Fault-Tolerant Programming Models and Computing Frameworks

Dissertation

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of The Ohio State University

By

Mehmet Can Kurt, B.S., M.S.

Graduate Program in Computer Science and Engineering

The Ohio State University

2015

Dissertation Committee:

Gagan Agrawal, Advisor
Saday Sadayappan
Radu Teodorescu
© Copyright by
Mehmet Can Kurt
2015
Abstract

Fault-tolerance on parallel systems has always been a big challenge for High Performance Computing (HPC), and hence it has drawn a lot of attention of the community. This pursuit in fault-tolerant systems is now important more than ever due to the recent advances in hardware. As the emergence of first multi-core and more recently many-core machines evince, computing power is constantly being increased with more number of processing cores resulting in more parallelism. In order to satisfy this demand and to increase power of individual components, chips are manufactured with decreasing feature sizes. Another trend is power optimization efforts, since it might not be feasible to run all system resources at their peak levels all the time due to the factors such as heat dissipation and maintaining a total power budget. These trends in hardware also change the way that scientific applications are implemented. The community designs new and diverse parallel programming models to harvest the available computing power in new hardware architectures. These models provide additional support to programmers so that they can achieve scalable performance by tuning applications via additional API, specifications or annotations. Unfortunately, these changes in hardware and software also bring new challenges. For instance, increasing number of components in HPC systems results in increasing probability of failure at the same time. Trends such as decreasing feature sizes and low voltage computing cause more frequent bit flip occurrences. Lastly, when incorporated incorrectly or inaccurately, programmer specifications for performance tuning might cause potential errors...
during execution. Considering these new problems, the community foresees that Mean Time Between Failures (MTBF) rates in the future are destined to decrease significantly so that the current fault-tolerance solutions will become completely inapplicable.

In this dissertation, we introduce fault-tolerance solutions in the context of existing and new parallel programming models and query and data-analysis frameworks. Our specific solutions target the three type of failures commonly seen; \textit{fail-stop failures}, \textit{soft errors} and \textit{programmer induced errors}. With proposed solutions, we address the following key challenges. (1) Replication is a standard technique employed in big data analysis platforms to ensure the availability of underlying data in presence of fail-stop failures. How should we create and organize data replicas so that we guarantee efficient recovery by preserving load balance among remaining processing units when failures occur? (2) Programming models are expected to play a key role in overcoming the challenges in future HPC systems including resilience. Can we design a programming model that exposes the core execution state and the most critical computations in an application through a set of programming abstractions? (3) With the help of these abstractions, can such a programming model automate application-level checkpointing and reduce the amount of checkpointed state? Can we use the same knowledge to detect silent data corruptions with low overheads by executing a subset of the computations in an application redundantly? (4) For checkpoint/restart solutions, can we design recovery techniques that do not enforce any assumptions on the number of processing units that the execution is restarted with? (5) Fault-tolerance has been mostly addressed in the context of SPMD paradigm. Is it possible to design fault-tolerance solutions against soft errors in different parallel programming paradigms such as task graph execution model? (6) In addition to fail-stop failures and soft errors due to manufacturing
issues and machine defects, can we also deal with the potential failures that are induced by programmer specifications while tuning an application to improve performance?

First, we presented the design and implementation of a fault-tolerant environment for processing queries and data analysis tasks on large scientific datasets. For two common query and data analysis tasks, we first provided a framework that employs the standard data indexing techniques and achieves high-efficiency of execution when there are no failures. Then, we showed how the framework recovers from failures up to a certain number of nodes efficiently and still maintains the load balance among remaining nodes after recovery completes. We achieved these goals by developing a data replication scheme, which we refer to as subchunk or subpartition replication. Our extensive evaluation demonstrated that our replication scheme outperforms the traditional solutions.

Second, we focused on designing a parallel programming paradigm that models computations and communication in iterative scientific applications through an underlying domain and interactions among domain elements. With proper abstractions, the proposed model hides the details of inter-process communication and work partitioning (including re-partitioning in presence of heterogeneous processing cores) from users. More importantly, it captures the most critical execution state and instructions in an application through the concepts of compute-function and computation-space object. The model supports automated, yet efficient, application-level checkpointing and at the same time detects soft errors occurring in processing cores and corrupting the main application state by a low-overhead redundant execution strategy. We analyze the performance of our programming model with various scenarios both on homogeneous and heterogeneous configurations.

Next, we directed our attention to task graph execution model, which is a different parallel programming paradigm than Single Program Multiple Data (SPMD) for which most
of the existing fault-tolerance solutions in the literature has been proposed. We designed a fault-tolerant dynamic task graph scheduling algorithm that recovers corrupted data blocks and meta-data in a task graph from arbitrary number of soft errors with low time and space overheads. We provided a task re-execution algorithm which is selective in the sense that only the corrupted portion of the task graph is recovered. Furthermore, as opposed to the traditional checkpoint/restart solutions, recovery is performed in a non-collective fashion so that only the threads observing the failure participate in the recovery process, whereas remaining threads continue normal execution. We evaluated our fault-tolerance solution extensively under different failure scenarios and showed the recovery overheads are negligible for the common case of small number of failures.

As our last work, we focused on another type of failure caused by the tuning efforts of runtime software and programmers to improve parallel execution performance. First, we proposed a memory management scheme to reduce the total memory consumption of applications expressed as a task graph. The presented optimization technique is based on recycling data blocks among tasks and it is able to handle task graphs with dynamic dependence relations efficiently in contrast to the common use-count based memory allocators. Recycling operations are dictated by functions which are either automatically explored by runtime or specified explicitly by user annotations. Regardless, an incorrect recycling operation might lead to data races and erroneous program output. Therefore, next, to detect such cases and still benefit from data block recycling, we proposed two algorithms which prune the space of candidate recycling functions and recover the effects of any invalid choice of recycling operation efficiently during execution. We demonstrated that the proposed schemes reduce the total memory consumption significantly and yet is able to avoid any potential hazards.
Dedicated to my grandfather Mehmet and my parents Gülay and Serdar.
Acknowledgments

First and foremost, I would like to express my gratitude to my advisor Dr. Gagan Agrawal for being the best advisor a PhD student can possibly have. His deep technical knowledge and experience on a wide range of topics has always fascinated me. He was very kind and understanding throughout this 5-year long adventure, even at times when I wasn’t at the top of my game. I would like to thank Dr. Sriram Krishnamoorthy for mentoring me during my internship in Pacific Northwest National Laboratory. He was like a co-advisor to me and I learned a great deal from him. Also, I would like to thank my committee members Dr. Saday Sadayappan and Dr. Radu Teodorescu for reviewing my dissertation and their valuable feedback.

Next I would like to thank the two angels in my life. My dear mother Gülay protected me with her love and prayers from 5000 miles away. My beautiful and sweet wife, Pelin, was always there supporting me no matter what. My father Serdar, my hero and role model in life, helped me through my struggles during these 5 years as he always does. My special thanks go to my parents in law Nezih and Gülçin and grandmother in law Ayten who were incredibly supportive and caring.

Both at Ohio State University and during my internship, I was very fortunate to work and meet with incredible people like Mücahid Kutlu (aka Müco), Gökçen Kestor-Gioiosa, Roberto Gioiosa, Mehmet Deveci, Emre Çinkılıç, Peren Aykas, Özgür Dalkılıç, Tekin
Biçer, Abdullah Sameh, Bin Ren, Yi Wang, Linchuan Chen, Xin Huo and Jiaqi Liu. They were always there when I asked for help and made life easier and more fun.

Lastly, my deepest gratitude goes to my extended family in Samsun, Turkey; my grandparents (who passed away, but will be always remembered) Aysel, Mehmet, Civan, Selvi, my dear aunts Tülay, Havva and Nurşen (who are like a second mother to me), Gülzade, Satı, Ayşe, my uncle Hikmet, my sweet cousins Hande and Duygu. I am blessed to have such an amazing family.
Vita

July 4, 1985 ........................................... Born - Samsun, Turkey

2008 ..................................................... B.S. Computer Engineering, Bilkent University

2011 ..................................................... M.S. Computer Engineering, Bilkent University

2010-present ................................. Graduate Teaching and Research Associate, The Ohio State University

Publications

Research Publications


**Fields of Study**

Major Field: Computer Science and Engineering
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>ii</td>
</tr>
<tr>
<td>Dedication</td>
<td>vi</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>vii</td>
</tr>
<tr>
<td>Vita</td>
<td>ix</td>
</tr>
<tr>
<td>List of Tables</td>
<td>xv</td>
</tr>
<tr>
<td>List of Figures</td>
<td>xvii</td>
</tr>
<tr>
<td>List of Algorithms</td>
<td>xxii</td>
</tr>
<tr>
<td>1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Motivation</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Summary of Our Contributions</td>
<td>4</td>
</tr>
<tr>
<td>1.3 Outline</td>
<td>8</td>
</tr>
<tr>
<td>2. A Fault-Tolerant Environment for Large-Scale Query Processing</td>
<td>9</td>
</tr>
<tr>
<td>2.1 Representative Dataset Types and Data Analysis Tasks</td>
<td>10</td>
</tr>
<tr>
<td>2.2 Fault-Tolerance Support for Datasets with Range Queries</td>
<td>11</td>
</tr>
<tr>
<td>2.2.1 Parallelization, Data Organization, and Indexing Support</td>
<td>12</td>
</tr>
<tr>
<td>2.2.2 Fault Tolerance Support</td>
<td>16</td>
</tr>
<tr>
<td>2.3 Computational Model for Aggregation Queries</td>
<td>23</td>
</tr>
<tr>
<td>2.3.1 Reducing Communication Volume</td>
<td>25</td>
</tr>
<tr>
<td>2.3.2 Fault Tolerance Support</td>
<td>27</td>
</tr>
<tr>
<td>2.4 Experimental Evaluation</td>
<td>30</td>
</tr>
</tbody>
</table>
### 5. Fault-Tolerant Dynamic Task Graph Scheduling

- **5.1 Background**
- **5.2 Problem Statement**
- **5.3 Task graph scheduling using NABBIT**
- **5.4 Fault-tolerant Scheduling**
- **5.5 Performance Analysis**
  - **5.5.1 Definitions**
  - **5.5.2 Work Analysis**
  - **5.5.3 Span Analysis**
  - **5.5.4 Completion Time Bounds**
- **5.6 Experiments**
  - **5.6.1 Overheads Without Failures**
  - **5.6.2 Overheads With Failures**
  - **5.6.3 Scalability Analysis**
- **5.7 Related Work**
- **5.8 Summary**

### 6. User-Assisted Memory Management for Dynamic Task-Graph Schedulers with Auto Correction Support

- **6.1 Background**
- **6.2 Our Approach: Store Recycling Functions**
- **6.3 Evaluating Store Recycling Functions**
  - **6.3.1 Checking Correctness of Recycling Functions**
  - **6.3.2 Exploring the Space of Recycling Functions**
- **6.4 Problem-Size Independent Store Recycling with Auto Correction Support**
- **6.5 Implementation**
  - **6.5.1 Task graph scheduling using NABBIT**
  - **6.5.2 Extensions to the NABBIT Scheduler**
- **6.6 Experimental Evaluation**
  - **6.6.1 Benchmarks and Applicability of Use-Count Based Garbage Collection**
  - **6.6.2 Comparison Between Single-Assignment and Recycling**
  - **6.6.3 Auto-recycling Overheads**
  - **6.6.4 Recycling Function Verification Costs**
  - **6.6.5 Re-execution Overheads**
- **6.7 Related Work**
- **6.8 Summary**
List of Tables

Table | Page
---|---
2.1 Main Terms Used in Formulation | 23
3.1 Execution Times (in secs) of Jacobi and Euler for DISC and Charm++ with varying number of shares under (a) homogeneous, (b) heterogeneous and (c) heterogeneous execution with runtime support for load-balance. | 71
4.1 Error detection rates for plain replication (rep) and replication with on the fly checksum and no replica computation-space object (rep+ofc+ncs) versions both without and with soft error injection. 0% rate for normal execution shows that there are no false positives | 94
5.1 Matrix size (N), block size (B), total number of tasks (T), total number of dependencies (E) and the critical path length (S) for each benchmark. | 119
5.2 Average, minimum, maximum, and standard deviation of the number of re-executed tasks in after notify scenario when the original set of failures on different task types (“v=0,” “v=last,” and “v=rand”) implies 512 task re-executions. The resulting average recovery overhead for each benchmark is presented in Figure 5.6. | 126
6.1 Dependence structures for LCS and Cholesky. Each tag in dependence structures correspond to functions provided by user for automatic exploration of recycling functions. | 143
6.2 Examples of automatically explored recycling functions with a two-hop predecessor traversal for LCS and Cholesky. | 144
6.3 Matrix size (N), block size (B), total number of tasks (T) and number of task classes (C) for each benchmark.“(d1)” and “(d2)” represent different problem sizes. | 151
6.4 Memory consumption (in GB) of single-assignment ("Single"), handwritten-recycle ("Hand") and auto-recycle versions that lead to minimum ("Min-Auto") and maximum ("Max-Auto") consumption. .................................................................................. 160

6.5 Number of recycling functions that are checked and actually verified as correct, when recycling functions are explored automatically with 1, 2, and 3 hops. Each pair "n_1/n_2" denotes that n_1 functions are determined to be correct after considering n_2 functions. ................................................................................................. 162
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>(a) 2nd-Order Hilbert Curve enumerates a data space by sixteen points. (b) Eight rectangles in data space are first sorted according to their Hilbert Value, then grouped into chunks each of which contains two rectangles. MBR of each chunk is shown in thick lines.</td>
</tr>
<tr>
<td>2.2</td>
<td>Master Node containing Hilbert R-Tree and Worker Nodes. The leaves of Hilbert R-Tree correspond to data chunks. Chunks are distributed to worker nodes in round-robin fashion.</td>
</tr>
<tr>
<td>2.3</td>
<td>(a) Replication stage generates two subchunks for each chunk. Subchunk $j$ for a chunk $i$ is denoted by $chunk_{ij}$. Distribution stage sends each subchunk to a different worker node in a round-robin fashion. (b) Each worker node stores its own chunk along with the subchunks of other nodes for fault-tolerance purpose.</td>
</tr>
<tr>
<td>2.4</td>
<td>Leaf and Subleaf Levels in Hilbert R-Tree with Fault Tolerance Support. Subleaf (subchunk) $j$ for a leaf (chunk) $i$ is denoted by $chunk_{ij}$.</td>
</tr>
<tr>
<td>2.5</td>
<td>(a) The nodes which are in the same rack do not send subchunks to each other. (b) After replication and distribution of subchunks, each worker node stores its own chunks along with the subchunks coming from the worker nodes outside of the rack.</td>
</tr>
<tr>
<td>2.6</td>
<td>A data space partitioned by three vertical and one horizontal cuts. Vertical cuts and horizontal cuts result in four x-ranges ($r_{x1}$, $r_{x2}$, $r_{x3}$, $r_{x4}$) and two y-ranges ($r_{y1}$, $r_{y2}$), respectively. Each partition is assigned to a different worker node denoted by $M_1$ to $M_8$.</td>
</tr>
<tr>
<td>2.7</td>
<td>Execution Time with No Replication and No Failures</td>
</tr>
</tbody>
</table>
2.8 Query Processing Times for Different Replication Schemes with Varying Chunk Size. (20% selectivity) - \( k \) is the number of subchunks for a chunk.  

2.9 Failure Overhead Ratio for Different Replication Schemes with Varying Selectivity.  

2.10 Execution Times for Different Replication Schemes with Rack Failures (Varying Rack Size). 64 worker nodes are used for parallelization.  

2.11 Execution Times and Analytical Trend with Different Partitioning Schemes. \(| r_x | = | r_y | = 10000, P(X) = P(Y) = 0.5, 1000 queries, no failures, 1 billion data objects and 64 worker nodes.  

2.12 Execution Times with Different Partitioning Schemes. \(| r_x | = | r_y | = 100000, P(X) = P(Y) = 0.5, 1000 queries, single machine failures, 1 billion data objects, 64 worker nodes.  

3.1 (a) A sample 2D-domain extending through \((x_{lo}, x_{hi})\) and \((y_{lo}, y_{hi})\) on X and Y dimensions, respectively. (b) 2D-domain decomposed into four subdomains by runtime system after programmer initializes the domain by the corresponding API function.  

3.2 Example code snippets for DISC (a) domain initialization, (b) attribute registration and (c) compute-function and computation-space object coupling and the main loop for a 3D molecular dynamics code.  

3.3 Common interaction types in scientific computing applications (a black data instance is interacting with gray data instances).  

3.4 setup_exchange_west() function (called by runtime system) which determines the x-coordinate range of points that subdomain \( S \) must send to its \( west \) neighbor at each exchange, when radius-based interaction is used.  

3.5 (a) A sample 2D domain partitioned with 1D-Decomposition (b) The same domain partitioned with 2D-Decomposition  

3.6 (a) The initial even partitioning of a grid structure with 10000 rows and 15000 columns among 6 processing units. Unit processing time of each unit \( P_i \) is denoted by \( t_i \). (b) The resulting partitions after automatic repartitioning is performed. Dashed lines show the previous position of each boundary, and red arrows show the direction that it has been shifted.
3.7 Homogeneous Configuration: Execution times of (a) MiniMD, (b) Euler, (c) Jacobi, and (d) Sobel for MPI and DISC Implementations on varying number of nodes with 4 cores each. ............................................. 65

3.8 Heterogeneous Configurations: Execution times of MiniMD, Euler, Jacobi and Sobel for MPI, DISC (default) and DISC with Heterogeneous Execution Support (disc-h) when 16, 32 and 64 cores are slowed down by 40%. For MiniMD and Euler, domain repartitioning is only performed once at the beginning. For Jacobi and Sobel, it is performed at the beginning and every 500th iteration. ................................................. 77

3.9 Heterogeneous Configurations: Execution times of MiniMD, Euler, Jacobi and Sobel for MPI, DISC (default), DISC with Heterogeneous Execution Support (disc-h) and DISC-perfect, when 64 cores (out of 256) are slowed down with varying random percentages. ............................................. 78

3.10 Normalized Execution Times of 4-way Jacobi, 8-way Jacobi and Sobel for DISC with 2 processes (with different computational power) on a single node. Process I’s share of workload is varied between 2.5% and 60% (rest is given to Process II). Asterisk (*) signs on the figure show DISC’s decision of work division between two processes for each application. Execution times are normalized against the time of the configuration that gives the best performance in each application. ............................... 79

4.1 Sample checkpoint files for a (a) 2D stencil and (b) 3D molecular dynamics application. Both files consist of two parts as meta-data and computation-space objects. ............................................. 82

4.2 Flow of execution at each iteration when replication strategy is in use. ................................. 86

4.3 Normal execution and checkpointing times of MPI and DISC implementations of four applications with varying number of nodes. ............................................. 89

4.4 Execution times of four applications without any replication (no rep), with plain replication (rep) and replication with improvements for cache utilization (rep+ofc and rep+ofc+ncs). Execution times for no rep with 1 process are 307.9, 398.9, 2686.2 and 213.2 seconds in Jacobi, Sobel, MiniMD and Euler, respectively. The same execution times for the best replication version rep+ofc+ncs are 316.2, 474.5, 2738.3 and 214.8 seconds. ................................. 92
5.1 A task graph example with task $A$ as the source and task $E$ as the sink. Dependence edges are drawn from producer to consumer. Black-shaded task $B$ fails during execution.

5.2 Routines used by the fault-tolerant task graph scheduler. Non-shaded portions correspond to the actions of the non-fault-tolerant NABBIT scheduler. Shaded portions indicate additions to the algorithm that make it fault tolerant. “A.key{key}” means that key replaces A.key in the fault-tolerant version. Auxiliary routines used for recovery (in catch blocks) are discussed in Section 5.4.

5.3 Additional routines to assist in the recovery from faults.

5.4 Speedup for the baseline and fault-tolerant versions in the absence of faults. Sequential times (in secs) for each benchmark are as follows: LCS (baseline=650, w/FT-support=668), SW (baseline=578, w/FT-support=579), LU (baseline=624, w/FT-support=613), Cholesky (baseline=338, w/FT-support=337), and FW (baseline=315, w/FT-support=371).

5.5 Impact of the time of failure (“before compute” and “after compute”) on the recovery overhead, when failures occur on different task types (“v=0,” “v=rand,” and “v=last”). (a) Failures cause the re-execution of 512 tasks; (b) failures cause 2% and 5% of the total number of tasks to be re-executed. In both (a) and (b), overheads are calculated by taking the sequential execution time of FT-support version, in the absence of faults, as the baseline. FT-support (evaluated in Figure 5.4) introduces an additional 10% overhead for FW, and negligible overhead for the other benchmarks.

5.6 Recovery overheads in the after notify scenario when the set of injected failures implies re-execution of 512 tasks, 2% and 5% of the total number of tasks in the graph.

5.7 Recovery overheads when varying the number of cores ($P$) for a failure scenario, causing the re-execution of (a) 512 tasks and (b) 5% of all the tasks. In both (a) and (b), failures are injected on “v=rand” tasks at the after compute failure point.

6.1 A task graph example, in which task $B$ can safely recycle the store assigned to task $A$, but not the one assigned to task $C$. 

xx
6.2 Basic actions of the store recycling verification algorithm. The implementa-
tion employs optimized versions of the atomic statements (discussed in
Section 6.3.1). ................................................................. 140

6.3 (a) Task graph for LCS. Green tasks form the set of candidates whose out-
put the yellow task can recycle. (b) Examples for three different task classes
in Cholesky shaded in different colors. ....................................... 142

6.4 Concurrent recycle detection on an example task graph. (a) and (b) show
the system state before and after task B recycles task A’s data item D. Task
C trying to recycle the same data item is forced by the runtime to allocate
new memory. ................................................................. 147

6.5 Eager recycle detection on an example task graph. (a) and (b) show system
state before and after task B recycles task A’s data item D. C detects that
A’s output has been recycled and initiates its re-execution. ................. 148

6.6 Execution time of LU, Cholesky and FW with single-assignment, use-
count, optimized handwritten-recycle, and auto-recycle versions. Perform-
ance of all benchmarks are evaluated on two different problem sizes la-
beled as “(d1)” and “(d2)”. Execution of single-assignment versions in LU
(d2), Cholesky (d2) and FW (d2) did not finish as they ran out of memory. . 155

6.7 Results of the same experimental setup in Figure 6.6 for SW, Rician, Heat2D
and LCS. Execution of single-assignment versions in SW (d2), Heat2D
(d2) and Rician (d2) did not finish as they ran out of memory. ............ 156

6.8 Correlation between number of reuses (x-axis) in verification and observed
memory consumption (in GB) (y-axis) in actual runs. Each point in the
figures represents a pair of these two values for an automatically generated
recycling function. .......................................................... 158

6.9 Costs associated with verification of recycling functions at 61 threads. . . 161

6.10 Distribution of incorrect recycling functions identified for each benchmark
in terms of their re-execution overheads (in percentage of original execu-
tion time). 4, 1424, 228, 1424, 30, 25 and 1500 functions are evaluated for
LCS, LU, Cholesky, FW, SW, Heat2D and Rician, respectively. .......... 163

6.11 Re-execution overheads (in percentage) with a representative incorrect re-
cycling function with varying number of threads. .......................... 164
List of Algorithms

1. Filtering Operation in Hilbert R-Tree ........................................... 20
2. Operations performed by the DISC runtime system in main processing structure when run _iteration_() function is invoked by the programmer. ... 60
3. TRYINITCOMPUTE(A, key, life, pkey) ............................................. 104
4. NOTIFYONCE(A, key, pkey, life) .................................................. 104
5. INITANDCOMPUTE(A, key, life) ..................................................... 104
6. NOTIFYSUCCESSOR(key, skey) ................................................... 104
7. COMPUTEANDNOTIFY(A, key, life) .............................................. 104
8. RECOVERTASKONCE(key, life) ..................................................... 107
9. ISRECOVERING(key, life) ............................................................ 107
10. RESETNODE(A, key, life) ........................................................... 107
11. REINITNOTIFYENTRY(T, key, S, skey, slife) ............................... 107
12. RECOVERTASK(key) ................................................................. 107
Chapter 1: Introduction

1.1 Motivation

High performance computing is undergoing a significant transformation, in the sense that performance is no longer the sole consideration while developing applications. Fault-tolerance, which is defined as providing dependable service in spite of faults occurring or having occurred [90], is becoming equally important as performance. Scientific and commercial computing today is performed by a large number of commodity machines with 10s/100s of thousands of cores targeting more parallelism. Emergence of multi-core and more recently many-core machines suggest that the number of components in the future HPC systems will be even greater. However, such an explosion in the number of components increases the probability of failure of any (or more) of these components at any given time. This as a result decreases the Mean Time Between two consecutive Failures (MTBF) in large HPC systems substantially. In fact, in some of the existing studies [125, 6], exascale systems are projected to fail every 3-26 minutes. Such failure rates would degrade the performance of commercial applications serving millions of transactions every hour drastically and also would make the completion of long-running scientific applications almost impossible. Hence, providing effective and efficient fault-tolerance capabilities both at hardware and software level is an absolute necessity.
The type of failures that a fault-tolerant system should deal with can be categorized into three. Fail-stop failures represent the class of failures, which cause a component in a large system to stop operating normally and become inaccessible. Fail-stop failures have been mostly addressed through checkpoint/restart based solutions [132, 86, 3, 38, 21, 78, 102, 116, 79]. In these solutions, each computing unit writes its current execution state periodically into a persistent storage either in a coordinated or uncoordinated fashion. Upon a failure, each unit reads its latest saved checkpoint file and rolls back the execution to the most recent error-free state. Currently, the most popular and only fully automated checkpoint/restart mechanism is system-level checkpointing [93, 106, 49, 12], which saves the entire application state from the operating system’s viewpoint by copying associated memory regions byte-by-byte. Unfortunately, the total cost of fault-tolerance support with system-level checkpointing, which is the sum of the costs of taking checkpoints (which increases as checkpointing frequency increases), the net time spent on re-computation (which increases as checkpointing frequency decreases), and the time spent on restart after a failure, can dominate the actual execution time. An analysis of a 100,000 core job, where each node has a MTBF of 5 years, indicates that these three costs can add up to 65% of the total execution time, i.e. only 35% of the time will be productively used [57]. One direction to reduce the high I/O overheads associated with system-level checkpointing is application-level checkpointing [13, 24, 62, 98, 111, 147]. However, this approach puts significant burden on users, since it requires them to point out which data-structures should be checkpointed and when as well as including recovery logic in application code.

Until recently, most of the work on fault-tolerance has focused on fail-stop failures. In recent years, there is a growing concern about a new class of failures, namely, soft errors or
silent data corruption. These errors involve bit flips in either processing cores, the memory, or the disk. Although radiation has been considered the main cause of such random bit flips [110], the recent technology trends such as use of smaller and smaller transistors and efforts to improve power-efficiency in hardware are now attributed as the cause of these faults occurring more frequently [126]. Many recent publications have summarized the observed frequency of these faults [56], for example, double bit flips occur daily at a national lab’s Cray XT5, and similarly, such errors were frequent in BG/L’s unprotected L1 cache. One popular approach to detect and recover from soft errors is replica execution [103, 56, 115, 117, 143, 114, 54]. Replication is orchestrated either by underlying hardware or software and exercised at varying granularity levels ranging between whole process and individual instruction replication. Regardless, the entire or a subset of computation is performed redundantly and the results are compared by means of a voting scheme. Unfortunately, most of the solutions based on replica execution require dedicated hardware and result in low resource utilization.

