is one). By changing the priority of the panel factorization tasks (using QUARK's task priority flags as mentioned in Algorithm 5), the execution of panel factorization can be scheduled earlier. Moving panel factorization earlier implies that a higher lookahead depth is used, unfolding more parallelism and generating more trailing matrix update tasks for the accelerator. Using priorities to improve lookahead results in approximately 5% improvement in the overall performance of the factorization. Figure 3.5 shows the update tasks being executed earlier in the trace.

Data layout: 1-D block cyclic data layout is applied to support multiple accelerators. Original matrix is initialized across all accelerators in a block-column cyclic fashion, with an approximately equal number of columns distributed to every

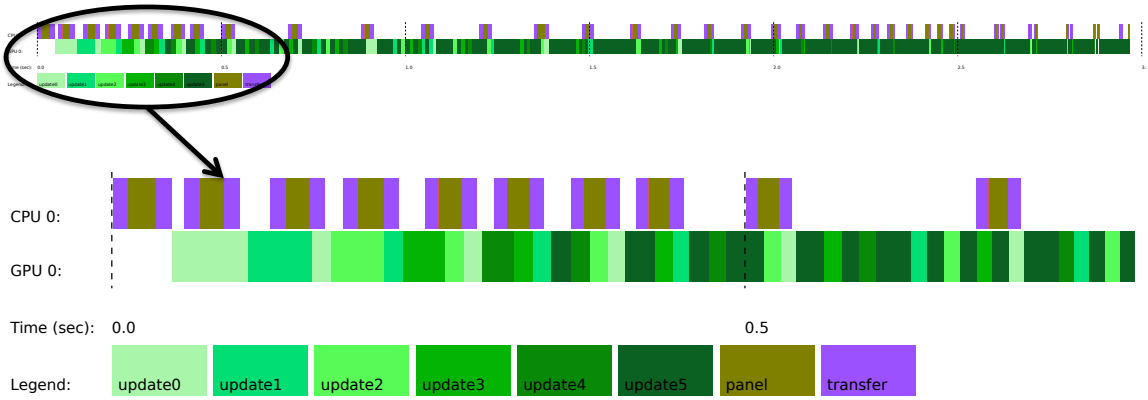

Figure 3.5: Trace of the Cholesky factorization on one 16-core Sandy Bridge CPU and one K20c GPU, using priorities to improve lookahead.

accelerator. It is important to note that data is allocated as one contiguous memory block on every accelerator, combining distinct column blocks together. This contiguous data layout allows large update operations to take place over a number of columns via a single Level 3 BLAS operation. It is much more efficient than storing distinct column blocks in separate memory segments and having multiple BLAS operations.

Hardware-Guided Data Distribution: It is demonstrated in experiments that the standard 1-D block cyclic data layout is hindering performance in heterogeneous multi-accelerator environments. Figure 3.6 shows the trace of the Cholesky factorization for a matrix of size $30,000$ on a hybrid system consisting of one K20c GPU, one Intel Xeon Phi (MIC) and one K20-beta GPU. The trace shows that the execution flow is bound by the performance of the slowest machine (the K20-beta GPU, second row) and thus we expect lower performance on this machine. We propose to re-adjust the data layout distribution to be hardware-guided by the use of the capability-weights. Using the QUARK runtime, the data is either distributed or redistributed in an automatic fashion so that each device gets the appropriate volume of data to match its capabilities. So, for example, for this system, using capability weights of K20c:MIC:K20-beta of 10:8:5 would result in a cyclic distribution of 10 columns of data being assigned to the K20c, for each 8 columns assigned to the MIC, and each 5

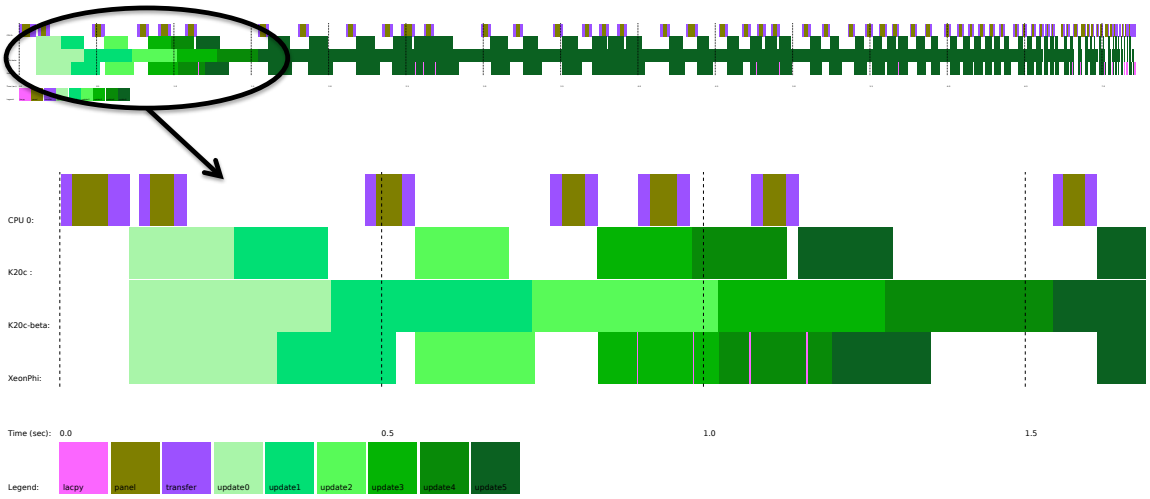

Figure 3.6: Cholesky factorization trace on one 16-core Sandy Bridge CPU and multiple accelerators (one K20c GPU, one Xeon Phi, and one K20-beta GPU), without enabling heterogeneous hardware-guided data distribution.

columns assigned to the K20beta. The superscalar execution environment can do this capability-weighted data assignment at runtime. Figure 3.7 shows the trace of the Cholesky factorization for the same example as above (a matrix of size $30K$ a node of the system D) when using the hardware-guided data distribution (HGDD) strategy. It is clear that the execution trace is more compact meaning that all the heterogeneous hardware are fully loaded by work and thus one can expect an increase in the total performance. For that we represent in Figure 3.8 the performance comparison of the Cholesky factorization when using the HGDD strategy. The curves in blue shows the performance obtained for a one K20c and one XeonPhi experiments. The dashed line correspond to the standard 1-D block-column cyclic distribution while the continuous line illustrate the HGDD strategy. We observe that we can reach an improvement of about 200-300 Gflop/s when using the HGDD technique. Moreover, when we add one more heterogeneous device (the K20beta GPU), here it comes to the complicated hardware situation, we can notice that the standard distribution do not exhibit any speedup. The dashed red curve that represents the performance of the Cholesky factorization using the standard data distribution on the system consisting of three different devices behaves closely and less efficiently than the one obtained with the

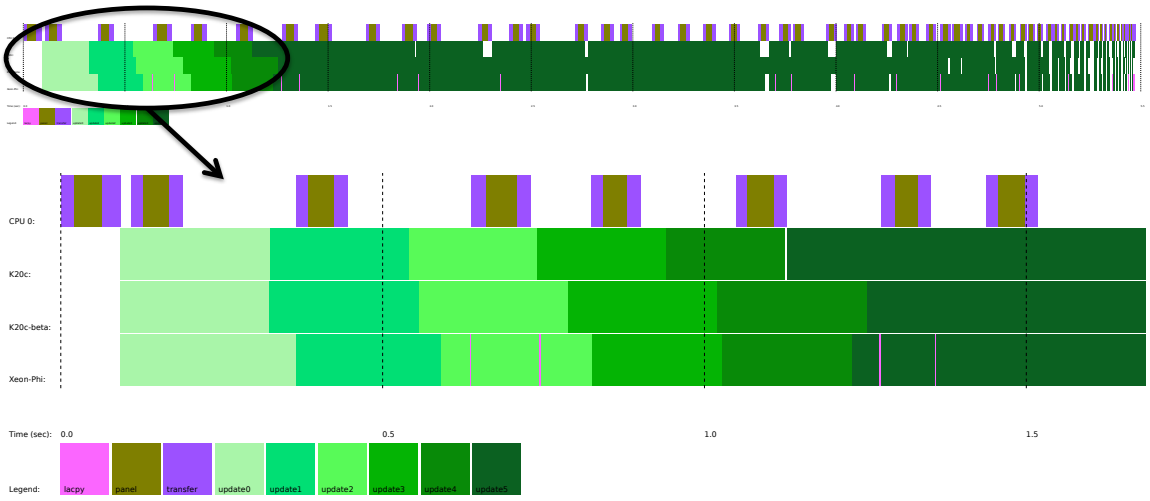

Figure 3.7: Cholesky factorization trace on one 16-core Sandy Bridge CPU and multiple accelerators (one K20c GPU, one Xeon Phi, and one K20-beta GPU), using the heterogeneous hardware-guided data distribution techniques (HGDD) to achieve higher hardware usage.

same standard distribution on two devices (dashed blue curve). This was expected, since adding one more device with lower capability may decrease the performance as it may slow the fast device down. The blue and red curves in Figure 3.8 illustrate that the HGDD technique exhibits a very good scalability for both algorithms. The graph shows that the performance of the algorithm is not affected by the heterogeneity of the machine, our proposed implementation is appropriate to maintain a high usage of all the available hardware.

3.5 Supporting Distributed-memory Heterogeneous Platforms

The compute nodes of large-scale machines contain a mixed-core approach to hardware, combining multi-core CPUs and GPUs or coprocessors, each of which appropriates for various work granularities. In this section, we describe the extended work of utilizing QUARK to develop Cholesky factorization in a distributed-memory environment. The extension includes a number of new contributions varying from the

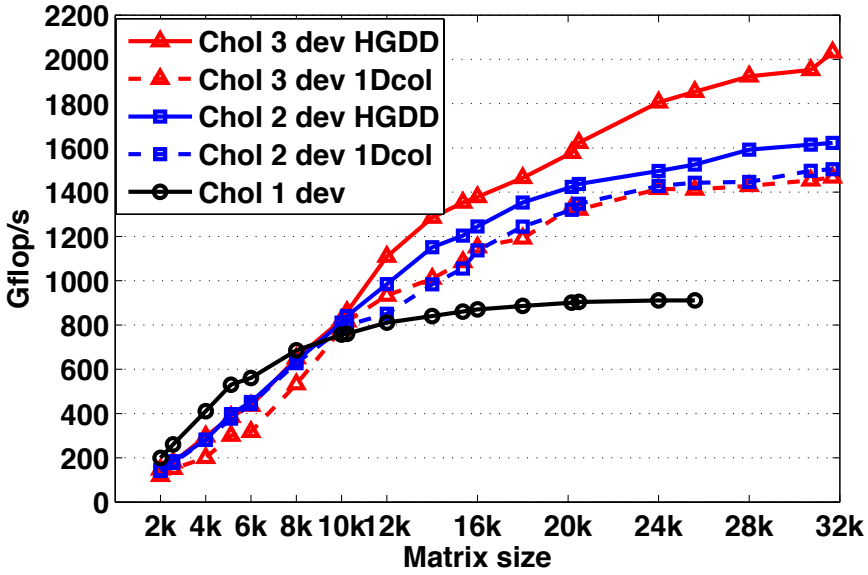

Figure 3.8: Performance comparison of the Cholesky factorization when using the hardware-guided data distribution techniques versus a 1-D block-column cyclic, on heterogeneous accelerators consisting of one K20c GPU (1dev), one Xeon Phi (2dev), and one K20-beta GPU (3dev).

way data is stored and moved to the way algorithms are split into tasks and scheduled for execution.

We extend the classical LAPACK algorithms into heterogeneous algorithms for distributed systems and give a description for the case of the Cholesky factorization. We designed a two-level block-cyclic distribution method to support the heterogeneous algorithms, as well as an adaptive task scheduling method to determine the splitting of work over the devices.

Algorithm 6 shows the starting point of our algorithmic considerations. The decomposition of the input matrix across both rows and columns is matched by the decomposition in double-nested loop to allow for static mapping to the hardware and flexible scheduling at runtime. This two-fold decomposition in the data domain and the algorithmic domain serves as facility of introducing lookahead Strazdins (1998) to increase efficiency through temporal and spacial overlap of communication, computation, and the mix thereof. Through this partitioning, we can take this concept beyond its inception and apply it in both domains (across matrix dimensions and

Algorithm 6: Right-looking blocked and tiled Cholesky factorization with a fixed blocking factor n_b.

Input : A—Symmetric positive definite
Input : n_b—Blocking factor
Output: L—Lower triangular
for $A_{i,i} \in \{A_{1,1}, A_{2,2}, A_{3,3}, \dots A_{*,*}\}$ **do**
    $A_{i,i} \in \mathbf{R}^{n_b \times n_b}$
    $L_{i,i} \leftarrow$ `UnblockedCholesky`$(A_{i,i})$
    for $A_{j,i} \in \{A_{i+1,i}, A_{i+2,i}, A_{i+3,i}, \dots A_{*,i}\}$ **do**
        $A_{j,i} \in \mathbf{R}^{n_b \times n_b}$
        $A_{j,i} \leftarrow L_{i,i}^{-1} \times A_{j,i}$
    for $A_{j,k}$ *where* $j, k > i$ **do**
        $A_{j,k} \in \mathbf{R}^{n_b \times n_b}$
        $A_{j,k} \leftarrow A_{j,k} - L_{j,i} \times L_{i,k}$

loop nests) simultaneously. The proper tracking of these, admittedly more complex, dependencies are offloaded to the runtime and thus only a minor burden is left to the algorithm developer – the custodial task of invoking the runtime and informing it about the dataflow structure.

Data Distribution: We use a multi-level hierarchy of data blocking rather than fixed blocking across nodes, cores, and devices. At the coarsest (global distributed) level we employ a 2D block cyclic distribution Choi et al. (1996), the main reasons being scalability and load balance, both of which are of concern at the level of parallelism and hardware size that we target. Inside a single node, the amount of concurrency can still be staggering, especially when we count GPU threads, floating-point ALUs, and hyper-threading contexts. More appropriate, however, is modeling the single node hardware unit as a moderately parallel entity with at most tens of computational units, be it GPU compute units or CPU cores. For such a hardware model, a 1D cyclic distribution is adequate to balance the load while still scaling efficiently. This 1D distribution has some additional benefits for matching the data layout to the panel-update style linear algebra algorithm.

MPI Communication: Our goal is to provide a level of abstraction that delivers portable performance on many kinds of heterogeneous systems. To that end, we propose a new methodology that avoids the all too common issue of the classical distributed programming model – the "*bulk-synchronous*" Valiant (1990) lock-step execution, which was used by ScaLapack Blackford et al. (1996). This model does not cope productively with the heterogeneity of the current processing units (large core-count many-core and heterogeneous systems), and neither can they overlap the communication nor account for the variability in runtime performance behavior. In a distributed-memory environment, explicit data movement tends to be the source of many parallel, and thus hard to develop. To alleviate the issue and to keep the overall ease of use and consistent notion of task-based runtime, we propose encapsulating MPI communication calls inside tasks. This turns the message passing primitives into data sources and sinks, which in turn makes it possible to ease the burden of manual tracking of asynchronous calls throughout the code and ensuring proper progress of the communication protocol. Additionally, the runtime provides basic flow control to limit the number of outstanding asynchronous events, which dovetails the issue of how many such non-blocking calls are acceptable for a given MPI implementation – a purely software engineering limitation that could potentially be hard to accommodate if done manually across a number of open source and vendor MPI libraries. When utilizing QUARK in a distributed-memory environment, the situation changes only slightly when one of the cores is devoted to only handle MPI-related activities. On occasion, the communication core might go underutilized due to high computation demand and low communication load but in the overall hardware mix with tens of cores per node, this does not pose an appreciative loss in total achieved performance. On the contrary, at the periods of heavy communication, the thread is either busy queuing new asynchronous sends and/or receives or providing progress opportunity to already executing MPI calls. With this scheme we achieve on-demand communication between nodes from the single message passing thread and shared memory concurrency within the node.

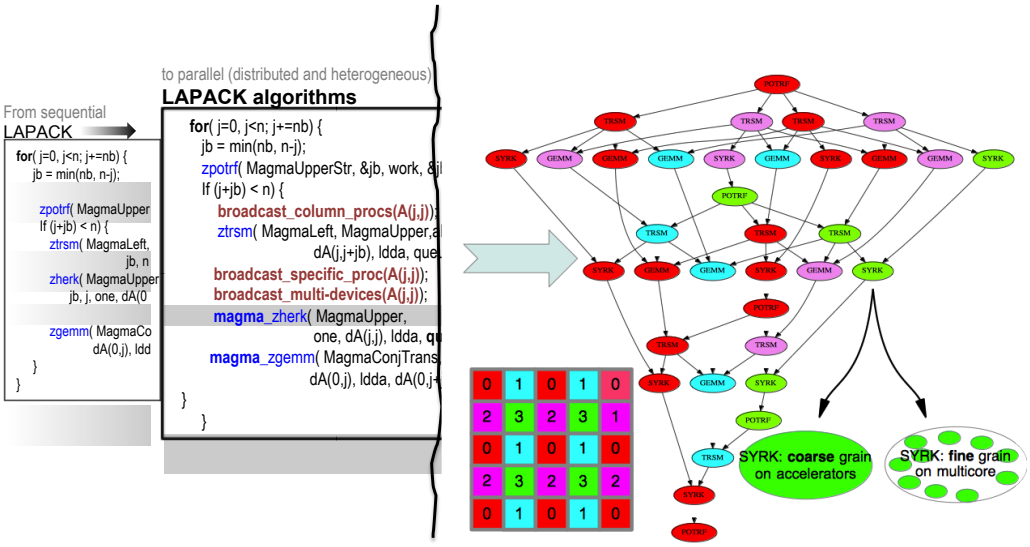

Figure 3.9: Performance scalability of Cholesky factorization on a multicore CPU and multiple GPUs (up to 6 K20c GPUs).

By combing all the extensions together, Figure 3.9 gives an overview of the design of Cholesky factorization in a distributed memory environment. Its algorithm looks like LAPACK (left), while a task superscalar runtime executes the underlying distributed algorithm (right). The execution can be viewed as a DAG with the tasks executed on nodes where the 2D block-cyclic data is located. In the example of Figure 3.9, a matrix consisting of 5×5 block-cyclic distributed tiles is executed on four distributed nodes, marked by different colors. MPI communication tasks, not shown for simplicity, are between nodes of different colors. One SYRK task is shown having adaptive grain sizes, depending on the hardware that the task is assigned to (CPU, GPU, Phi).

3.6 Experimental Results

In this chapter, several algorithmic and programming techniques have been proposed to address the challenge of obtaining good performance across multiple accelerators.

The efficient strategies used to schedule and exploit parallelism across multi-way heterogeneous platforms will be highlighted in this section through extensive experiments performed on the four hybrid systems.

3.6.1 Performance in shared-memory environment

Our experiments are performed on a number of shared-memory systems with different accelerators.

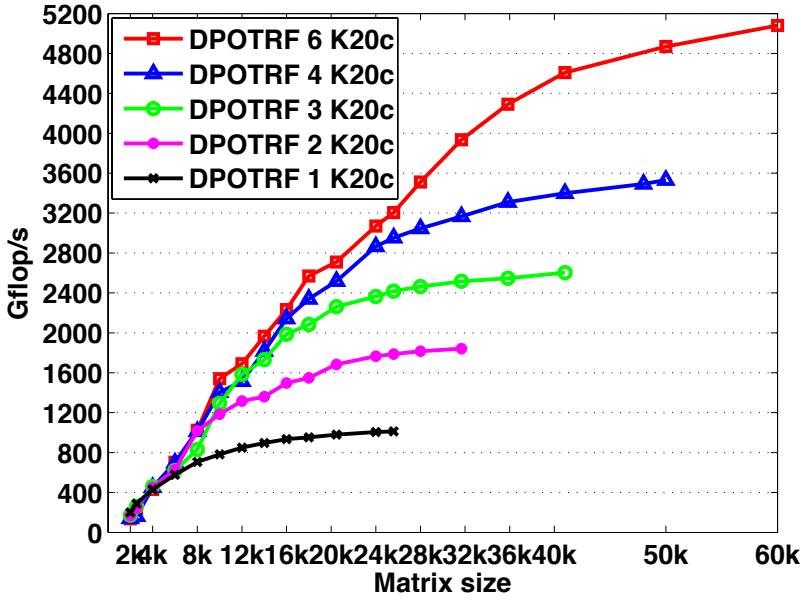

Figure 3.10: Performance scalability of Cholesky factorization on a multi-core CPU and multiple GPUs (up to 6 K20c GPUs).

Figure 3.10 shows the performance scalability of the Cholesky factorization in double precision on a system equipped with dual-socket, 8-core Intel Xeon E5-2670 (Sandy Bridge) processors and six NVIDIA K20c GPUs. The curves show performance in terms of Gflop/s. We note that this also reflects the elapsed time, e.g., a performance that is two times higher, corresponds to an elapsed time that is two times shorter. On this system, our heterogeneous multi-device implementation of Cholesky factorization shows very good scalability. For a $60,000$ matrix, the Cholesky factorization achieves 5.1 Tflop/s when using all the 6 K20c GPUs. Figure 3.11 shows

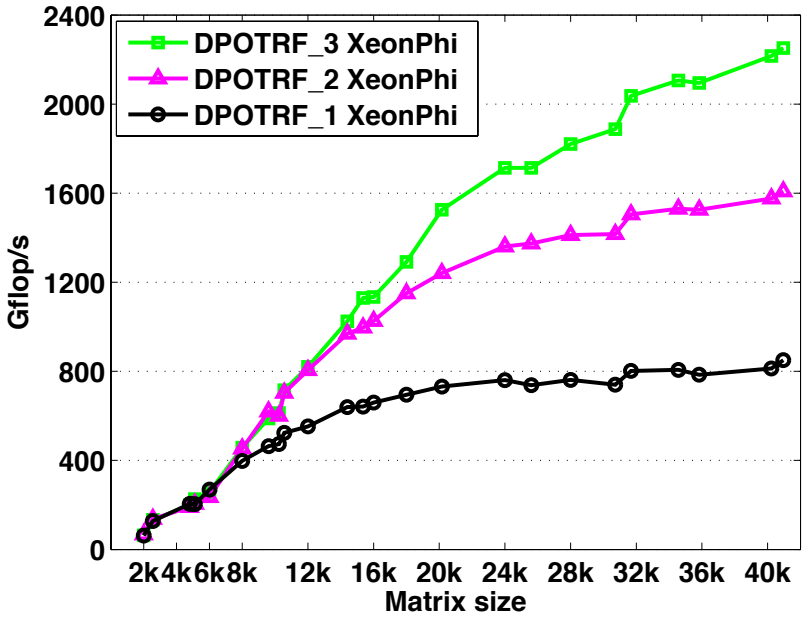

Figure 3.11: Performance scalability of Cholesky factorization on a multi-core CPU and multiple Xeon Phi coprocessors (up to 3 Xeon Phi coprocessors).

similar performance trends of Cholesky factorization when using a system equipped with dual-socket, 8-core Intel Xeon E5-2670 (Sandy Bridge) processors and three Intel Xeon Phi KNC coprocessors. For a matrix of size $40,000$, the Cholesky factorization reaches up to 2.3 Tflop/s when using the 3 Intel Xeon Phi KNC coprocessors.

3.6.2 Performance in distributed-memory environment

We also evaluate our unified programming model on distributed-memory systems. We conduct our experiments on two distributed systems, featuring GPUs and MICs, respectively: System A has 120 nodes connected with Mellanox InfiniBand QDR. Each node has two Intel Xeon hexa-core X5660 CPUs running at 2.8 GHz, and three NVIDIA Fermi M2090 GPUs. System B has 48 nodes connected by an FDR InfiniBand interconnect providing 56 Gb/s of bi-directional bandwidth. Each node features two 8-core Intel Xeon E5-2670 CPUs (Sandy Bridge), running at 2.6 GHz, and two Intel Xeon Phi 5110P coprocessors with 8 GiB of GDDR memory each. We use weak scalability to evaluate the capability of our algorithm to solve potentially

larger problems when more computing resources are available. We set the problem size for a single node to $30\,000 \times 30\,000$ matrix.

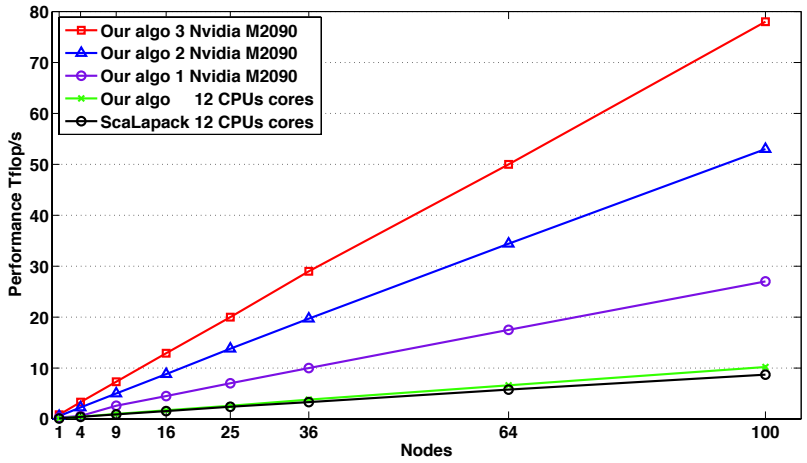

Figure 3.12: Weak scalability (horizontal reading) strong scalability (vertical reading) of the distributed multi-device Cholesky factorization on System A.

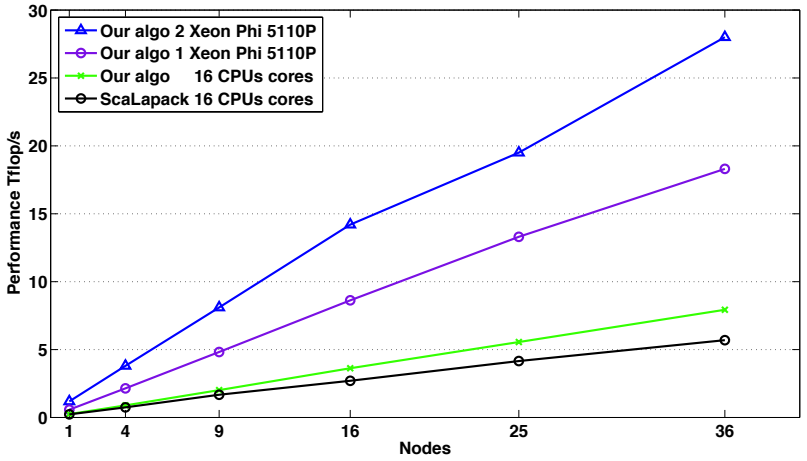

Figure 3.13: Weak scalability (horizontal reading) strong scalability (vertical reading) of the distributed multi-device Cholesky factorization on System B.

Figure 3.12 illustrates the performance of the Cholesky factorization on System A – distributed platform with GPU accelerators. We plotted the best performance obtained by the state-of-the-art ScaLapack software as implemented by the Intel MKL, and tuned for the best blocking factor n_b across multiple runs. We also plotted the performance obtained by our algorithm when using only the CPUs.

This allowed us to compare fairly with the ScaLapack approach. We can see that our implementation is between 15% to 20% faster than its ScaLapack counterpart and we achieved perfect weak scaling – a result we were expecting. The ScaLapack approach follows the classical "*bulk-synchronous*" technique, meaning that, at every phase of the factorization there is a synchronization. Thus, there is a synchronization between the three phases of the Cholesky algorithm. The bottleneck of the ScaLapack approach compared to our proposed dynamic technique can be summarized by the following observations:

- during the diagonal tile factorization, only one processor is working in ScaLapack while in our technique, when a processor is performing the diagonal factorization of step i, the other processors are still applying updates from step $i-1$.
- ScaLapack cannot hide the overhead of the communication because it issues only blocking message passing calls, while in our approach, the communication is hidden since it is handled by a separate thread and thus when a communication is in progress, the other threads are busy with computational kernels.
- Close to the end of the factorization, there is not enough work to keep the processors fully occupied, this is a bottleneck for the ScaLapack approach, while its effect is minimized for the algorithm we proposed because of the multi-dimensional lookahead technique.

Figure 3.12 also shows the weak scalability for our algorithm when adding either 1, 2, or 3 GPUs. This experiment demonstrates a good weak scalability when using heterogeneous hardware. Enabling more GPUs on each node brought the performance up in a proportionate fashion. The performance obtained on 100 nodes using 3 NVIDIA M2090 GPUs is about 78 Tflop/s for a fixed problem size of 30 000 per node. Similarly, our experiments on System B illustrates the same behavior. Our approach exhibits a very good scalability when using 1 or 2 Intel Xeon Phi KNC 5110P

coprocessors. The performance obtained on 36 nodes using 2 Xeon Phi coprocessors is about 28 Tflop/s for a fixed problem size of 30 000 per node.

3.7 Conclusion

This chapter proposes a unified programming model for developing high-performance dense linear algebra in multi-way heterogeneous environments using a task-based runtime. In particular, we present best practices and methodologies from the development of high-performance dense linear algebra for accelerators. We also present how judicious modifications to a task-based runtime are used to ensure that we meet two competing goals: (1) to obtain high fraction of the peak performance for the entire heterogeneous system, (2) to employ a programming model that would simplify the development. Our task superscalar runtime environment allows simple serial algorithmic implementations that are flexible enough to achieve high performance on both shared-memory and distributed-memory heterogeneous environments.

Chapter 4

Soft Error Resilient Design for a Task-based Runtime

4.1 Introduction

As the scale of modern computing systems grows, failures will happen more frequently. On the way to exascale computing, a generic, low-overhead and resilient extension becomes a desired aptitude of any programming paradigm. In this chapter we explore three additions to a dynamic task-based runtime to build a generic framework providing soft error resilience to task-based programming paradigms. The first recovers data corruption by re-executing the minimum number of tasks, the second takes critical checkpoints of the data flowing between tasks to minimize the necessary re-execution, while the last one takes advantage of algorithmic properties to recover data corruption without re-execution. These three resilient mechanisms have been implemented in the PaRSEC task-based runtime framework Cao et al. (2015). Experimental results validate our design and quantify the overhead introduced by these mechanisms.

The rest of this chapter is organized as follows: Section 4.2 introduces and the background of task-based scheduling using PaRSEC and the impact of soft errors

on applications using task graph scheduling. Section 4.3 proposes three mechanisms to protect applications against soft errors, including: correcting Sub-DAG strategy, sub-DAG & data logging composite strategy and algorithm based fault tolerance strategy. Section 4.4 describes the design of merging data logging mechanism into PaRSEC runtime level to provide automatic resilience for application running on PaRSEC. Section 4.5 shows the experimental results on Titan supercomputer at Oak Ridge National Laboratory and Section 4.6 concludes this chapter.

The portion of this chapter has been published in the following publication of mine:

- Chongxiao Cao, Thomas Herault, George Bosilca, Jack Dongarra, "Design for a Soft Error Resilient Dynamic Task-based Runtime", Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International

4.2 Problem Statement

While most of the techniques introduced are generic, in this chapter, we will illustrate the soft error resilient design using the tiled Cholesky factorization Buttari et al. (2009). This algorithm factors an $N \times N$, symmetric, positive-definite matrix A into the product of a lower triangular matrix L and its transpose, i.e., $A = LL^T$ (or $A = U^T U$, where U is upper triangular). We implement it using a tiled linear algebra algorithm in which linear algebra operations are represented as a set of tasks that operate on square blocks of data (the tiles), and are dynamically scheduled based on the dependencies among them and on the availability of computational resources.

Algorithm 7 describes the tiled Cholesky factorization algorithm and Figure 4.1 shows the snapshot of this factorization on a 4×4 tile matrix at step $k = 1$. The algorithm consists of four computational kernels: **POTRF** (Cholesky factorization), **TRSM** (triangular solver), **SYRK** (symmetric rank-k update) and **GEMM** (general matrix-matrix multiplication) that each operates on a tile (the matrix A is tiled in $NT \times NT$ tiles of size $nb \times nb$, and $A[m][n]$ represents a whole tile of A). It is important

Algorithm 7: Tiled Cholesky Factorization Algorithm

```
for k = 0...NT − 1 do
    A[k][k] ← POTRF(A[k][k])
    for m = k + 1...NT − 1 do
        A[m][k] ← TRSM(A[k][k], A[m][k])
    for n = k + 1...NT − 1 do
        A[n][n] ← SYRK(A[n][k], A[n][n])
    for m = n + 1...NT − 1 do
        A[m][n] ← GEMM(A[m][k], A[n][k], A[m][n])
```

to note that only the lower or upper triangular part of the input matrix is allocated and initialized as Cholesky factorization operates on a symmetric matrix. Figure 4.1 also demonstrates that all operations only update data in lower triangular part of the matrix. In order to implement this tiled algorithm on a distributed-memory platform, the classical 2D block cyclic distribution is applied as it guarantees good scalability and satisfactory load balancing following the "owner computes" strategy. As shown in Figure 4.2, a 4×4 tile symmetric matrix is distributed evenly on a 2×2 grid of processes. Implemented in PaRSEC, the Cholesky factorization can be described in the form of DAG consisting of tasks and data dependencies. Figure 4.3 shows the corresponding DAG for Cholesky factorization of a 4×4 tile matrix on a 2×2 process grid. Each of the four computational kernels has been represented by different type of tasks. Data dependencies between tasks can be local or remote, depending on the position the predecessor task generating the input flow. Also, the input dataflow may also be overwritten after the task execution. For example, in Figure 4.3, a **TRSM** task overwrites the input from its predecessor task **GEMM**, and a **POTRF** task overwrites the input from its predecessor task **SYRK**.