In addition to fail-stop failures and soft errors, yet another and very frequently seen type of failures are errors introduced by programmers [112] as part of performance optimization efforts. The emergence of the multi-core era has led to an increased interest in the design of effective yet practical parallel programming models. To achieve scalable performance, existing models [25] are accompanied by heavily optimized runtime systems which support further tuning of implemented applications. These tuning efforts range from task scheduling and prioritization to affinity control and memory management and can be expressed within the application by programmers through an extended set of language constructs/API. Such efforts might become a potential source of error (just like fail-stop failures and soft errors) if they are not incorporated carefully. Hence, the ability to recover
from such errors induced by programmers is extremely important for any programming model with configurable runtime behavior.

In this dissertation, we present fault-tolerance solutions against the three type of failures (fail-stop failures, soft errors and programmer induced errors) emphasized above. Proposed solutions are designed and implemented in the context of both existing and new programming models and large scale data processing frameworks.

1.2 Summary of Our Contributions

A Fault-Tolerant Environment for Large-Scale Query Processing: In our first work, we consider the problem of supporting query processing and data analysis on a massive scientific dataset, when parts of the underlying data become inaccessible due to fail-stop failures. Particularly, we consider both processing of data to compute various derived data products, and queries from various scientists, which would require data reduction, both applied on spatial or spatio-temporal datasets. We have the following goals, when such analysis is performed over a large dataset hosted on a cluster: 1) high efficiency of execution of a particular data analysis task or query, when there are no failures, 2) ability to handle failure of up to a certain number of nodes, and 3) only a modest slowdown in processing times of data analysis task or a query when there are failures.

We present a fault-tolerant environment to process range queries on spatial datasets and aggregation queries on a collection of points on a 2-dimensional space. Our framework is designed to handle two failure scenarios: single machine failure and rack failure. We show how we replicate the data, manage the metadata, and process queries to achieve the three goals we listed. First, we use well-known partitioning and indexing mechanisms to achieve high parallel efficiency while processing these datasets. Second, by replicating data, we
are able to keep the system operational even in the presence of failures. Specifically, we use subchunk or subpartition replication, which allows us to meet our third goal, i.e. query workload during failures is shared appropriately among the remaining (operational) nodes. This property is very valuable for such systems, since failures can be frequent with growing cluster sizes, and moreover, users will like to see high query processing efficiency even in presence of failures.

A Parallel Programming Model With Support for Heterogeneous Execution and Low Overhead Fault-Tolerance: As our second work, we introduce fault-tolerance solutions for iterative scientific applications in the context of a new parallel programming model. Considering the current technology trends and challenges for future HPC systems, we first make the following observations on the existing distributed memory programming models. (1) Programming models today do not allow automated and efficient application-level checkpointing. A programming model which can meet this requirement would reduce the costs of fault-tolerance support significantly. (2) They do not have the ability to capture the core execution state and main computations in an application. Such knowledge can be extracted with programming abstractions and can be used to protect applications against soft errors through a low overhead replication strategy. (3) Almost none of them allows an application involving communication to be load balanced across a set of heterogeneous resources. Moving forward, it is critical that programming models allow heterogeneous resources to be used effectively.

This work describes a programming model and an associated runtime system we have developed to address the above needs. The main concepts in the programming model are that of a domain and interactions between the domain elements. Once this information is
captured through our APIs, interprocess partitioning of work and communication is automatically handled by the system. We first show how the runtime system can automatically re-partition the work among heterogeneous processors or nodes using a novel approach based on a non-linear optimizer. Next, we show how a set of abstractions, particularly the concepts of compute-function and computation-space objects, can leverage low-overhead fault-tolerance solutions against hard and soft errors within the programming model. While computation-space objects enable automated application level checkpointing, replicated execution of compute-functions helps detect soft errors with low overheads.

**Fault-Tolerant Dynamic Task Graph Scheduling:** Next, we direct our attention to scalable and efficient recovery techniques from soft errors in the context of a different programming paradigm; *task-graph execution model*. Although there is a large code base of applications written with popular computing paradigms such as BSP (Bulk Synchronous Programming) and SPMD (Single Program Multiple Data), these models generally, even the best implementations, can not hide the latency caused by the communication and synchronization among processes involved in computation. In order to achieve massive parallelism and satisfy resilience needs in large-scale systems, task-graph execution model stands out as a promising direction and it has been already revisited by popular parallel runtimes such as CILK [8, 18] and CnC [25].

In this work, we consider the design, implementation and evaluation of scheduling algorithms that can continue execution of an application specified through a task graph to completion despite *detectable* soft errors. The detailed algorithms we have developed and implemented are specific to the NABBIT system [4], which is a provably time-efficient framework for scheduling task graphs using work stealing. We adapt the NABBIT dynamic
task graph scheduling algorithm to support scalable recovery from soft errors that impact individual tasks. The recovery is performed in a non-collective fashion without interfering with threads not impacted by the fault. At the same time, the threads needing a waiting task efficiently perform the recovery without incurring significant blocked or idle time. Our approach can recover from arbitrary number of task failures while incurring very low overheads in the absence of faults.

User-Assisted Memory Management for Dynamic Task Graph Schedulers with Auto Correction Support: As our last work, we address the third type of failures we consider; programmer induced errors. As an example of such a case, we focus on user-assisted memory management for dynamic task graph schedulers. The traditional approach to limit memory consumption in task graphs is garbage-collection through a reference-counting scheme, in which each data item is associated with a counter that tracks the number of usages on that data item so far. Once fully consumed, memory segment of the data item is reclaimed by runtime. Despite being effective, reference-counting scheme requires the programmers to specify an exact usage count for each data item during production time. However, in task graphs with dynamic dependence relations, such knowledge might not be available a priori. An alternative approach in which programmers provide memory recycling hints through explicit annotations can overcome this limitation, but runtime should deal with any misspecifications that might lead to erroneous execution state.

First, we present a novel approach to dynamically recycle the memory locations assigned to data items as they are produced by tasks. Recycling operations are dictated by functions which are either automatically explored by runtime or specified explicitly by programmer annotations. We develop algorithms to identify memory-efficient store recycling
functions by systematically evaluating the validity of a set of (user-provided or automatically generated) alternatives. Because recycling functions can be input data-dependent, we have also developed support for continued correct execution of a task graph in the presence of a potentially incorrect store recycling function. Experimental evaluation demonstrates that our approach to automatic store recycling incurs little to no overheads, achieves memory usage comparable to the best known solutions, often produces recycling functions valid across problem sizes and input parameters, and efficiently recovers from incorrect choice of store recycling functions.

1.3 Outline

In Chapter 2, we present our fault-tolerant large scale query processing framework that currently supports range queries on spatial data and aggregation queries on point datasets. In Chapter 3, we first describe our domain-interaction based programming model that provides efficient heterogeneous execution for iterative scientific computation patterns commonly used. Chapter 4 extends the programming model presented in Chapter 3 and shows how the programming abstractions used in this model leverage automated application-level checkpointing as well as low-overhead replica execution to detect soft error occurrences. Chapter 5 introduces our fault-tolerant dynamic task graph scheduler and presents the details of the algorithm that recovers from detectable soft errors. In Chapter 6, we present our user-assisted memory management approach for dynamic task graph schedulers and show how incorrect memory optimization efforts are detected and fixed automatically by the runtime. Finally, we conclude in Chapter 7.
Chapter 2: A Fault-Tolerant Environment for Large-Scale Query Processing

One of the most critical challenges facing computing today is the “big data” problem. As datasets are increasing in size, the data management and processing needs are being met with added parallelism, i.e., by involving more nodes and/or cores in the system. Hence, as in any large-scale systems, fault-tolerance is also a major problem for systems that maintain a large dataset or data collection, and perform parallel search or retrieval on this data. Loss of underlying data due to fail-stop failures is intolerable, since user queries and analysis tasks need to be served accurately as well as in a timely manner. Hence, data replication, which is the traditional approach to ensure availability of data in big data platforms, should be done in a way that load balance among computing units is still preserved even after failures occur.

In this work, we present the design and implementation of a fault-tolerant environment for processing queries on large scientific dataset. Our systems meet the following three requirements that we consider essential for any such environment: 1) high efficiency of execution of a particular data analysis task or query, when there are no failures, 2) ability to handle failure of up to a certain number of nodes, and 3) only a modest slowdown in processing times of data analysis task or a query when there are failures. We address these challenges by developing a new data replication scheme, which we refer to as subchunk or
subpartition replication. Our system currently supports two types of queries: range queries on spatial data and aggregation queries on point datasets, but the underlying ideas can be extended to other query types as well.

2.1 Representative Dataset Types and Data Analysis Tasks

The approach for supporting fault-tolerance, particularly, achieving high efficiency of data analysis in the presence of failures, depends upon the nature of the dataset and the data analysis and queries performed on the dataset. In this work, we consider two specific types of datasets with the associated query workloads. In both cases, we are assuming that a certain type of queries or data analysis tasks form the primary workload. Thus, our data placement strategy can focus on optimizing these types of queries. All other queries can still be executed on the system, though not with the same efficiency, especially, when failures arise. As the two types of datasets and workloads we consider arise frequently in practice, our solutions are widely applicable.

The first type of dataset is a large spatial dataset, in which each data object covers an area on a 2-dimensional space. We describe each data object by a rectangle, using four values, denoted as \((min_x, max_x, min_y, max_y)\). These four values indicate the boundaries of the rectangle along the \(x\) and \(y\) dimensions, respectively. Such datasets arise in geographic information systems (GIS), very-large-scale integration (VLSI) designs and computer-aided design (CAD) applications, in which rectangles present a good approximation of the space that each geometric entity covers. The dominant query workload on such a dataset comprises spatial searches or range queries, each of which has another rectangle as its parameter. Such a query involves returning the data objects whose rectangles overlap with the area covered by the query.
The second type of dataset we consider is as follows. The dataset consists of a set of points in a 2-dimensional space. Each data point is \((x, y, \text{attribute}1, \text{attribute}2, \ldots)\), where \(x\) and \(y\) are the coordinates of the point in space. Outputs of large-scale simulations often follow this format.

The primary workload on such datasets comprises a variety of \textit{data reduction} tasks, which can all be viewed as aggregation queries. An aggregation query involves three parameters: an \textit{aggregation dimension}, an \textit{aggregation function} and a \textit{selection criterion}. The first argument, aggregation dimension, indicates the axis that the values need to be aggregated. If the value is \(x\), we aggregate a specified attribute of the points that have the same \(x\) coordinate value. As a result of this operation, we have a vector that contains an aggregation value for each distinct \(x\) coordinate on the data space. Aggregation function, which is the second argument, can be any of the reduction functions such as minimum, maximum, sum, average, or other user-specified function. The third argument specifies the value that need to be aggregated, and also provides a \textit{selection criterion}, so we can exclude the data points that do not satisfy that criterion from the aggregation operation. An example query is “\textit{for each unique \(x\) coordinate, find the average of the first attribute of all points where the first attribute is greater than 100}”.

\subsection*{2.2 Fault-Tolerance Support for Datasets with Range Queries}

The first type of dataset and workload we consider involves spatial objects, on which range queries are executed. We initially describe the computational set-up used to parallelize such queries, and the well-known data organization and indexing mechanisms typically used for such cases. Then, in the context of the particular data organization and indexing mechanism, we describe our approach for enabling fault-tolerance.
Figure 2.1: (a) 2nd-Order Hilbert Curve enumerates a data space by sixteen points. (b) Eight rectangles in data space are first sorted according to their Hilbert Value, then grouped into chunks each of which contains two rectangles. MBR of each chunk is shown in thick lines.

2.2.1 Parallelization, Data Organization, and Indexing Support

Our work used the well known master/worker model, where the master node initiates the processing of a query. Each worker node stores a different portion of the underlying spatial data. The master sends each worker node the information about the sub-query that they need to execute. The unit of data for the purpose of data management is a block or a chunk. **Chunk size** is a preset parameter for the system and it indicates the number of data objects that each data chunk stores. We define **minimum bounding rectangle (MBR)** of a data chunk as the rectangle that covers all of the data rectangles (objects) that are contained in that data chunk. Similar to the representation of a data rectangle (object), we represent MBR of a data chunk by four values denoted as \((min^c_x, max^c_x, min^c_y, max^c_y)\).
To help improve performance, a *spatial indexing structure* is typically used to help filter the data regions. While indexing has been a topic of extensive study in the database community, certain solutions have been widely used. R-Tree, proposed by Guttman [69], is a well-known tree structured access method, which groups the data objects that are closely positioned on data space together and represents them by a minimum bounding rectangle. The original R-Tree has been extended to create several variants, which differ from their ancestors by applying different data grouping and insertion strategies. For instance, Hilbert R-Tree[82] applies a technique based on Hilbert space-filling curves [72] to group the nearby data objects together and it is proven to be very effective. Thus, in this work, we use Hilbert R-Tree as the representative spatial filtering structure on which our solution for supporting fault-tolerant query processing is implemented.

A Hilbert R-Tree is built as follows. In the very first step, a Hilbert space-filling curve is chosen. Such a curve visits any n-dimensional data space without crossing itself. Figure 2.1(a) shows a 2nd-order Hilbert Space Filling Curve and how it enumerates a data space by generating sixteen Hilbert points. Once a Hilbert Curve is selected, next, data objects in the dataset are sorted according to their *Hilbert value*. We define the hilbert value of a data object as the hilbert point closest to the center of its rectangle. After sorting is done, the successive data objects are packed into chunks, so each chunk holds a number of data objects which are located more closely on data space than the data objects contained in the other chunks. In Figure 2.1(b), eight data rectangles (objects) in a dataset are sorted according to their Hilbert values (the sorted list is shown just below the figure). In the same figure, each chunk created out of this sorted list contains two data objects and MBR of a chunk is shown in thick lines. Once the chunks are created, in the remaining steps, these
chunks are packed into a number of non-leaf nodes and this process is repeated until there is a single root node left.

In order to improve the resource utilization of the system and maximize the throughput, we want each worker node to perform an equal amount of work for each query. Distributing the data chunks that correspond to the leaf nodes in Hilbert R-Tree is a critical step in this direction. A range query on spatial data is likely to return the data rectangles that are positioned close to each other. Therefore, effective declustering of neighboring data objects among the available worker nodes results in better division of work. Distributing the chunks in a round-robin fashion to the worker nodes gives results in effectively declustering, since similar data chunks go to different nodes.

There are several design options that can be considered in term of organizing the indexing structure for a parallel environment. One approach can be to build a separate R-Tree on each available worker node. In this design, once the query definition is fetched from master node, each worker node would run the query on its own R-Tree. However, in such a design, every worker has to be involved in query execution regardless of whether it has data relevant to the query. Another approach is to create a single R-Tree on the entire dataset and place it at the master node. In our framework, we apply this second approach since it is the more common solution in existing studies [85] and it provides a more centralized view for fault-tolerance, as we explain in next section. One possible downside of a single centralized R-Tree might be that it can become a bottleneck. Because we target read-only scientific datasets, this concern can be addressed by having multiple copies of the entire R-tree on different nodes, which can all serve as masters.

Figure 2.2 gives an overview of our framework. In this example, the system consists of one master node and four worker nodes and we use the same eight data objects shown in
Figure 2.2: Master Node containing Hilbert R-Tree and Worker Nodes. The leaves of Hilbert R-Tree correspond to data chunks. Chunks are distributed to worker nodes in round-robin fashion.

As indicated previously, Hilbert R-Tree is stored at the master node and leaves of the tree correspond to data chunks that are created by grouping the data objects in the sorted list in Figure 2.1(b). For each leaf node (data chunk) in Hilbert R-Tree, we keep an entry holding three pieces of information: $cid$, $mid$ and $MBR$. $cid$ is a number that identifies that chunk uniquely in the system, and $mid$ is the id of the worker node storing it. As the figure illustrates, for each non-leaf node in the tree, we keep a separate $MBR$, covering data chunks stored under that branch, and $child_{1..c}$ pointers for each child node. $MBRs$ and $child_{1..c}$ pointers stored in non-leaf nodes are used for navigation within the tree during filtering process of a query.

The overall operation of the system is as follows. By utilizing Hilbert R-Tree’s filtering ability, the master node filters unrelated data regions from search space and creates a list of data chunks that can possibly hold data rectangles intersecting with the query rectangle.
Note that although MBR of each chunk is stored in the tree, master node does not make any computations to check if MBR of a chunk itself intersects with the query. This intersection check would bring an additional overhead and we decided to leave that task to worker nodes. Next, to each worker node, the master node sends $cids$ of the chunks that need to be processed by that node along with the query definition. Each worker node then checks each chunk in that list; if the chunk intersects with the query rectangle, the data objects stored in that chunk are tested against query. The intersecting data objects are sent to the master node.

### 2.2.2 Fault Tolerance Support

In our study, we deal with two specific failure scenarios: individual machine failures and rack failures, which can be described by a number of machines connected to the same network hardware becoming unavailable at one instance. In both of these scenarios, we assume that a failure can only occur in between processing of two queries, and hence we do not focus on intra-query failures. Note that query types and data analysis tasks we are supporting have a short-running nature.

When a worker node in the system fails, we lose all the data stored under that node. Considering the nature of the range search on spatial data, this loss is intolerable, since we cannot completely answer any query that accesses data from this node. The obvious method to address this problem is data replication. Adapting data replication solution to our framework, we can replicate the data chunks under each worker node and keep the replicas in any one of the remaining nodes present in system. However, this solution comes with a price. Replicating all of the data chunks under a node and storing the replicas only under a single node means that the node that stores the replicas performs work twice as
Figure 2.3: (a) Replication stage generates two subchunks for each chunk. Subchunk $j$ for a chunk $i$ is denoted by $chunk_{ij}$. Distribution stage sends each subchunk to a different worker node in a round-robin fashion. (b) Each worker node stores its own chunk along with the subchunks of other nodes for fault-tolerance purpose.

much as any other node, when there is failure. This would definitely results in degradation of the overall performance due to poor load balance. For this reason, we propose another replication and distribution scheme which let us provide fault-tolerance while preserving load balance.

**Main Idea:** Instead of replicating the chunks stored under a node in entirety, we break each chunk into a predefined number of **subchunks**. Assuming that chunk size is $a$ and subchunk size is $b$ (number of data objects in a subchunk), we group the first consecutive $b$ data
objects in the original chunk into one subchunk, the following \( b \) data objects into another subchunk, and so on. After repeating this subgrouping procedure for every successive \( b \) data objects, we end up with \( k (\approx \lceil a/b \rceil) \) number of subchunks for each chunk. For the next step of our fault-tolerance scheme, we distribute these subchunks in round-robin fashion to the remaining nodes in the system. As a result, the copy of a data chunk is shared among a number of worker nodes as distinct pieces encapsulated in subchunk form, instead of being kept by a single node in entirety. Figure 2.3 shows an example setup where we have four nodes each of which stores only one data chunk. In the same example, first (replication stage), we create two subchunks \((k = 2)\) out of each data chunk and then (distribution stage) distribute them to other worker nodes in round-robin fashion.

When a failure occurs in the system, the processing work that would be done by the failed node normally is shared among the other worker nodes. Because of the way the original data chunks are created during Hilbert R-Tree construction, each chunk contains data objects that are closely located in data space. Since we select successive data objects while creating the subchunks from a chunk, we know that data objects within each generated subchunk also maintain proximity. This property prevents the blind grouping of data objects into subchunks and leads to more compact MBRs in representing the area a subchunk covers.

**Query Processing and Metadata:** Query processing with such fault-tolerance works as follows. The master node keeps track of which worker nodes are alive at any time and assigns the tasks that needed to be done to the available worker nodes when there is a failure. However, organization of metadata is critical for enabling this process. Specifically, where a data chunk is stored, which subchunks belong to that particular chunk, and where these subchunks are located, needs to be known by the master node. Though many designs
are feasible, we chose to embed this information into the Hilbert R-Tree itself. This design choice not only leads to the data in master node to be kept as compact as possible, but also makes it possible for Hilbert R-Tree to be used both as a filtering structure and a failure-management tool at the same time.

In order to build the fault-tolerance meta-data into Hilbert R-Tree, we change its inner structure by adding a subleaf level to the bottom of the tree. Introduced subleaf level consists of the entries about subchunks created out of each data chunk. Figure 2.4 illustrates this subleaf level and the leaf level just above it. For each node in subleaf level, we keep a MBR along with a unique subchunk id (scid) and the machine id (mid) of the worker node that stores the subchunk. We also modify leaf node structures so that each of them now contains an array of pointers subleaf\textsubscript{1...k} pointing to the corresponding subchunk entries for the chunk that leaf node represents.

The complete protocol is as follows. First, the master node processes the query definition on Hilbert R-Tree as described previously. When the search in R-Tree reaches a leaf node, the master node looks at the id of the worker node that the chunk is stored in and checks if that node is alive or not. If the worker node is alive, it appends the chunk id of the chunk to the end of the request list of that worker node. If the worker node is
Algorithm 1 Filtering Operation in Hilbert R-Tree

1: FILTER(Q, N)
2: input: Q: query object
3: N: a node in Hilbert R-Tree
4: output: \( R_{1...M} \): array of request lists each of which contain chunk/subchunk ids for a
      worker node (M is the number of worker nodes)
5: // check each child of N
6: for child = N.child\(_1\) → N.child\(_c\) do
7:   if isLeaf(child) then
8:     // check if worker node holding the chunk is alive
9:       if isAlive(child.mid) then
10:          // append cid of chunk to the end of
11:             // corresponding request list
12:             \( R_{child.mid} \leftarrow \langle child.cid \rangle \)
13:       else
14:         // fault-tolerance support: go to each subchunk,
15:         // append scid of subchunk to the end of
16:         // corresponding request list
17:         for subleaf = child.subleaf\(_1\) → child.subleaf\(_k\) do
18:             \( R_{subleaf.mid} \leftarrow \langle subleaf.scid \rangle \)
19:       else
20:         // check if child intersects with the query
21:         if isIntersect(Q,child.MBR) then
22:             FILTER(Q, child)
not alive, master node goes one level deeper in the tree and reaches the subleaf level of the chunk. Here, for each subchunk, it appends the id of that subchunk to the request list that is associated with the worker node storing it. Note that master node again does not make any computation to check if MBR of a subchunk intersects with the query, and leaves this task to workers. After initial filtering on Hilbert R-Tree, master node sends each alive worker node both the query definition and the request list containing the chunk and subchunk ids that worker node needs to process. Algorithm 1 illustrates this filtering procedure in detail.

**Discussion:** One disadvantage of our master-worker computation model might seem as introducing a single point of failure. This limitation can be easily addressed by employing multiple master nodes, each with its own copy of the indexing structure. Another concern could be as follows. Using large chunk size in conjunction with small sized subchunks, which are only used during failures, might possibly involve complicated processing at the master node. Instead, one can choose another approach, where we have small sized chunks and their replicas might be distributed over the other nodes in a random fashion. However, this approach leads to a large number of non-leaf and leaf nodes (chunks) and results in a significant increase in the time required for navigation within the Hilbert R-tree, and also in communication between the master and the worker nodes.

**Handling Rack Failures:** So far, we explained how our fault-tolerance mechanism works for single machine failure scenarios. The same idea can be extended to rack failure scenarios with slight modifications. The subchunk creation step in the single machine failure case is directly applicable to rack failure case. However, distribution of the subchunks should differ because of the following. Assume that node\(_A\) and node\(_B\) are two nodes in the system and they are in the same rack. Simple round-robin distribution might put some portions of replica of the data stored in node\(_A\) under the node\(_B\). In order to prevent such cases, we can
<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c_v$</td>
<td>number of vertical cuts on data space</td>
</tr>
<tr>
<td>$c_h$</td>
<td>number of horizontal cuts on data space</td>
</tr>
<tr>
<td>$r_x$</td>
<td>x-coordinate range</td>
</tr>
<tr>
<td>$r_y$</td>
<td>y-coordinate range</td>
</tr>
<tr>
<td>$r_{x_i}$</td>
<td>a smaller range within $r_x$</td>
</tr>
<tr>
<td>$r_{y_i}$</td>
<td>a smaller range within $r_y$</td>
</tr>
<tr>
<td>$</td>
<td>r_x</td>
</tr>
<tr>
<td>$</td>
<td>r_y</td>
</tr>
</tbody>
</table>

Table 2.1: Main Terms Used in Formulation

introduce a rack-aware policy to our distribution scheme. In this scheme, when distributing the subchunks of a chunk in a node, we do not use the nodes that are in the same rack with that node. Thus, the only nodes that receive the subchunks of a node are the ones that are located outside of that particular rack.

Figure 2.5 illustrates the exact procedure for subchunk replication and distribution when rack-awareness policy is applied. In the figure, we use four worker nodes which are grouped under two racks.

### 2.3 Computational Model for Aggregation Queries

Aggregation queries have different characteristics than range queries. First, they involve communication across worker nodes to compute the results. Second, since every data point in the system needs to be processed for a complete aggregation, load balancing problem is trivially solved as long as data is equally partitioned among the worker nodes.

The main symbols used in describing aggregation queries and related formulations are summarized in Table 1. In our discussion, we assume a two dimensional dataset, though our methodology can be trivially extended for larger dimensional cases as well.
Figure 2.5: (a) The nodes which are in the same rack do not send subchunks to each other. (b) After replication and distribution of subchunks, each worker node stores its own chunks along with the subchunks coming from the worker nodes outside of the rack.

The master node first divides data space into equal data partitions and assigns each partition to a worker node. The shape of data partitions is determined by the number of imaginary vertical \( (c_v) \) and horizontal cuts \( (c_h) \) on the data space. Vertical cuts divide the X-coordinate range \( (r_x) \) into \( c_v + 1 \) number of smaller ranges \( (r_{xi}) \) and the size of each range \( (|r_{xi}|) \) is the same. Similarly, horizontal cuts divide the Y-coordinate range \( (r_y) \) into \( c_h + 1 \) number of smaller ranges \( (r_{yi}) \) with equal size \( (|r_{yi}|) \). The resulting smaller data
ranges define the range of coordinate values that the data points in a partition can take.

We define the set of worker nodes that share some X-coordinate range $r_{xi}$ as a “X-range set” and denote it by $R_{xi}$. Likewise, we define the set of worker nodes that share some Y-coordinate range $r_{yi}$ as a “Y-range set” and denote it by $R_{yi}$. Figure 2.6 shows an example data space partitioned by three vertical ($c_v = 3$) and one horizontal cuts ($c_h = 1$) which produce four smaller X-coordinate ($r_{x1..4}$) and two smaller Y-coordinate ranges ($r_{y1..2}$). As a result, the following range sets are produced by the partitioning; $R_{x1} = \{M_1, M_2\}$, $R_{x2} = \{M_3, M_4\}$, $R_{x3} = \{M_5, M_6\}$, $R_{x4} = \{M_7, M_8\}$, $R_{y1} = \{M_1, M_3, M_5, M_7\}$ and $R_{y2} = \{M_2, M_4, M_6, M_8\}$.