Soft errors can happen at random moment and memory location, normally in the form of a bit flip. Figure 4.4 demonstrates how data corruption is propagated in a DAG when a soft error occurs. If this failure happens during the execution of a **TRSM** task on Node 1 (marked using a red cycle in the figure), till the end of the

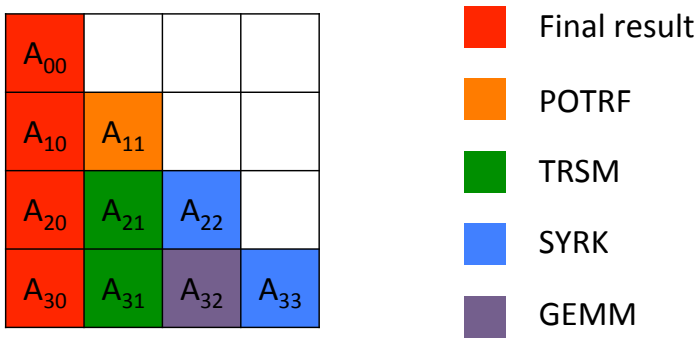

Figure 4.1: Step $k = 1$ of a Cholesky factorization of 4x4 tile matrix.

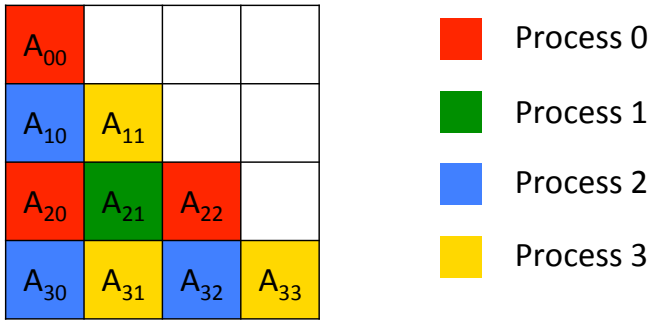

Figure 4.2: Example of a tile 2D block cyclic distribution.

factorization the corrupted data flow would have been propagated to the following 6 tasks (marked using blue cycles in the figure), ruining in total 35% of the tasks in this example.

As stated in the beginning of this chapter, we plan to design a solution at the runtime level that has low overhead and is applicable to a wide range of applications. One possible technique to offer low overhead solution is to prevent the corrupted data flow from propagating to the failed task's successors, therefore the failed task should be recovered right after the soft error happens, and before it is able to propagate the corrupted data flow. By utilizing task graph scheduling, the recovery of a failed task can be also represented in the form of a DAG. By inferring the data dependencies in recovery, a task-based runtime is able to exploit maximum parallelism by overlapping

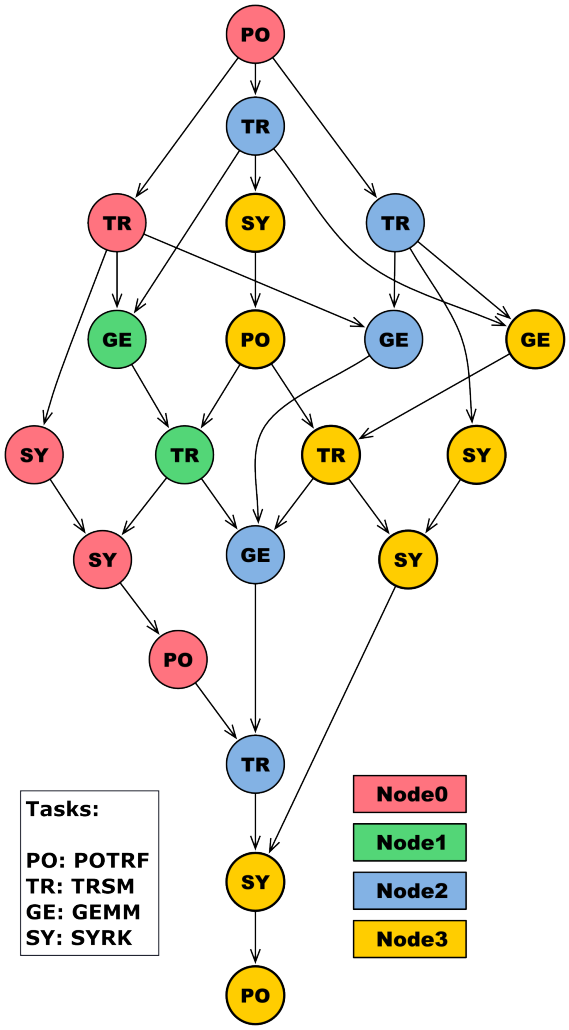

Figure 4.3: DAG of the Cholesky factorization of a 4x4 tile matrix on a 2x2 process grid.

the recovery with original execution. Dependency conflict is solved correctly by runtime and the concurrent recovery provides a low-overhead solution to deliver correct result. Also, as task graph scheduling only triggers the recovery when a failure happens, this dynamic feature avoids extra synchronization in a distributed-memory environment, keeping the failure free execution almost unaltered from the original, non resilient, execution.

On the other hand, algorithmic methods to mitigate the impact of data corruption on specific applications are well developed. For example, in dense linear algebra,

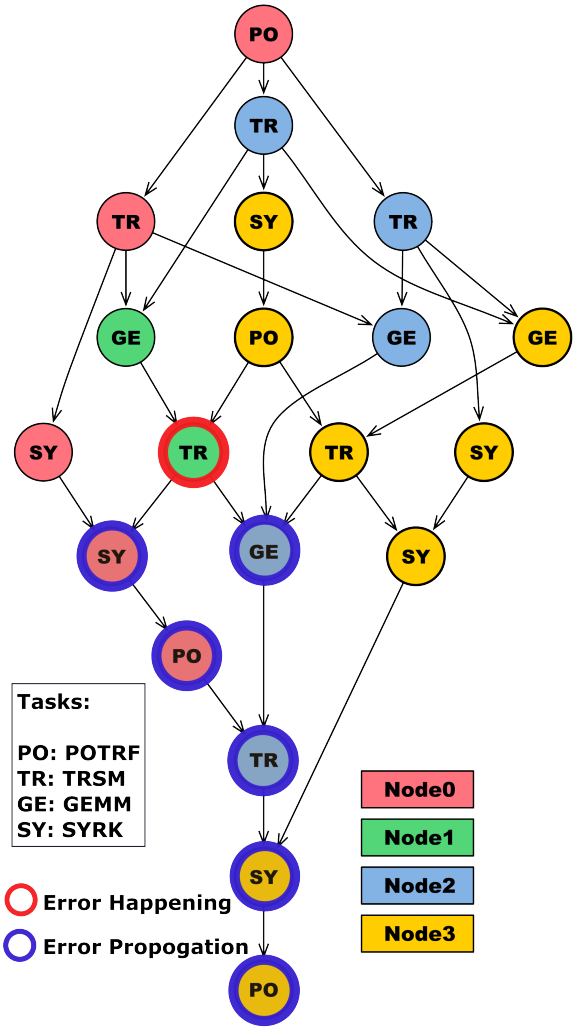

Figure 4.4: DAG of the Cholesky factorization of a 4x4 tile matrix on a 2x2 process grid, and a possible scenario of a soft error propagation (starting from the task surrounded by a red line).

Sherman-Morrison formula is applied to recover from one soft error during an LU factorization, by introducing very small overhead Du et al. (2011a). However, in this chapter, we are not going to design fault tolerant techniques based on modifying the original algorithm of an application. As this algorithmic level design is not generic and requires modification for every targeted application. We are focusing on dealing with the challenge of designing an independent generic strategy that can be easily applicable to any task-based programming paradigm.

4.3 Design of Soft Error Resilience in PaRSEC

4.3.1 Sub-DAG Mechanism

For any task graph based application, its represented DAG is required to be stored during execution as this DAG provides hints for task discovery and scheduling. In PaRSEC, a DAG is represented by a concise format called Parameterized Task Graph (PTG), which expresses the tasks and their data dependencies in a symbolic way, independent of the problem size Bosilca et al. (2013). PaRSEC runtime engine takes advantage of this concise representation to discover and schedule tasks without unfolding the entire DAG in memory, reducing the memory requirement for storing the DAG and exchanging the computation cycles to traverse the DAG with cycles to compute the successors of a task.

When a soft error happens, the output, and potentially some of the input, of the failed task is corrupted. A low-overhead solution to recover data corruption is to suspend all the successors of the failed task until it is recovered. A generic way to recover a failed task in a DAG is to re-execute it to deliver correct result. The re-execution requires its predecessors to provide the input again. However, task graph scheduling releases input dataflow after a task completes in order to reduce memory cost during execution. As a result, the input dataflow for a failed task is not saved, and it requires its predecessors to be re-executed as well. This backward traverse will go along the opposite direction of the original data flow until it reaches the source task of each of the necessary data. Considering the fact that the input for an application comes from a read-only stable storage and is not affected by soft errors, as long as the runtime is able to retrieve any information of the DAG, the correct result of any task in the DAG can be regenerated. According to runtime's feature of expanding DAG at any level during execution, a straightforward idea to recover from soft errors is to reuse the original data and DAG to recompute the missing data. Based on this idea, we exploit the capability of PaRSEC's PTG representation to dynamically retrieve

all the predecessors of a failed task. In this mechanism, a failed task is replaced by re-executing a *correcting* sub-DAG consisting of this task and all its predecessors. The runtime provides the functionality of regenerating and scheduling the sub-DAG dynamically.

Figure 4.5 demonstrates an example of the correcting sub-DAG for the failed **TRSM** task in Figure 4.4. Compared with the original DAG, the size of this sub-DAG has been reduced to minimum, only consisting of tasks related to re-executing the failed task. Re-executing this sub-DAG ensures that the failed task and its predecessors are recomputed only once from the original input data. As analyzed before, the recovery here is not sequential, it is executed in parallel with other available tasks in the original DAG. When implementing this correct sub-DAG mechanism in PaRSEC, every computing node on a distributed-memory platform owns its scheduling engine and is able to parse the concise PTG representation to unfold the DAG at any level. After a soft error is detected, the computing node re-executing the failed task triggers a global creation of the correcting sub-DAG by broadcasting a recovery message to other computing nodes.

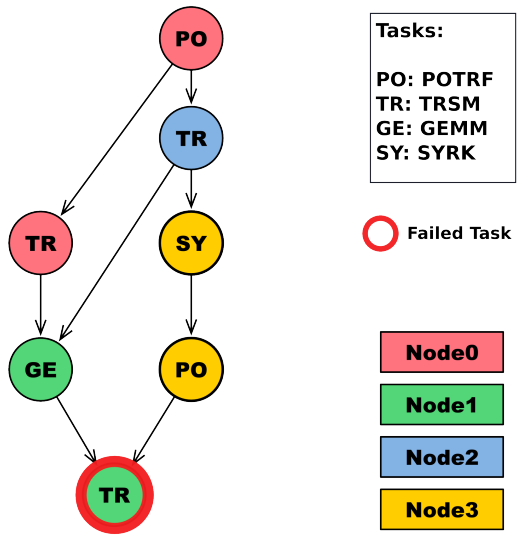

Figure 4.5: Correcting sub-DAG for the failed **TRSM** task.

This correcting sub-DAG strategy is designed at application level, and treats every task as a generic object. Thus, it can be integrated into a task-based runtime to support any application running on it. In the following, we analyze the computing overhead and storage overhead of this mechanism:

The **Computing Overhead** is proportional to the position of the failed task in the DAG. If the failure happens in the early stage of the execution, that means the size of the correcting sub-DAG is small and the computing overhead will be relatively low. On the other hand, if the failure happens in the late stage of the execution, the size of the sub-DAG will be large and the computing overhead will be relatively high. We analyze the computing overhead by investigating the algorithm. Figure 4.6 shows an example when a failure happens in the middle of the Cholesky factorization. Failure can strike four types of tasks, and the recovery cost of the **POTRF** task is minimum, as it is the predecessor of all the other three types of tasks. We compute the overhead as the number of additional floating point operations (FLOPs) to re-execute. The recovery of a **POTRF** task takes the same amount of FLOPs as a Cholesky factorization on the top left submatrix that encompasses the failed **POTRF** task. On this example, it is a half-size submatrix $A[[0, N/2], [0, N/2]]$, as marked by dark blue line in Figure 4.6. We use the cost of recovering failed **POTRF** in Kth column as the theoretical computing overhead. It is computed as:

$$FLOP_{Orig} = \frac{1}{3}N^3 \qquad FLOP_{Extra} = \frac{1}{3}K^3$$

$$Overhead_{Comp} = \frac{FLOP_{Extra}}{FLOP_{Orig}} = (\frac{K}{N})^3$$

Table 4.1 summarizes the computing overhead for Cholesky when failures happen at different stages of execution. In the failure-free case, the correcting sub-DAG is never created and there is no performance penalty.

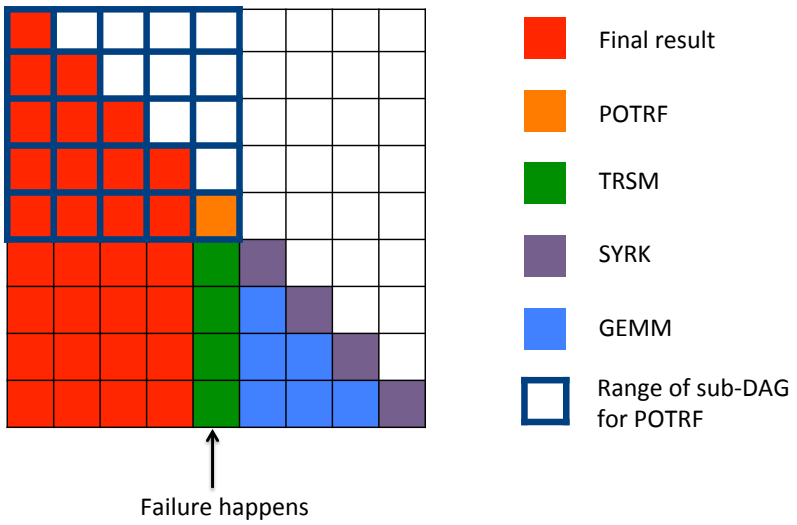

Figure 4.6: Example when a failure happens in the middle of a factorization.

Table 4.1: Computing overhead of sub-DAG mechanism for Cholesky factorization.

Failure Position	Beginning	Middle	End	No Failure
$Overhead_{Comp}$	$(\frac{nb}{N})^3$	12.5%	100%	0

Storage Overhead: This sub-DAG mechanism requires extra memory space to execute the correcting sub-DAG. Thus in the worst case, when the last task in original DAG is failed, another $N \times N$ symmetric, positive-definite matrix is allocated and the storage overhead is 100%.

4.3.2 Sub-DAG & Data Logging Composite Mechanism

As analyzed above for the correcting sub-DAG mechanism, the re-execution always starts from the beginning of the DAG because the intermediary data is released during failure-free execution. The computing overhead explodes when a failure happens in the late stages of the execution, up to 100% to recover the final task of the factorization, meaning that the whole application needs to be recomputed.

In this strategy, the previous approach is augmented by adding a data logging mechanism to limit the necessary rollback and therefore to reduce the number of re-executed tasks. When recovering a failed task, only the predecessors after the newest saved intermediary data are required to be re-executed.

In this approach, every matrix tile is treated as a data logging unit. We define a logging interval β, meaning that a copy of dataflow is reserved in memory after every β updates. Logging interval β can be modeled as a function of failure rate, task execution time and checkpoint time Daly (2006). The optimal value of β is not discussed in this chapter and we set it to a constant. Dataflow is logged locally on every computing node. By combining all local logged data together, a fully-fledged snapshot exists during the execution. Logged data is saved to local memory and we assume that the probability that both the data and its saved copy are corrupted by correlated failures is negligible. Figure 4.7 shows an example when $\beta = 2$ is applied . The faded tasks here mean those tasks already completed, and those intermediary dataflows updated twice are saved into local memory. It is important to mention that only Read/Write (RW) flows require to be reversed during DAG execution, Read flows are final results of matrix factorization and will not be modified in future. Figure 4.8 demonstrates how to recover the last task in Cholesky factorization using data logging mechanism. Here only those tasks generating intermediary flows below the logged wave are required to re-execute. Also, tasks generating final results are not required to re-execute, as final results can be retrieved from corresponding matrix tiles directly. By utilizing this mechanism, the size of correcting sub-DAG is trimmed to much smaller compared with original sub-DAG mechanism. In this case, only two tasks, one **SYRK** task and one **POTRF** task are re-executed to the failure.