We use a standard 2-phase algorithm to parallelize aggregation queries. In the first phase (local aggregation), master node parses query specification and sends this information to each worker node. A worker node aggregates an attribute of the points that share the same coordinate value. Repeating the same aggregation operation for each unique coordinate along the specified axis results in a partial result vector for the small coordinate range that the node is assigned to. For instance, in Figure 2.6, if the aggregation dimension is specified as X-axis, worker nodes $M_1$ and $M_2$ aggregate values for points whose X-coordinate values are within the boundaries defined by range $r_{x1}$. As a result, they both produce a vector which is composed of individual aggregation values for each unique X-coordinate value in range $r_{x1}$. Similarly, if aggregation dimension is given as Y-axis, worker nodes $M_1, M_3, M_5$ and $M_7$ perform the same operation for each unique Y-coordinate value that lies in $r_{y1}$ and each generates a partial result vector.

Partial aggregation vectors generated in local aggregation phase must be combined in order to obtain final aggregation results. Therefore, the second phase (global aggregation) requires communication among the worker nodes that share the same coordinate range. For
Figure 2.6: A data space partitioned by three vertical and one horizontal cuts. Vertical cuts and horizontal cuts result in four x-ranges \((r_{x1}, r_{x2}, r_{x3}, r_{x4})\) and two y-ranges \((r_{y1}, r_{y2})\), respectively. Each partition is assigned to a different worker node denoted by \(M_1\) to \(M_8\).

this purpose, one of the worker nodes from each range set is selected as the combiner for the corresponding small range. A combiner is responsible from receiving the partial result vectors generated by all of the other worker nodes defined in the range set and producing the final result for that range by combining these received vectors. Again in Figure 2.6, \(M_1\) can be selected as the combiner for X-coordinate range \(r_{x1}\), so \(M_2\) should send the partial vector that \(r_{i1}\) generates for a query to \(M_1\) when the query specifies X-axis as the aggregation dimension. Likewise, \(M_7\) can be assigned as the combiner for Y-coordinate range \(r_{y1}\), meaning that \(M_1\), \(M_3\) and \(M_5\) send their partial result vectors to \(M_7\) in presence of a query with Y-axis as the aggregation dimension. Once global aggregation phase is performed by combiners, they return results to user and another aggregation query is fetched by master node.

2.3.1 Reducing Communication Volume

Since load balance is automatically achieved, reducing the communication volume is crucial for performance. The communication volume involved for global aggregation phase
depends on the size of X-coordinate range \(|r_x|\) and Y-coordinate range \(|r_y|\) and can be reduced by choosing a data partitioning scheme accordingly. When partitioning underlying data at startup time, each vertical cut introduces an additional \(|r_y|\) amount of communication volume to the global aggregation phase for queries that defines Y-axis as aggregation dimension, whereas each horizontal cut introduces an additional \(|r_x|\) amount of communication volume for queries with X-axis as aggregation dimension. This means that a partitioning scheme with \(c_v\) number of vertical cuts and \(c_h\) number of horizontal cuts results in \(|r_x| \times c_h\) and \(|r_y| \times c_v\) total communication volume for X-axis and Y-axis, respectively.

Since \(|r_x|\) and \(|r_y|\) values depend on dataset and they are known beforehand, this observation can be used to choose a partitioning scheme in favour of the range which has more number of unique coordinate values. For instance, in cases where \(|r_x|\) is much more greater than \(|r_y|\), using more number of vertical cuts is a better option compared to using more number of horizontal cuts, since it would result in less communication volume on average if queries with different aggregation dimensions are seen at the same frequency.

However, in reality, individual frequencies might differ and they must be considered as well when deciding on the partitioning scheme. Assuming that the typical query workload is known in advance and therefore, the frequencies of aggregation along the two dimensions are known, the following methodology can be applied.

We define expected communication volume for a query by the following equation:

\[
V_{comm} = P(X) \times |r_x| \times c_h + P(Y) \times |r_y| \times c_v
\]  

(2.1)

Assuming the probabilities of aggregation along X-axis and Y-axis \((P(X)\) and \(P(Y)\)) are known, choosing a number of vertical cuts and horizontal cuts according to the following ratio minimizes \(V_{comm}\):
\[ \frac{c_v}{c_h} \approx \frac{P(X) \times |r_x|}{P(Y) \times |r_y|} \]  

(2.2)

### 2.3.2 Fault Tolerance Support

As explained in range queries section, the traditional replication scheme for fault-tolerance suggests to replicate a data partition as a whole and store it under a single worker node. However, we have emphasized that such kind of replica generation and storage leads to imbalances in the workload of different worker nodes when there are failure(s). Therefore, to be able to provide fault-tolerance and preserve load balance at the same time, we propose a subpartitioning scheme for aggregation queries by adapting the main idea behind subchunk replication scheme used in range queries. However, unlike range queries, we need to consider both load balance and reduction of communication volume while deciding on a subpartitioning scheme.

We use the following method. First, we divide each data partition into a number of subpartitions \( (M') \) by using a number of vertical \( (c_v') \) and horizontal cuts \( (c_h') \) and replicate the data points in each subpartition. Then, each subpartition replica is assigned to a worker node that is different from the node that stores the parent partition. Thus, at query time, the processing work that would have normally been performed by the failed node is shared among the worker nodes that host the subpartitions of the parent data partition. One special case occurs if the failed node is a combiner that is assigned the task of global aggregation for a certain coordinate range. In such a case, that global aggregation task should be performed by another node in the system. To solve this problem, for each coordinate range, we assign one primary and one backup combiner, which is selected from the corresponding range set if possible.
As a result of initial partitioning and subpartitioning, each subpartition created out of a partition has x-coordinate and y-coordinate ranges with sizes \( \frac{|r_x|}{(c_v+1)(c_v'+1)} \) and \( \frac{|r_y|}{(c_h+1)(c_h'+1)} \), respectively. Since each subpartition is to be assigned to a worker node, in addition to the partial results for its own partition, a worker node also has to send the partial results for assigned subpartitions to corresponding combiners for global aggregation phase. Clearly, this produces an additional communication volume to the query. The total additional communication volume introduced by a subpartitioning scheme for a partition can be written as \( \frac{|r_x|}{(c_v+1)(c_v'+1)} \times M' \) for X-axis and \( \frac{|r_y|}{(c_h+1)(c_h'+1)} \times M' \) for Y-axis queries. Using different number of vertical and horizontal cuts for subpartitioning has an impact on produced additional communication volume. Therefore, \( c_v' \) and \( c_h' \) parameters can be chosen by considering the initial partitioning scheme along with dataset and query characteristics. Moreover, distribution of subpartitions to worker nodes is also important since an intelligent distribution may lead to further reductions in communication volume, as compared to a random one.

**Distribution of Subpartitions:** Here, we assume that \( c_v' \) and \( c_h' \) parameters have been already selected and we focus on subpartition distribution. We can reduce the additional communication volume introduced by a subpartitioning scheme by assigning subpartitions to worker nodes which share the same coordinate ranges. Such an assignment does not change the range of coordinate values stored in these nodes, so they do not need a separate send operation for the partial result vectors generated for subpartitions. For instance, in Figure 2.6, assigning subpartitions of the partition in worker node \( M_2 \) to worker nodes in the range set \( R_{x1} (M_1) \) decreases the communication volume overhead for queries with X-axis as the aggregation dimension. Similarly, assigning the same subpartitions to worker nodes in range set \( R_{y2} (M_4, M_6 \text{ and } M_8) \) provides a reduction in the communication volume.
overhead for queries that specify Y-axis as the aggregation dimension. We define expected 
communication volume overhead ($V_{\text{over}}$) introduced by a subpartitioning scheme for each 
partition by the following expression, capturing the above observation:

$$V_{\text{over}} = \frac{P(X) \times |r_x|}{(c_v + 1) \times (c_v' + 1)} \times (M' - M_x)$$

$$+ \frac{P(Y) \times |r_y|}{(c_h + 1) \times (c_h' + 1)} \times (M' - M_y)$$

(2.3)

In the formula, assuming that the node that stores the parent partition is in X-range set 
$R_{x_i}$ and Y-range set $R_{y_j}$, $M_x$ represents the number of subpartitions that are assigned to the 
worker nodes in $R_{x_i}$, whereas $M_y$ represents the number of subpartitions that are assigned 
to the worker nodes in $R_{y_j}$. The maximum values that $M_x$ and $M_y$ can take are $c_h$ and $c_v$, 
respectively.

Rather than distributing subpartitions randomly, using a distribution which increases 
$M_x$ and $M_y$, and therefore, reduces the communication volume overhead. We use the 
following distribution rule to reduce the value computed by Expression 2.3, irrespective of 
the parameters $M'$, $c_v'$, and $c_h'$:

- **if** $\frac{P(X) \times |r_x|}{(c_v + 1) \times (c_v' + 1)} > \frac{P(Y) \times |r_y|}{(c_h + 1) \times (c_h' + 1)}$, first distribute subpartitions to the worker nodes in 
  $R_{x_i}$, then to the nodes in $R_{y_j}$ and assign the remaining ones to any other node.

- **else if** $\frac{P(X) \times |r_x|}{(c_v + 1) \times (c_v' + 1)} < \frac{P(Y) \times |r_y|}{(c_h + 1) \times (c_h' + 1)}$, first distribute subpartitions to the worker 
  nodes in $R_{y_j}$, then to the nodes in $R_{x_i}$ and assign the remaining ones to any other node.

- **else** distribute subpartitions to worker nodes in $R_{x_i}$ and $R_{y_j}$ in round-robin fashion.
2.4 Experimental Evaluation

In this section, we present results from a number of experiments we conducted to evaluate our framework. All experiments are performed on a cluster comprising of machines with two quad-core 2.53 GHz Intel(R) Xeon(R) processors with 12 GB RAM running under Red Hat Enterprise Linux Server release 6.1, and Gigabit ethernet as the interconnect.

In all our experiments, we compare the performance of our schemes against default schemes, which include those with no support for fault-tolerance and those with chunk-level/partition-level replication. Chunk-level/partition-level replication is used in other systems, though we did not find a suitable system for comparison. For example, Hadoop, the open source version of MapReduce, is designed for data processing, and not query handling. It does not have any built-in indexing support, neither does it target short-running queries.

2.4.1 Evaluation for Range Queries

For range queries, we compared two different versions. The first version has no fault-tolerance support, implying that there is no data replication and the system is not able to function under a failure. On the other hand, the second version has data replication support and is able to serve queries even under single machine failure and rack failure scenarios. For each experiment, we use a spatial dataset with arbitrary rectangle sizes randomly positioned on data space. Unless noted otherwise, this dataset consists of 1 billion data objects and 1000 queries are executed on it. We report aggregate execution times over these 1000 queries.
(a) Query Processing Times with Varying Chunk Size (64 Worker Nodes)

(b) Query Processing Times with Varying Number of Worker Nodes (chunk size = 10000)

Figure 2.7: Execution Time with No Replication and No Failures
**Normal Execution:** With the first experiment, we aim to see the impact of using different chunk sizes (the number of objects in a chunk) and number of worker nodes on query performance when there is no failure in the system. Figure 2.7(a) illustrates the execution time of the version with no fault-tolerance support. 64 worker nodes are used for parallelism and each query spans 5% of the data space. The time is divided into three components: time spent in processing the Hilbert R-Tree at the master node (initial pruning), denoted as HR-Tree, time spent for communication between master and workers (Comm Time), and time spent at worker node for processing unfiltered chunks (Worker Time). Clearly, using a small chunk size to organize the data objects results in a vast number of chunks and increases the overheads introduced by processing Hilbert R-Tree and communication. As the chunk size is increased, execution time decreases up to a certain point, but increasing it beyond 10000 objects impacts load balance negatively and reduces the pruning rate of unrelated regions. Therefore, setting chunk size parameter to a moderate value during the initial configuration of the system leads to better overall performance.

Figure 2.7(b) illustrates the scalability of the system by using the same experimental setup in (a). This time, we fixed the chunk size to 10000 objects and measured the execution time when different number of worker nodes are used. We can see that there is a near-linear scalability.

Next, we measure the overheads introduced by the version with fault-tolerance support when there is no failure. We ran various query workloads with different selectivity values by employing different number of worker nodes. Since failure handling protocols do not interfere with the normal execution of the system, we haven’t observed any statistically meaningful overheads. We do not include figures with detailed results because of space limitations.
Single-Machine Failures: Next, we compare the performance of the traditional replication scheme (chunk replication) and the subchunk replication scheme we have introduced when the system has a single machine failure. For the traditional approach, we assign the replicas generated for the data stored in each node to another node in entirety. Figure 2.8 reports the execution times for chunk and subchunk replication schemes when chunk size is varied. 64 worker nodes are used for parallel query processing and each query spans 20% of the data space. For each chunk size, we show the effect of using different number of subchunks ($k$) on the subchunk replication scheme. We also include the time when there is no failure as a baseline. Note that chunk replication is represented by $k = 1$ in the figure. Our subchunk replication approach outperforms chunk replication in all of the configurations. Setting $k$ to a larger value implies that more subchunks are created out of each chunk. This provides a better load balance when there is a failure. When chunk size is 10000, which gives
Figure 2.9: Failure Overhead Ratio for Different Replication Schemes with Varying Selectivity.

the best result for 5% selectivity (Figure 2.7(a)) and the second best for 20% selectivity, our subchunk replication scheme with \( k = 20 \) provides a 46% improvement over chunk replication.

Figure 2.9 shows the ratio of failure overhead of our subchunk replication scheme to the failure overhead introduced by chunk replication scheme when varying \( k \). Here, again, we use 64 worker nodes for parallelism and set the chunk size to 10000 objects. We make the same overhead comparison with three different selectivities; 5% (“S 5%”), 20% (“S 20%”) and 40% (“S 40%”). Ideally, the overhead ratio should follow the line represented with “Ideal (1/k)” in the figure, since dividing a chunk into more subchunks should provide a reduced overhead that is projected by the line \( 1/k \). However, in practice, using more number of subchunks can introduce some extra costs and increase the overheads. The results shows that, for different values of \( k \), the lines for 5% and 20% selectivities almost follow the ideal line. The overhead ratio with 40% selectivity is even smaller than the projected
ideal values for the first few $k$ values along the x-axis. The reason of this behaviour is the following. When chunk replication is used, a worker node simply performs the intersection test between query and a replica chunk by using chunk’s MBR that covers a larger area than MBR’s of subchunks for that same chunk would cover. Therefore, when subchunk replication is in use, not only the load balance among workers improves, filtering rate (the percentage of chunk and subchunks that fail the intersection test) at worker nodes also goes up, due to tighter MBRs. This effect shows itself better at higher selectivity values since worker nodes need to process more and more number of chunks.

**Rack Failures:** We next evaluate the performance of our subchunk replication scheme under rack failure scenarios. For this experiment, we set chunk size again to 10000 objects and vary the rack size to see its effect on our scheme. Each query spans 20% of the space.
Figure 2.11: Execution Times and Analytical Trend with Different Partitioning Schemes. $|r_x| = |r_y| = 10000$, $P(X) = P(Y) = 0.5$, 1000 queries, no failures, 1 billion data objects and 64 worker nodes.

Figure 2.10 shows the results for chunk replication scheme ($k = 1$) and subchunk replication schemes when $k$ is equal to 2, 5, 10, and 20. We also included the time under normal execution as a baseline. When rack size is small, execution time decreases with increasing $k$. However, in cases where rack size is large, large $k$ values do not improve performance further since the overheads such as processing Hilbert R-Tree and communication between master and workers start to dominate. Nevertheless, our subchunk replication scheme continues to outperform the chunk replication scheme in every configuration.

2.4.2 Evaluation for Aggregation Queries

**Normal Execution:** The first experiment we conducted for aggregation queries considers the impact of different partitioning schemes on query performance when there is no failure in the system. In Figure 2.11, we perform aggregation queries on each dimension at the
same frequency \((P(X) = P(Y) = 0.5)\) and compare the execution times of different partitioning schemes when the range of coordinate values on each axis \((|r_x| \text{ and } |r_y|)\) are also the same. We report the execution times with bars and also show the expected communication volume defined in Expression 2.1 for each partitioning scheme with a line. Each time bar is further divided into two; execution time for queries with X (“X-Agg”) and Y (“Y-Agg”) as the aggregation dimension. As expected, the extreme partitioning schemes such as \(64 \times 1\) and \(1 \times 64\) show the worst performance since each leads to a large communication volume along one of the aggregation dimensions. The partitioning scheme \((8 \times 8)\), which is also suggested by the ratio defined in Expression 2.2, provides the best performance for this dataset and query configuration. For other dataset configurations with varying \(P(X)\),

Figure 2.12: Execution Times with Different Partitioning Schemes. \(|r_x| = |r_y| = 100000, P(X) = P(Y) = 0.5, 1000 queries, single machine failures, 1 billion data objects, 64 worker nodes.
we observed that the expected communication volume is a good indicator that can help to select a suitable partitioning scheme. Again, for space considerations, we haven’t included the corresponding figures.

**Single-Machine Failures:** Next, we compare the performance of traditional replication scheme (partition replication) and subpartitioning replication scheme when there is a failure in the system. For partition replication, we replicate the data stored in each partition and store it under a different worker node in entirety. Figure 2.12 illustrates the execution times of the different replication schemes when the number of subpartitions ($M'$) is varied. For each $M'$, the first two bars correspond to times without failure and with failure when partition replication is used, respectively. We define the trivial subpartitioning schemes $4 \times 1$, $1 \times 4$ for $M' = 4$ and $1 \times 8$, $8 \times 1$ for $M' = 8$ as “1D”. For each $M'$, we only show the best one among them. Similarly, we refer the subpartitioning schemes $2 \times 2$, $4 \times 2$ and $2 \times 4$ as “2D”. For both of 1D and 2D subpartitioning, we labeled the ones that distributes subpartitions randomly with “Rand” and the ones that use rule-based distribution with “Rule”. Figure shows that regardless of the subpartitioning scheme and distribution, our subpartitioning replication scheme outperforms partition replication, and rule-based distribution provides improvements over random-distribution in each case. Here, an interesting observation is that increasing number of subpartitions from 4 to 8 does not necessarily lead to speedups, unless the subpartitions are distributed by our rule-based distribution method.

### 2.5 Related Work

We now compare our work with related efforts on parallel query processing and scientific data processing and management.
Parallelization of query processing is an active topic for at least the last two decades [46, 66]. Initial work [74, 75] focused on efficient data-parallel spatial join algorithms and use of different spatial data structures such as PMR Quadtrees, R-trees and R+-trees. PARADISE project [47] emerged as a large-scale effort that designs and implements a database system to handle GIS type of applications. Another study [23] examined different approaches for a parallel spatial join on a hybrid of shared-nothing and shared-memory architectures. There have been a number of efforts on providing good data declustering for multiple disks to improve parallel I/O [52, 16, 55, 85]. None of these efforts have considered fault-tolerance or data replication. Parallelization of aggregation queries has also received attention. Shatdal et al. [128] propose three different algorithms for parallel processing of aggregation task expressed through SQL queries. Focusing on aggregation-join queries, another study [138] provides three parallel methods depending on the attribute that is used in partitioning of data. Gao et al. [61] present several parallel main-memory based algorithms to perform aggregation operations on a temporal database. Once again, none of these works involve any kind of fault-tolerance mechanism for the system to stay operational in case of failures. One effort toward handling node unresponsiveness has been given by Chu et al. [35]. This study is in context of cluster based Internet services and employs a timeout scheme to prune slow or unresponsive nodes.

MapReduce paradigm [40] provides a more convenient failure protocol. In Hadoop [144], the open-source version of Map-Reduce, each file is split into blocks and replica(s) are created for each block, and distributed over the cluster. When a computing node becomes inoperable, the task at the failed node is restarted at an alternate node. Several studies (e.g. [84]) indicate that a single machine failure in Hadoop may cause a 50% increase in completion time. To reduce this overhead, another effort [148] proposes a passive
task-replication scheme, where, for a map task, a number of backup nodes are kept in ready state to interfere in case of any failures. Yet another effort [17] addressed same problem on a Map-Reduce variant called FREERIDE. Another set of studies on Map-Reduce Fault-Tolerance [15, 5] focus on Byzantine failures, where the nodes produce incorrect/inconsistent outputs due to the corrupted local state or incorrect processing of requests. In either case, the solutions developed in the context of map-reduce do not apply to data management and processing done outside of a map-reduce implementation. MapReduce, as we stated earlier, is not meant to support the queries we have considered.

Database community also has been working to fulfill the special needs required by large-scale data-intensive science applications and many prototypes have been built towards this goal [133, 80, 135]. However, none of these projects define any failure handling protocols.

2.6 Summary

In this work, we presented the design of a fault-tolerant environment that can process queries and data analysis tasks on data hosted in a large cluster. Our framework supports range queries on spatial data and aggregation queries on point datasets, but the proposed approaches can be extended to other query types and data analysis tasks as well. Besides delivering high efficiency under normal execution, our design enables the system to be operational when there is a single-machine failure or a rack failure. We use subchunk and subpartition replication, which allows us to preserve load balance among the remaining nodes in the presence of failures. Experimental evaluation shows the effectiveness of our replication and distribution schemes over the more traditional or basic replication approach (chunk replication and partition replication).
Chapter 3: DISC: A Domain-Interaction Based Programming Model
With Support for Fault-Tolerance and Heterogeneous Execution

In previous chapter, we focused on big data processing platforms and presented a fault-tolerant large scale query processing environment against fail-stop failures. In this chapter, we direct our attention to Single Program Multiple Data paradigm and propose a new parallel programming model with support for heterogeneous execution and automated fault-tolerance against fail-stop failures and soft errors. As we move towards the next generation of supercomputing, resilience and heterogeneity in computing units are becoming a norm, rather than an exception. Existing programming models, especially for distributed memory execution, typically have been designed to facilitate high performance on homogeneous systems, with no consideration for resilience. For scientific computations, we do not have programming models that can both handle inter-node heterogeneity and failures, while maintaining programmability and efficiency. We argue that, with a set of proper programming abstractions, a programming model can automatically guard applications against both fail-stop failures and soft errors, and also can deal with heterogeneity in computing platforms.
This work focuses on the above two goals and describes a programming model and associated runtime system we have developed to address the above needs. Proposed programming model is based on Domain and domain element Interaction concepts and particularly targets scientific applications. It is based on the observation that certain patterns or templates, like structured and unstructured grids, and particle interactions are extremely common in scientific computing. While these applications have different communication patterns, they are similar in an important way, i.e., they have an underlying domain, and most of the computation occurs due to the interactions among domain elements. Our programming model supports an API by which the domain, interaction among domain elements, and functions for updating any attributes of these domain elements can be explicitly specified. Starting from this model, inter-process partitioning of the work and the communication is handled automatically by the runtime system.

Our programming model offers several benefits over other models. First, because the partitioning of work and data are implicit, work can be partitioned among a heterogeneous collection of resources, and even re-partitioned on the fly. In comparison, with a programming model like MPI, work distribution tends to be static, making it extremely hard to deal with heterogeneous resources. Second, the simplified API makes it feasible to support automated, yet efficient, application-level checkpointing. This can reduce checkpointing overheads, while also alleviating the need for programmers inserting checkpointing and restart logic. Third, programming abstractions within the model expose the data structures and computations that are the most susceptible to soft error occurrences. Hence, soft errors are detected efficiently with low overheads by using a partial replication strategy.

For clarity, we divide our contributions with this programming model into two, which are explained in this chapter and the next. In the remaining sections of this chapter, we
first introduce the basics of our programming model and show how we achieve the first benefit, that is automatic work partitioning/re-partitioning on the fly in presence of heterogeneous processing elements. We extend our programming model and explain how the set of abstractions intrinsic to this model leverage low-overhead fault-tolerance support in Chapter 4.

3.1 A Domain-Interaction Based Model

As stated earlier, our goal is to create a high-level programming model that can support automated resilience and work distribution for heterogeneous computing. At the same time, we want the programming model to not require extensive compiler support (since providing a robust and efficient implementation for such models tends to be extremely challenging). Rather than supporting a truly general model, we focus on supporting a model in which most popular scientific applications can be expressed. To consider the needs of scientific applications, we take inspiration from several popular scientific computing dwarfs that were summarized by Phil Collela (and further generalized in Berkeley’s landscape on parallel computing). Their observation has been that scientific applications follow one of the seven templates or patterns: structured grid or stencil computations, unstructured grids, N-body simulations, generalized reductions, dense linear algebra, sparse linear algebra, and spectral methods (FFTs).

Within this set of templates or patterns, we further observe that standard libraries are available and popularly used for dense and sparse linear algebra and FFTs. Thus, our claim is that a very large fraction of popular scientific applications developed today using a general purpose parallel model (e.g. MPI or one of the PGAS languages) involves
one or more of the following patterns: structured or unstructured grids, generalized reductions, or N-body interactions. Note that in recent years, there have been several efforts that have focused on one of these classes, e.g. MapReduce for generalized reductions [41], domain-specific work targeting stencil computations [34, 137], and irregular reductions [45], though we note that none of these works have, with the exception of the work on MapReduce, considered work redistribution across heterogeneous units.

Our goal is to create one framework for all these classes of applications. Particularly, we observe that despite a vast difference in underlying communication patterns, there are important similarities across these distinct patterns. First, applications involving stencil computations, unstructured grids, or N-body simulations consistently have an iterative structure. Second, many of key data-structures in them are used to store values associated with points (or cells or particles) in a multi-dimensional domain. This domain might represent a dense two or three-dimensional grid, as in stencil kernels, or it might consist of points which are sparsely located over the grid, as in unstructured grid based computations. Finally, there exists some form of interaction among the points forming the domain. The nature of the interaction might be defined explicitly by using additional data-structures (such as a list of edges, or a cut-off distance), or it might be implicitly implied from the location of grid cells. Most of the execution time is spent on calculating new values for point attributes by considering these interactions and performing the necessary updates. For instance, in a molecular dynamics simulation, each point is considered to be interacting with points within a certain radius, and coordinate and other attributes of the point is updated based on these interactions during each iteration. Similarly, in stencil computations, each grid point is updated with weighted contributions from its nearest neighbors (surrounding grid points).
Now, let us consider the implementation of these applications using one of the popular parallel programming models today, like MPI or any of the PGAS models. Considerable amount of effort is devoted to all or most of the following steps: bookkeeping the initial data partitioning and task assignment, identifying set of data instances that must be sent/received for data exchange, preparing (initializing, packing, and unpacking) input and output communication buffers, placing lines of code for actual communication, and so on. Besides the complexity of these steps, with such programming style, it is not feasible to repartition the work across processes if there are differences in processing speed. Thus, our premise is that a programming model that can hide details of data distribution and communication from programmers, and perform operations such as data exchange automatically, will not only simplify parallel programming, but can also help provide data and work redistribution if the processes are heterogeneous.

We now elaborate on the key concepts of our programming model.

### 3.1.1 Domain and Subdomain

Our programming model treats the entire input space as a multi-dimensional *domain*, which consists of *domain elements*. For an application to be implemented by our model, programmers should first provide pertinent information about the domain. This information is used to initialize the runtime system and it includes 1) whether the domain represents a structured grid, an unstructured grid, or a particle set, 2) number of dimensions and boundary values for each dimension and 3) the type of the interaction among domain elements. Programmers can let DISC automatically decide on the initial partitioning, or provide the preferred strategy (e.g. along a certain dimension or multiple dimensions) to the runtime system. After initialization is done, the underlying runtime system decomposes the entire
domain into non-overlapping regions (subdomains) and assigns each region to a unique processing unit. Since the subdomain decomposition and assignment is performed by the runtime, it is able to hold a high-level view of the entire domain. Figure 3.1 illustrates such a view for a sample application with points in a 2D-domain.