Computing Overhead: In this mechanism, the computational cost of recovery does not depend on the failure position. Moreover, in the Cholesky factorization, the input of any task is either the owner tile for which there was a data copy logged at most β operations ago, or a final output of another task, that is validated before the re-executed task could start. Thus, any failed task output can be recovered by

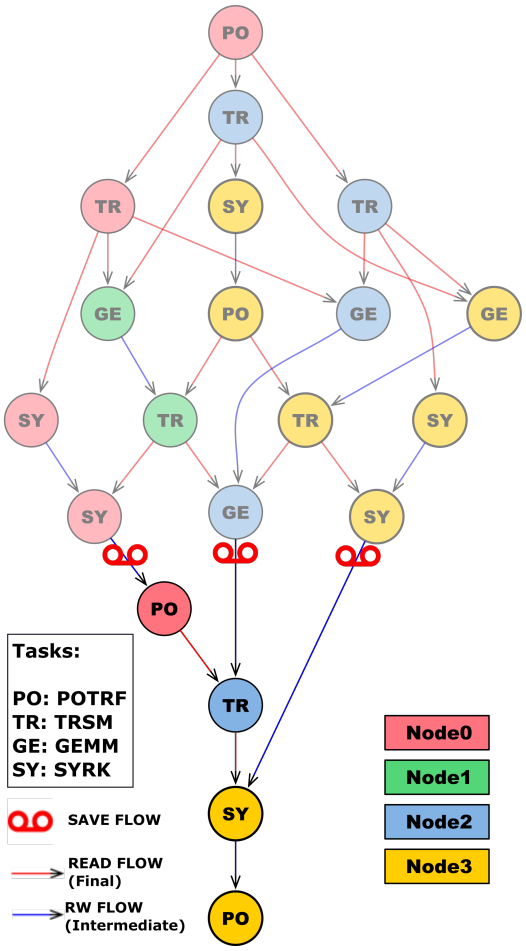

Figure 4.7: An example of combining correcting sub-DAG with data logging method when logging interval $\beta = 2$.

re-executing at most β previous tasks on the same tile. The number of FLOPs of a task computed as $C \cdot nb^3$, where C is 1/3 for **POTRF**, 1 for **TRSM**, 1 for **SYRK** and 2 for **GEMM**. We set C to 2 to provide a conservative bound when estimating computing overhead for any failed task. The theoretical computing overhead can be computed as:

$$\begin{aligned} FLOP_{Extra} &= \beta 2nb^3 \\ Overhead_{Comp} &= \frac{FLOP_{Extra}}{FLOP_{Orig}} = \frac{\beta 6nb^3}{N^3} \end{aligned}$$

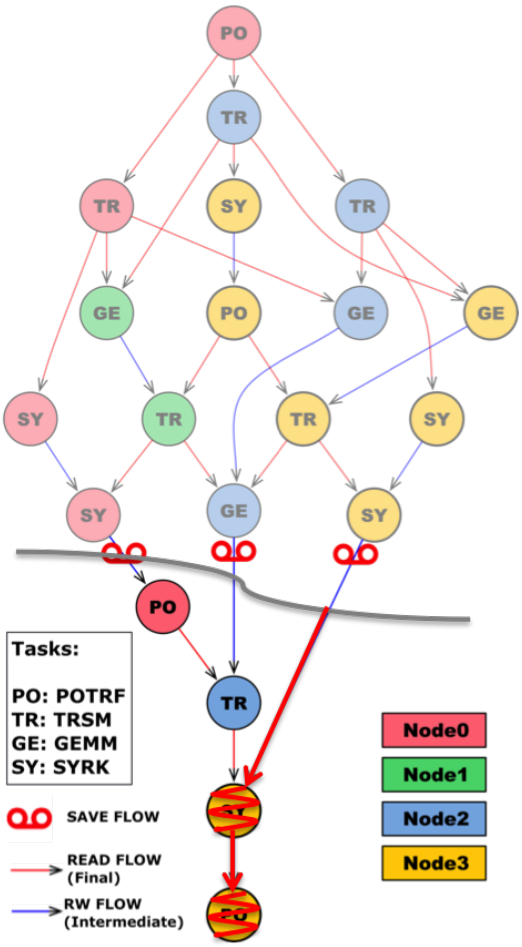

Figure 4.8: An example of recovering a soft error with data logging method when logging interval $\beta = 2$.

Table 4.2 summarizes the computing overhead of the Cholesky factorization using this sub-DAG & data logging composite strategy. The overhead in failure free execution is close to 0 because the cost of logging data into local memory is negligible.

Table 4.2: Computing overhead of sub-DAG & Periodic Checkpoint mechanism for Cholesky factorization.

Failure Position	Beginning	Middle	End	No Failure
$Overhead_{Comp}$	$\frac{nb^3}{N^3}$	$\frac{\beta 6nb^3}{N^3}$	$\frac{\beta 6nb^3}{N^3}$	≈ 0

Storage Overhead: This mechanism needs to allocate the same size of matrix as input to store data flowing snapshot periodically, even if the initial data is available on a stable storage. Thus, the storage overhead is 100%.

4.3.3 Algorithm-Based Fault Tolerance Mechanism

The two application level mechanisms described above take advantage of the task graph of the application to recover from failures by re-executing the minimum number of tasks. A different approach, potentially less generic, is to use an algorithmic invariant to completely avoid re-execution. This approach is based on Algorithm Based Fault Tolerance (ABFT) techniques, with well known solutions for most of the dense and sparse linear algebra kernels. In order to recover from data corruption on a matrix tile by using ABFT, additional information would be attached to this tile to provide error correction functionality if necessary.

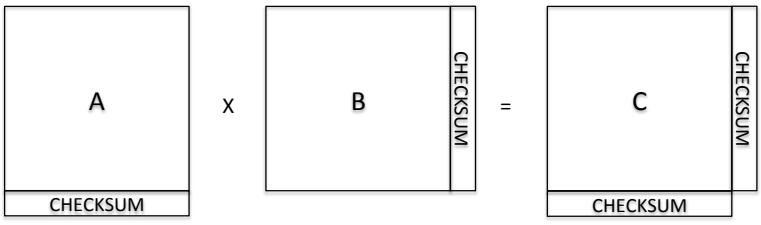

Figure 4.9: An ABFT matrix multiplication example.

ABFT was firstly introduced by Huang and Abraham to detect and correct soft errors in systolic arrays Huang and Abraham (1984). ABFT techniques are based on the idea of maintaining consistency of the computing data and recovery data, by applying appropriate mathematical operations on both original data and recovery data Du et al. (2012). Typically, for linear algebra operations, additional rows and/or columns are attached to input matrix to be maintained as checksums. An example of ABFT enabled matrix multiplication is shown in Figure 4.9. The corresponding matrices A, B, C have the following relationship:

$$A * B = C$$

A is appended with a column checksum vector $e^T A$ while B is appended with a row checksum vector Be. It is demonstrated as follows that the checksum relationship for matrix C keeps consistent after computation:

$$\begin{bmatrix} A \\ e^T A \end{bmatrix} \begin{bmatrix} B & Be \end{bmatrix} = \begin{bmatrix} AB & ABe \\ e^T AB & e^T ABe \end{bmatrix} = \begin{bmatrix} C & Ce \\ e^T C & e^T Ce \end{bmatrix}$$

In this task level approach, we attach two column checksum vectors to every tile in the input matrix. Note that we do not modify the factorization itself: the checksums are only attached original data layout. Figure 4.10 gives the example of the matrix snapshot after attaching two column checksum vectors to a 4×4 tile matrix. It is also important to mention that this ABFT based approach also provides an efficient soft error detector. In order to detect errors in one matrix tile, one checksum vector is sufficient. Furthermore, detecting and correcting n errors in one matrix tile require at least $n + 1$ checksum vectors Wu and Chen (2014). For example, let's consider an $n \times n$ matrix $A = [a_1, a_2, ..., a_n]$ and two n vectors

$$e_1 = (1, 1, ..., 1)^T \qquad e_2 = (1, 2, ..., n)^T$$

Two column checksum vectors are defined as:

$$c_1 = e_1 A \qquad c_2 = e_2 A$$

Assume that an error happens at the (i, j) element of A:

$$a'_{i,j} = a_{i,j} + \gamma$$

Now we can first decide the jth column of A is inconsistent with checksums:

$$\alpha_1 = \sum_{k=1}^{n} a_{k,j} - (c_1)_j = \gamma \neq 0$$

ABFT techniques provide an efficient soft error detector, which can be used to complement or replace existing hardware error detection mechanisms using ECC memory. By attaching a single checksum vector to every matrix tile, we can also implement a failure detector for the two application level mechanisms described above. When a soft error happens, the checksum vector can determine the ith element of the failed column causes the inconsistency:

$$\alpha_2 = \sum_{k=1}^{n} k a_{k,j} - (c_2)_j = i\gamma$$

$$\alpha_2/\alpha_1 = i$$

For error correction, the value of $a_{i,j}$ is corrected by simply subtracting α_1.

For the remainder of this chapter, we consider the case of a single soft error per execution. We simulate it by introducing a significant bit-flip into the exponent of a floating point data and let the runtime detect and recover the state of the computation. It is to be noted that for some tasks, a single soft-error is propagated inside a task and translates into multiple data corruption. While the ABFT mechanism presented here is not able to correct the data in this particular case, it can still be trusted as a detection mechanism. In this case, the two mechanisms from Section 4.3.1 and 4.3.2 can successfully complement the ABFT approach. In the following analysis of overhead, we focus on the case where failures can be recovered using an ABFT mechanism.

Computing Overhead: Using ABFT techniques, extra FLOPs are introduced *for each task* from maintaining the consistency of checksums and validating result. After attaching checksum vectors, the matrix size becomes $(1+2/nb)N \times (1+2/nb)N$.

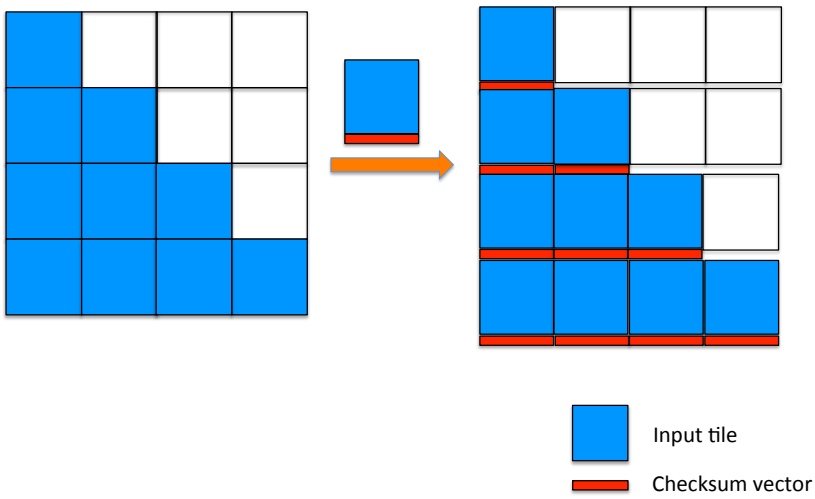

Figure 4.10: Attaching checksum vectors to a 4x4 tile symmetric matrix.

The number of FLOPs of the Cholesky factorization on this larger matrix is:

$$FLOP_{New} = \frac{1}{3}((1 + \frac{2}{nb})N)^3$$

The cost of maintaining checksum is:

$$FLOP_{Chk} = \frac{1}{3}((1 + \frac{2}{nb})N)^3 - \frac{1}{3}N^3$$

The computing overhead of maintaining checksums is:

$$Overhead_{Chk} = \frac{FLOP_{Chk}}{FLOP_{Orig}} = (1 + \frac{2}{nb})^3 - 1$$

The number of extra FLOPs of correcting error in one task comes mostly from using checksum vectors to detect inconsistency, which is $2nb^2$. The detection operation is a matrix vector multiplication and we wrap this operation into the task definition of **POTRF**, **TRSM**, **SYRK** and **GEMM**. Failure detection is enabled upon the completion of every task automatically and doesn't modify original DAG.

If one failure is detected, only nb floating point operations are required to locate the error position and only one FLOPs is required to add the error back to the corrupted matrix element. These $nb+1$ operations are negligible comparing with the large amount of operations in maintaining checksums and detecting errors, and thus are discarded in overhead estimation. There are approximately $(N/nb)^3/6$ tasks in Cholesky factorization, thus we estimate the total cost of correcting error as:

$$FLOP_{Corr} = \frac{N^3}{3nb}$$

The computing overhead of correcting error is:

$$Overhead_{Corr} = \frac{FLOP_{Corr}}{FLOP_{Orig}} = \frac{1}{nb}$$

The total computing overhead of this mechanism is:

$$\begin{aligned} Overhead_{Comp} &= Overhead_{Chk} + Overhead_{Corr} \\ &= (1+\frac{2}{nb})^3 - 1 + \frac{1}{nb} \end{aligned}$$

This task level mechanism recovers the soft error internally, avoiding possible task re-execution. Thus the recovery overhead does not depend on the failure position in the DAG. As shown in Table 4.3, the computational overhead remains constant for one failure case as every task itself has become resilient after attaching checksum vectors.

Table 4.3: Computing overhead of ABFT mechanism for Cholesky factorization.

	One Failure	No Failure
$Overhead_{Comp}$	$(1+\frac{2}{nb})^3 - 1 + \frac{1}{nb}$	$(1+\frac{2}{nb})^3 - 1 + \frac{1}{nb}$

Storage Overhead: The ABFT mechanism requires allocating extra memory to store checksum vectors. For every $nb \times nb$ tile, the size of 2 checksum vectors is $nb \times 2$, thus the total storage overhead is $2/nb$. In tiled dense linear applications, the

tile size is tuned to optimize the efficiency of the operation and the parallelism of the application. This often translates in nb in hundreds, which make the extra memory requirement of storing checksum vectors negligible.

4.4 Fault Tolerant Layer in PaRSEC

As mentioned is Section 4.3.2, data logging mechanism is generic and low-overhead in nature and is able to be integrated into any application that can be expressed as a DAG of tasks with labeled edges designating data dependencies. Here we move this mechanism into the runtime level of PaRSEC to provide automatic resilience for non fault tolerant applications on PaRSEC. Three major functionalities are implemented in this fault tolerant layer to ensure a resilient running environment for DAG-based applications:

1. Reserving minimum dataflows for protection.
2. Minimizing number of re-executed tasks for recovery.
3. Minimizing extra memory used for data logging.

Also, this fault tolerant layer exports a configuration interface for application developers and auto tuning tools to setup optimal logging scheme for specific application. Figure 4.11 explains how this fault tolerant layer works. An original application without any fault tolerant features is submitted into PaRSEC in a form of DAG. Whenever a task is completed, fault tolerant layer will check whether any output dataflow needs to be reserved based on user/tool defined data logging scheme. Moreover, every time when a dataflow reaches logging point or becomes final data, this fault tolerant layer will release previous saved version for this dataflow, in order to keep memory overhead to minimum, and to guarantee that there is always one active logging wave during application execution for recovering potential failures. On the other side, if a failure reported from underlying hardware (i.e., ECC memory)

or application's own algorithmic feature, this fault tolerant layer will generate a minimum DAG consisting of all the tasks from the newest logging wave to the failed task and execute it in parallel with other non failed tasks in original DAG.

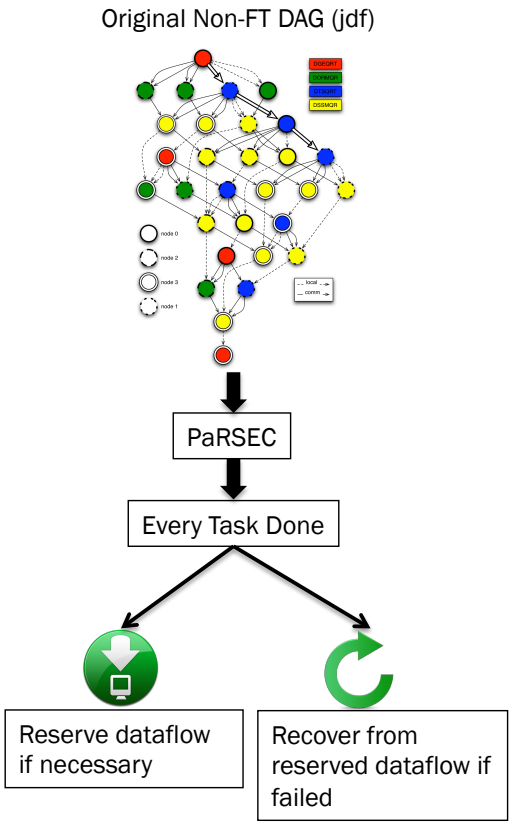

Figure 4.11: Fault Tolerant Layer in PaRSEC, Supporting Non Fault Tolerant Applications.

4.5 Experimental Results

4.5.1 Experiment Setup

The Titan supercomputer at Oak Ridge National Laboratory is used as experimental platform. Titan has 18,688 nodes with Cray custom high-speed interconnect, each node contains a 16-core AMD Opteron 6274 CPU with 32 GiB of DDR3 ECC memory

and an Nvidia Tesla K20X GPU with 6 GiB GDDR5 ECC memory. Our experiments are tested on the CPU section of Titan, and those mechanisms are generic to be extended to support GPU section of Titan. Up to 256 computing nodes of Titan are used in performance evaluation, and for every computing node, 8 CPU cores are used to explore parallelism inside a node. At the software level, we use GCC 4.8.2 as compiler and Cray LibSci 12.2.0 to provide basic linear algebra subroutines (BLAS).

In the following experiments of validating overhead, we inspect the overhead of three proposed mechanisms for soft errors and the overhead of protection and recovery from fault tolerant layer of PaRSEC separately. In order to investigate the practical overhead of three mechanisms comparing with the theoretical overhead analyzed before in this chapter, three resilient mechanisms enabled Cholesky factorizations are tested, and failures are injected as single bit-flip inside one task during the execution. Failure is triggered in the middle of Cholesky factorization, as indicated in the Figure 4.6. Also, in order to investigate the protection and recovery overhead from fault tolerant layer in PaRSEC, we launch a non resilient QR factorization and inject a failure in the middle of QR factorization.