As a concrete example, consider a molecular dynamics application such as MiniMD which simulates the motion of a large number of atoms in three-dimensional space throughout a predefined number of time-steps. When implemented using DISC model, the three-dimensional space is treated as an N-body simulation domain and each atom in the simulation corresponds to a domain element. Figure 3.2(a) shows a code snippet that initializes a 3D simulation domain on DISC for MiniMD.

### 3.1.2 Attributes

Each domain element in a DISC domain has associated coordinate values. In some domain types such as structured grids, coordinate values of domain elements might stay fixed during the entire execution and can be inferred directly from the boundary values of...
// provide domain information and initialize DISC runtime
DomainProps props;
props.set_ndims(3); // number of dimensions
props.set_min_bounds(0, 0, 0); // x, y, z min—bounds
props.set_max_bounds(XMAX, YMAX, ZMAX); // x, y, z max—bounds
props.default_partitioner(true); // let DISC partition the domain
props.set_interaction(RADIUS_BASED, 2.5); // radius based interaction pattern with
cutoff distance of 2.5
NBodyDomain domain(props);

(a)

DoubleAttribute velocities[3]; // x, y, z velocities
/* fill in attribute object velocities with initial values of x, y, z velocities */
domain.register_attributes(&velocities);

(b)

DoubleAttribute computation_space[6]; // new x, y, z coords and velocities
domain.add_compute_function(minimd_kernel, &computation_space);

// apply minimd—kernel for a number of iterations
for(int it = 0; it < N_ITERATIONS; it++) {
  // process current iteration
  domain.run_iteration();
  ...
}

(c)

Figure 3.2: Example code snippets for DISC (a) domain initialization, (b) attribute registration and (c) compute-function and computation-space object coupling and the main loop for a 3D molecular dynamics code.
assigned subdomains. However, for other domain types, they might be updated periodically and their initial values should be explicitly provided by programmers. In addition to coordinates, each domain element can also be associated with a set of attributes. For instance, each atom in MiniMD has three additional attributes that store velocity values of the corresponding atom on x, y and z axis. The key advantage of DISC model is its ability to perform data exchange operations based on the interaction pattern automatically and to re-partition the domain on the fly in presence of heterogeneity by migrating domain elements within the domain. To fulfill both of these promises, programmers register coordinates and attributes of domain elements within each subdomain via DISC API, so that the runtime is informed of the data structures that maintain associated information on each domain element. Figure 3.2(b) shows how attributes of domain elements in MiniMD are passed to the runtime through DISC objects called DoubleAttribute.

### 3.1.3 Interaction Between Domain Elements

The key idea of the programming model is that an application developer specifies the interactions between the domain elements, and communication is automatically handled by the runtime system based on these interactions. The nature of interactions among entities
within a domain can vary across applications. Figure 3.3 demonstrates some of the commonly used interaction types found in scientific computations. In stencil computations, each element is updated with weighted contributions from a set of its nearest neighbors (grid-based interactions). In a molecular dynamics application, the interactions among points are defined by a cutoff radius $r_c$, and each point interacts with only the points within that radius (radius-based interaction). Yet another type of interaction arises when an unstructured grid is used. Here, rather than a nearest-neighbor or a cutoff radius, the set of point interactions is defined explicitly by a data structure typically specifying the set of edges (explicit-list interaction).

3.1.4 Compute-Function and Computation-Space

In DISC model, each processing unit performs computations for the assigned portion of the domain. In other words, the domain elements that a processing unit processes lie within the boundaries of the subdomain that has been assigned to it by the runtime. DISC requires programmers to express underlying computation, which typically comprises of calculating new values for attributes associated with domain elements, in a single or a set of functions referred as compute-functions. Compute-functions generally host the portion of code on which most of the execution time is spent. Programmers specify these functions by passing function pointers to the runtime. At each iteration during a program’s execution, the runtime invokes these functions in the order that they are specified.

For each compute-function, programmers explicitly declare one or more objects called computation-space. A computation-space object coupled with a compute-function stores the results of computation carried out by that function. It generally contains an entry for
each domain element in the corresponding subdomain and programmers perform any updates related to the domain elements directly on the computation-space object itself. This way, the runtime is aware of what additional data structures along with coordinates and attributes describe the domain elements in a subdomain completely. This abstraction leverages automated migration of domain elements within the domain if needed. Note that mapping a value in computation-space to the corresponding domain element can be inferred from domain type in most cases. Otherwise, programmers can pass additional functions to the runtime that dictate this mapping.

In MiniMD, atoms interact with other atoms in a given radius and this interaction results in recomputation of coordinates and velocities of each atom at every time-step. The code snippet in Figure 3.2(c) reflects this by defining six computation-space objects (three for new coordinates and three for new velocities). These objects are coupled with the compute-function \texttt{minimd\_kernel} and passed to the runtime via DISC API.

### 3.1.5 Fault-Tolerance Support

The notion of compute-functions and computation-space objects along with attributes of domain elements play an important role in realization of fault-tolerance support within the DISC programming model. With the abstractions that they provide, these concepts help revealing both the most critical portion of computations and also the core execution state in an application. Particularly, computation-space objects and attributes leverage low-cost and automated application-level checkpointing since they represent the current state of execution. Moreover, since the results of computations expressed in compute-functions are directly reflected on computation-space objects, redundant execution of compute-functions and comparison of replicas reveal any silent data corruptions impacting the execution state.
The details of these fault-tolerance solutions are presented in Chapter 4 along with a comprehensive evaluation of associated overheads.

### 3.2 Runtime Communication Generation from Domain-Interaction API

As we have stated earlier, the key underlying idea and innovation in the programming model is the automatic and runtime management of the communication using the knowledge of the interactions between the domain elements. This section focuses on how this is accomplished.

Note that despite the difference in the nature of interactions across applications, one common feature in all of the applications we target is as follows. Given one subdomain, at the beginning (or the end) of an iteration, one needs to acquire new values of all outside points that any local point from this subdomain might be interacting with. Based on the knowledge of domain and subdomains, as well as the interactions, our runtime system is able to perform such communication and place the updated values of coordinates & attributes of interacted points in a structure for programmer’s use. Towards this goal, for each process, the runtime system should first identify the data points that should be sent to other processes. The process to determine these points and the communication routine used for exchange depends on the interaction type.

#### 3.2.1 Grid-based Interactions

For this interaction type, each processing unit needs the grid points that are located immediately on the outside of the boundaries of its subdomain. To obtain these data points, the runtime system performs a single exchange operation with each of its immediate neighbors; first with *west* and *east*, and then with *south* and *north* (for a 2D-domain). Once
// px, py: indices of subdomain S in processor grid
// neighs[][]: holds boundaries of every neighbor
// (neighs[px][py] holds the boundaries of subdomain S)
// rc: cutoff distance
setup_exchange_west(double rc) {
    // number of exchanges with west neighbor, initially 0
    nexchanges=0; i=0;
    // max. x−coord of points that west neighbor is interacting
    wb_xmax=neighs[px][py−1].xmax+rc;
    do {
        // lowest x−coord of points that S should send in this exchange
        bounds[nexchanges].lo=neighs[px][py+i].xmin;
        // highest x−coord of points that S should send in this exchange
        bounds[nexchanges].hi=min(neighs[px][py+i].xmax, wb_xmax);
        nexchanges++; i++;
    } while(wb_xmax >= neighs[px][py+i].xmax);
}

Figure 3.4: setup_exchange_west() function (called by runtime system) which determines the x-coordinate range of points that subdomain S must send to its west neighbor at each exchange, when radius-based interaction is used.

the runtime system receives these nearby interacting data points from the surrounding subdomains, it places the corresponding values in existing data structures in such a way that preserves the regular access pattern of stencil kernels and leverages data locality.

3.2.2 Radius-based Interactions

The data exchange routine here is different than the routine carried out for grid-based interactions, since each subdomain needs data points in a box that extends through \((x_{\text{min}}−r_c, x_{\text{max}}+r_c)\) on x and \((y_{\text{min}}−r_c, y_{\text{max}}+r_c)\) on y-dimension, where \((x_{\text{min}}, x_{\text{max}}, y_{\text{min}}, y_{\text{max}})\) indicate the assigned boundaries of the subdomain and \(r_c\) is the cutoff distance.

For this interaction type, each subdomain is aware of the boundaries of its neighboring subdomains, and thus, implicitly, the range of data points that it should send/receive to/from a neighbor. The corresponding algorithm (a similar approach used in [108]) works in two
steps for a 2D-domain (three steps for 3D-domains), with each step applied on a single dimension. In the first step, each subdomain $S$ sends the points that its immediate west and east neighbors are interacting with. Assuming $l_x$ (equal to $S.x_{\text{max}} - S.x_{\text{min}}$) is the length that $S$ spans on x-dimension, if $l_x > r_c$, it is sufficient that $S$ only sends its own local points in the x-range ($S.x_{\text{min}}, S.x_{\text{min}} + r_c$) to the west, and local points in the x-range ($S.x_{\text{max}} - r_c, S.x_{\text{max}}$) to the east neighbor. Similarly, $S$ receives the points that it is interacting with from these neighbors. However, if $l_x \leq r_c$, $S$ should make more than one exchange, since some of the points that $S$ is responsible to send would be present in a subdomain other than its immediate surrounding neighbors. In that case, at each subsequent exchange, $S$ sends a neighbor some of the points that it obtains from the opposite neighbor in previous exchange. For instance, at second exchange, $S$ should send the west neighbor the points in the x-range ($S.x_{\text{max}}, S.x_{\text{min}} + r_c$), which are received from the east neighbor at first exchange. The same logic also works in the opposite direction with the east neighbor. Figure 3.4 shows the routine that a subdomain $S$ uses to find the total number of exchanges and the x-range of the points that it should send to the west neighbor at each exchange. In second step of the algorithm, $S$ follows the same communication pattern with its north and south neighbors and the algorithm completes.

When the runtime system gathers these data points from neighboring subdomains, it appends their coordinates and attributes to the end of the same data structures that initially hold the coordinates and attributes of registered local points in the subdomain.

### 3.2.3 Explicit-list based Interactions

Such interactions are usually provided by an input dataset in the form of number pairs. Each number in such a pair corresponds to a unique identifier for a data point and implies
that there exists an edge (interaction) between those points. Therefore, in contrast to the previous two types, explicit-list based interactions do not depend on the spatial configuration of the data points. To support such interactions, programmer can provide a unique global id-number for each data point during the local point registration process. After registration, for each local point, the points that it interacts with can be passed to runtime system by invoking an API function (currently via `disc_add_interaction()` function), which takes three arguments; 1) global id-number of local point, 2) global-id number of interacted point, and 3) coordinates of interacted point. Once the programmer adds all of the interactions, runtime system automatically determines the subdomains that contain the interacted points by comparing point coordinates with boundaries of the subdomains and performs data exchange with those subdomains during actual communication. Thus, as another major difference from the first two type of interactions, communication for explicit-list based interactions might occur with any of the subdomains, not just the neighbors. After updated values of interacted points are obtained from corresponding subdomains, they are appended to the end of existing data structures, as in radius-based interactions. Currently, DISC runtime only supports static unstructured meshes, in which explicit interaction lists are not updated once execution starts.

### 3.3 Automatic Repartitioning of Domains

A key aspect of our programming model is automated partitioning of the domain for array-based computations, to allow efficient execution on heterogeneous collection of cores and/or nodes. The motivation, as we stated earlier, arises because of static as well as dynamic variations. Static variations, where different nodes and/or cores have different processing power, but the power does not change during execution, are more likely because of
smaller feature sizes, greater heterogeneity due to possible use of low-voltage execution, and usage of an accelerator or co-processor. Dynamic variations can arise as power optimizations are applied at each node during execution. Even with static variations, the key challenge for the programmer is that variations are not known before execution, and cannot be (and should not be) hard-coded in the applications. If static variations occur, partitioning just needs to be performed once, at the start of the execution. If dynamic differences arise, repartitioning needs to be performed periodically.

We have developed techniques for (re-)partitionig scientific computations based on multi-dimensional domains for heterogeneous settings. As we have stated earlier, existing work that can handle such heterogeneity relies either on over-decomposition [81] or needs a task model for specification of the computation [30]. The former incurs substantial overheads (which we will demonstrate experimentally), while the latter is not a natural way to express scientific computations involving communication. In our approach, the DISC runtime system maintains the load balance in the system by shrinking the area of
subdomains that are assigned to slower nodes/cores and sharing this extra-work among the remaining processing units. However, while doing so, it maintains rectilinear partitions, which is important for keeping communication manageable.

Our partitioning approaches can be applied either once at the start of the computation (if heterogeneity is static) or periodically (if relative speeds of processors may change dynamically). In either case, the master node collects two pieces of information from each processing unit $i$: the time spent on main computation, $T_i$, and the number of local points in its subdomain, $n_i$. Once gathered, master node computes a unit processing time ($t_i$) for each process by simply dividing $T_i$ with $n_i$. When differences are static, this information may be obtained in the beginning using profile runs. The calculated unit times are a good indicator of the performance of a processing unit, and they can be utilized to decide if an assigned subdomain should be expanded, or shrunk.

### 3.3.1 Repartitioning for 1D-Decomposition

The method used to preserve load-balance for domains partitioned along a single dimension (1D-Decomposition) is simple. The size of each subdomain along the partitioning dimension needs to be inversely proportional to its unit processing time $t_i$. Consider the example in Figure 3.5(a). In subdomain assignment, the problem reduces to finding new values for $x_{r1}, x_{r2}, x_{r3}, x_{r4}, x_{r5}$, and $x_{r6}$. These new values are calculated by the expression:

$$x_{ri} = x_r \times \frac{1/t_i}{t_{sum}}$$  \hspace{1cm} (3.1)

where $x_r$ is the length of the total range on X-dimension and $t_{sum}$ is defined as $\sum_{i=1}^{6} \frac{1}{t_i}$.
3.3.2 Repartitioning for 2D or 3D-Decomposition

Expanding/shrinking subdomain boundaries in 2D (or 3D) Decomposition is different than one-dimensional case in the sense that it is highly desirable to maintain rectilinear partitions, because communication can become extremely complex otherwise. The need for maintaining rectilinear sections leads to interesting tradeoffs. For instance, in Figure 3.5(b), to increase the subdomain assigned to processing unit \( P_1 \), we can increase either \( x_{r1} \), the interval that it expands on X-axis, or \( y_{r1} \), the interval that it expands on Y-axis. However, the change that we make on a subdomain boundary also affects the boundaries of other subdomains. For instance, increasing \( x_{r1} \) also increases the workload of processing unit \( P_4 \).

We have developed a novel approach for handling this problem, which uses a non-linear optimizer. Specifically, the subdomain assignment problem in Figure 3.5(b) can be expressed as a non-linear optimization problem, where we have the objective:

\[
\min \ T_{\text{max}} \tag{3.2}
\]

with the following constraints:

\[
\begin{align*}
    x_{r1} \times y_{r1} \times t_1 &\leq T_{\text{max}} & x_{r2} \times y_{r1} \times t_2 &\leq T_{\text{max}} \\
    x_{r3} \times y_{r1} \times t_3 &\leq T_{\text{max}} & x_{r1} \times y_{r2} \times t_4 &\leq T_{\text{max}} \\
    x_{r2} \times y_{r2} \times t_5 &\leq T_{\text{max}} & x_{r3} \times y_{r2} \times t_6 &\leq T_{\text{max}} \\
    x_{r1} + x_{r2} + x_{r3} &\leq x_r & y_{r1} + y_{r2} &\leq y_r \\
    x_{r1}, x_{r2}, x_{r3}, y_{r1}, y_{r2} &> 0
\end{align*}
\]
Figure 3.6: (a) The initial even partitioning of a grid structure with 10000 rows and 15000 columns among 6 processing units. Unit processing time of each unit $P_i$ is denoted by $t_i$. (b) The resulting partitions after automatic repartitioning is performed. Dashed lines show the previous position of each boundary, and red arrows show the direction that it has been shifted.

In other words, the goal is to create rectilinear sections, such that the maximum across the processing times for all processors is minimized. Now, the challenge is in solving such a problem at runtime, where efficiency is important. We first invoke an external software, AMPL [59], which is a well-known mathematical programming software to formulate optimization problems. AMPL calls appropriate solvers (in our case, we used MINOS) to find the optimal solution(s) for the model and returns the results to the master node. After receiving the results, the master node decides on (new) boundaries of subdomains and informs all processing units in the system.

3.3.3 Example Scenario

As an illustration of the re-partitioning decisions made by the non-linear optimizer, we present the following hypothetical scenario. Figure 3.6(a) shows the original partitioning of a grid structure with 10000 rows and 15000 columns to six equal partitions. Suppose now,
out of 6 processing units, \( P_1 \) runs 50% and \( P_4 \) runs 17% slower than the remaining units and this difference in processing power is captured by unit processing time values \( t_1 \) and \( t_4 \) as shown in the figure. Our re-partitioning scheme generates the following formulation for such a case by relaxing the problem:

\[
\begin{align*}
  x_{r1} \times y_{r1} \times 6.0 & \leq T_{max} & x_{r2} \times y_{r1} \times 5.0 & \leq T_{max} \\
  x_{r3} \times y_{r1} \times 5.0 & \leq T_{max} & x_{r1} \times y_{r2} \times 10.0 & \leq T_{max} \\
  x_{r2} \times y_{r2} \times 5.0 & \leq T_{max} & x_{r3} \times y_{r2} \times 5.0 & \leq T_{max} \\
  x_{r1} + x_{r2} + x_{r3} & = 15000.0 & y_{r1} + y_{r2} & = 10000.0 \\
  x_{r1}, x_{r2}, x_{r3}, y_{r1}, y_{r2} & > 0
\end{align*}
\]

Figure 3.6(b) illustrates the new values for \( x_{r1}, x_{r2}, x_{r3}, y_{r1}, y_{r2} \) and the new partitions. To minimize the objective function given in Equation 3.2, the workload of processing units \( P_1 \) and \( P_4 \) are first decreased by moving the leftmost vertical boundary towards left and the rest of the workload is distributed evenly among the remaining units. Next, since \( P_1 \) is faster than \( P_4 \), its corresponding workload is slightly increased by shifting the bottom most horizontal boundary up and reducing \( P_4 \)'s workload further.

Note that the repartitioning scheme described above will likely not work well when data points are placed non-uniformly on space. For applications or datasets where this is the case, we can improve the method as follows. We can maintain a data structure similar to a heat-map, which holds the number of points at different regions of domain by using a
Algorithm 2 Operations performed by the DISC runtime system in main processing structure when run_iteration() function is invoked by the programmer.

```c
Stats* runtime_stats;
// perform Step 1 and Step 2 as many times as requested
while needMoreComputation() do
    // Step 1. perform automatic data-exchange
    if interaction_type==grid_based then
        exchange_near_neigh();
    else if interaction_type==radius_based then
        exchange.spatial();
    else if interaction_type==explicit_list then
        exchange_selective();

    // Step 2. perform main-computation
    runtime_stats=run_compute_functions();

    // Step 3. run load-balance mechanism
    if isTimeToLoadBalance(load_balance_freq) then
        set_new_boundaries(runtime_stats);
```

binning approach. This data structure can be used to determine the real impact of moving a particular boundary.

### 3.4 Implementation: Putting it Together

In our current implementation, DISC programs follow SPMD structure, with each process having a main loop that initiates the communication and computation tasks for the application, as shown in Figure 3.2(c). In each iteration of the main loop, a set of functions is invoked by a specific processing structure in a particular order to carry out the main computational tasks and maintain the load-balance. This processing structure is executed by the runtime system when run_iteration() function, shown by the pseudocode in Algorithm 2, is called within the application code.
The first operation (Step 1) performed in the processing structure is the data exchange to obtain the interacting domain elements residing in other subdomains. In order to perform the exchange, the runtime system first checks the type of interaction (grid-based, radius-based, or explicit-list based) that exists among the domain elements and invokes the corresponding routine. All low level details, such as initializing, packing, and unpacking input/output communication buffers, are taken care of by the invoked function automatically. As a result of this step, coordinates of the interacting domain elements along with any other registered attributes are placed in a data structure by the runtime system, so that they can be accessed by the compute-function(s). After data exchange, the next step (Step 2) is to invoke \texttt{run_compute_functions()} function, which makes a call to one or more user-defined compute-functions. These functions process domain elements in the sub-domain and update computation-space objects accordingly. Steps 1 and 2 may be repeated if needed by the application. Upon completion of computational tasks, in Step 3, the runtime system optionally runs the load-balancing mechanism. The frequency of initiating load-balance procedures can be determined by setting a specific parameter. Once this step is initiated, the runtime system examines the key statistics (the total time and the number of data points that have been processed), which are returned by previous step, and decides if domain re-decomposition is necessary to improve the overall performance. If so, it assigns new boundaries for each subdomain and makes the necessary changes on internal structures. The details of the procedure used for subdomain reassignment is explained in Section 3.3.

Though our current implementation follows an SPMD structure and uses MPI as the underlying library, other models, including fine-grained models like data-flow, can also...
be supported in the future. For example, we can support a more asynchronous execution, possibly through fine-grained data-exchange routines and compute-functions. Since the interactions between domain elements are clearly exposed to our runtime system, data-dependencies among different domain elements are automatically revealed and can be managed at the runtime for asynchronous or data-driven execution.

3.4.1 Example application code

Below, we provide the complete code for a stencil application (Jacobi) to demonstrate how DISC runtime is initialized, how attributes of domain elements are registered and how compute-functions are coupled with computation-space objects. The main computation for Jacobi is implemented in `jacobi_kernel` function and it consists of updating each grid cell with weighted values from neighboring cells.

```c
#include "structured_domain.h"
#include "double_attribute.h"
#include "utils.h"

// global variables including domain object
StructuredDomain* domain;
int n_rows, n_cols;

// compute function
void jacobi_kernel(vector<Attribute*>& computation_spaces)
{
    DoubleAttribute* output_space = (DoubleAttribute*) computation_spaces[0];
    DoubleAttribute* att = (DoubleAttribute*) domain->get_attribute(0);

    double* output_space_buff = output_space->get_array();
    double* att_buff = att->get_array();
    int width = n_cols+2;

    for(int i = 1; i <= n_rows; i++)
    {
        for(int j = 1; j <= n_cols; j++)
        {
            int center = i*width+j;
            int north = center-width;
            int south = center+width;
            int west = center-1;
            int east = center+1;
```

}
}

int main(int argc, char* argv[])
{
  // setup domain
  DomainProps props;
  props.set_ndims(2);
  props.set_domain_type(STRUCTURED_GRID);
  props.set_min_bounds(0, 0);
  props.set_max_bounds(1024, 1024);
  props.set_interaction(GRID_BASED, 1);
  domain = new StructuredDomain(props);

  // get assigned subdomain bounds
  Bounds<int>* local_bounds = domain->local_bounds;
  n_rows = local_bounds->hi[0] - local_bounds->lo[0];
  n_cols = local_bounds->hi[1] - local_bounds->lo[1];

  // define, populate and register attributes
  DoubleAttribute* val_att = new DoubleAttribute(0, (n_rows+2)*(n_cols+2), true);
  double* val_att_buff = val_att->get_array();
  for(int i = 1; i < n_rows+1; i++)
    for(int j = 1; j < n_cols+1; j++)
      val_att_buff[i*(n_cols+2)+j] = (double) (i+j) / 100;
  domain->register_attributes(val_att);

  // create computation-space object and couple it with compute-function
  DoubleAttribute* computation_space = new DoubleAttribute(0, (n_rows+2)*(n_cols+2));
  domain->add_compute_function(&jacobi_kernel, 1, &computation_space);

  // attributes and compute-functions are registered.
  // prepare communication, because it’s time to start execution.
  domain->setup_communication();

  // execute jacobi kernel
  for(int i = 0; i < N_ITERS; i++) {
    domain->run_iteration();
    domain->swap_buffers();
  }

  // finalize DISC runtime
  domain->finish();
}
3.5 Experiments

In this section, we present results from a number of experiments we conducted to evaluate the efficiency and capabilities of our programming model. All experiments are performed on a cluster where each node has two quad-core 2.53 GHz Intel(R) Xeon(R) processors, with 12 GB RAM, executing RedHat Enterprise Linux Server release 6.1, and Gigabit ethernet as the interconnect. Our programming model is implemented in C++ language and uses MPICH2 (version 1.4.1p1) as the underlying communication library. The same version was used for executing MPI applications used for comparison. For all experiments, except for the one focusing on CPU-GPU heterogeneity, the comparisons have been performed over a varying number of nodes ranging between 16 and 128.

Our evaluation is based on four applications, chosen to demonstrate the programming model’s capability of supporting applications involving different communication patterns. Thus, we chose one molecular dynamics application (MiniMD), one application involving an unstructured grid (Euler), and two smaller kernels involving stencil computations (Jacobi and Sobel). The MPI version of MiniMD was obtained from the Mantevo suite, whereas for other applications, we wrote MPI versions ourself. For each application, first, we compared the performance of our programming model with the corresponding MPI implementations under execution on homogeneous nodes, and under execution with a distributed memory system where nodes have different speeds. Second, to show the advantage of our model over existing systems where over-decomposition can be used to handle heterogeneity, we compared our performance against Charm++ on Jacobi and Euler. For this experiment, we used Charm++ version 6.5.1. Lastly, as a demonstration of our model handling a different kind of heterogeneity, we conducted a set of experiments on a single

1Please see https://software.sandia.gov/mantevo
CPU-GPU node for two stencil kernels, where we used a NVIDIA Tesla C2050 graphics card with Cuda version 5.5.

3.5.1 Comparison with MPI Implementations

Execution on Homogeneous Nodes: MPI provides a simple API for application development and its implementations have been heavily optimized for execution on homogeneous configurations. Thus, we first disable heterogeneous execution support feature of DISC
and compare its performance with MPI on a homogeneous set of nodes to see how much overhead DISC abstractions cause on normal execution. Because we offer a higher-level API on top of MPI, we can expect to see certain overheads from our programming model in this case.

We compare the performance of MPI and DISC as we increase the number of nodes, using 4 cores per node in each case. In MiniMD, we simulate the behavior of 4 million atoms, whereas we use 12 million nodes for Euler. On the other hand, for both Jacobi and Sobel, we use a grid structure with 400 million elements. We run each application up to 1000 iterations. Figure 3.7(a) shows that implementing MiniMD with our programming model brings only a 2.7% overhead on average and both of the versions scale well with increasing number of nodes. Now, as shown in Figure 3.7(b), implementing Euler in our programming model only causes an average overhead less than 1%. An important point to note here is the following. For MPI implementation of Euler, before starting the main computation, we first partition the nodes in the simulation with METIS, a commonly used graph partitioner, so that each partition has roughly the same size and the number of cross-edges among different partitions is minimized. This, as a result, leads to an even division of workload among processing units and also reduces the communication cost during the actual execution significantly (the partitioning cost is not included in the reported execution times). With our programming model, initial partitioning is handled by the runtime system. Figure 3.7(c) and 3.7(d) demonstrate the execution times of Jacobi and Sobel. Compared to the MPI versions, DISC implementations have an average overhead of 0.5% for Jacobi and 3.8% for Sobel. In overall, DISC model does not incur any significant overheads over MPI implementations and its relative speedups are at least as good as those from MPI.
Another question is how DISC and MPI compare while keeping the number of nodes fixed, and increasing the number of cores used within each node. We conducted an experiment using 16 nodes, and increasing the number of cores used per node from 1 to 8. We do not show the detailed results here, but as a quick summary, our findings are: 1) when we compare speedups of each application over 1 core executions, the maximum difference favoring MPI implementation between any pair of speedup values is only 0.04, 2) the speedups of Euler, Jacobi, and Sobel using 8 cores per node (over 1 core per node) is between 6.99 and 7.86 with DISC. MiniMD scales poorly with 8 cores for both MPI and DISC due to the increasing communication cost. We believe that this is caused by the periodic communication pattern of this application and can be addressed with a better process-to-core binding.

**Execution on Heterogeneous Configurations:** In this set of experiments, we evaluate the performance of our model when the cluster includes a heterogeneous collection of units. As elaborated before, in the near future, heterogeneity might arise either due to the static or dynamic differences in processing power. Because current clusters almost always have homogeneous set of nodes, we simulated heterogeneous behavior by slowing down four or more nodes, using synthetic load that consumed nearly 40% of the computation cycles on each core. Using a total of 64 nodes (with 4-cores used on each node), we created several different configurations. Particularly, ‘‘256+0’’ is the homogeneous case, and ‘‘240+16’’, ‘‘224+32’’, and ‘‘192+64’’ imply that 16 (4 nodes), 32 (8 nodes), and 64 (16 nodes) of the cores are slower than others.

Figures 3.8(a), 3.8(b), 3.8(c) and 3.8(d) show the performance of MPI, DISC (default), and DISC with heterogeneous execution support (“disc-h” in the figures) for MiniMD, Euler, Jacobi and Sobel, respectively. Here, DISC (default) implies our system without
support for heterogeneity. Both MPI and DISC (default) continue to use the same sized sub-domains even when the computing power of different units vary. In contrast, DISC with heterogeneous execution support reacts to the heterogeneity in the system and performs automatic re-partitioning. In the results we report in Figure 3.8(a), MiniMD is run for 500 iterations and uses 108 million atoms. Distribution of the domain for heterogeneous units is performed once at the beginning. Since the initial partitioning of the domain involves using more than one dimension, repartitioning is performed by utilizing the non-linear optimizer explained in Section 3.3. As shown by the first scenario in the figure, the overhead to support heterogeneous execution is less than 0.1%, when we have a homogeneous set of resources. In all other scenarios, where sixteen or more adjacent units are slower, there is a 51-54% increase in the execution time for both MPI and DISC (default) implementation. With heterogeneous execution support, this slowdown is reduced to 10.6%, 11.7% and 14.6% for each scenario. Furthermore, in all cases, it takes only around 0.03 seconds for the mathematical optimization solver (MINOS) to find an optimal solution for the problem.

Figure 3.8(b) reports the results for Euler, which is run for 10000 iterations. Because the initial partitioning is one-dimensional, repartitioning mechanism is performed by using the first technique and is only invoked once at the beginning of execution. When the resources are homogeneous, heterogeneous execution support degrades the performance by 3.2%, mostly because the load balancer reacted to small temporary changes in the processing power of cores and some of the auxiliary data structures used by the Euler application code have been readjusted. For other scenarios, the slowdown experienced by the DISC (default) implementation ranges between 67-73%, but heterogeneous execution reduces these overheads to 41.6%, 45.2%, and 47.9% for “240+16”, “224+32” and “192+64” cases,
respectively. Again, for all cases, re-partitioning decisions are made by the load-balancer in less than 0.14 seconds.

In Figure 3.8(c) and 3.8(d), we present results for Jacobi and Sobel. For these experiments, the number of elements in the grid structure is 900 million and each application is run up to 5000 iterations. Once again, heterogeneous execution mechanism makes use of the non-linear optimizer and is invoked at the beginning and then every 500th iteration. For Jacobi, under homogeneous execution scenario, re-partitioning causes only a 0.33% overhead. For all other cases, both MPI and DISC implementation experience approximately a 47-51% slowdown. Our support for heterogeneous execution reduces the slowdown to 8.4%, 15.9% and 25.2% for the three heterogeneous configurations, respectively. On the other hand, for Sobel, both implementations slow down around 56%, and DISC with heterogeneous execution support reduces it to an average of 14.3%. Once again, for both Jacobi and Sobel, the time required for MINOS to solve the underlying non-linear problem is negligible.

Next, we further evaluate the ability of DISC to maintain load-balance with heterogeneity, by creating scenarios where processing units become slower by varying percentages. Figure 3.9 shows results for each application when 1) all 256 cores have the same processing power (“0%”), 2) 64 cores out of 256 are overloaded with synthetic workload that consumes 25% of the computation cycles (“25%”), and 3) each of the 64 cores is overloaded differently with a workload consuming a random percentage between 25% and 50% (“25-50%”). In addition to MPI, DISC (default) and DISC with heterogeneous execution support (“disc-h”), we also include a hypothetical version of DISC, called “disc-perfect”. Execution times for the “disc-perfect” version are calculated by prorating the performance on the homogeneous setting by a factor equal to the ratio of the computational power of the
original cluster vs. the computational power of the heterogeneous cluster. Assuming DISC (default)’s execution time is $T_{\text{disc}}$ under the homogeneous “0%” case, the execution time of “disc-perfect” for the “25%” case is $T_{\text{disc}} \times 1.0625$, since 25% of the cores have become slower by 25%. Depending on the distribution of random slowdown percentage values, for the “25-50%” case, “disc-perfect”’s execution time can vary between $T_{\text{disc}} \times 1.06$ and $T_{\text{disc}} \times 1.14$. It should be noted that the “disc-perfect” version not only assumes that perfect load-balance has been achieved, but also that the performance depends only on the available computation cycles (ignoring communication), and loss of computation cycles can be accurately assessed by noticing rate of execution on any node.

Figure 3.9(a) reports the results for MiniMD. Both MPI and DISC (default) experience 25% and 83% slowdown for “25%” and “25-50%” scenarios, respectively. Support for heterogeneous execution reduces these slowdowns to 9.1% and 17.7%. As Figure 3.9(b) illustrates, Euler runs 36% and 111% slower under two heterogeneous scenarios than their respective homogeneous configurations and “disc-h” manages to decrease these slowdowns to 25.2% and 54.8%. Figure 3.9(c) and 3.9(d) show the results for Jacobi and Sobel. In Jacobi, MPI and DISC (default) experience slowdowns around 25% and 80% for respective scenarios and “disc-h” improves the execution time by reducing the slowdowns to 7.5% and 35.2%. Lastly, in Sobel, slowdowns of 27% and 90% are reduced to 9% and 18.8% with our heterogeneous execution support.

When we compare the results for “disc-h” with “disc-perfect”, for MiniMD and Sobel, we see that automatic repartitioning strategy of DISC provides a performance very close to the best theoretical slowdowns of 6% (for “25%” case) and 10% (for “25-50%” case). However, although DISC improves the execution times over MPI implementations significantly in Euler and Jacobi too, the slowdowns are much more above the optimal values,
Table 3.1: Execution Times (in secs) of Jacobi and Euler for DISC and Charm++ with varying number of chares under (a) homogeneous, (b) heterogeneous and (c) heterogeneous execution with runtime support for load-balance.

<table>
<thead>
<tr>
<th>Charm++ (Number of Chares)</th>
<th>DISC</th>
<th>64</th>
<th>128</th>
<th>256</th>
<th>512</th>
<th>1024</th>
<th>2048</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jacobi</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(a) Homog.</td>
<td>448.7</td>
<td>481.24</td>
<td>481.56</td>
<td>486.81</td>
<td>481.3</td>
<td>482.66</td>
<td></td>
</tr>
<tr>
<td>(b) Heter.</td>
<td>741.2</td>
<td>793.51</td>
<td>792.49</td>
<td>825.36</td>
<td>808.7</td>
<td>798.33</td>
<td>794.92</td>
</tr>
<tr>
<td>(c) Heter. LB</td>
<td>523.05</td>
<td>611.33</td>
<td>618.54</td>
<td>612.22</td>
<td>588.92</td>
<td>591.29</td>
<td>596.4</td>
</tr>
<tr>
<td>Euler</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(a) Homog.</td>
<td>325.44</td>
<td>383.29</td>
<td>382.94</td>
<td>385.96</td>
<td>393.2</td>
<td>423.43</td>
<td>502.22</td>
</tr>
<tr>
<td>(b) Heter.</td>
<td>530.04</td>
<td>642.43</td>
<td>635.59</td>
<td>632.05</td>
<td>650.23</td>
<td>695.98</td>
<td>813.97</td>
</tr>
<tr>
<td>(c) Heter. LB</td>
<td>428.11</td>
<td>637.63</td>
<td>635.49</td>
<td>490.18</td>
<td>534.42</td>
<td>596.59</td>
<td>639.72</td>
</tr>
</tbody>
</table>

especially for “25-50%” case. Our detailed analysis of the results revealed that estimation of a core’s processing power with unit processing times only (introduced in Section 3.3) could not completely capture the effects of heterogeneity on these applications. The main reason of this behavior for Jacobi is its main computation phase, where each grid point is updated by using four surrounding neighbors, is not compute-intensive enough for our heterogeneity measurement technique to calculate a good estimation. Furthermore, our analysis revealed that Euler’s communication phase is very vulnerable to any heterogeneity in the system, since 30% of its normal execution time is spent on data exchange. The greater the number of processing units used and slowed, the more its communication phase deteriorates.

### 3.5.2 Comparison with Charm++ Implementation

As we have stated throughout, over-decomposition, followed by runtime load balancing, is an existing mechanism for handling heterogeneity across nodes. Charm++ [81] is a
mature system, used for many large-scale applications, with a large volume of underlying research on load balancing [130]. Thus, we compare the performance of our programming model with Charm++ under different execution scenarios. In the results we demonstrate in Table 3.1, Jacobi (with 6.4 billion elements for 100 iterations) and Euler (with 12 million nodes for 1000 iterations) are run on 16 nodes. We executed Charm++ implementations with a varying number of Chares (cooperating message-driven objects corresponding to sub-domains) from 16 to 2048. The number of Chares represents the degree of over-decomposition. For heterogeneous execution scenarios ((b) and (c)), 4 nodes are slowed down. In scenario (c), load-balance mechanism is activated only once at the beginning of the execution. We tested different load-balancing strategies for Charm++ (RandCentLB, GreedyLB, RefineLB, NeighborLB), and here we only report the best one (RefineLB), which moves objects away from the most overloaded processors to reach the average.

When comparing our system with Charm++ for Jacobi, we see that Charm++ is 7.2% slower even on homogeneous settings. The difference is slightly larger when neither systems is using load balancing. However, Charm++’s load balancing ability is very dependent on the number of Chares used. Best performance with Jacobi is seen with 512 Chares, as the system does not have enough granularity to load balance with fewer Chares, and a larger number of Chares only introduce additional overheads. The best version with Charm++, based on the assumption that an optimal number of Chares is used, is still slower by 12.6% over our system.

The trends are similar with Euler, though performance differences are more substantial. Charm++ is slower by at least 17.8% even on a homogeneous configuration. As Euler has more communication, the impact of the choice of number of Chares is even more significant. In the heterogeneous setting, with load balancing support, Charm++ obtains best
performance with 64 Chares, where it is still 14.5% slower than DISC. Moreover, the performance with Charm++ is very sensitive to the number of Chares, and there is significant slowdown both with fewer or a larger number of Chares. Furthermore, in dealing with the same level of heterogeneity, the ideal number of Chares for the two applications is very different. We will, however, note that Charm++ is primarily designed to handle dynamic applications that generate new tasks during their execution, and not the heterogeneity of the parallel system. In comparison, our system cannot handle an application with dynamically generated tasks.

3.5.3 Decomposition Across CPU and an Accelerator

In this set of experiments, we consider a different type of heterogeneity resulting from the usage of an accelerator or co-processor together with CPU. If both devices have to be used for computation, the domain needs to be partitioned based on the relative speeds of the devices for the particular application. Note that certain co-processors today, like the Intel MIC, allow programming models like MPI to be used directly, meaning that DISC can be directly utilized as well. In other cases, such as when an NVIDIA GPU is used as the accelerator, our framework can be used for workload partitioning. Our current experiment has been conducted with the latter setup. Note that in this setup the programmer only needs to write a separate CUDA kernel and place the lines of code in `compute-function` to offload the work to GPU and load the results back. In other words, the extra coding-effort for using GPU does not involve any operations to support dynamic repartitioning.

In Figure 3.10, we run DISC implementations of 4-way Jacobi, 8-way Jacobi and Sobel with gridsize of 400 million elements for 100 iterations. Each application is executed by
using 2 processes on a single node only. Process I processes its entire workload on assigned CPU, whereas Process II makes calls to a GPU accelerator in its compute-function to speedup its execution. Hence, two processes have different computational powers. For each application, we first control the workload between two processes manually by changing their corresponding boundaries. Specifically, we tested different configurations for share of workload varying from \((2.5\%, 97.5\%)\) to \((60\%, 40\%)\), where the first and second number in each parantheses corresponds to share of Process I and Process II, respectively. Then, we let DISC partition the workload automatically by activating its heterogeneous execution support at the beginning of applications. In the figure, execution times are normalized against the minimum execution time in each application. Asterisk (*) signs on the figure show the share of workloads between two processes that DISC automatically detects towards an improved execution time. For 4-way Jacobi, DISC assigns 35\% of the workload to Process I. Assigning more work to Process II, DISC decreases Process I’s share to 30\% for 8-way Jacobi and 12.5\% for Sobel, since these applications are more compute intensive. When we compare DISC’s decisions with manually controlled experiments, we see that DISC’s automatic partitioning decision finds the share of workloads that achieves near-optimal (if not optimal) execution time.

### 3.6 Related Work

Parallel programming model design and implementation has been extensively studied over the past 3 decades. Here, we restrict ourself to models that support heterogeneous execution and/or those for distributed memory programming.
MPI [131] has been the most popular distributed memory programming model. Since MPI is based on programmer controlled data partitioning and communication, repartitioning cannot be performed automatically. OpenMPI implementation [67] can handle heterogeneity in terms of machine representation or network interface differences, but does not provide ability to repartition the work. Another distributed memory model that has been actively used is Charm++ [81], which is also used to create Adaptive MPI [1]. Charm++ and AMPI can both support load balancing, but only with user over-decomposing the problem. We have shown a performance advantage of our solution over Charm++. However, it should be noted that Charm++ can handle load imbalance arising because of dynamic behavior of an application, whereas our system cannot currently do so.

PGAS models have also become popular for parallel programming. Models like Global Arrays [105] and UPC [37] face similar issues as MPI in dealing with any possible heterogeneity. Languages that are based on more advanced compiler support, such as X10 [29] and Chapel [129], are more likely to be able to handle re-partitioning, though the challenge for these languages is in supporting more complex applications efficiently. For example, we are not aware of any study that demonstrates performance comparable to MPI for an application of the level of complexity of MiniMD from these languages. The same is also true for the earlier generation of compiled parallel languages, like High Performance Fortran and its variants [22, 73, 149].

Scheduling and load balancing have been extremely well studied in high performance computing [60, 48, 30]. Most of the load balancing work assumes that the application has already been decomposed as a set of tasks. We are not aware of any work that can decompose an array-based computation, perform load balancing, and then automatically handle communication.
In recent years, programming models for heterogeneous computing have emerged in the context of accelerator based systems [91, 10, 127, 134, 121, 146, 68, 140, 83]. Again, none of this work can automatically partition the problem and then handle communication. One exemption is the Qilin system [96], which performs work distribution between CPU and GPU based on offline learning. Our work, in comparison, is based on runtime monitoring of progress.

3.7 Summary

This work has described a parallel programming model that can handle most popular classes of scientific applications and automatically partition the workload (and manage communication) across heterogeneous processes. The model is based on relatively simple and portable runtime support (i.e. does not require any compiler support). We have shown how stencil computations, unstructured grid computations, and particle interaction based applications can be expressed using our programming model. Detailed performance evaluation using four applications has shown: 1) on homogeneous settings, our system only has a 2% slowdown over MPI, 2) on heterogeneous settings, we outperform MPI and Charm++ by an average of 34.2% and 13.5%, respectively, and 3) our heterogeneous execution scheme can also effectively divide the work between a CPU and an accelerator. In Chapter 4, we show how this programming model can be extended for low-overhead fault-tolerance support.
Figure 3.8: Heterogeneous Configurations: Execution times of MiniMD, Euler, Jacobi and Sobel for MPI, DISC (default) and DISC with Heterogeneous Execution Support (disc-h) when 16, 32 and 64 cores are slowed down by 40%. For MiniMD and Euler, domain repartitioning is only performed once at the beginning. For Jacobi and Sobel, it is performed at the beginning and every 500th iteration.
Figure 3.9: Heterogeneous Configurations: Execution times of MiniMD, Euler, Jacobi and Sobel for MPI, DISC (default), DISC with Heterogeneous Execution Support (disc-h) and DISC-perfect, when 64 cores (out of 256) are slowed down with varying random percentages.
Figure 3.10: Normalized Execution Times of 4-way Jacobi, 8-way Jacobi and Sobel for DISC with 2 processes (with different computational power) on a single node. Process I’s share of workload is varied between 2.5% and 60% (rest is given to Process II). Asterisk (*) signs on the figure show DISC’s decision of work division between two processes for each application. Execution times are normalized against the time of the configuration that gives the best performance in each application.
Chapter 4: Low-Overhead Fault-Tolerance Support using DISC Programming Model

In previous chapter, we described the basics of DISC programming model, which models iterative scientific applications with an underlying domain and interactions among domain elements. In this chapter, we explain how DISC programming model leverages low-overhead fault-tolerance solutions for recovering from fail-stop failures and detecting soft error occurrences. As we emphasized throughout this dissertation, future HPC systems will have a significantly greater number of cores than the current systems. Additionally, current technology trends in chip manufacturing and power optimization efforts indicate that soft errors, which have an undeterministic nature in terms of time and location, will be observed more frequently. These two factors together imply that Mean Time Between Failures (MTBF) will become so small that the time required to complete a checkpoint can exceed MTBF making the existing approaches completely inapplicable. These developments are imposing new challenges for application programmers. One one hand, they need to be able to manually implement efficient application-level checkpointing and recovery. Even more challenging for them is to implement techniques for dealing with soft errors. One pressing question is whether programming models can help automate fault-tolerant solutions.
In this work, we address this question in the context of the DISC programming model. DISC supports an API by which the domain, interaction among domain elements, and functions for updating any attributes of these domain elements can be explicitly specified. Starting from this model, inter-process partitioning of the work and the communication is handled automatically by the runtime system. In this chapter, we examine another important application of this programming model. We extended DISC model so that it also leverages low-overhead fault-tolerance support. We show that the abstractions that DISC model provides to hide the details of process communication and work partitioning/re-partitioning help also identify the main execution state and the functions that are the most susceptible to soft errors. Exposure of such vital program state and instructions is utilized in order to implement two fault-tolerance mechanisms within the runtime. First, with the concept of computation-space objects, DISC API makes it feasible to support automated, yet efficient, application-level checkpointing. This as a result can reduce checkpointing overheads significantly. Second, with the concept of compute-functions, DISC runtime is capable of detecting soft errors using a partial replication strategy. Here, only the set of instructions most likely to corrupt the main execution state is executed with redundancy and the results are compared efficiently with computed checksums.

4.1 Fault-Tolerance Support

We now describe two fault-tolerance approaches that have been implemented for the applications developed using DISC model.

4.1.1 Checkpointing

DISC model automates application-level checkpointing, alleviating the need for expensive system-level checkpointing that is normally used for programming models like MPI.
Like any checkpointing-based approach, we assume the existence of a persistent storage where the checkpoint files can be written into.

Two important questions for application-level checkpointing are: 1) when should checkpoints be taken, and 2) what data structures will be needed to restart the computation in case of a failure, and therefore, need to be checkpointed. It turns out that the DISC model simplifies these decisions. Particularly, the end of an iteration of the time-step loop (after data exchange and main computation have been completed by the runtime system as shown in Algorithm 2 of Chapter 3) is a natural point for taking the checkpoint. Compared to system-level checkpointing, we get a coordinated checkpoint (in the sense that there is no need for message logging for recovery), while not requiring any time-consuming coordination between processes.
Now, let us return to the question of which data structures need to be checkpointed. DISC model encapsulates the computational progress made on each domain element in objects that we introduced in Chapter 3; attribute and computation-space objects. At each iteration, attribute objects store the current information associated with domain elements, whereas computation-space objects capture the updates on them performed through compute-functions. As a concrete example, if we consider MiniMD, after each time-step, the attributes and the computation-space objects contain previous and updated coordinate and velocity values of each atom. The collection of attribute and computation-space objects represent the main execution state of applications at any given time. This collection along with the high-level information such as initial domain decomposition (boundaries of each subdomain) can be used to recover the state of DISC runtime and the underlying application completely.

If an application has multiple compute-functions, not all computation-space objects may be live at the end of an iteration of the time-step loop, i.e, certain computation-space objects could have been consumed already. Moreover, some of the attribute objects might entirely depend on and be calculated from a small set of remaining attributes without incurring a significant recomputation cost. This implies that during failure recovery not all of the attributes and computation-space objects are needed to recreate the execution state of domain elements. Some of them can be ignored by the checkpointing mechanism to save bandwidth, hence time, and also storage space. While compiler analysis can provide this information, our model currently asks the programmers to explicitly annotate this information by passing additional arguments during instantiation of these objects. This way, programmers can explore the tradeoff space in checkpointing the entire domain state vs. recalculation of a small portion from saved data structures. Note that any other application
state besides the ones associated directly with domain elements should be explicitly checkpointed by programmers. However, considering the computation patterns that DISC model targets, such additional state is limited and recomputed efficiently from checkpointed attribute and computation-space objects.

Checkpointing frequency as well as other important information like the file path where the checkpoint files will reside can be set via DISC API. To the head of checkpoint file, we insert some meta-data information including the current iteration number, and also the boundaries of the subdomain that attribute and computation-space objects represent. This meta-data is utilized to reconstruct the application state during recovery. Figures 4.1(a) and (b) illustrate the content of sample checkpoint files, which are taken at the 20th iteration of a 2D stencil grid computation and a 3D molecular dynamics application. In both (a) and (b), only the computation-space objects are saved.

**Recovery:** During recovery from a failure, DISC model is able to restart the computation both with the same or a fewer number of processes, unlike the current checkpointing approaches in MPI, which can only allow restart with the same number of processes. For instance, assuming that there are $N$ processing units in the system before the failure, if the computation is restarted with a fewer number of nodes, say $N - 1$, the domain is decomposed into $N - 1$ subdomains.

Whether with the same or fewer number of nodes, the most critical operation for recovery is to recreate the computational state of a subdomain from existing checkpoint files. If a processing unit has been assigned the same subdomain as before, it will be sufficient to access that subdomain’s checkpoint file and load its content into computation-space object in entirety. However, after decomposition, a change in subdomain boundaries is very likely.
Therefore, each processing unit may need to read several checkpoint files. In such cases, the metadata information mentioned previously is utilized to filter down the checkpoint files either completely or at least partially, i.e. we check if there is an intersection between processing unit’s newly assigned subdomain and the boundaries of the subdomain that the checkpointed computation-space object represents.

Once computation-space objects for the new domain have been reconstructed from the checkpoint files, the application can restart from the iteration in which the last checkpoint was taken.

4.1.2 Replication

Soft error detection has drawn significant attention from community in recent years. Such error detection could be from a variety of sources including hardware or software error detection codes such as ECC, symptom-based error detectors [70] and application-level assertions. One approach to detect such errors is to create two or more independent threads of execution and compare the execution state of different threads. This work has been done at multiple levels – replication at process level [56] or replication at the instruction level [54]. However, trivial replication of the entire program execution and comparison of resulting computation might incur significant overheads. We claim that concepts of compute-functions and computation-space objects in DISC model can be used to implement a partial replication strategy to reduce associated overheads substantially.

Compute-functions contain the lines of code to which majority of the program execution time is devoted. A soft error in combinatorial logic components including register values, ALUs and pipeline latches is most likely to occur when processing cores carry out the instructions expressed in compute-functions. Since computations, and hence updates on
Figure 4.2: Flow of execution at each iteration when replication strategy is in use.

domain elements, defined in compute-functions are directly reflected on the computation-space objects coupled with them, a soft error occurring during the execution of these functions eventually corrupts the computation-space objects, either directly and transitively. This observation suggests that soft errors can be efficiently detected by replication of compute-functions only and cross-comparison of their associated computation-space objects after each iteration. Note that replication mechanism described next assumes that processor components other than the memory are susceptible to soft errors. A produced value is assumed to be resilient once it leaves the processor and is stored back in memory. Control flow variables are protected by other means such as invariant assertions against the possibility of causing fatal errors such as segmentation faults.

**Replication mechanism:** Figure 4.2 demonstrates the execution flow at each iteration when this partial replication strategy is implemented in the DISC runtime. After data exchange operations are performed, the runtime splits the main execution thread into two as *original* and *replica*. Each thread is associated with its own computation-space object,
but they both invoke the same compute-function in parallel. During compute-function execution, both original and replica threads use the same set of input space, i.e. attributes of domain elements and any global data structures in application code. Sharing the same memory space, except the computation-space objects, leads to a significant reduction in overall memory footprint of replication strategy.

Currently, the replication strategy in DISC model makes the assumption that compute-functions provided by the programmer are side-effect free, meaning that they do not modify any global data structures. This is mainly to avoid possible race conditions. Note that one can synchronize original and replica threads by pragma directives with respect to the threading library used by the DISC runtime.

**Checksum calculation:** After both threads finish executing the compute-function, they calculate a checksum value over their own computation-space object. We employ integer module operation as the checksum function. Regardless of their data type, we treat the bit representation of values in computation-space objects as an integer and accumulate them into a single sum [100]. After checksum calculation, the two threads merge and checksum values are compared by the main thread. If the values match, application advances to the next iteration. Otherwise, DISC runtime ceases the execution and informs the programmer that a soft error has been detected and a recovery procedure should be initiated.

**Improvements for cache utilization:** The initial replication scheme calculates checksums over computation-space objects once individual threads finish execution of compute-functions. Although checksum calculation can be performed quite efficiently, especially in architectures with vector units, accessing the entire computation-space objects once again
leads to a large number of cache misses, and hence to high overheads, especially when computation-space objects are large. To remedy this, we present an improvement on top of the plain replication scheme presented previously. Instead of performing it in a separate step, we incorporate checksum calculation directly into compute-functions. Particularly, pure compute-functions provided by programmers are modified in a way that entries in a computation-space object contribute to the checksum on the fly, right after they are assigned a value. On the fly checksum calculation increases temporal locality of overall replication strategy and helps us avoid the data access costs incurred by an isolated checksum calculation phase.