Both Cholesky factorization and QR factorization are implemented in double precision with tile size $nb = 200$ that was tuned to reach the highest performance of trailing matrix update operation (i.e., the **GEMM** operation and **TSMQR** operation respectively), while still allowing a large amount of parallelism at reasonable matrix sizes. To serve as a comparison base, we use the standard Cholesky and QR factorization implemented in PaRSEC without adding any soft error resilient mechanism.

We pursue weak scalability experiments to evaluate the capability of the proposed fault tolerant strategies to handle potentially larger problems when more computing resources are available. For these experiments, we fix the memory used on each node and increase the matrix size accordingly when we increase the number of nodes. The input matrix size for single-node experiments is set to 6000 for Cholesky factorization

and 8000 for QR factorization, and is scaled to $6000\sqrt{P}$ and $8000\sqrt{P}$ respectively, where P is the number of nodes.

In all experiments reported in this section, we take 5 runs and report the average (arithmetic mean) performance.

4.5.2 Performance of Sub-DAG Mechanism

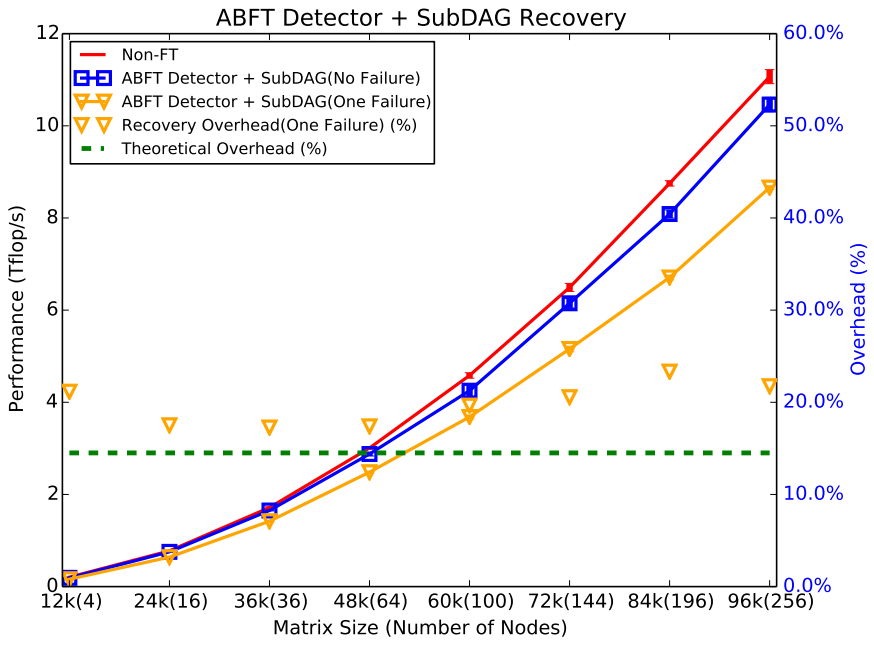

Figure 4.12: Weak scalability of correcting sub-DAG mechanism compared to non fault tolerant Cholesky.

Figure 4.12 shows the performance and overhead of the Cholesky factorization on Titan with the correcting sub-DAG mechanism when one failure happens during the execution. We inject a failure in one **GEMM** task when factorization goes to the middle column of the matrix. Here a failure detector is implemented using ABFT methodology by add one checksum vector to every matrix tile. The red curve is the performance of non fault tolerant Cholesky factorization implemented in standard PaRSEC. The blue curve is the performance of fault tolerant version in failure-free execution. Comparing with non-fault tolerant performance, its overhead

is introduced by the failure detector. The theoretical overhead for one failure is about 15%, including failure detecting overhead using one checksum vector from Table 4.3 and failure recovering overhead as computed from Table 4.1 (as explained, this is the cost of computing a Cholesky factorization on a matrix of half size). We can see that the overhead of one-failure case varies around 15% to 20%, which is close to theoretical overhead.

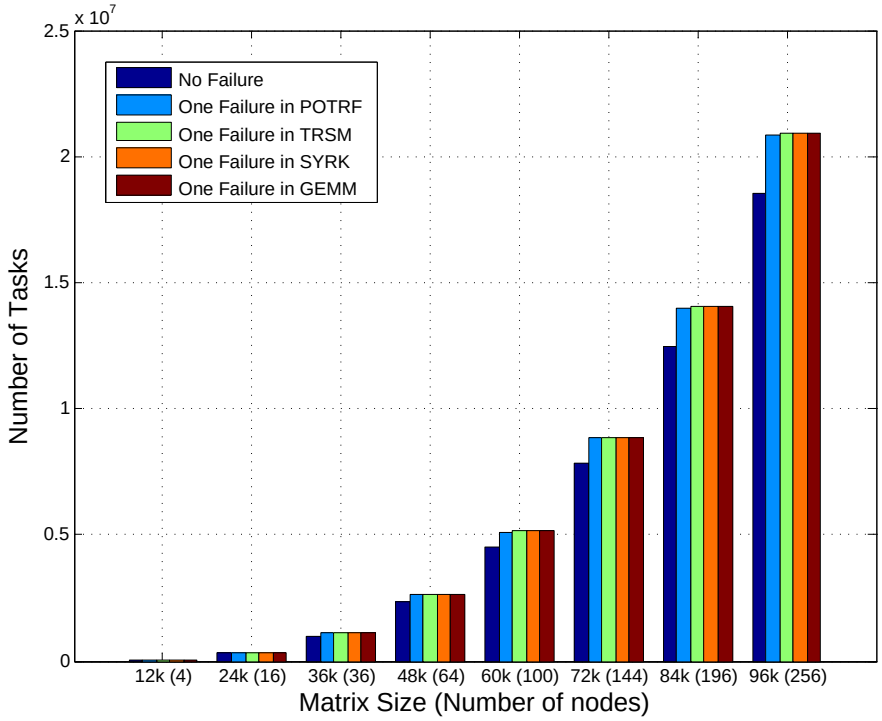

Figure 4.13: Number of total tasks of the correcting sub-DAG mechanism in failure-free and one-failure cases.

As failure could strike any of the four types of tasks in Cholesky factorization. Figure 4.13 shows the number of total tasks of the correcting sub-DAG mechanism in different cases. It indicates that the number of re-executed tasks for four types of tasks doesn't change much. Recovering from failures in **POTRF** requires the fewest number of task re-executions since **POTRF** is on the critical path of DAG and is the predecessors of the other three types of tasks.

4.5.3 Performance of Sub-DAG & Data Logging Composite Mechanism

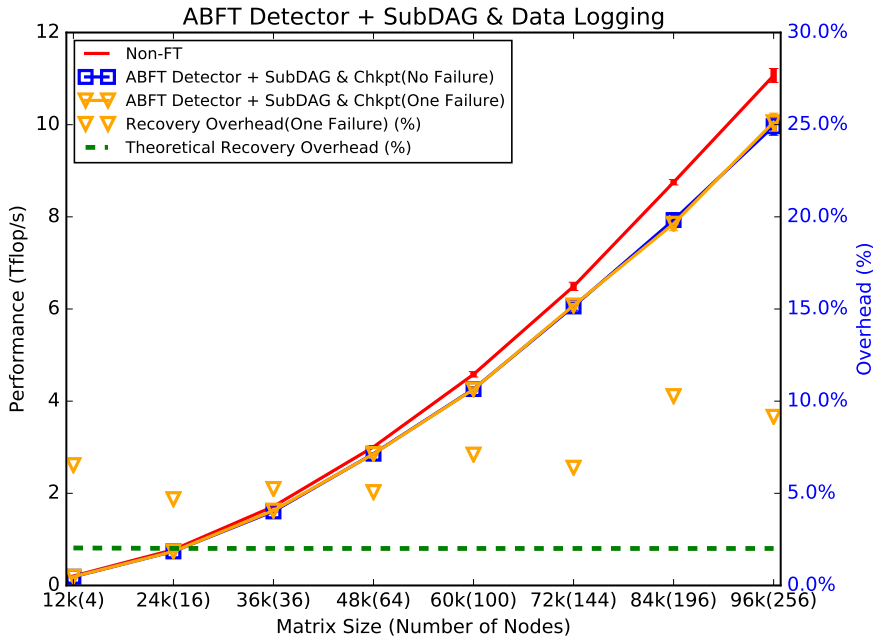

Figure 4.14: Weak scalability of correcting sub-DAG & data logging composite mechanism compared to non fault tolerant Cholesky.

Figure 4.14 shows the performance and overhead of the Cholesky factorization on Titan with the correcting sub-DAG & data logging composite mechanism when one failure is injected during the execution. The same as the previous experiment, a failure detector is also implemented using ABFT methodology by add one checksum vector to every matrix tile. The checkpoint interval β is set to 10, i.e. a copy of one matrix tile is saved to memory locally after 10 updates. Since data logging mechanism reserves intermediary dataflow and limits the maximum number of re-executed tasks to 10, for one failure case, the failure is injected in one **GEMM** task without loss of generality. Recovering from data corruption in the other three types of tasks have similar overhead. Based on the discussion in Section 4.3.2 and Section 4.3.3, the theoretical overhead includes recovery overhead which is close to 0 (as explained,

only 10 tasks are required to re-execute comparing with a large number of original tasks) and failure detection overhead which is close to 2%.

We can see that the overhead of the one-failure case fluctuates around 7%. Comparing with small theoretical overhead, the practical one also includes the noise of the measurement. These results validate our analysis that in-memory data logging mechanism reduces the number of re-executed tasks drastically and the cost spent for logging intermediary dataflow remains negligible.

4.5.4 Performance of ABFT Mechanism

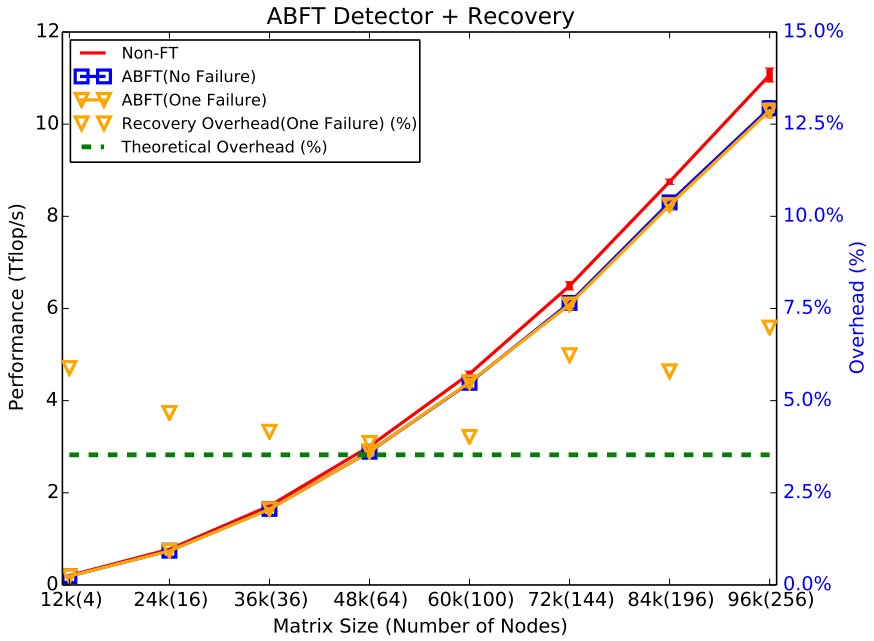

Figure 4.15: Weak scalability of ABFT mechanism compared to non fault tolerant Cholesky.

Figure 4.15 presents the performance and overhead of Cholesky factorization with task level fault tolerant support using ABFT technique on Titan. ABFT based mechanism provides both failure detection functionality and failure correction functionality. Each task in original execution DAG is validated at completion, and corrective actions are initiated when this validation fails. For the one-failure case, we

inject the failure in one **GEMM** task, and the erroneous matrix element will not be propagated inside the task thus it can be recovered using checksums. The theoretical overhead is obtained from Table 4.3. The results show that the overhead of recovering one failure fluctuates from 5% to 7.5%, and does not increase when application size and number of nodes increase, and remains close to the theoretical overhead. Also, it is important to note that the difference between failure-free performance and one-failure performance is negligible. Compared with failure-free case, only nb more FLOPs are required to locate the error position and only one FLOP is required to correct the wrong matrix element. These extra $nb + 1$ operations are negligible considering the total number of FLOPs is $(1/3)N^3$ in the Cholesky factorization.

Compared with the previous two application level mechanisms, this task level mechanism has higher overheads in fault-free case because of the cost of maintaining checksums. At the contrary, the additional cost to recover from failures is very small in task level mechanism since it does not require task re-execution.

4.5.5 Overhead of Detection Mechanism

For the first two application level mechanisms we use ABFT based method to provide accurate and effective failure detectors. Here we investigate the practical detection overhead in Cholesky factorization introduced ABFT based failure detector. Figure 4.16 presents the performance of a Cholesky factorization with the correcting sub-DAG mechanism on 60k matrix using 100 nodes, and highlights the cost and overhead of using ABFT of adding a single checksum to each matrix tile to implement the soft error detection mechanism. Detection overhead includes extra FLOPs spent in maintaining the checksum and validating results. This computational cost is paid on each task, regardless if it is a task of the original DAG, or a task of the correcting sub-DAG. The results validate that if ABFT detector is enabled, the overhead cost can increase up to 6%.

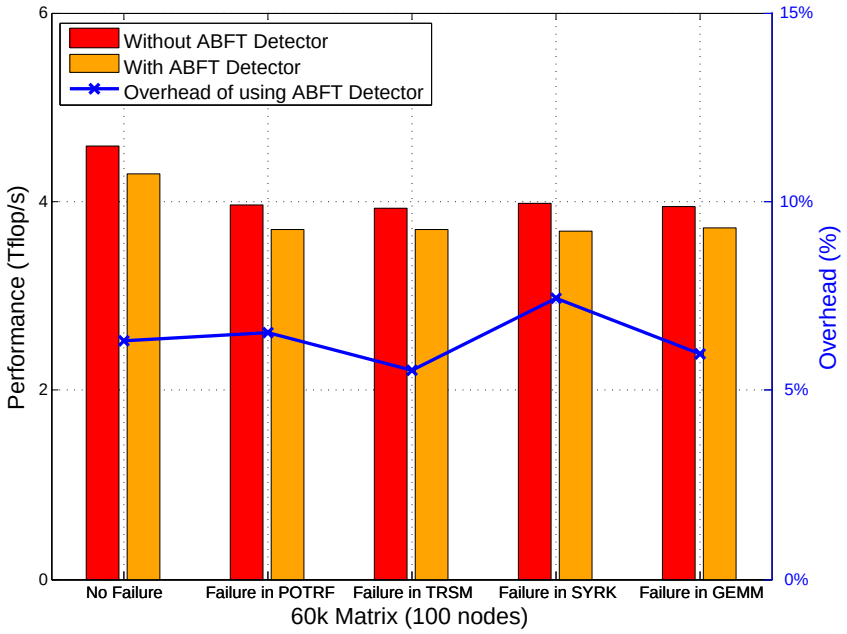

Figure 4.16: Performance and overhead of using ABFT as a detection mechanism for the correcting sub-DAG approach without failures and with one failure.

4.5.6 Performance of Fault Tolerant Layer in PaRSEC

In this experiment, we investigate the protection and recovery overhead from fault tolerant layer in PaRSEC by using QR factorization. Failure is injected to a **TSMQR** task when the factorization goes to the middle column of the matrix. Figure 4.17 shows the performance of QR factorization running on PaRSEC when fault tolerant layer is enabled, comparing with fault tolerant layer is disabled. Here we set data logging interval to 10. The performance of one failure case and failure-free case are very close as data logging interval is small. Comparing with the performance of QR factorization running on PaRSEC without resilient support, the overhead from one failure case and failure free case fluctuate around 2%, which is introduced from the cost of logging intermediary dataflow and is very small.

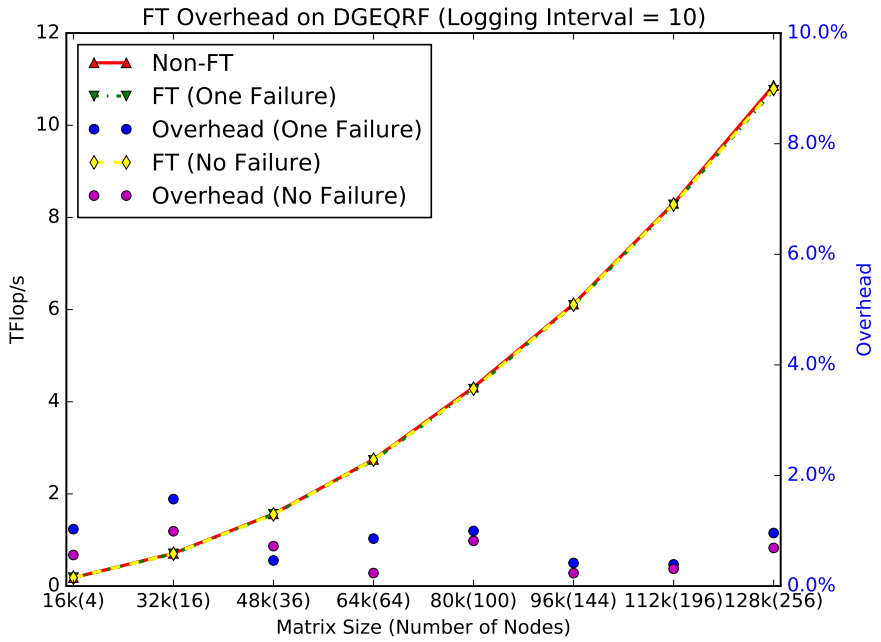

Figure 4.17: Overhead of Fault Tolerant Layer in PaRSEC on QR Factorization.

4.6 Conclusion

This chapter proposes three soft error resilient mechanisms designed for a dynamic task-based runtime. The proposed extensions provide resilience at two different levels of granularity: coarse granularity, automatic solutions at the application level and fine granularity, algorithm-based solutions at the task level. At the application level, a correcting sub-DAG mechanism is used to recover from failures by re-executing minimum number of tasks from beginning to retrieve lost information. A composite mechanism combining sub-DAG with data logging saves intermediary dataflow between tasks during the execution to reduce the amount of necessary re-executions. These two application-level mechanisms are generic and can be integrated into any task-based dynamic runtime, providing automatic resilient support for applications running on it. As task-based approaches decompose the application into smaller and less complicated tasks, it is feasible to take advantage of the intrinsic algorithm properties of tasks to provide validators allowing to detect, and possibly

recover, from soft errors. Additionally, a soft error detector based on ABFT technique is proposed to provide detection ability for any DAG-based application exhibiting ABFT properties (as described by Huang and Abraham Huang and Abraham (1984)), that can successfully complement a hardware-level failure detector. We also present how the generic data logging mechanism is merged into PaRSEC runtime to provide automatic resilience for non fault tolerant applications over PaRSEC. Detailed experiments have been tested on Titan supercomputer at ORNL, and experimental results validate the proposed fault tolerant mechanisms and highlight the low overhead of the current implementation in PaRSEC framework.

Chapter 5

Hard Error Resilient Design for a Task-based Runtime

5.1 Introduction

While many types of failures can strike a distributed memory cluster Schroeder and Gibson (2007), the focus of the work in this chapter is on the most common case: the hard error, that is, the fail-stop model. In this model, failure is in the form of node outage. The failed cluster nodes stop working and the corresponding data is lost. When a hard error happens, the application is interrupted due to lost of data and computing resource. A hard error could occur at any moment and affect any parts of the application's data. We introduce two generic approaches, which augment the data logging mechanism for soft error in Section 4.3 to adapt hard-error environment. To be more specific, we propose non-volatile storage approach and remote data logging approach to protect intermediary dataflow and final data for DAG-based applications.

The rest of the chapter is organized as follows: Section 5.2 introduces the impact of hard errors on distributed memory systems for DAG-based applications. Section 5.3 presents the two mechanisms to protect critical data against hard errors, including

non-volatile storage mechanism and remote data logging mechanism. Section 5.4 concludes this chapter.

5.2 Problem Statement

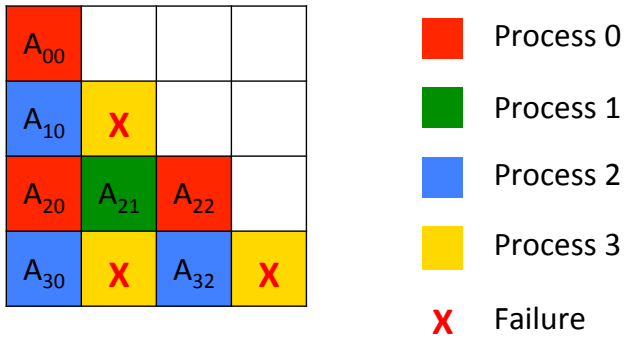

Figure 5.1: Global view of the matrix when a process fails.

In this chapter, we continue to use Cholesky factorization as a case study to illustrate our design as Chapter 4. Here, we consider hard error in the form of process failure. When a process fails in the process grid, the data resident on that process will be all gone. Figure 5.1 shows the status of the matrix when a hard error strikes. The original matrix layout is 2D block cyclic distribution and a hard error strike process 3 during execution. All the matrix tiles resident on process 3 are gone. Figure 5.2 shows the corresponding stats in the DAG. Multiple tasks are failed when process 3 fails. These tasks include: (1) completed tasks generating final data in the matrix; (2) running tasks generating current intermediary dataflows in the DAG; (3) future tasks on the failed node. The impact of a hard error is more complicated to handle compared with a soft error, as there is more dataflow corruption after a hard error and failures are propagated to more tasks. For example in figure **??**, failure is propagated to all successors requiring data on process 3.

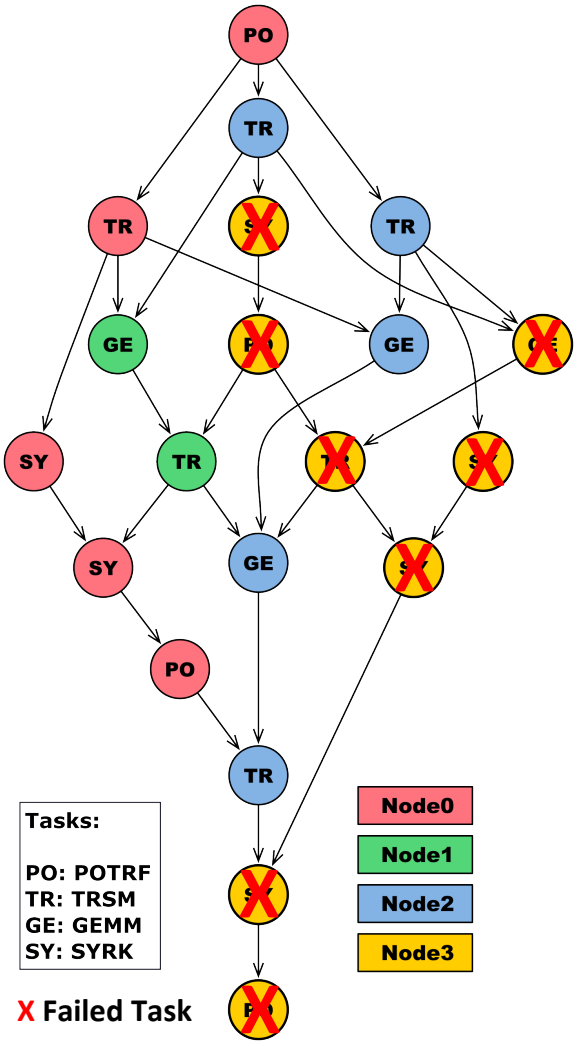

Figure 5.2: DAG of the Cholesky factorization of a 4×4 tile matrix on a 2×2 process grid, and a possible scenario of a hard error happens on process 3.

5.3 Design of Hard Error Resilience in PaRSEC

In Chapter 4, data logging mechanism has been presented as a generic and low-overhead scheme to recover soft errors in DAG-based applications. The idea of data logging mechanism is based on reducing the size of re-executing DAG for a failed task by reserving intermediary dataflow during execution. After a hard error occurs, the status of a DAG-based application can be viewed as an extended one after a soft error happens, including:

1. Multiple on-going tasks are failed when a hard error happens.

2. Final result of the application on the failed process which may serve as Read input for other future tasks, is also unavailable.

In this section, we present two mechanisms, that is, non-volatile storage mechanism and remote data logging mechanism, to extend data logging method to protect against hard errors.

5.3.1 Non-volatile Storage Mechanism

In the form of process failure, all the data resident on the failed process is lost, including ongoing dataflow and reserved dataflow. The first mechanism to augment data-logging mechanism is utilizing non-volatile storage. Non-volatile storage is a type of computer storage that can retrieve stored information even after having been power cycled (turned off and back on). Non-volatile storage devices include read-only memory, flash memory, ferroelectric RAM, most types of magnetic computer storage devices (e.g. hard disk drives, floppy disks, and magnetic tape), optical discs, and early computer storage methods such as paper tape and punched cards Wikipedia (2017b). Non-volatile storage can be served as secondary storage for reserved data in a hard-error environment. The form of main memory on today's computer system is random access memory (RAM), which is volatile, implying that when a compute node is crashed, any information saved in main memory is lost. By combining non-volatile storage and main memory together, intermediary dataflow as well as final data will be saved in both locations. Whenever a hard error happens, failed process would retrieve saved data from non-volatile storage, while other non-failed processes would retrieve saved data from main memory, and collaborate together to rebuild the sub-DAGs for recovery. Considering performance, two types of non-volatile storage can be utilized in our design:

1. **Solid-state drive (SSD)**: A SSD is a solid-state storage device that uses integrated circuit assemblies as memory to store data persistently Wikipedia

(2017c). Compared with hard disk drives, SSDs are typically more resistant to physical shock, run silently, have lower access time, and lower latency.

2. **Non-volatile random-access memory (NVRAM)**: NVRAM is random-access memory that retains its information when power is turned off (non-volatile) Wikipedia (2017d). This feature is different with existing dynamic random-access memory (DRAM) and static random-access memory (SRAM), which are only able to save data when power is on. Flash memory is the most popular NVRAM memory on market today.

Nowadays, building systems equipped with NVRAM remains costly. Due to its high cost, NVRAM is usually considered as compelling storage technologies for future supercomputers. For example, the next generation supercomputer at Oak Ridge Leadership Computing Facility (OLCF), named SUMMIT, will arrive in 2017 and be ready for users in 2018. In SUMMIT, Each node will have over half a terabyte of coherent memory addressable by all CPUs and GPUs, plus an additional 800 gigabytes of NVRAM Hemsoth (2015). Here we propose a design of utilizing non-volatile storage to save necessary information for a dynamic task-based runtime. We implement our design in machines equipped with local SSDs, and it is applicable for future machines equipped with NVRAM.

Reviewing the implementation of data logging mechanism in Chapter 4, every compute core saves necessary output dataflow to main memory after task completion. After using non-volatile storage as a secondary destination to reserve dataflow and final result, data movement toward non-volatile storage can be implemented in following two ways:

1. Every compute core stores its own data to non-volatile storage.

2. Assigning a separate core to handle data movement to non-volatile storage.

Considering the imbalance between slow sequential I/O operations and fast parallel multi-core task executions, using method 1 would force task execution on

every compute core to halt until I/O is available. In our design, we use method 2 as it doesn't require explicit wait for I/O. Figure 5.3 illustrates how data movement to non-volatile storage is implemented in PaRSEC. In this example, on process 3, intermediary dataflow is stored every 2 updates (blue flow), and final result (red flow) is also backed up in SSD. Whenever a compute core completes a task and reaches saving point, this compute core pushes the target data into the storage queue (using SSD in our implementation) and continues to execute next available task. A separate core handles I/O with SSD, it keeps moving data out of the storage queue and saving in SSD. Data movement to SSD is asynchronous in this case, and there is no idle gap for any compute core to wait for I/O to be available.

We investigate the overhead of using SSD to reserve necessary data against hard error by using Cholesky factorization. As every node only backs up data in local SSD, the overhead is dominated by local SSD access. The configuration of the experiment platform is: 2 Intel E5520 CPUs running at 2.27 GHz and 1 SSD connected by SATA interface. The Read/Write bandwidth of this SSD is about 800 MB/s. Cholesky factorization with matrix size 6000 and tile size 200 is tested, performance is compared with standard PaRSEC without fault tolerant features. Figure 5.4 shows the performance of using non-volatile storage mechanism with different saving interval. The Cholesky factorization using standard PaRSEC runs at about 60 GFlop/s, while performance of using SSD is under 20 GFlop/s. Remember the result of in Section 4.5, data logging mechanism only introduces close to 0 overhead as the cost of accessing main memory is negligible. Here, the overhead is dominated by the cost of saving results into SSD. Bandwidth of current generations of main memory (Random-access memory) is listed in table 5.1. Comparing with the SSD bandwidth on the experiment platform, the transfer speed of SSD is $10 - 20$ times slower than main memory, meaning that the saving overhead of using SSD is $10 - 20$ times larger than using main memory. Also, data movement to SSD is sequential, and in order to protect again hard errors, more data is saved in this case. These two factors also contribute the overhead of using SSD. The overhead of this non-volatile

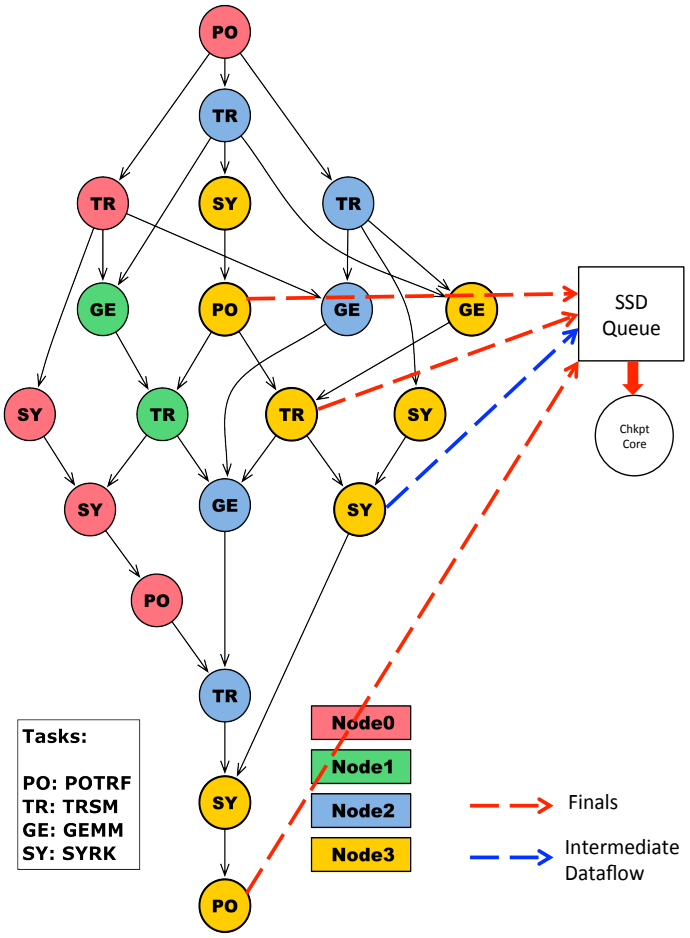

Figure 5.3: DAG of the Cholesky factorization of a 4x4 tile matrix on a 2x2 process grid, using non-volatile storage (SSD) mechanism on process 3.

mechanism mostly depends on the transfer bandwidth of underlying storage devices. In future if faster storage devices such as NVRAMs are available, this mechanism is adaptable and provides lower overhead. Furthermore, the red bar in the result shows the performance of using saving interval of 30. As the size of matrix is 6000 and tile size is 200, the total number of factorization steps is 30. This gives us a measurement of overhead that if SSD is only used to protect input and output of an application.

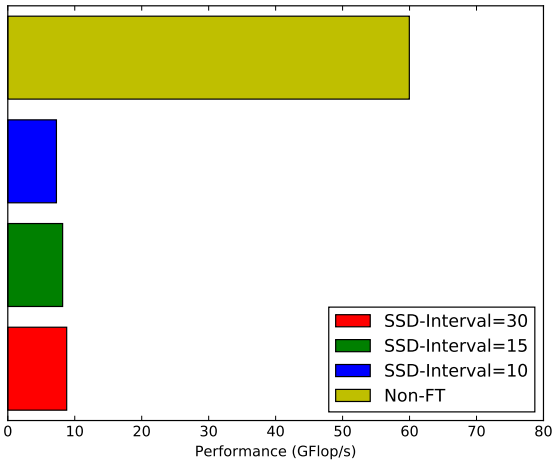

Figure 5.4: Performance of Cholesky Factorization with Non-volatile Storage Mechanism.

Table 5.1: Bandwidth of Current Generations of RAM

RAM Type	DDR3-1066	DDR3-1333	DDR3-1600	DDR4-2133	DDR4-2400
Bandwidth	8533 MB/s	10667 MB/s	12800 MB/s	17066 MB/s	19200 MB/s

5.3.2 Remote Data Logging Mechanism

The idea of non-volatile storage mechanism presented in Section 5.3.1 is to put reserved dataflow and final result into devices that can retrieve information even after having been power cycled (turned off and back on). The overhead of such mechanism is limited by the Read/Write bandwidth of storage device. Here we present another mechanism to extend data logging method in a hard error-prone environment by utilizing main memory on a remote node. As shown in figure 5.5, whenever a dataflow in the DAG reaches a logging point or becomes final result of the application, corresponding compute node generating this dataflow sends a copy of it to a remote buddy node, and retrieves the copy back from the buddy node if a hard error occurs later. As using remote buddy node to backup dataflow increases the amount of data transferred in the network, the selection strategy of buddy node can impact the performance of protected applications. An optimal selection of a buddy node involves the application's algorithmic feature and hardware performance, which

is not discussed in this work. Here, we use a simplified strategy to set the buddy node as original node rank +1. Figure 5.6 describes how this remote data logging mechanism is working on node 3 of Cholesky factorization of a 4×4 tile matrix on a 2×2 process grid. Similar as non-volatile storage mechanism in section 5.3.1, intermediary dataflow is stored every 2 updates (blue flow), and final result (red flow) is also backed up in remote buddy node. Whenever a compute core completes a task and reaches saving point, this compute core creates a communication task of sending this target data, submits this communication task to the communication engine of PaRSEC and continues to execute next available task. Communication engine on each node handles data interaction with remote buddy node by using asynchronous send and receive activities.