Another source of overhead is the need to create a second copy of computation-space objects. Having additional computation-space objects for replica threads both increases the total memory footprint and at the same time diminishes overall cache utilization. Thus, as a second improvement, we avoid creating replica computation-space objects by modifying compute-functions further. Particularly, assignments to computation-space objects in replica thread are replaced by instructions that accumulate the assigned variables to the checksum values instead. Having no replica computation-space object in replica threads results in further improvements in data locality. In experimental results section, we demonstrate how these two optimizations affect performance of replication strategy, especially for applications with large output space.

4.2 Experiments

In this section, we present results from a number of experiments we conducted to evaluate the fault-tolerance solutions that we implemented within DISC model. Our evaluation is based on the same four applications in Chapter 3; a molecular dynamics application
Figure 4.3: Normal execution and checkpointing times of MPI and DISC implementations of four applications with varying number of nodes.

(MiniMD), one application involving an unstructured grid (Euler), and two smaller kernels involving stencil computations (Jacobi and Sobel).

4.2.1 Checkpointing

One of the key advantages of DISC model is the support for automated application-level checkpointing. We now show how the cost of checkpointing with our approach compares with the only automated solution currently available with MPI, which is system-level checkpointing. Moreover, we also examine how the total execution time of our system and MPI versions compare, when checkpointing overheads are included.
For checkpointing support in MPI implementations, we used MPICH2-BLCR, which is one of the most popular system-level checkpoint/restart libraries. Experiments in this section are performed on a cluster where each node has two quad-core 2.53 GHz Intel(R) Xeon(R) processors, with 12 GB RAM, executing RedHat Enterprise Linux Server release 6.1, and Gigabit ethernet as the interconnect. The comparisons have been performed over a varying number of nodes ranging between 16 and 128 (with only one core at each node), consistent with our focus on distributed memory parallelism. Both in this and next section, we repeated each experiment 5 times and report the average results.

Figure 4.3(a) and (b) demonstrate the execution times of Jacobi and Sobel, as we increase the number of nodes. Gray portions of the bars correspond to normal execution times, whereas red portion on top of each bar shows the additional time spent for checkpointing. For both applications, we use a grid structure with 400 million elements, execute them for 1000 iterations and trigger checkpoint mechanism every 250 iterations. Compared to the MPI versions, our model’s implementations have average overheads less than 1% for Jacobi and 4% for Sobel in normal execution times. The size of each global checkpoint in Jacobi and Sobel is 6 GB for MPI and 3 GB for our model. Corresponding figures show that checkpointing operations in our model are completed approximately in half of the time than MPI.

Figure 4.3(c) and (d) report the same results for MiniMD and Euler. In MiniMD, we simulate the behavior of 4 million atoms, whereas we use 12 million nodes for Euler. We run each application for up to 1000 iterations and take checkpoints every 100 iterations. Results show that implementing MiniMD and Euler with DISC brings an average overhead less than 5% in normal execution without checkpointing. In MiniMD, each global checkpoint of MPI version is nearly 2 GB in size, whereas with our programming model, the
application-level checkpoint is only 192 MB. Consequently, on the average, checkpointing
time of MPI is nearly 12 times higher. As the number of nodes increases, checkpointing
times increase, due to the fact that more nodes are contending for pushing the data to the
network file system at the same time. In Euler, the global snapshot size is again 2 GB
for MPI, and 640MB with our programming model. As a result, the time required for
checkpointing in MPI is nearly 4 times higher.

It is also useful to note that in all cases, after adding the normal execution and check-
pointing times, our model is faster. In some of the cases, particularly, execution of MiniMD
and Euler on 128 nodes, our model reduces the total execution time at least by a factor of
2, when checkpointing overheads are included. Furthermore, we can see that with increas-
ing number of nodes, as well as with increasing complexity of applications, the relative
advantage of our model increases. The former is because of increasing contention for I/O
related to checkpointing, whereas, the latter is because a full application has many more
structures than those that need to be checkpointed at the application level. Because Jacobi
and Sobel are small templates, the application-level checkpoint is nearly 50% of the size of
system-level checkpoint. In comparison, for a more complex application like MiniMD, the
ratio is close to 10%. Thus, we can see that for most applications, we can expect significant
performance from our model.

4.2.2 Replication

Next, we present the results for DISC implementations of the previous applications,
when we replicate compute-function execution in each process. We evaluate our partial
replication approach on Intel Xeon Phi 7110P many-core coprocessor. The reason for
choosing this architecture is that many-core systems are likely to be common in the exascale
Figure 4.4: Execution times of four applications without any replication (no rep), with plain replication (rep) and replication with improvements for cache utilization (rep+ofc and rep+ofc+ncs). Execution times for no rep with 1 process are 307.9, 398.9, 2686.2 and 213.2 seconds in Jacobi, Sobel, MiniMD and Euler, respectively. The same execution times for the best replication version rep+ofc+ncs are 316.2, 474.5, 2738.3 and 214.8 seconds.

era, where soft errors will also be more likely. Specifically, the coprocessor we have used has 61 cores running at 1.1GHz with 32KB L1 cache, 512 KB L2 cache per core and 8GB device memory for all cores, and is capable of running 244 hardware threads with hyperthreading support. All applications were compiled by Intel icpc-13.1.0 compiler with -O3 optimization with auto vectorization flag on. Each process replicates the compute-function execution step using OpenMP multi-threading library. We run all applications for 100 iterations. To mitigate the impact of system noise, we dedicate core0 of Xeon Phi to the
OS and pin DISC processes to hardware threads between core1 and core60. Original and replica threads in each process are pinned to the same core, except configurations where we have 1 and 30 processes.

Figure 4.4(a) and (b) present the replication results for Jacobi and Sobel. For each application, we compare the performance of four DISC versions; 1) execution without any replication (no rep), 2) execution with plain replication (rep), 3) execution with replication and on the fly checksum calculation (rep+ofc), and finally 4) execution with replication, on the fly checksum calculation and no replica computation-space (rep+ofc+ncs). All DISC versions are run with 1, 30, 60, 120 processes. For 240 processes, we only report the results for no replication version, since the replication versions utilize all of the 240 hardware threads with 120 processes. The figure shows that for Jacobi at 120 processes DISC replication versions rep, rep+ofc and rep+ofc+ncs have 118%, 44% and 33% overheads, respectively, over execution with no replication. Note that because the 240 thread no rep version does not have better performance over the 120 thread version, the results from 120 threads can be used to establish overheads of replication over the most efficient execution without replication. For Sobel, with the same number of processes, the overheads are 102%, 51% and 45%. These results indicate that two improvements over the plain replication scheme lead to significant reductions in total overhead by reducing data access costs during checksum calculation and improving overall cache utilization.

Figure 4.4(c) and (d) present the results for MiniMD and Euler. At 120 processes, DISC replication strategy causes 13%, 15% and 9% overheads in MiniMD, respectively for rep, rep+ofc and rep+ofc+ncs versions. In Euler, the same overheads are 34%, 41% and 24%. Although the overheads with the plain replication version itself is quite small, we see that
Table 4.1: Error detection rates for plain replication (rep) and replication with on the fly checksum and no replica computation-space object (rep+ofc+ncs) versions both without and with soft error injection. 0% rate for normal execution shows that there are no false positives

<table>
<thead>
<tr>
<th></th>
<th>normal execution</th>
<th>with error injection</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>rep</td>
<td>rep+ofc+ncs</td>
</tr>
<tr>
<td>Jacobi</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>Sobel</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>MiniMD</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>Euler</td>
<td>0%</td>
<td>0%</td>
</tr>
</tbody>
</table>

the suggested improvements do not lead to substantial benefits compared to Jacobi and Sobel. This is mainly due to the fact that computation-space objects in MiniMD and Euler have a smaller size and they fit in the L2 cache of Xeon Phi cores. Another potential reason is the following. Xeon Phi employs software and hardware-based data prefetching to reduce data access latencies. The prefetching mechanism works very aggressively for stencil kernels and accessing the same data within a core both by original and replica threads might lead to capacity and conflict misses. Furthermore, an existing analysis on Xeon Phi in [53] reports drops in bandwidth when different threads access the same memory space simultaneously due to the effects of contention at the interconnect level. Hence, we believe that any data locality optimization such as on the fly checksum calculation and no replica computation-space object for kernels such as stencils result in substantial improvements.

On the other hand, due to the irregular data access patterns in MiniMD and Euler, the amount of data prefetching is limited. The overhead for plain replication is not too high to begin with and the improvements in rep+ofc and rep+ofc+ncs versions are less visible.

As the last experiment, we show how effective DISC partial replication strategy is in detecting soft errors. Table 4.1 reports error detection rates when the four applications
are run both when there is no soft error occurrence during execution and when a single soft error is injected. Error injection is done manually by flipping a single bit of a random stack variable during the execution of compute-functions. We repeat the same experimental setup for two versions; plain replication (rep) and replication with on the fly checksum and no replica computation-space (rep+ofc+ncs). Each configuration is performed 50 times and error detection rates show how many times DISC detected an error in these runs as a percentage. Results show that when there is no soft error injection, error detection rate for both versions is 0% meaning that DISC replication strategy does not produce any false positives. Moreover, in Jacobi, Sobel and MiniMD, both versions are able to detect injected soft errors and achieve 100% error detection rate. As the only exception, in Euler, plain replication version detects only 24% of injected errors, whereas rep+ofc+ncs again achieves a 100% detection rate. This is due to the fact that in Euler each corrupted stack variable makes two contributions to the computation-space objects, one being positive and the other negative. When checksums are calculated in plain replication scheme, positive and negative contributions seem to cancel out each other reducing overall detection rate. In contrary, rep+ofc+ncs version is insusceptible to such cancellation, since checksums are calculated by using the corrupted assigned values directly and ignoring their sign.

4.3 Related Work

Fault-tolerance for high performance computing against hard errors has been extensively studied. Much of this research specifically targets MPI [132, 86, 3, 38, 21, 78, 102]. Recent efforts on optimizing the process include combination of coordinated and uncoordinated checkpointing [116] and compression for reducing the overheads of checkpointing [79]. Another approach is algorithm-level fault-tolerance [107, 7, 32, 39], where an
algorithms properties are exploited (typically to build-in redundancy). While this approach can overcome many of the overheads of general checkpointing, it has two key limitations: 1) as the name suggests, the solution is very specific to a particular algorithm, and 2) the fault-tolerant algorithm needs to be implemented by the programmer manually while developing the application. As for soft errors, the general detection approach is through redundant execution. This redundancy can be achieved at various levels. For instance, in [103], each computing node in execution is paired with a buddy node that performs the same work. Paired nodes checkpoint and exchange their local state periodically and the resulting computations in paired nodes are cross compared through their respective checkpoints. [56] provides a new MPI implementation that creates replica MPI tasks and performs online verification only on MPI messages during communication. Studies in [115, 117] execute all dynamic instructions in a program twice by redundant threads and compare the first and second result. If there is a mismatch, both threads restart execution from the faulty instruction. There have been some efforts to reduce the overheads associated with redundancy; [143] exploits high-level program information at compile time to minimize data communication between redundant threads, whereas [114] explores the partial redundant threading spectrum, in which only a dynamic subset of instructions are duplicated to near single threaded execution performance (in the expense of limited fault coverage). [54] combines redundant threading with symptom-based detectors by quantifying the likelihood that a soft-error impacting an instruction creates a symptom (branch mispredicts, cache misses). Resultingly, it only duplicates the ones that can not generate any such symptoms. Although the proposed solutions achieve significant reductions in associated overheads, none of them attempts to implement redundancy at the programming model level. As we show in next
sections, proper abstractions at programming model level can expose the most vital program state and can help automate redundant execution with small overheads.

4.4 Summary

In this work, we presented how DISC, the parallel programming model (presented in Chapter 3) for iterative scientific applications based on structured, unstructured grids and N-body simulations, is extended to leverage low-overhead fault-tolerance support. We showed that the existing abstractions in DISC model for automated inter-process communication and work partitioning/re-partitioning can be also used for automated application-level checkpointing and replicated execution to detect soft error occurrences. Experimental evaluation shows that checkpointing in DISC model provides significant improvements over system-level checkpointing scheme and soft errors can be detected by a partial replication strategy with low overheads.
Chapter 5: Fault-Tolerant Dynamic Task Graph Scheduling

Most of the work on fault-tolerance in parallel systems has focused on fail-stop failures, where a node completely halts, in the context of Single Programming Multiple Data (SPMD) paradigm for which we proposed fault-tolerance solutions in previous chapter. In this work, we focus on dealing with detectable soft errors in a different programming paradigm; task-graph execution model.

We consider the design, implementation, and evaluation of task graph scheduling algorithms that can continue execution of an application specified through a task graph to completion despite faults. The goal of the work is to minimize the slowdown of the application in the presence of soft errors. There is a considerable amount of work on fault-tolerant task graph scheduling in the real-time systems community [109, 58, 65], but these efforts require replicated task execution and/or duplicated state to support efficient failover. In comparison, our goal is to minimize the impact of faults without significantly impacting the performance or resource utilization during normal execution.

While both our problem formulation as well as the main ideas in our solution are general and applicable to task graph execution on shared and distributed-memory platforms, the detailed algorithms we have developed and implemented are specific to the NABBIT system. NABBIT [4] is a framework for scheduling task graphs in a provably time-efficient manner using work stealing. In this work, we adapt the NABBIT dynamic task graph scheduling
algorithm to support scalable recovery from soft errors that impact individual tasks. The recovery is performed in a *non-collective* fashion without interfering with threads not impacted by the fault. Simultaneously, the threads requiring a waiting task efficiently perform the recovery without incurring significant blocked or idle time. The presented approach can recover from an arbitrary number of task failures while incurring very low overheads in the absence of faults.

### 5.1 Background

In our work, a task graph is represented by a directed acyclic graph with vertices representing tasks and edges representing dependences between them, pointing from a source of a dependence to its destination. A task/vertex cannot begin execution until all tasks it depends on, referred to as its *predecessors*, complete. Vertices with no incoming dependences are referred to as *source* vertices, and those with no outgoing dependences are referred to as the *sink* vertices.

The task graph representation in this work encompasses models that represent the dependence between tasks rather than between tasks and data blocks [20, 25]. This representation can be derived from such a bipartite dependence graph by transitively constructing the task-task dependences. We do not require that each data block be constructed exactly once. Instead, we allow updates to data blocks, as long as the dependences specified ensure that all uses of a data block causally precede a subsequent definition (considered the *next version*) of the same block. Each task is considered synonymous with the definitions of data blocks it effects. A single task can produce multiple data blocks, and two tasks have a dependence between them if any data block defined by one task is used by the other. Figure 6.1 shows an example task graph with A as the source and E as the sink task. Note
Figure 5.1: A task graph example with task $A$ as the source and task $E$ as the sink. Dependence edges are drawn from producer to consumer. Black-shaded task $B$ fails during execution.

that task $A$’s data could be overwritten by task $C$ because all uses of $A$’s output, except by $C$ itself, are complete when task $C$ begins execution.

Tasks we are considering in this work are side-effect free and have locality of effect in the sense that tasks in the graph can not invoke each other, can not change the variables in global scope, or generally can not interact with outside world. The changes made in a task are only effective within the scope of that task and given the same set of input data blocks they always produce the same set of output data blocks.

5.2 Problem Statement

Our goal is to handle task scheduling in the presence of soft faults or errors. A soft fault can result in a bit flip in combinational or sequential logic. This can propagate into an incorrect arithmetic result or memory state. When undetected, this can lead to what is referred to as silent data corruption. A soft error affecting a task affects the computation only if the description of the task or any of its outputs is affected. Therefore, we focus on recovery from corruption of data blocks or task descriptors and, more specifically, on recovery from
such corruption once it is detected. There is significant ongoing research in error detection, which can stem detect errors from a variety of sources: hardware or software error detection codes, such as ECC; symptom-based error detectors [70]; application-level assertions; etc. Sequential execution elements (memory allocator, call stack, etc.), information on the task graph structure embedded in the application, the work stealing runtime, and the application data structures beyond the data blocks operated on by tasks are assumed to be made resilient through other means. We also assume that once an error is detected, all subsequent accesses to that object will observe the error.

In this work, we are interested in runtime scheduling algorithms for dynamic task graphs that can recover from corruption of task descriptors and data blocks. We consider algorithms that do not require or complement checkpoint-restart and, particularly, do not require using stable storage during task graph processing. Collective recovery approaches, such as those with checkpointing and restart, would synchronize all threads, possibly rolling them back to a prior execution. These approaches will require the overhead of synchronization even when there are no failures, and, with frequent errors, the application’s progress may be extremely slow. Our overall objective is to minimize overheads in the absence of faults with recovery costs proportional to the amount of work lost, while ensuring high resource utilization (i.e., not involving unnecessary repeated execution or replication).

For an illustration of the challenges involved, consider a specific snapshot of the execution of the task graph in Figure 6.1. In the snapshot, task B fails right after its computation, and the failure is detected by the thread operating on task B. Now, even before such a failure is detected, the threads processing tasks C and D could have observed the computation of task B and started their respective computations. Thus, the first challenge is to ensure that the threads executing tasks C and D are aware of the fault in the computation of task
B. Then, task B’s output has to be recomputed (or recovered) before computations of C and D are restarted. For efficiency, it is important that task B recovers only once—not twice—which could occur possibly at the initiation of two different threads executing C and D.

Yet another complication arises because of the reuse or overwriting of memory that takes place. Recall from our earlier example that task C reuses the space allocated by task A for its output (as the only other use of A’s output is by B, which needs to finish before C’s execution). Thus, even before C is aware of B’s failure, it could be overwriting A’s output. However, A’s output is required for recomputing B, as well as for restarting the computation of C once B has been recomputed. This implies that A will have to be recovered as well. Finally, similar to the preceding discussion, it is important that A also recovers only once.

Thus, handling such recovery correctly and efficiently for a general dynamic task graph is a significant challenge. This problem is addressed in the rest of the work, in the context of an existing scheduler used in the NABBIT framework.

5.3 Task graph scheduling using NABBIT

In this section, as a prelude to presenting the scheduling scheme in the presence of failures, we describe the elements of the task graph to be specified by the user and outline the scheduling algorithm in the absence of faults. The fault-tolerant scheduling algorithm relies on the following information from the user about the task graph:

Task key: A unique identifier for each task used to relate different references to the same task without the need for a pre-allocated task object.

Sink task: The task that transitively depends on all other tasks in the task graph.
**Predecessors and successors:** Functions that return an ordered list of predecessors and successors of a task, given its key. This information is used to reveal the dependences among tasks.

**Compute:** A function that defines the main operations to be performed by each task.

Once this information is available through library-provided classes, the task graph scheduling algorithm captures the structure of the task graph. The task scheduling algorithm is built on the NABBIT task graph scheduler [4], which is a provably efficient scheduler based on work stealing. Figure 5.2 presents the routines used by the task scheduler. For this section, we only refer to the non-shaded portions of the routines and ignore the gray shaded parts, which show how the runtime is modified to support failure recovery.

The tasks in the task graph are referred by keys (type int64_t), and their execution is controlled by the runtime through a concurrent hash map. A created task is atomically inserted into the hash map using the INSERTTASKIFABSENT routine and obtained later with a call to GETTASK. Note that the hash map stores the pointers to the tasks and not the tasks themselves.

For each task, the runtime holds the following fields:

**(int) join:** A counter, referred to as the join counter, that tracks the number of outstanding predecessors for a task. This counter is the basic unit of completion notification. It is initialized with the number of predecessors and decremented whenever a predecessor completes execution. Given that a task can be notified in parallel by any of its predecessors, while it is potentially being operated upon, the join counter is incremented and decremented atomically. A task is ready to be executed when its join counter is zero.

**(int64_t*) notifyArray:** An array (initially empty) that stores the successors that are enqueued to be notified once the task finishes its execution. Similar to the join counter, mutual
TRYINITCOMPUTE(A, key, life, pkey)
1: inserted = INSERTTASKIFABSENT(pkey)
2: # get task descriptor of A's predecessor (B)
3: (B, blife) = GETTASK(pkey)
4: if (inserted)
5: # B has just been inserted: explore B
6: spawn INITANDCOMPUTE(B, pkey, blife)
7: try
8: if(B.overwritten) throw;
9: finished = true
10: lock(B)
11: if (B.status < Computed)
12: # B should notify A once computed
13: add A.key{key} to B.notifyArray
14: finished = false
15: unlock(B)
16: catch
17: finished = false
18: RECOVERTASKONE(pkey, blife)
19: if (finished)
20: # B has already been computed
21: NOTIFYONE(A, key, pkey, life)

INITANDCOMPUTE(A, key, life)
1: INIT(A)
2: # traverse immediate predecessors of A
3: for pkey ∈ predecessors(A,key{key})
4: spawn TRYINITCOMPUTE(A, key, life, pkey)
5: NOTIFYONE(A, key, key, life)

NOTIFYSUCCESSOR(key, skey)
1: (S, slife) = GETTASK(skey)
2: NOTIFYONE(S, skey, key, slife)

COMPUTEANDNOTIFY(A, key, life)
1: try
2: COMPUTE(A)
3: A.status = Computed
4: n = SIZEOF(A.notifyArray)
5: notified = 0
6: # notify all successors enqueued in notify array
7: while (notified < n)
8: for i ∈ [notified, n)
9: skey = A.notifyArray[i]
10: spawn NOTIFYSUCCESSOR(key, skey)
11: notified = n
12: lock(A)
13: n = SIZEOF(A.notifyArray)
14: if (notified == n) A.status = Completed
15: unlock(A)
16: catch
17: if(error in A)
18: else RESETNODE(A, key, life)

NOTIFYONE(A, key, pkey, life)
1: try
2: # get index of pkey in the ordered list of preds
3: ind = CONVERTPREDKEYTOINDEX(key, pkey)
4: success = ATOMICBITUNSET(A.bitVector, ind)
5: # notify A only if the vector bit was set
6: if (success)
7: val = ATOMICDECANDFETCH(A.join)
8: # execute A if join counter is zero
9: if (val==0) COMPUTEANDNOTIFY(A, key, life)
10: catch
11: RECOVERTASKONE(key, life)

Figure 5.2: Routines used by the fault-tolerant task graph scheduler. Non-shaded portions correspond to the actions of the non-fault-tolerant NABBIT scheduler. Shaded portions indicate additions to the algorithm that make it fault tolerant. “A.key{key}” means that key replaces A.key in the fault-tolerant version. Auxiliary routines used for recovery (in catch blocks) are discussed in Section 5.4.
exclusion is ensured among concurrent operations on a task’s notify array by protecting
them with a lock associated with each task.

**int status:** The execution status of a task at the moment. The possible values are Visited,
Computed, and Completed. Once a task has been inserted into the hash map, its status is
set as Visited.

The execution of a task graph begins with the creation and insertion of the sink task
into the hash map followed by an invocation of the **INITANDCOMPUTE** function. **INIT-
ANDCOMPUTE** initializes the task and traverses its immediate predecessors through calls
to **TRYINITCOMPUTE**. Note that in general, if the predecessors of a task are all computed,
the task can be executed right away using the **COMPUTEANDNOTIFY** routine. Otherwise,
the task registers itself to the notify array of each predecessor that is not ready. Invoking
**INITANDCOMPUTE** and **TRYINITCOMPUTE** in a recursive fashion, the execution expands
the task graph and eventually reaches one of the source tasks with no incoming depen-
dences. **COMPUTEANDNOTIFY** executes such a task’s **COMPUTE** function (provided by
the user), updates its status as Computed, and begins notifying the successors (via **NO-
IFYSUCCESSOR**) registered in its notify array. After there are no more successors left
in the notify array, the task changes its status to **COMPLETED**. Any successor traversing
this task after the status change can see the completion of the task’s execution and can di-
rectly decrement its join counter. If a successor’s join counter is observed to be zero, it is
scheduled for execution using the same **COMPUTEANDNOTIFY** routine.

The actions of the task graph scheduler are parallelized using Cilk-style support for
recursive parallel (strict) computations. In particular, the creation and computation of the
predecessors of a given task are concurrent and can be executed by different threads. The
work-stealing scheduler randomly finds work, in terms of segments of task graph traversals to be performed, until the root of the computation—the sink task—is completed.

The actions of individual threads coupled with the scheduling properties of work stealing were used to show that this task graph scheduling algorithm is provably efficient. In particular, given a task graph with work $T_1$ (the time it takes to execute the task graph on a single processor), critical path length $T_\infty$ (the time it takes to execute on an infinite number of processors), and maximum degree $d$ (maximum number of predecessors and successors), the running time is $O(T_1/P + T_\infty \min\{P,d\})$, where $P$ is the number of available processors. The running time is asymptotically optimal for task graphs where the degree of each node can be bounded by a constant. Even for task graphs with large degree nodes, the running time is close to optimal, modulo a small amount of synchronization overhead.

5.4 Fault-tolerant Scheduling

Figure 5.2 presents the pseudocode for the fault-tolerant dynamic task graph scheduler. The statements and function parameters shaded in gray are introduced to the NABBIT scheduler to support fault-tolerance. These routines are supported by the recovery routines shown in Figure 5.3. The changes to the NABBIT algorithm are two-fold. First, different phases of the algorithm that access a task object or a data block are enclosed in try-catch statements. Any detected errors are assumed to throw an exception, which is caught to trigger the recovery procedure. Second, additional function parameters and data structures are introduced to support the recovery.

We describe our recovery approach in terms of how the algorithm supports fault recovery by providing key guarantees. These guarantees are then combined to show that the
RECOVERONCE(key, life)
1: if (!ISCOVERING(key, life))
2: RECOVERTASK(key)

ISCOVERING(key, life)
1: inserted = INSERT_RECORD(R, key, life)
2: if (inserted) return false
3: stored = GET_RECORD(R, key)
4: success = ATOMIC_CAST_SWAP(stored, life-1, life)
5: return !success

RECOVERY(key)
1: repeat
2: try
3: success = true
4: # insert the new incarnation of the task
5: (T, life) = REPLACE_TASK(key)
6: T.recovery = true
7: # traverse successors to recreate notify arr.
8: T.succs = GET_SUCCS(key)
9: for skey ∈ T.succs
10: (S, slife) = GET_TASK(key)
11: REINIT_NOTIFY_ENTRY(T, key, S, skey, slife)
12: spawn INIT_AND_COMPUTE(T, key, life)
13: catch
14: if (!ISCOVERING(key, life))
15: success = false
16: until (success)

RESET_NODE(A, key, life)
1: try
2: A.join = 1 + SIZE_OF(A.preds)
3: SET_ALL_BITS(A.bitVector)
4: INIT_AND_COMPUTE(A, key, life)
5: catch
6: RECOVERONCE(key, life)

REINIT_NOTIFY_ENTRY(T, key, S, skey, slife)
1: try
2: # ignore Computed and Completed tasks
3: if (S.status ≠ Visited) return
4: ind = CONVERT_PRED_KEY_TO_INDEX(skey, key)
5: if (S.bitVector[ind] == 1)
6: lock(T)
7: # skey’s waiting notification from T
8: add skey to T.notifyArray
9: unlock(T)
10: catch
11: if (error in S)
12: RECOVERONCE(skey, slife)
13: else
14: throw

Figure 5.3: Additional routines to assist in the recovery from faults.
proposed approach can execute a task graph to completion with the correct result irrespec-
tive of the number of faults. Now, we present a descriptive outline of the correctness proof.
We shall use the example task graph in Figure 6.1 and execution snapshot described in
Section 6.1 for illustration.

**Guarantee 1.** Each failure is recovered at most once.

For example, we want task B in Figure 6.1 to be recovered once and not by every
observer of the fault. We associate each creation and insertion of a task object into the hash
map with a *life number*. This life number is tracked through the call stack whenever that
task is processed. Thus, a failure detected is associated with a particular life number of
that task, referred to as its *incarnation*. When multiple threads detect the failure, they try
to simultaneously recover that task using the RECOVER TASK ONCE routine. As the first
step, the ISRECOVERING routine checks whether recovery of the current incarnation of
the task (the task with the current life number) already has been initiated. The first thread
to observe this condition performs the actual recovery. To facilitate this, we maintain a
separate concurrent hash map (denoted as R in Figure 5.3) that associates a given key with
the most recent life number for which a recovery process has been initiated. The recovery
table is empty initially, and a record for a given key is inserted only if a failure occurs on the
corresponding task. When a task fails the first time, the thread that inserts the record into R
performs the recovery. For subsequent failures on the same task, the thread that succeeds in
updating the existing record performs the recovery. Note that a failed task whose successors
already have been computed is not recovered, because no other task attempts to access such
a task.

**Guarantee 2.** A task’s status is correctly recovered.
A task’s status dictates the action a thread takes when it is encountered. As indicated before, a task’s status could be Visited (created but not computed), Computed (COMPUTE function has been executed), or Completed (all enqueued successors have been notified). Rather than attempt to restore a recovered task’s status from a backup or snapshot, we treat a task being recovered as a newly created task and begin its processing by checking the status of its predecessors. This is shown in the RECOVERTASK routine, where the recovering thread calls INITANDCOMPUTE to redo the task’s execution. Any successor requesting this task from the hash map (using its key) would get this new incarnation and handle it as they would a non-failed predecessor.

**Guarantee 3.** *The join counter of a task object is decremented exactly once per predecessor.*

Consider the scenario where task B decremented D’s join counter before failure and attempts to do so again after recovery. D’s join counter would now be zero, allowing it to be executed, even though its predecessor task C might not have been computed. We avoid these scenarios by associating each join counter with additional information about which predecessors have notified it. In particular, we retain a *bit vector* that tracks if the join counter has been decremented for a particular predecessor in the ordered list of predecessors. This bit vector is initialized to 1 for all bits. Each bit is unset when the corresponding predecessor is observed to have been computed, or when that predecessor issues a notification to this task. The join counter is decremented only if that bit is set. This change to NABBIT is shown in NOTIFYONCE in Figure 5.2. The CONVERTPredKEYToINDEX routine, given a predecessor key, returns the predecessor’s index in the ordered list of predecessors, so that the corresponding bit in the vector can be unset.

**Guarantee 4.** *Every task waiting on a predecessor is notified.*
Successors enqueued in the notify array of a failed task are expected to be notified by this task when it computes. Not notifying any such task can result in some tasks never being executed, leading to a hung execution state. In the example, task B needs to correctly handle the notification to tasks C and D. Recovering this information (mainly, the notify array) using some form of duplicated storage would require additional support to recover from multiple failures. We work around this issue by altering the base task graph scheduling algorithm’s behavior in the presence of failures. This is shown in REINIT_NOTIFYENTRY. A task being recovered traverses all of its successors with Visited status to check their bit vector and observe if they have been notified. Any successor not yet notified is considered to have been enqueued for notification before the failure and is enqueued into the reconstructed notify array. When this task computes, all such enqueued tasks are notified. This ensures that tasks C and D are notified when task B has been recovered and recomputed. This *eager* notification deviates from the NABBIT algorithm and could potentially result in execution of tasks in a different, albeit still correct, order impacting its optimality guarantees. We prove in Section 5.5 that this change does not violate NABBIT’s asymptotic parallel efficiency guarantee.

**Guarantee 5.** *Failures in data blocks observed during task computation are recovered.*

Note that faults in the data blocks could also be observed in application code, denoted by the COMPUTE routine in COMPUTEANDNOTIFY, when a thread is executing the COMPUTE function to perform the actual computation represented by the task. We assume that these detected errors are reported back to the runtime through exceptions. When an exception is caught in the COMPUTEANDNOTIFY routine, we first identify which task’s fault resulted in the failure. Then, we check if the task being computed is the source of the error. If so, that task is recovered. If not, we begin processing the task anew, using the
RESETNODE routine, by atomically resetting the notification bit vector and join counter and traversing its predecessors to verify if any of them have failed. As just described, the bit corresponding to each predecessor is unset upon notification, and the join counter is decremented exactly once per predecessor.

**Guarantee 6.** Failures observed during recovery are recursively recovered.

We would like to recover from failures that might affect tasks while they are being recovered. As shown in the RECOVERTASK routine, the recovery of a failed task involves updating its entry in the hash map with a new incarnation of the task using the REPLACEETASK routine, initializing the notify array by inspecting its successors, and re-executing the task as if it were a normal task. If an error occurs when such a re-execution is being performed, the next incarnation of the task is inserted into the hash map, and the re-execution begins anew. Such errors can happen an arbitrary number of times and still be recovered. In the algorithm, this can be observed by the fact that operations in the recovery routines in Figure 5.3 are themselves enclosed in try-catch statements.

**Lemma 1.** Every task is executed only after all of its predecessors have been computed, and the final output of every task is computed from the same inputs with and without failures.

*Proof Sketch.* Each predecessor decrements a task’s join counter exactly once. When the join counter becomes zero and the task is ready to be computed, all of its predecessors have been computed. If a task observes a failure in the task descriptors or data associated with one of its predecessors, it attempts to identify and recover from that failure. This continues until all predecessors execute without faults, and the resulting data blocks are fault-free. Fault-free execution of predecessors produces the same inputs to a task even if the predecessors had experienced prior failures.
Lemma 2. A task whose predecessors fail is eventually executed.

Proof Sketch. A task might observe the failure of one of its predecessors during its computation or when it is checking its predecessors. Any such failed predecessor ensures that it notifies this task exactly once. Note that a task may have been notified by an earlier incarnation of a predecessor before it failed. If the task begins its user computation while the failed predecessor is being recovered, it would reset its bit vector, traverse the predecessors one more time, and observe this failure. This task either observes the recovered state of its predecessor and is computed, or is eventually notified by any outstanding predecessors, including the predecessor being recovered. The last predecessor to notify executes this task.

Lemma 3. The sink task is executed to completion irrespective of the number of failures.

Proof Sketch. The predecessors of a sink task might observe a failure in themselves, or in one of their predecessors, during their computation. All of these failures can be recovered from, as explained in Lemma 2. Thus, all predecessors of the sink task execute to completion. If the sink task itself fails, it is observed by the thread processing the sink task or by one of its predecessors. As such, failures in the sink task or its predecessors are recovered. When the sink task and all of its predecessors are in an error-free state, the sink task is executed.

Theorem 1. The task graph execution produces the same result with and without faults.

Proof Sketch. Lemma 3 showed that the sink task is executed irrespective of the number of failures encountered. All tasks are assumed to be stateless, meaning every execution of a task produces the same output for the same inputs. Lemma 1 shows that the sink task
is executed with the correct non-faulty inputs, producing the same final result with and without faults.

When data blocks are reused by multiple tasks in a graph, additional re-execution may be required. For example, task A’s output has been partially overwritten by task C. However, re-execution of task B (and later task C) requires the output from A. During normal execution, the dependences specified ensure that all uses of a particular version of a data block are complete before the task that produces the next version of the data block is allowed to execute. However, a fault might result in the need to use such a data block version after it has been overwritten. Our algorithm tracks such overwrites and re-executes these tasks by treating them as if they failed.

The fault-tolerant algorithm does not affect the performance of the base task graph scheduler in the absence of faults, except through the addition of atomic operations to maintain the bit vector. The recovery focuses on tasks that failed and is performed by threads that need to operate on them without affecting other threads. While single-assignment task graphs only incur these costs, reuse of data buffers could result in additional re-execution in trying to reproduce the inputs to the failed tasks. In the experimental evaluation, we show that this overhead is not significant in practice, and could be ameliorated by retaining the intermediate versions in memory. While errors can still affect these retained versions, the impact of such re-execution also can be minimized.

5.5 Performance Analysis

In this section, we provide a theoretical analysis of the runtime of the program on \( P \) processors in the presence of faults. As there is failure recovery built into the system, the running time depends on the number of times each node fails and is recovered. In particular,
we prove an \textit{a posteriori} bound—our bound depends on the number of times each node failed and was recovered. We first calculate the upper bounds on the total work and span of any execution. Then, we translate these bounds to completion time bounds using known theoretical bounds on completion times of series-parallel programs using randomized work stealing [8, 18].

5.5.1 Definitions

Consider a task graph $\mathcal{D} = (V, E)$. Each node $A \in V$ has a list $\text{in}(A)$ of immediate predecessors and a list $\text{out}(A)$ of immediate successors. Therefore, $|\text{out}(A)|$ and $|\text{in}(A)|$ denote the out- and in-degrees of $A$, respectively. For simplicity in stating the results, we assume that every node is a successor of a unique node $\text{root}(\mathcal{D})$ with no incoming edges and a predecessor of a unique node $\text{final}(\mathcal{D})$ with no outgoing edges. Let $\text{paths}(A, B)$ be the set of all paths in $\mathcal{D}$ from node $A$ to node $B$.

Note that actual execution of the computation is non-deterministic for several reasons. A node may be executed more than once due to failures or because the memory containing the result has been freed since the node executed and another successor needs this result. In addition, the actual execution of a dynamic task graph may depend on the input and the schedule. Hence, our completion time bounds will be for a particular execution. Each possible execution can be represented as an execution graph (more specifically, DAG) $\mathcal{E}$. Each execution of $\mathcal{D}$ leads to a potentially different $\mathcal{E}$.

We define several notations for subgraphs of an execution graph $\mathcal{E}$. For a particular execution graph $\mathcal{E}$, we define a function $N$ such that $N(\mathcal{E}, A)$ is the number of times task $A$ is executed in $\mathcal{E}$. In addition, let $\text{com}_i(\mathcal{E}, A)$ be the subgraph corresponding to $i$th execution of the compute function and $\text{comNot}_i(\mathcal{E}, A)$ be the subgraph corresponding to the $i$th
execution of the compute and notify function. For any subgraph $\mathcal{E}'$ of an execution DAG, we denote the work of the subgraph as $W(\mathcal{E}')$ and the span as $S(\mathcal{E}')$.

To analyze the running time, we must examine executions of the worst-case parameters of $\mathcal{E}$. We define that the total work done by an execution $\mathcal{E}$ of $\mathcal{D}$ is $W(\mathcal{E})$, and the span is $S(\mathcal{E})$. We will overload the notation, indicating that $W(\mathcal{D}^N)$ is the maximum work among all execution graphs that have the same function $N$, and $S(\mathcal{D}^N)$ is the maximum span among all of these execution graphs. Similarly, for each node $A$, we define $W(\text{com}(A))$ as the maximum time a compute function can take over all execution graphs $\mathcal{E}$ and all executions of that compute function. Similarly, $S(\text{com}(A))$ is the maximum span of the compute function of $A$ over all execution possibilities of $A$.

### 5.5.2 Work Analysis

To calculate the work of a task graph execution, we first construct (pessimistic) bounds on the time NABBIT spends waiting at synchronization operations. Let $L_W(A)$ be the maximum over all executions of the time spent on atomic decrements.

**Lemma 4.** Any execution of $\mathcal{D}$ has work at most

$$W(\mathcal{D}^N) = O\left(\sum_{A \in V} N(A)(W(\text{com}(A)) + \sum_{B \in \text{out}(A)} N(B) + L_N(A)) + L_J(A)\right),$$

where

$$L_J(A) = O\left(\sum_{B \in \text{out}(A)} \min\{|\text{in}(B)|, P\}\right) \text{ and }$$

$$L_N(A) = O\left(\sum_{C \in \text{in}(A)} \min\{|\text{in}(C)|, P\}\right).$$

**Proof.** The first term arises from the work of the compute functions because each node is executed $N(A)$ times. The second term arises from the fact that every time $A$ is executed, we must look through the notify array to see if any of them should be notified. Note that each successor $B$ of $A$ can appear many times in the notify array, in particular, $N(B)$ times. The term $L_N$ stems from the fact that a node may be added many times to its predecessor’s
notify array, and there is potential contention on this array. Finally, the term $L_j$ bounds the amount of time we can spend waiting to decrement the join counter. Every time a node $B$ is notified and its join counter is decremented, the decrement may have to wait for other updates. We do not multiply this quantity by $N(A)$ as the failure model ensures that each node is notified at most once.

5.5.3 Span Analysis

The nondeterministic nature of the computation complicates a direct calculation of $S(D^N)$. Instead, we construct a new, deterministic execution DAG $E^N$, which is parametrized by the function $N$. The span of this DAG is an upper bound on the span of the computation where the number of times each node is executed is dictated by the function $N$.

In this DAG, for a node $A$, there are $N(A)$ executions of the compute function represented by DAGs $\text{com}_1(A), \text{com}_2(A), \ldots, \text{com}_{N(A)}(A)$. We omit $E$ from this notation and assume that each of these DAGs is the worst-case DAG for the compute function over all executions. At the end of each compute function, we assume there is a notification to all of the successors, but none of these notifications actually succeed. Finally, in the graph $E^N$, we add edge from the end of DAG $\text{com}_i(A)$ to $\text{com}_{i+1}(A)$ as these computations must occur sequentially one after the other.

More importantly, in $E^N$, we make two assumptions. First, for each node, we define the method $\text{comNot}^*(A)$ to be the same as the original method, except that all possible recursive calls always occur. In other words, $\text{comNot}^*(A)$ always makes recursive calls for all of its successors. Second, we assume that only the last execution of each node $A$ actually manages to inform all of the successors. In other words, the last call to $\text{com}(A)$ is
the one that turns into \( \text{comNot}^\ast(A) \). Therefore, the first instance of \( \text{com}_1(B) \) of \( B \) has an edge from the last instance of \( \text{com}_N(A) \) of \( A \) if \( A \) is a parent of \( B \).

**Lemma 5.** With a dynamic execution graph \( \mathcal{E} \) generated by \( \mathcal{D} \), where a node \( A \) is executed \( N(A) \) times, the span of \( \mathcal{E} \) is at most \( S(\mathcal{E}^N) \).

**Proof Sketch.** Consider the longest path to any subDAG \( \text{com}_1(A) \) in \( \mathcal{E} \) and in \( \mathcal{E}^N \). Both paths must contain at least one instance of \( \text{comNot}(B) \) for every \( B \) that precedes \( A \). However, it is easy to see that the path in \( \mathcal{E}^N \) is not shorter because everyone is always successfully notified by the last execution of every node.  

**Lemma 6.** The span of the computation is \( S(\mathcal{E}^N) \) is at most

\[
S(\mathcal{E}^N) \leq O\left( \max_{p \in \text{paths}(\text{root}, \text{final})} \left\{ \sum_{X \in p} N(X) S(\text{com}(X)) + \sum_{Y \in \text{out}(A)} N(Y) + L_N(X) \right\} + \sum_{(X,Y) \in p} L_S(X,Y) \right)
\]

where

\[
L_S(X,Y) = O\left( \min \{|\text{in}(Y)|, P\} \right) \quad \text{and} \quad L_N(A) = O\left( \sum_{C \in \text{in}(A)} \min \{|\text{in}(C)|, P\} \right).
\]

**Proof sketch.** The first term is due to the fact that node \( X \) is executed \( N(X) \) times (one after the other), and each execution has the span \( S(\text{com}(X)) \). The second term comes from the fact that each of these compute functions tries to notify all of the nodes in the notify array but fails to do so. The term \( L_N \) is due to the contention on the notify array. Finally, the last term stems from the fact the last execution of \( X \) successfully notifies all of \( X \)’s successors. The term \( L_S(X) \) accounts for the contention cost of decrementing the join counter for \( Y \), where \( Y \) is a descendant of \( X \). In the worst case, this decrement might have to wait for \( \min \{|\text{in}(Y)|, P\} \) other decrements.
5.5.4 Completion Time Bounds

We have bounded the work and span of the execution graph using the characteristics of the task graph. Now, we relate these bounds back to the execution time using a work-stealing scheduler.

For any task graph $\mathcal{D}$, where a node $A$ is executed $N(A)$ times according to the failure model, define $T_1$ as the time it takes to execute $\mathcal{D}$ on a single processor. Define $T_\infty$ as the time it takes to execute $\mathcal{D}$ on an infinite number of processors, assuming no synchronization overhead. We have

$$T_1 = \sum_{A \in V} N(A)(W(\text{com}(A)) + |\text{out}(A)|)$$

and

$$T_\infty = \max_{p \in \text{paths(root, final)}} \left\{ \sum_{X \in p} N(A)S(\text{com}(X)) \right\}.$$

Using Lemmas 4 and 6, and the analysis of a work-stealing scheduler [8, 18], we obtain the following upper bound for the completion time of the task graph on $P$ processors:

**Theorem 2.** Consider a task graph $\mathcal{D}$, where each node $A$ is executed $N(A)$ times. The graph has the maximum degree $d$ and maximum path length (number of nodes on the longest path in the task graph from root to final) $M$. Also, $N = \max_{A \in \mathcal{D}} N(A)$. With probability at least $1 - \epsilon$, NABBIT executes $\mathcal{D}$ on $P$ in time

$$O \left( \frac{T_1}{P} + T_\infty + \log(P/\epsilon) + NMd + NL(\mathcal{D}) \right),$$

where $L(\mathcal{D}) = O \left( (|E|/P + M) \min\{d, P\} \right)$.

**Proof sketch.** From [8, 18], a Cilk-like work-stealing scheduler completes a computation with work $W$ and span $S$ in time $O(W/P + S + \log(P/\epsilon))$ on $P$ processors with probability at least $1 - \epsilon$. To prove the completion time, we relate the work $W(\mathcal{E}^N)$ and span $S(\mathcal{E}^N)$ to $T_1$ and $T_\infty$. 

118
Table 5.1: Matrix size (N), block size (B), total number of tasks (T), total number of dependencies (E) and the critical path length (S) for each benchmark.

<table>
<thead>
<tr>
<th></th>
<th>LCS</th>
<th>LU</th>
<th>Cholesky</th>
<th>FW</th>
<th>SW</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>512Kx512K</td>
<td>10Kx10K</td>
<td>10Kx10K</td>
<td>5Kx5K</td>
<td>6Kx6K</td>
</tr>
<tr>
<td>B</td>
<td>2Kx2K</td>
<td>128x128</td>
<td>128x128</td>
<td>128x128</td>
<td>128x128</td>
</tr>
<tr>
<td>T</td>
<td>65536</td>
<td>173880</td>
<td>88560</td>
<td>64000</td>
<td>132650</td>
</tr>
<tr>
<td>E</td>
<td>195585</td>
<td>508760</td>
<td>255960</td>
<td>308880</td>
<td>262600</td>
</tr>
<tr>
<td>S</td>
<td>510</td>
<td>238</td>
<td>238</td>
<td>120</td>
<td>1475</td>
</tr>
</tbody>
</table>

The proof follows from Lemmas 4 and 6. We bound the in- and out-degrees of nodes by $d$ and bound expressions that compute maximum over paths $p$ in terms of $M$. Bounding the work in Lemma 4 using the maximum degree, we know that

$$W(\mathcal{E}^N) = T_1 + N|E|\min\{d_i, P\}.$$ 

Similarly, we can use Lemma 6 to show that $S(\mathcal{E}^N)$ is not more than

$$O\left( T_\infty + NMd_o + NM\min\{d_i, P\}\right).$$

We note a few things about this bound. It reduces to the normal NABBIT bound when there are no failures and $N(A) = 1$ for all $A$. Second, it is asymptotically optimal for constant degree graphs, graphs whose degree can be bounded by a constant. Even when the degree is not bounded, we do not expect the terms that depend on the degree to have much impact on the runtime all nodes have sufficient work.

5.6 Experiments

We evaluated our scheduling algorithm on a 48-core Redhat 4.1.2-54 Linux system, consisting of four sockets, each with a 12-core AMD Opteron 2.3 GHz processor, and 256 GB of memory. Our implementation is built on top of Cilk++ 8503 x86_64 release,
and all benchmarks are compiled with Cilk++ compiler (based on gcc 4.2.4) with “-O3” optimization. We attempted to mitigate the impact of system noise and variability by using only 44 of the 48 cores for our experiments and setting the affinity of each thread to a specific core. In all experiments reported in this section, we take 10 runs and report the average (arithmetic mean). Standard deviations are presented as error bars.

We evaluated our implementation using five benchmarks: LCS (longest common sub-sequence); Smith-Waterman (local sequence alignment algorithm [95]); Floyd-Warshall (all-pairs shortest path algorithm in a weighted graph); and two dense linear algebra kernels, which are LU decomposition and Cholesky factorization. The configuration used for each benchmark is shown in Table 6.3. LCS, Smith-Waterman (SW), and Floyd-Warshall (FW) are implemented using the recursive definitions of the corresponding dynamic programming solutions. We evaluated single-assignment and memory reuse strategies for implementing these benchmarks. In the memory reuse version, we allocate a set of data blocks and reused them to store the outputs of subsequent tasks whenever possible. In all cases, except LCS, the memory reuse implementation resulted in improved performance. Thus, we used this version for subsequent evaluation. Note that memory reuse increases the potential cost of our algorithm by requiring recomputation of tasks because their outputs have been overwritten. We expect the overheads of our fault-tolerance scheme for the single-assignment implementations to be lower. Memory reuse is not applicable to LCS because each task’s output is part of the computation’s final output and cannot be reused.

We observed that the cost of fault recovery for Floyd-Warshall significantly depended on the exact location of the fault. This is due to the impact of varying costs incurred by the recursive recomputation of inputs to a failed task. We adapted the implementation to retain two versions per data block, doubling the memory requirement to minimize the impact
of such cascading recomputation. Note that neither version of a data block is considered
checkpointed as either, or potentially both, could fail. Our resilience approach would be
equally applicable to the single-assignment approach for implementing Floyd-Warshall and
might be preferable when optimized memory management policies make that implemen-
tation comparable or better performing than the two-version implementation. In general,
such choices point to the trade-offs between different task graph representations for a given
application rather than the scheduling algorithm.

5.6.1 Overheads Without Failures

We evaluated the impact of fault-tolerance support on performance in the absence of
failures. Figure 5.4 shows speedups achieved by the implementation with no fault-tolerance
support (baseline) and by our fault-tolerant implementation for each benchmark. Unlike
the fault-tolerant version, the baseline version includes no additional data structures or
statements introduced for fault-tolerance. Results indicate that these additional structures
do not incur substantial overheads. One exception is Floyd-Warshall, with nearly 10%
overhead on the execution with 44 cores. This is due to the memory management and
additional cache misses introduced by the two versions maintained for each data block.

5.6.2 Overheads With Failures

We evaluate overheads in the presence of failures by injecting faults into execution. For
each figure in this section, recovery overhead is defined as the increase in the execution time
(over the fault-tolerant version in the absence of failures) and displayed as a percentage. To
simulate faults, we a priori identify the tasks that would fail and the point in their lifetimes
where they would fail. When a fault is injected, a flag is set to mark the fault, which is then
observed by a thread accessing that task. We evaluate the following fault scenarios:
Figure 5.4: Speedup for the baseline and fault-tolerant versions in the absence of faults. Sequential times (in secs) for each benchmark are as follows: LCS (baseline=650, w/FT-support=668), SW (baseline=578, w/FT-support=579), LU (baseline=624, w/FT-support=613), Cholesky (baseline=338, w/FT-support=337), and FW (baseline=315, w/FT-support=371).
**Amount of work lost:** We randomly inject failures in the task graph to effect the loss of a constant amount of work or a certain percentage of the total work. A fault affects both a task and the data blocks it has computed. For every experiment, we verify the fault injection by ensuring that the number of tasks recovered matches the loss of work in terms of number of tasks, intended. This cannot be guaranteed in some scenarios (descriptions follow).

**Time:** The cost incurred by a fault depends on the point in a task’s lifetime at which the fault affects it. A task lifetime consists of three phases: before compute, after compute, and after notify. A task failing in the before compute phase has traversed its predecessors and is waiting for one or more notifications to be scheduled for execution. A task that has completed its main operations and is about to notify its successor tasks, is considered to be in the after compute phase. Once a task finishes notifying its successors, it transitions into the after notify period. A task may spend a significantly larger fraction of its lifetime in before compute and after notify phases as these phases could potentially involve several recursive task computations.

The impact of a fault on a task is different in each of these scenarios. A task in the before compute phase has not performed its computation, incurring a computation cost. Recovering such a task, while incurring the cost of resetting its state and tracking its predecessors, does not result in task re-execution overhead. In contrast, failures occurring in after compute require the task to be re-executed, potentially incurring significant overhead. Statically analyzing the impact of faults in the after notify phase is difficult due to fact that some tasks might not recover if all successors of a failed task finish their computation before the fault can be injected.

**Task type:** We define three sets of tasks for the injection of failures: \( v=0 \), \( v=last \), and \( v=rand \). Tasks with the \( v=0 \) label denote tasks that produce the first version of a data
block. The $v=last$ label denotes tasks that produces the last version. A failure on a $v=0$ task causes either 0 or 1 task re-execution, depending on the time at which it is injected. Conversely, the failure of a $v=last$ task can trigger a chain of re-executions, where all of the tasks that produce the previous versions of a particular data block get re-executed. Representing a case between these two extremes, we identify a $v=rand$ task as a task that produces a random $i^{th}$ version ($0 \leq i \leq n$) of a data block, where $n$ is the last version number of the same data block.

We show the impact of the timing of failure (before compute and after compute) when we inject failures on three different task types ("$v=0$,” “$v=rand,” and “$v=last””) in Figure 5.5(a). In each of the six injection scenarios, the injected failures cause the recovery to execute a total of 512 tasks corresponding to less than 1% of all tasks for each application, with the largest observed overhead being 0.96%. As expected, the before compute cases incur negligible overheads as recovering such failures does not lose any work done prior to the failure. In comparison, after compute scenarios lead to loss of computed work, bringing an observable but still small overhead on all benchmarks. Moreover, we observe there is no clear difference in the impact of failures between the before compute and after compute scenarios, for the various task types. We repeated the same experiment for scenarios with only 1, 8, and 64 task re-executions and did not observe any statistically significant overheads. The relevant figures are not displayed due to space constraints.

Figure 5.5(b) demonstrates results for the same setup, but with injected failures causing 2% and 5% of the total number of tasks to be re-executed. For this experiment, we report the results for “$v=rand$” tasks only because the available amount of “$v=0$” and “$v=last$” tasks in most of our benchmarks are below 5%. As in the previous figure, before compute failures hardly bring any overhead, whereas after compute failures cause, at most, 3.6%
Figure 5.5: Impact of the time of failure (“before compute” and “after compute”) on the recovery overhead, when failures occur on different task types (“v=0,” “v=rand,” and “v=last”). (a) Failures cause the re-execution of 512 tasks; (b) failures cause 2% and 5% of the total number of tasks to be re-executed. In both (a) and (b), overheads are calculated by taking the sequential execution time of FT-support version, in the absence of faults, as the baseline. FT-support (evaluated in Figure 5.4) introduces an additional 10% overhead for FW, and negligible overhead for the other benchmarks.
Table 5.2: Average, minimum, maximum, and standard deviation of the number of re-executed tasks in after notify scenario when the original set of failures on different task types (“v=0,” “v=last,” and “v=rand”) implies 512 task re-executions. The resulting average recovery overhead for each benchmark is presented in Figure 5.6.