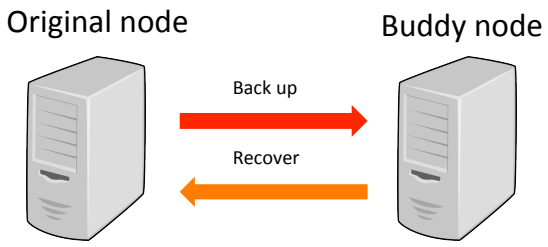

Figure 5.5: Remote Data Logging Mechanism.

In the direction of implementing remote data logging mechanism in the runtime level of PaRSEC, the termination of computer node in a distributed-memory environment is different in the context of fault tolerant execution compared with the one in original task graph execution context. In original PaRSEC, tasks are assigned to different compute node based on data locality. In order to determine when the computation has completed, every node actively detect whether all assigned tasks are done and no more data send request to other nodes is pending. Based on such a termination detection mechanism, every node will close as soon as possible. As shown in figure 5.7, remote data logging mechanism could not work under this termination

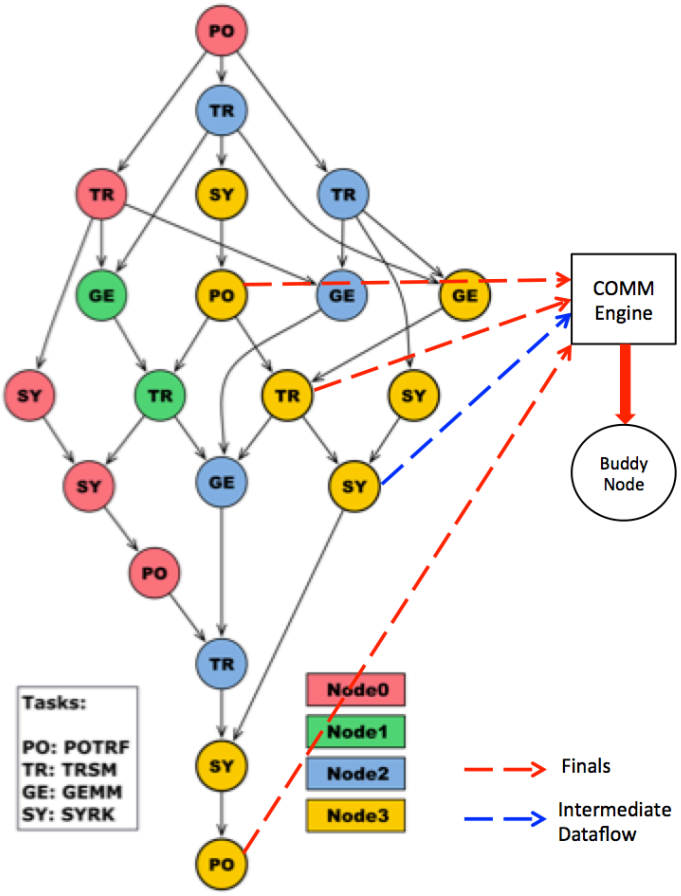

Figure 5.6: DAG of the Cholesky factorization of a 4x4 tile matrix on a 2x2 process grid, using remote data logging mechanism on process 3.

scheme because it is not guaranteed that buddy node will terminate later than original node, leaving data backup and recovery unreliable.

In order to enable remote data logging mechanism to work in PaRSEC, the node termination agreement in PaRSEC is modified. All compute nodes must agree that all of them are idle and no more work is available. To implement such a collective way of termination, many schemes are possible, ranging from centralized schemes using shared counters and termination detection servers to fully distributed schemes Dinan (2010). We have implemented a collective version of termination for PaRSEC in fault tolerant context using shared counters. On every compute node, a shared counter is

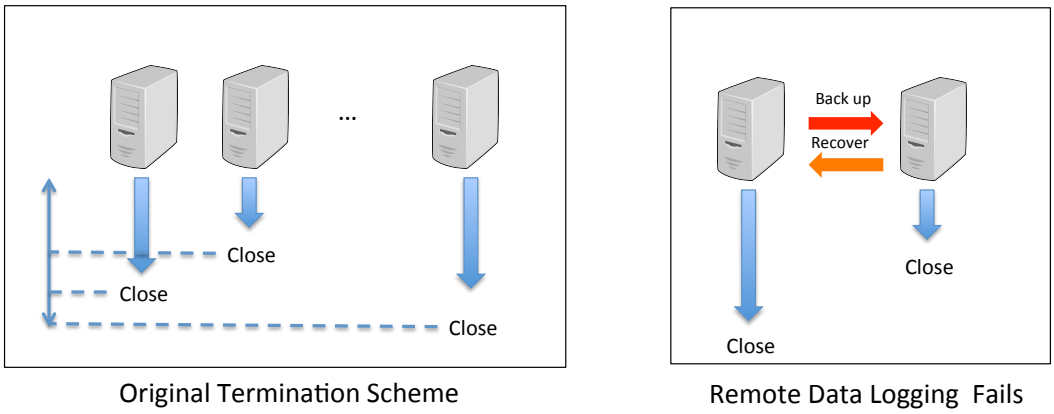

Figure 5.7: Remote data logging mechanism is unreliable in original PaRSEC termination scheme.

initialized as the number of total compute nodes in current process grid. Whenever a compute node finishes local tasks, it will broadcast its completion information to all other nodes to update shared counter. After all nodes reach to idle status, the shared counter is set to 0 and all nodes are ready to terminate. Figure 5.8 describes the implementation of this modified termination scheme, under such scheme, a buddy node will keep running after local tasks are completed, preparing for receiving logged data and possible recovery.

The overhead of extra messages added to communication level in this remote data logging mechanism is essential, as it delays the transfer of original dataflows in the DAG and activation of available tasks. As we take a simplified decision for setting remote buddy node as original node's rank +1, it is possible that the buddy node is one of original node's direct successors due to application's algorithmic feature. For example, in Cholesky factorization, a **POTRF** task generates final output for current matrix tile and broadcasts its result to all the matrix tiles below it doing **TRSM**

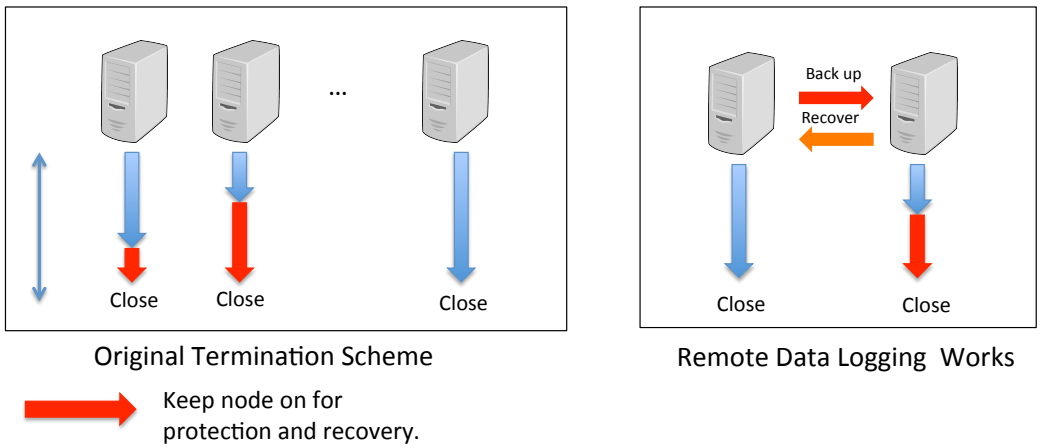

Figure 5.8: Modified PaRSEC termination scheme for remote data logging mechanism.

tasks. Figure 5.9 demonstrates an example that the remote buddy node happens to be one of the **POTRF** task's successors. In this case, a duplicated data transfer is committed to communication engine and it should be avoided in optimal scenario for reducing extra communication cost. In worst case, if an application needs to log data remotely for every task and remote buddy node belongs to one of the successors in every step, the overhead of extra messages added to communication engine could be 100%.

A dynamic scheme for selecting remote buddy node has been implemented in PaRSEC. When a task reaches a logging point, runtime firstly checks whether there is any remote node exists as one of its successors. If this task doesn't have successors or all its successors are local tasks, then its output is logged remotely to original rank +1. Otherwise, the runtime chooses the first remote successor as the buddy node automatically. Also, on the receiver's side, it is important to tell whether an incoming dataflow needs to be logged or not. A data logging message has been merged

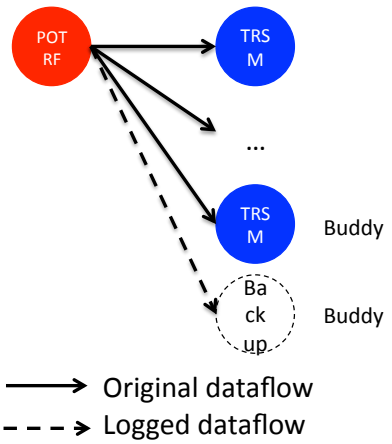

Figure 5.9: An example of duplicated data transfer in Cholesky factorization using remote data logging mechanism.

into original dataflow message, and this logging message tells the receiving node to log the incoming dataflow or not. As the position of remote buddy node is decided dynamically during execution, it is important to retrieve the same information back during recovery. In Section 4.3, the PTG feature has been utilized to dynamically expand the DAG to any direction as wanted. Here during recovery, we take advantage of PTG representation of tasks in PaRSEC again, expand the DAG one level deeper for failed tasks on crashed node to find correct buddy node. Also, the buddy node is failed node's rank +1 if the failed task has no remote successors.

The overhead of using remote data logging mechanism has been investigated. We use Cholesky factorization as the application and conduct experiments on a 16 nodes cluster at University of Tennessee. Every node has 2 Intel E5520 CPUs running at 2.27GHz, and they are connected with Infiniband-20G network. Weak scalability experiments are carried to evaluate the overhead of this remote data logging mechanism. We set tile size to be 200 and the matrix input size for single-node experiments to 6000 for Cholesky factorization, and scale it with $6000\sqrt{P}$ where P

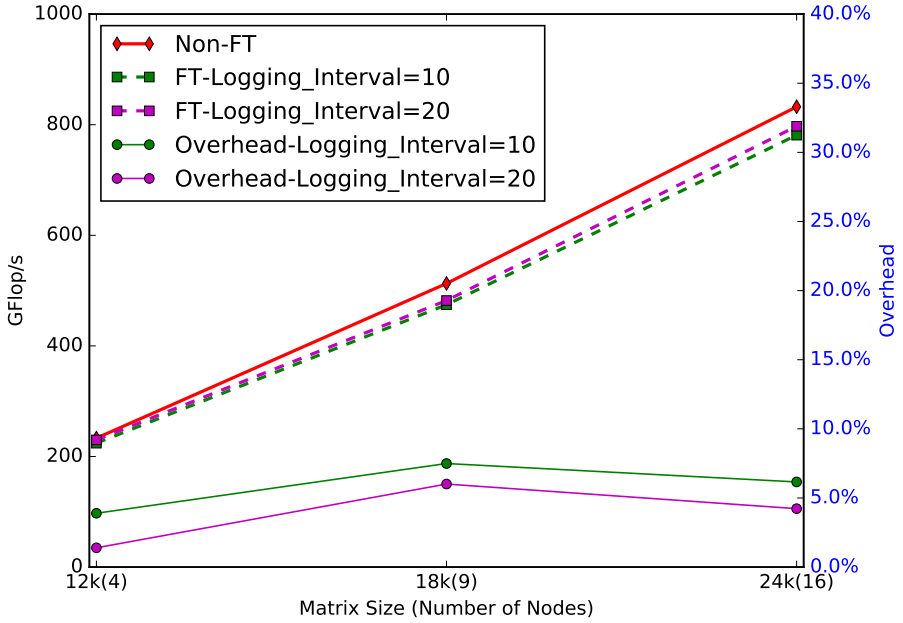

Figure 5.10: Overhead of remote data logging mechanism in Cholesky Factorization using infiniband-20G.

is the number of nodes. Figure 5.10 shows the overhead of this mechanism. Data logging interval is set to 10 and 20 separately, and performance is compared with non fault tolerant version of PaRSEC. From the results, we can see that the protection overhead for both data logging intervals are under 10%, this means that the cost of using remote data logging mechanism to protect application is acceptable and it is feasible to continue to integrate with fault tolerant MPI library (e.g., ULFM) to design a recovery mechanism. Another observation from the results is that the overhead of logging data every 10 updates and logging data every 20 updates is very close. As factorization goes on, the actual matrix size where tasks are updating is shrinking, that means the amount of data that needs to be logged remotely is decreasing during execution. Also, the dynamic scheme of deciding remote buddy node reduces extra data added to communication engine. Even using 10 interval the data will be logged more frequently, the total amount of data added to communication engine is close to using 20 interval.

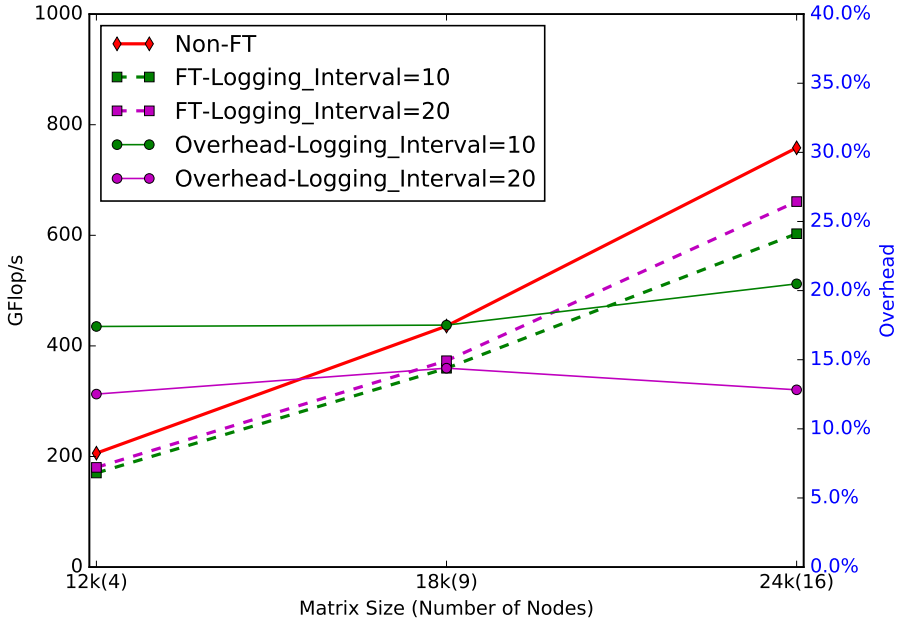

Figure 5.11: Overhead of remote data logging mechanism in Cholesky Factorization using infiniband-10G.

As we know by using remote data logging mechanism, extra massages added to communication engine would delay the transfer of original dataflow and activation of available tasks. The bandwidth of network could also impact the performance of this mechanism. We conduct another set of experiments on another 16 nodes cluster at University of Tennessee with slower network interconnection. Here, every node is equipped with 2 Intel Westmere-EP CPUs running at 2.13GHz, and they are connected with Infiniband-10G network. Network bandwidth is only half compared with previous cluster and we conduct the same weak scaling experiments for Cholesky factorization. Figure 5.11 shows the results and we can see can the overhead of using data logging interval 10 and interval 20 is around 15% to 20%. Compared with the overhead on faster network interconnection cluster, extra messages related to remote logging take more time to transfer as the network bandwidth is smaller here, resulting in higher delay in original DAG execution.

For today's supercomputers and future supercomputers, interconnect is high-bandwidth. For example, Titan supercomputer at Oak Ridge National Laboratory is equipped with Cray Gemini interconnect which has 20 GB/s bandwidth and next generation supercomputer Summit at Oak Ridge National Laboratory will use dual-rail Mellanox EDR InfiniBand interconnects, providing 23 GB/s data sharing between the nodes Hemsoth (2015). These high-bandwidth interconnection implies that remote data logging mechanism is an effective solution to protect DAG-based applications against hard errors on supercomputers.

5.4 Conclusion

This chapter describes two feasible mechanisms designed for a dynamic task-based runtime, for handling hard errors. The proposed extensions are implemented in PaRSEC and ensure resilience by utilizing non-volatile storage and remote node for protection. A non-volatile storage mechanism is proposed to extend data logging mechanism in previous soft error related work to protect against hard errors, by storing final result and intermediary dataflow into secondary storage. We also present a remote data logging mechanism to protect data against hard errors by relying on compute nodes cooperatively. Critical data will be backed up both in local memory and in remote buddy node, which makes the rebuild of recovery DAG available when a hard error happens. We conduct experiments on 16 nodes clusters to measure the cost of these two mechanisms, and proves the feasibility of this two mechanisms to protect against hard errors toward exascale computing.

Chapter 6

Conclusions and Future Work

6.1 Conclusion

In this dissertation, we have identified two critical issues toward future exascale computing: 1) the gap between hardware peak performance and practical performance of developing applications on complicated programming environment of today's and future supercomputers; 2) lacking efficient resilient support for task-based runtime systems while future exascale systems will be subject to failures much more frequently than current petascale systems. To address these existing issues, we designed a unified programming model to utilize a dynamic task-based runtime to develop high performance dense linear algebra applications on heterogeneous platforms and distributed-memory platforms. Moreover, fault tolerant mechanisms for both soft and hard errors are designed for a task-based runtime and implemented in PaRSEC system.

Toward alleviating the disparity between hardware peak performance and application performance, the unified programming model takes advantage of a lightweight task-based runtime to manage the resource-specific workload, and to control the dataflow and parallel execution of tasks. Under unified algorithmic development, tasks are abstracted across different underlying heterogeneous resources, including

multi-core CPUs, GPUs and Intel Xeon Phi coprocessors. Several optimization schemes are presented to improve performance by increasing priorities of critical tasks and splitting appropriate workload size for different devices. Cholesky factorization is implemented in this approach in both shared-memory and distributed-memory platforms, demonstrating the effectiveness of this unified design and proving its full adaption to a wide range of accelerators.

In addition, to solve the emerging resilient challenge as the scale of modern computing systems grows, fault tolerant mechanisms are designed for a task-based runtime to protect applications against both soft and hard errors. For soft errors, three fault tolerant mechanisms are proposed to provide resilience at two levels of granularity: At the application level, a correcting sub-DAG mechanism is designed to recover from data corruption by re-executing minimum number of tasks from beginning to regenerate correct data. A composite mechanism combining sub-DAG with data logging saves necessary intermediary dataflows during the execution to reduce the number of re-executed tasks in recovery. At the task level, ABFT technique is applied to take advantage of the intrinsic algorithmic properties of tasks in DAG to provide validators allowing to detect and recover from soft errors. By applying ABFT mechanism, application is decomposed into smaller and less complicated tasks with self resilient features. As application level data logging mechanism is generic and adaptable to any task graph based application, a fault tolerant layer is implemented in PaRSEC system to provide automatic resilience for non fault tolerant applications running over PaRSEC. Experiments on large scale cluster have confirmed the proposed mechanisms all meet the design criteria in terms of error correction and performance overhead. This altogether offers very promising alternatives to the currently widely used checkpointing/restart method with much less overhead.

For hard errors, generic data logging mechanism is extended in two ways, by utilizing local reliable storage and remote compute node's memory, to guarantee resilience. A non-volatile storage mechanism is proposed to store final result and intermediary dataflows into secondary storage. Our implementation in PaRSEC is

adaptable to different storage devices, ranging from SSD to NVRAM. A remote data logging mechanism is also designed to protect applications against hard errors. Final result and intermediary dataflows are saved not only in local memory but also in a remote buddy node's memory. When a hard error happens, the task execution on the failed node is rebuilt by combining saved information on corresponding buddy node and other non-failed nodes.

6.2 Future Work

With the quick development of accelerators in performance, more and more domain scientific applications have been re-designed to exploit accelerator's massive parallelism feature. The utilization of task-based runtime in this work addresses challenges in developing high performance dense linear algebra applications. Moving the design of the unified framework using task-based runtime toward supporting the development of other popular research applications, such as Deep Neural Networks, Computational Fluid Dynamics, will be addressed as part of the future work.

In addition, this work addresses data protection against hard errors. Implementation of hard-error recovery for a task-based runtime requires resilient support from underlying MPI library. ULFM Herault et al. (2015) provides new interfaces for MPI that enables distributed programs using MPI to restore message passing functionality affected by hard errors. An interesting area is to integrate fault tolerant design with ULFM, to meet the goal of providing high reliability against hard errors.

Another interesting area is that every fault tolerant mechanism deals with a single type of error, either soft error or hard error. In a failure prone context, there is no guarantee that only one type of error will occur. Future work will include designing integrated protection against both hard errors and soft errors in task-based runtimes, providing a robust resilient solution for incoming exascale computing systems.

Bibliography

Agrawal, K., Leiserson, C. E., and Sukha, J. (2010). Executing task graphs using work-stealing. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pages 1–12. 19

Agullo, E., Demmel, J., Dongarra, J., Hadri, B., Kurzak, J., Langou, J., Ltaief, H., Luszczek, P., and Tomov, S. (2009). Numerical linear algebra on emerging architectures: The plasma and magma projects. *Journal of Physics: Conference Series*, 180(1):012037. 12, 27

Agullo, E., Dongarra, J., Nath, R., and Tomov, S. (2011). A fully empirical autotuned dense qr factorization for multicore architectures. In *Proceedings of the 17th International Conference on Parallel Processing - Volume Part II*, Euro-Par'11, pages 194–205, Berlin, Heidelberg. Springer-Verlag. 13

AMD (2015). clblas libraries: clblas 2.2. https://github.com/clMathLibraries/clBLAS. 12

Anderson, E., Bai, Z., Bischof, C., Blackford, L. S., Demmel, J., Dongarra, J. J., Du Croz, J., Hammarling, S., Greenbaum, A., McKenney, A., and Sorensen, D. (1999). *LAPACK Users' Guide (Third Ed.)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA. 24

Augonnet, C., Thibault, S., Namyst, R., and Wacrenier, P.-A. (2009). StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, Euro-Par '09, pages 863–874, Berlin, Heidelberg. Springer-Verlag. 13

Avizienis, A., Gilley, G. C., Mathur, F. P., Rennels, D. A., Rohr, J. A., and Rubin, D. K. (1971). The star (self-testing and repairing) computer: An investigation of the theory and practice of fault-tolerant computer design. *IEEE Trans. Comput.*, 20(11):1312–1321. 14

Ayguadé, E., Badia, R. M., Igual, F. D., Labarta, J., Mayo, R., and Quintana-Ortí, E. S. (2009). An extension of the starss programming model for platforms with multiple gpus. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, Euro-Par '09, pages 851–862, Berlin, Heidelberg. Springer-Verlag. 12

Besseron, X., Jafar, S., Gautier, T., and Roch, J.-L. (2006). CCK: An Improved Coordinated Checkpoint/Rollback Protocol for Dataflow Applications in KAAPI. In *ICTTA'06*. 20

Blackford, L. S., Choi, J., Cleary, A., Petitet, A., Whaley, R. C., Demmel, J., Dhillon, I., Stanley, K., Dongarra, J., Hammarling, S., Henry, G., and Walker, D. (1996). Scalapack: A portable linear algebra library for distributed memory computers - design issues and performance. In *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing*, Supercomputing '96, Washington, DC, USA. IEEE Computer Society. 18, 43

Board, T. O. A. R. (2013). Openmp 4.0. http://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf. 13

Bosilca, G., Bouteiller, A., Danalis, A., Faverge, M., Herault, T., and Dongarra, J. J. (2013). Parsec: Exploiting heterogeneity to enhance scalability. *Computing in Science Engineering*, 15(6):36–45. 3, 56

Bosilca, G., Bouteiller, A., Danalis, A., Herault, T., Lemarinier, P., and Dongarra, J. (2011). DAGuE: A Generic Distributed DAG Engine for High Performance Computing. In *IPDPS Workshop*, pages 1151–1158. 9

Buttari, A., Langou, J., Kurzak, J., and Dongarra, J. (2009). A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Comp.*, 35(1):38–53. 51

Cao, C., Dongarra, J., Du, P., Gates, M., Luszczek, P., and Tomov, S. (2014). clmagma: High performance dense linear algebra with opencl. In *Proceedings of the International Workshop on OpenCL 2013 & 2014*, IWOCL '14, pages 1:1–1:9, New York, NY, USA. ACM. 4, 22, 27

Cao, C., Herault, T., Bosilca, G., and Dongarra, J. (2015). Design for a soft error resilient dynamic task-based runtime. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 765–774. 50

Cappello, F. (2009). Fault tolerance in petascale/ exascale systems: Current knowledge, challenges and research opportunities. *The International Journal of High Performance Computing Applications*, 23(3):212–226. 2

Cappello, F., Geist, A., Gropp, B., Kale, L., Kramer, B., and Snir, M. (2009). Toward exascale resilience. *Int. J. High Perform. Comput. Appl.*, 23(4):374–388. 10

Chan, E., Quintana-Orti, E. S., Quintana-Orti, G., and van de Geijn, R. (2007). Supermatrix out-of-order scheduling of matrix operations for smp and multi-core architectures. In *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '07, pages 116–125, New York, NY, USA. ACM. 13

Chandy, K. M. and Lamport, L. (1985). Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75. 16

Choi, J., Dongarra, J., Ostrouchov, S., Petitet, A., Walker, D., and Whaley, R. C. (1996). *A proposal for a set of parallel basic linear algebra subprograms*, pages 107–114. Springer Berlin Heidelberg, Berlin, Heidelberg. 42

Cosnard, M., Jeannot, E., and Yang, T. (1999). Slc: Symbolic scheduling for executing parameterized task graphs on multiprocessors. In *Proceedings of the 1999 International Conference on Parallel Processing*, pages 413–421. 9

Daly, J. T. (2006). A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Gener. Comput. Syst.*, 22(3):303–312. 60

Dinan, J. (2010). *Scalable Task Parallel Programming in the Partitioned Global Address Space*. PhD thesis, The Ohio State University. 88

Du, P., Bouteiller, A., Bosilca, G., Herault, T., and Dongarra, J. (2012). Algorithm-Based Fault Tolerance for dense matrix factorizations. In *PPoPP'12*, pages 225–234. ACM. 18, 63

Du, P., Luszczek, P., and Dongarra, J. (2011a). High performance dense linear system solver with soft error resilience. In *IEEE Intl. Conf. on Cluster Computing*, pages 272–280. 18, 55

Du, P., Luszczek, P., Tomov, S., and Dongarra, J. (2011b). Soft error resilient qr factorization for hybrid system with gpgpu. In *Proceedings of the Second Workshop on Scalable Algorithms for Large-scale Systems*, ScalA '11, pages 11–14, New York, NY, USA. ACM. 18

Elnozahy, E. N. M., Alvisi, L., Wang, Y.-M., and Johnson, D. B. (2002). A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408. 16

Fechner, B., Honig, U., Keller, J., and Schiffmann, W. (2008). Fault-tolerant static scheduling for grids. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–6. 19

Gautier, T., Besseron, X., and Pigeon, L. (2007). Kaapi: A thread scheduling runtime system for data flow computations on cluster of multi-processors. In *Proceedings of the 2007 International Workshop on Parallel Symbolic Computation*, PASCO '07, pages 15–23, New York, NY, USA. ACM. 20

Geist, A. (2016). How to kill a supercomputer: Dirty power, cosmic rays, and bad solder. *IEEE Spectrum*. 10

Haidar, A., Ltaief, H., YarKhan, A., and Dongarra, J. (2011). Analysis of dynamically scheduled tile algorithms for dense linear algebra on multicore architectures. *Concurr. Comput. : Pract. Exper.*, 24(3):305–321. 9, 10

Hemsoth, N. (2015). Programming challenges on the road to summits peak. *The Next Platform*. 83, 94

Herault, T., Bouteiller, A., Bosilca, G., Gamell, M., Teranishi, K., Parashar, M., and Dongarra, J. (2015). Practical scalable consensus for pseudo-synchronous distributed systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 31:1–31:12, New York, NY, USA. ACM. 18, 97

Howlett, J. and Rota, G. C. (1980). *History of Computing in the Twentieth Century*. Academic Press, Inc., Orlando, FL, USA. 14

Huang, K.-H. and Abraham, J. A. (1984). Algorithm-Based Fault Tolerance for matrix operations. *IEEE Trans. Comput.*, 33(6):518–528. 17, 63, 78

Intel (2016). Intel math kernel library. https://software.intel.com/en-us/intel-mkl/. 12

Jia, Y. (2015). *Algorithm-Based Fault Tolerance for Two-Sided Dense Matrix Factorizations*. PhD thesis, University of Tennessee. 11

Jia, Y., Bosilca, G., Luszczek, P., and Dongarra, J. J. (2013). Parallel reduction to hessenberg form with algorithm-based fault tolerance. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 88:1–88:11, New York, NY, USA. ACM. 18

Jia, Y., Luszczek, P., and Dongarra, J. (2016). Hessenberg reduction with transient error resilience on gpu-based hybrid architectures. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 653–662. 18

Johnson, T. (1993). A concurrent dynamic task graph. In *Parallel Processing, 1993. ICPP 1993. International Conference on*, volume 2, pages 223–230. 19

Khronos OpenCL Working Group (2009). The opencl specification, version: 1.0 document revision: 48. 27, 28, 29

Kurt, M. C., Krishnamoorthy, S., Agrawal, K., and Agrawal, G. (2014). Fault-tolerant dynamic task graph scheduling. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 719–730, Piscataway, NJ, USA. IEEE Press. 20

Lyons, R. E. and Vanderkulk, W. (1962). The use of triple-modular redundancy to improve computer reliability. *IBM J. Res. Dev.*, 6(2):200–209. 14

Mittal, S. and Vetter, J. S. (2016). A survey of techniques for modeling and improving reliability of computing systems. *IEEE Transactions on Parallel and Distributed Systems*, 27(4):1226–1238. 15

Mouallem, P., Crawl, D., Altintas, I., Vouk, M., and Yildiz, U. (2010). A fault-tolerance architecture for kepler-based distributed scientific workflows. In *Proceedings of the 22Nd International Conference on Scientific and Statistical Database Management*, SSDBM'10, pages 452–460, Berlin, Heidelberg. Springer-Verlag. 19

NIVIDIA (2017). cublas libraries: cublas 8.0. https://developer.nvidia.com/cublas. 12

Qin, X. and Jiang, H. (2006). A novel fault-tolerant scheduling algorithm for precedence constrained tasks in real-time heterogeneous systems. *Parallel Comput.*, 32(5):331–356. 19

Randell, B. (1975). System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, SE-1(2):220–232. 16

Russell, D. L. (1980). State restoration in systems of communicating processes. *IEEE Trans. Softw. Eng.*, 6(2):183–194. 16

Schroeder, B. and Gibson, G. A. (2007). Understanding failures in petascale computers. *Journal of Physics: Conference Series*, 78(1):012022. 79

Song, F., Tomov, S., and Dongarra, J. (2012). Enabling and scaling matrix computations on heterogeneous multi-core and multi-gpu systems. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, pages 365–376, New York, NY, USA. ACM. 12

Strazdins, P. E. (1998). Lookahead and algorithmic blocking techniques compared for parallel matrix factorization. In *10th International Conference on Parallel and Distributed Computing and Systems, IASTED*, Las Vegas, USA. 22, 26, 41

Strom, R. and Yemini, S. (1985). Optimistic recovery in distributed systems. *ACM Trans. Comput. Syst.*, 3(3):204–226. 15, 16

Top500 (2016). http://www.top500.org. 1

Valiant, L. G. (1990). A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111. 43

Wikipedia (2017a). https://en.wikipedia.org/wiki/ECC_memory. 15

Wikipedia (2017b). https://en.wikipedia.org/wiki/Non-volatile_memory. 82

Wikipedia (2017c). https://en.wikipedia.org/wiki/Solid-state_drive. 82

Wikipedia (2017d). https://en.wikipedia.org/wiki/Non-volatile_random-access_memory. 83

Wu, P. and Chen, Z. (2014). FT-ScaLAPACK: Correcting soft errors on-line for ScaLAPACK Cholesky, QR, and LU factorization routines. In *HPDC'14*, pages 49–60. ACM. 18, 64

YarKhan, A., Kurzak, J., and Dongarra, J. (2011). Quark users' guide: Queueing and runtime for kernels. Technical report, Innovative Computing Laboratory, University of Tennessee. 9

Yoon, D. H. and Erez, M. (2009). Memory mapped ecc: Low-cost error protection for last level caches. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 116–127, New York, NY, USA. ACM. 15

Vita

Chongxiao Cao was born in Lishui, Zhejiang, China, on November 25, 1985. He received his Bachelor of Engineering in Automation and Master of Engineering in Systems Engineering from Xi'an Jiaotong University, Xi'an, Shaanxi, P.R.China in 2008 and 2011, respectively. In August 2011, he was enrolled in the University of Tennessee as a doctoral student in Computer Science. During his graduate studies, he worked as a graduate research assistant in Innovative Computing Laboratory (ICL) under the guidance of Dr. Jack Dongarra and Dr. George Bosilca. He was involved in two federally funded projects related to high performance computing: MAGMA and PaRSEC. His current research interests include high performance computing, heterogeneous computing and fault tolerance. Chongxiao Cao is expected to receive a Doctor of Philosophy degree in Computer Science in May 2017.