<table>
<thead>
<tr>
<th></th>
<th>LCS</th>
<th>LU</th>
<th>Cholesky</th>
<th>FW</th>
<th>SW</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>last</td>
<td>rand</td>
<td>0</td>
<td>last</td>
</tr>
<tr>
<td>Avg</td>
<td>443</td>
<td>448</td>
<td>442</td>
<td>473</td>
<td>3606</td>
</tr>
<tr>
<td>Min</td>
<td>448</td>
<td>431</td>
<td>431</td>
<td>484</td>
<td>43585</td>
</tr>
<tr>
<td>Max</td>
<td>453</td>
<td>446</td>
<td>4245</td>
<td>483</td>
<td>483</td>
</tr>
<tr>
<td>Std</td>
<td>8</td>
<td>9</td>
<td>8</td>
<td>10</td>
<td>1206</td>
</tr>
</tbody>
</table>

Figure 5.6: Recovery overheads in the after notify scenario when the set of injected failures implies re-execution of 512 tasks, 2% and 5% of the total number of tasks in the graph.

and 8.2% overheads for the “2%” and “5%” cases, respectively. Broadly, the amount of re-execution overhead is proportional to the amount of work lost.

Next, we evaluate the impact of faults injected in the after notify phase for different types of tasks. We present these results separately because the impact of after notify faults depends on the specific benchmark and types of faults introduced. In one extreme case, all of the successor tasks may have already used the failing predecessor’s output by the time

126
the failure is injected, implying no re-executed work. As another extreme, if the successor discovers the predecessor’s failure after it has started to overwrite the previous versions of its output data block, separate recoveries must be initiated to restore both the failing predecessor’s output and the previous versions of the successor’s output. Table 5.2 reports the statistics for the actual number of re-executed tasks (average, min, max, and standard deviation) for each benchmark when the initial set of failures injected at the after notify period implies 512 task re-executions. Results show that after notify failures on different task types can have a distinct impact on each benchmark, especially in benchmarks where the number of uses on the last version of data blocks depends on the problem size (as in LU, Cholesky, and SW). Such failures can lead to a high number of re-executions with large standard deviation values. In contrast, applications such as LCS, where each data block has, at most, three uses, the re-execution amounts are low and similar for all task types. Figure 5.6 demonstrates the average overheads for the setup in Table 5.2 along with the scenarios where the injected failures imply re-execution of 2% and 5% of the total number of tasks (on “v=rand” tasks only). For most cases, the overheads do not exceed 2.5% and 6.5% for “2%” and “5%” scenarios, respectively.

5.6.3 Scalability Analysis

For this analysis, we focus on failures that occur at the after compute time period on “v=rand” tasks. Figure 5.7 plots the scalability of the fault-tolerance mechanism when varying the number of cores are used. As Figure 5.7(a) shows, a constant number of re-executions hardly has any effect on the overall execution, whereas, in Figure 5.7(b), we observe an increasing trend in the recovery costs when more cores are employed. Generally, when a data reuse strategy is applied for subsequent versions of data blocks, each failure in
Figure 5.7: Recovery overheads when varying the number of cores (P) for a failure scenario, causing the re-execution of (a) 512 tasks and (b) 5% of all the tasks. In both (a) and (b), failures are injected on “v=rand” tasks at the after compute failure point.
the system leads to a chain of task re-executions, which cannot be run in parallel. The chain involves the producer task for the corrupted data block, as well as any tasks that produce a previous version for the same data block. These chains lack concurrency to keep all threads busy and comprise the biggest scalability challenge for any task graph execution scheme and related fault-tolerance mechanisms. Unsurprisingly, this loss in available concurrency is more visible with a higher number of cores, as shown by Figure 5.7(b). Nevertheless, the overhead of our fault-tolerance mechanism still does not exceed 6.5% for most cases, reaching a maximum of 8.2% in Smith-Waterman on 44 cores.

5.7 Related Work

The challenges of dealing with an architecture that can expose soft errors to programmers has received some attention in recent years. The area of approximate computation has emerged [51, 119, 19, 28, 141, 118, 120], with the idea being that a programmer can explicitly declare what computations can be performed approximately. The target domains have been multimedia, data mining, and machine learning. In comparison, scientific programmers are typically unwilling to accept a lack of precision unless supported by rigorous mathematical analysis. Erez et al. have experimented with the notion of containment domains [36], which can confine errors in certain program segments. This approach also requires additional programmer effort. Another direction has been to use replication of processes [57, 104]. While this approach does not require additional programmer effort, it decreases resource utilization efficiency, increasing costs and power budget. Our solution does not require additional programmer effort, and has much lower costs compared to replication. However, it does assume that soft errors can be detected in a timely fashion. In
other efforts, Li et al. [92] recently studied soft error vulnerabilities in scientific programs, but their work does not provide a specific solution for addressing the problem.

**Algorithm-level solutions for fault tolerance:** Algorithm-level fault tolerance solutions have been a topic of investigation for almost three decades. In this approach, the idea is that recovery can be performed by using already existing information in running processes, and, if the algorithm itself does not contain such redundant information, it can be added by modifying them [76, 31, 39, 94, 33]. While this approach can handle most types of soft errors, it is specific to a particular algorithm, and cannot be applied to all algorithms.

**Task graph scheduling with knowledge of computation and communication times:** Many task graph scheduling approaches assume the structure of the task graph and weights associated with the vertices and edges, corresponding to task computation and inter-task communication times, are known in advance [88]. Fault tolerance approaches under these assumptions include both entirely offline approaches [71, 64, 14, 109] and approaches that include some dynamic decision making [136, 58]. These approaches typically employ task duplication, by a fixed or variable amount, to tolerate failures of individual tasks [109, 65] while minimally impacting the task graph execution latency. Given our focus on minimizing fault tolerance overheads rather than minimizing execution latency, we do not assume such complete *a priori* knowledge and do not employ task duplication.

**Fault tolerance for task-based computations:** Maehle and Markus [97] presented fault-tolerant scheduling of data flow programs on distributed systems by preserving the inputs to tasks so a task can be recovered from a failure using its inputs. In general, recovery from multiple failures requires multiple checkpoints. Vrvilo et al. [142] consider fault tolerance for Concurrent Collections using checkpoints of the current execution frontier of the data flow graph. Using single-assignment data allows checkpointing to be performed in parallel.
with computation once the data has been created. Charm++ supports message-driven execution and employs message logging and checkpointing to recover from faults [104, 101]. Our approach complements these schemes and can increase the time between checkpoints in computations that can be structured as task graphs.

**Fault-tolerant key-value store:** Key-value stores have been made resilient against a variety of errors. Dynamo [43]; MapReduce [42], which operates on key-value pairs; and Linda [11, 145], which operates on tuples, provide fault tolerance through replication. These approaches typically store significant application state in the key-value store, requiring expensive techniques to ensure fault tolerance. In our scheme, the values in the hash map are individual scalars that track tasks and can be made resilient with low overhead. We observed that application data blocks and computation dominate the total time (>96% in most cases) and space consumed during execution. Thus, the hash map can be made resilient with minimal impact on overall application performance.

We are not aware of any prior work that combines provably efficient task scheduling and scalable error recovery.

### 5.8 Summary

We presented a fault-tolerant dynamic task graph scheduling algorithm that recovers from faults without global co-ordination and can efficiently interleave recovery from faults and normal execution to avoid scalability limitations. The algorithm was shown to be asymptotically optimal for graphs whose degree can be bound by a constant. In the absence of faults, the fault-tolerant version was shown to not incur significant overheads compared to the non-fault-tolerant version. Experimental evaluation through injection of faults at various stages of execution showed the presented algorithm can efficiently recover from an
arbitrary number of faults with costs that are roughly proportional to the amount of work lost.
Chapter 6: User-Assisted Memory Management for Dynamic Task-Graph Schedulers with Auto Correction Support

In our last work, we focus on a completely different type of failures; errors induced by programmers during efforts for performance optimizations. To achieve scalable performance, existing programming models [25] are accompanied by runtime systems which support further tuning of implemented applications. These tuning efforts can be expressed within the application by programmers through an extended set of language constructs/API. Such efforts might become a potential source of error if they are not incorporated carefully. Hence, the ability to recover from such errors is extremely important for any programming model with configurable runtime behavior. As an example for such tuning efforts and potential errors introduced, this work focuses on memory-efficient execution of programs structured as task graphs and processed by a dynamic scheduler.

To motivate the problem, consider a data-flow or a single-assignment task graph. Each task in such a task graph produces a unique data item, which is not subsequently modified by other tasks. This representation maximally exposes the concurrency in the program and enables most effective utilization of available hardware parallelism. However, naively scheduling such task graphs can quickly overrun the memory available in a parallel system. Traditional garbage collection schemes (mark-and-sweep, reference counting, and others)
treat any data item with a valid pointer as being alive. Given that the definition-use relationships in a dynamically executing task graph are not known \textit{a priori}, any given data item might have a potential future use, and thus cannot be garbage collected, until the task graph completes execution. Other approaches to optimize memory usage often require additional information on the number of uses or the last use of a given data item [87, 123].

In this work, we present an alternative approach based on \textit{recycling} the store across tasks in a task graph. Task A is said to recycle the store occupied by task B if A’s output overwrites the memory occupied by B’s data item, effectively erasing B’s output. Recycling operations are dictated by store recycling functions which are provided by user specifications. We first address a number of issues related to realizing a recycling based task graph execution. We identify the key characteristics of correct and efficient store recycling functions. Next, we present an algorithm to verify that a user-provided store recycling function is correct (e.g., the recycling does not cause any overwrites to live data or lead to a data race). To ease programming, we have developed mechanisms for such detection to be carried out with low overheads during program execution and enable recovery from incorrect overwrites within the same run. Thus, the programmer can execute an application with a potentially incorrect recycling function, and still obtain a correct execution without a large slowdown inspite of the potential errors.

To further improve programmability, we present an approach towards automatically deriving a correct recycling function. Specifically, we consider compositions of incoming dependences as potentially valid recycling functions, determine which of them is correct, assess the reuse facilitated by correct functions, and choose the one that leads to the largest number of recycled stores. The correctness determination can be done by execution on a smaller problem size. While this does not ensure correctness of recycling function on
a different input dataset (since dependencies can be input dependent), recall that we can further verify the correctness with low overheads during a production run.

6.1 Background

A task graph is a directed acyclic graph consisting of nodes denoting tasks and edges denoting inter-task dependences. An edge from task A to task B indicates that task B cannot begin execution until task A has completed execution. Task A is said to be a predecessor of task B, whereas task B is a successor of task A. Certain task graph models such as Concurrent Collections [25] employ alternative representations with explicit tasks and data items – tasks produce data items once the data items they depend on are ready. However, we will employ a representation of task graphs consisting only of tasks to simplify the discussion. In particular, each task is assumed to produce one data item, allowing us to treat a data item and the task that produces it as being synonymous.

Task graph models typically distinguish between intermediate data items and data items exposed to the outside environment, referred to as output data items. Though the output data items cannot be deallocated during the execution of the task graph, efficient execution requires design of memory reclamation strategies for the intermediate data items. Various memory management strategies have been studied in the literature for these intermediate data items.

Some task graph schedulers are agnostic of data items and leave the burden of memory management to the user. When using these schedulers, the tasks in the task graph explicitly map the outputs of multiple tasks to the same memory region. This is a tedious and error-prone process – the tasks sharing the same memory region should not limit concurrency or lead to incorrect memory overwrites.
Reference counting garbage collectors and those in managed runtime systems (e.g., mark and sweep collection schemes) declare an object with no references from live variables as dead and ready to be deallocated. During dynamic task graph execution, all data items can be accessed by any pending tasks. Therefore, no tasks can be garbage collected until task graph execution completes.

Another approach to memory management requires the user to specify the number of uses or the last use for each data item. This information is independent of the schedule while enabling eager memory reclamation. On the other hand, providing this information imposes additional burden on the programmer in analyzing the task graph to compute the exact number of uses for each data item. Thus, in this work, we are interested in the design of automated approaches to dynamically schedule task graphs in a memory-efficient fashion.

6.2 Our Approach: Store Recycling Functions

In this section we describe our overall approach to aiding memory-efficient execution of task graphs.
First, while scheduling under a strict memory constraint is highly desirable, it is also extremely challenging to achieve for dynamic schedulers. In addition, applications typically run in environments with some flexibility in their memory usage. While reducing memory usage is beneficial, the use of the absolute minimum memory is not often necessary. To this end, rather than mapping to a bounded store, we focus on recycling the store as the execution proceeds. Store recycling allows us to specify memory management constraints in terms of tasks in the task graph. We define the task $A$ to be a recycle predecessor of task $B$ (and task $B$ to be a recycle successor of task $A$), if the output of $B$ occupies the same memory region as the output of the task $A$. In other words, the task $B$ recycles the store assigned to the task $A$.

Explicit representation of a recycle predecessor/successor for each task can be prohibitively expensive. We observe that tasks in a task graph can often be organized into classes, with all tasks in a class sharing the same data access patterns, number of predecessors, and ordering relationships. We associate a store recycling function with each task class. This often allows the recycling function to be represented compactly in a problem-size independent fashion.

Consider a store recycling function that maps task $B$ to task $A$, so that task $B$’s output occupies the same memory region as the output of task $A$. A schedule that employs such a store recycling function should order the execution of $A$ to precede $B$. Therefore, a store recycling function imposes an ordering constraint between a task and its recycle predecessor. A store recycling function is correct if and only if every valid schedule for the task graph satisfies this ordering constraint. The correctness of a store recycling function is therefore defined as follows:
A store recycling function is correct if and only if it does not violate causality and does not lead to a data race.

Recycling can potentially impose some additional constraints on the task graph. Consider the task graph shown in Figure 6.1. Task A’s output has been computed and consumed by the time task B can execute. Therefore, the task B can safely recycle the store assigned to task A. However, tasks B and C are concurrent. If the task B was to recycle the store assigned to the task C, an additional constraint will be introduced. The introduction of such additional dependence edges can lead to performance penalties. The scalability achievable by task graph schedulers is fundamentally limited by the degree of parallelism available, or alternatively, the critical path length of the computation. However, different schedulers can exhibit different performance depending on the structure of the task graph, even if the critical path length is not affected. Therefore a stronger condition to determine the efficiency of a store recycling function is defined as follows:

A store recycling function is efficient if it does not introduce any additional dependence constraints in the task graph.

In other words, any dependence induced by the store recycling function should be already implied by the edges in the task graph. Neither of these conditions checks for early overwriting of task outputs. Specifically, store recycling can cause a task’s output to be overwritten before all of its uses are complete. We formulate the above requirements and avoid overwriting by imposing the following constraints on store recycling functions:

1. **Constraint 1:** Every task is a successor of all immediate successors of its recycle predecessor.

2. **Constraint 2:** Two tasks A and B can recycle the same task C’s store only if task A (or B) can recycle the store associated with task B (or A).
We illustrate these constraints using the task graph in Figure 6.1. Task $B$ and $C$ are successors to the immediate successors of $A$ (tasks $D$ and $E$), and can thus recycle $A$’s store. However, they both cannot recycle $A$’s store due to the second constraint. On the other hand, both tasks $B$ and $F$ can recycle $A$’s store based on the above constraints. Note that a valid recycling function is reflexive, with a task recycling to itself interpreted as allocating new memory for its output, symmetric, and transitive. Therefore a valid store recycling function forms a partial order.

6.3 Evaluating Store Recycling Functions

In this section, we describe our approach to evaluating a collection of user-specified store recycling functions in terms of correctness and efficiency.

6.3.1 Checking Correctness of Recycling Functions

Consider the verification of a single recycling function. During verification, we mimic the actions of the recycling function without actually performing them. The actual task graph execution could be performed using a different or no recycling function. For ease of discussion, in remaining parts of this section, we refer to the recycle operations being evaluated as if they are being employed in the execution.

The basic actions of the verification algorithm are shown in Figure 6.2. The two constraints above imposed on a store recycling function relate to causality relationships between tasks. We therefore employ vector clocks to track causality [113] and check each constraint by comparing the relevant vector clocks. A simple scheme is to execute the task graph and retain vector clocks, predecessor, and successor information for all tasks, and verify all tasks on termination. However, this could incur high overheads. We therefore
on init of task descriptor T:
  T.recycle_succ = [] # recycling successors
  T.succ = [] # dependence successors

on execution start of task T:
  Trpred = recycle(T)
  assert Trpred happens-before T
  atomic Trpred.recycle_succ += [T]
  atomic rsucc = Trpred.recycle_succ
  for Tp in rsucc:
    if Tp == T: break
    atomic Tp.recycle_succ += [T]
    memfence
    atomic succ = Tp.succ
    assert Tp happens-before T
    for s in succ:
      assert s happens-before T

on access to output of Tp in task T:
  T.clock = T.clock U {Tp.clock}
  atomic Tp.succ += [T]
  memfence
  atomic rsucc = Tp.recycle_succ
  for rs in rsucc:
    assert T happens-before rs

Figure 6.2: Basic actions of the store recycling verification algorithm. The implementation employs optimized versions of the atomic statements (discussed in Section 6.3.1).

perform on-the-fly checks. In particular, we atomically track information on the dependence successors and store recycle successors for each task. We ensure that each access to a data item and recycling operation is valid.

We allow tasks to recycle themselves or invalid task identifiers. A task identifier is invalid only if no such task exists in the task graph. This is a useful mechanism to handle source tasks and tasks with only a subset of the dependences as other tasks. We do not assume prior information on the successors of a task. We track and verify each use as we encounter it. This allows us to check for incorrectly ordered recycle and access operations.

Note that the algorithm is a simplified version of the implementation. Atomic statements only operate on individual fields or task descriptors and are implemented in a lock-free fashion using atomic instructions where possible. Atomic list append operations are
implemented using atomic instructions on their length field. Rather than copy entire lists as shown in the algorithm, we exploit the fact that these are append-only lists and atomically copy the list length and operate on the list without a lock.

The algorithm to verify a recycling function is specified in the form of actions triggered by various events in the task graph scheduler. The algorithm itself is independent of the specifics of the task scheduler employed and is therefore compatible with any task graph scheduler.

**Evaluating Multiple Recycling Functions** The algorithm to verify a recycling function does not require that the same recycling function be employed by the task scheduler. We exploit this property to verify multiple recycling functions simultaneously. In the absence of a recycling function, the total memory usage is the total size of the data items produced by all tasks in the task graph. A store recycle operation to a valid task avoids the allocation of new memory. Therefore, total memory usage when executing a task graph under a store recycling function is inversely proportional to the number of recycle operations performed, and the amount of memory allocation avoided. We therefore track number of recycle operations for each recycle function being evaluated. If multiple recycle functions happen to be correct, we choose the recycling function with the minimal memory usage for subsequent executions of the task graph.

**6.3.2 Exploring the Space of Recycling Functions**

The two constraints that we define for a correct recycling function imply that a recycling function is correct if and only if it maps each task $B$ in task graph to another task $A$, where execution of both $A$ itself and all of $A$’s immediate successors causally precede $B$. Therefore, candidates that $B$ can recycle safely should be a subset of $B$’s immediate and
transitive predecessors. Figure 6.3(a) demonstrates a small task graph example for LCS (Longest Common Subsequence), in which dependence edges are drawn from producer to consumer. Each task consumes data items that are produced by its left, upper, and diagonal neighbors. In the figure, the green tasks satisfy both constraints and their outputs can be recycled by the yellow task. Such candidates for a task can be explored by traversing the task graph starting from the given task and exploring the task graph towards the source tasks (tasks with no incoming dependences) by using predecessor relations provided by the user.

We compactly enumerate the possible traversal paths based on the recycling functions for a given task graph derived from the dependences. The paths are enumerated to a bounded depth, referred to as hops, to limit the cost of verification. This enumeration of traversal paths requires information on the dependence structure of the task graph from the user. Dependence structure information describes the number of immediate predecessors for any task and a function to reach each of them. Table 6.1 summarizes the necessary
Table 6.1: Dependence structures for LCS and Cholesky. Each tag in dependence structures correspond to functions provided by user for automatic exploration of recycling functions.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Task Classes</th>
<th>Dependence Structures</th>
</tr>
</thead>
<tbody>
<tr>
<td>LCS</td>
<td>lcs</td>
<td>lcs: (LEFT, UPPER, DIAG)</td>
</tr>
<tr>
<td>Cholesky</td>
<td>cholesky</td>
<td>cholesky:(MULT)</td>
</tr>
<tr>
<td></td>
<td>trisolve</td>
<td>trisolve:(MULT, CH)</td>
</tr>
<tr>
<td></td>
<td>mmult</td>
<td>mmult:(MULT, TR1, TR2)</td>
</tr>
</tbody>
</table>

information that user provides as dependence structure of tasks in LCS. Each task in LCS generally depends on three tasks referred to as LEFT, UPPER and DIAG. Therefore, the user provides three functions to reach the LEFT, UPPER, and DIAG predecessor for any given task. We look for predecessors of a task reachable from that task using any of the three functions within a bounded number of hops of the given task. Some example recycling functions that are explored automatically (with a 2-hop search) using this information are shown in Table 6.2.

LCS is a uniform benchmark, in which all tasks in execution perform similar operations. In contrast, tasks in most other benchmarks perform different computations, hence might have completely different dependence structures. A good example is Cholesky, illustrated in Figure 6.3(b). A typical parallel implementation of Cholesky factorization consists of three different task classes: (1) Cholesky algorithm on diagonal blocks, (2) triangular solve operation on blocks under diagonal, and (3) matrix-matrix multiplication for other blocks below the diagonal. In the figure, these tasks are represented with red, yellow, and gray
<table>
<thead>
<tr>
<th>Recycling Functions</th>
<th>LCS</th>
<th>Cholesky</th>
</tr>
</thead>
<tbody>
<tr>
<td>func0</td>
<td>lcs: (hop1 to DIAG, hop2 to LEFT)</td>
<td>cholesky: (hop1 to MULT)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>trisolve: (hop1 to MULT)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mmult: (hop1 to MULT)</td>
</tr>
<tr>
<td>func1</td>
<td>lcs: (hop1 to DIAG, hop2 to UPPER)</td>
<td>cholesky: (hop1 to MULT)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>trisolve: (hop1 to MULT)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mmult: (hop1 to MULT, hop2 to MULT)</td>
</tr>
<tr>
<td>func2</td>
<td>lcs: (hop1 to DIAG, hop2 to DIAG)</td>
<td>cholesky: (hop1 to MULT)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>trisolve: (hop1 to MULT, hop2 to MULT)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mmult: (hop1 to MULT)</td>
</tr>
</tbody>
</table>

Table 6.2: Examples of automatically explored recycling functions with a two-hop predecessor traversal for LCS and Cholesky.

colors, respectively and tasks $A_{0,0,0}$, $A_{1,0,0}$, $A_{2,0,0}$ and $A_{2,1,0}$ are labeled as representatives for each task class. Clearly, $A_{2,0,0}$ and $A_{2,1,0}$ have different dependence structures. In such cases, a single dependence structure provided by user is not enough to represent the structure of different task classes. As a result, the traversal path represented by chosen recycling function would not be applicable to some tasks. This would cause these tasks to allocate new memory for their output and would increase memory consumption significantly due to missed recycling opportunities.

Therefore, to generate the most effective recycling functions for all benchmarks, we request the following overall information from the user:

- number of task classes in computation,
- a general dependence structure for each task class in terms of number of predecessors, and
• a function to reach each predecessor in dependence structure of each task class.

In addition, a small problem size from the user to perform the verification before the actual large-scale calculation would ensure that the verification costs remain low. We discuss the use of such a recycling function on a larger problem size in the next section. In case there is more than one task class in the task graph, we explore possible set of recycling functions for each class, and then generate the \textit{combined} recycling functions that represent an individual recycling function for each task class. We verify the correctness of these combined functions as explained above. An example dependence structure provided for Cholesky and the combined recycling functions generated automatically are shown in Table 6.1 and Table 6.2.

Assuming there are \( m \) task classes in a computation, the number of combined recycling functions that we verify is \( \prod_{i=0}^{m-1} p_i \), where \( p_i \) is number of individual recycling functions explored for task class \( i \). Each \( p_i \) increases further with the number of hops explored. Our experimental analysis shows that we are able to verify a large number of combined recycling functions efficiently.

\section*{6.4 Problem-Size Independent Store Recycling with Auto Correction Support}

The algorithm in Section 6.3 identifies candidate recycling functions for arbitrary execution schedules of a given task graph. Programs often produce structurally similar but not identical task graphs for different inputs or problem sizes. Our representation of the recycling function in terms of predecessors relative to a task naturally makes it problem-size independent. However, a recycling function valid for one problem size (or input) might not
be valid for a different input or problem size. Repeated verification of a candidate recycling function for every input to a program can be prohibitively expensive, especially for large problem sizes. In this section, we present our approach to ensure continued correct execution of a task graph despite the chosen recycling function being invalid.

We ensure correct execution by providing the following guarantees.

**Guarantee 1.** A recycling function does not result in concurrent recycling.

A recycling function might map two different tasks in execution onto the same task. Such mapping results in a data race if one task does not causally precede the other. Although we are able to detect such violations in verification phase using vector clocks, we remove vector clocks and related mechanisms during normal execution to minimize performance impact.

Instead, to detect concurrent recycling efficiently, we hold an additional `active_write` flag in each allocated memory (for task’s output) and protect it with a lock for synchronized access. When a task is about to perform a valid recycle operation, it raises this flag as an indication that memory for corresponding data item is currently being written by a task for its output. During the time `active_write` flag is raised, any other task trying to recycle the same data item checks the flag’s value, notices that it is currently in use and allocates new memory. Once the task that raises the flag (i.e., the current definition of the data item) finishes execution successfully, it changes the flag’s value back to `false` for future recycle operations on the same data item.

Figure 6.4 illustrates how concurrent recycling on the same data item is detected. In both (a) and (b), completed tasks (A, E and F) are shown in green, whereas yellow represents tasks (B) that are currently executing. In (a), fields shown for A indicate that A’s output resides in data item D and A’s output has not been recycled yet. B wants to recycle
Figure 6.4: Concurrent recycle detection on an example task graph. (a) and (b) show the system state before and after task $B$ recycles task $A$’s data item $D$. Task $C$ trying to recycle the same data item is forced by the runtime to allocate new memory.

$A$’s output and does so successfully. The change in system state after the recycle operation is shown in (b). After recycle operation is completed, another task $C$ attempts to recycle $A$’s output, but fails to do so since $D$’s active write flag (“awrite” in the figure) indicates that another task ($B$) is currently overwriting $D$. As a result, $C$ allocates new memory for its own output.

**Guarantee 2.** A data item recycled before all of its uses are done is correctly computed again by re-execution of its producer task.

A data item might be recycled by a task even if there are still remaining successors which will access that data item later in the execution. We guarantee correct execution in such scenarios by employing an overwrite detection and an accompanied re-execution mechanism at runtime.

Each task holds an additional recycled flag that indicates if the output of the task has been overwritten by another task. To detect a premature recycle operation, each task checks the recycled flag of all of its immediate predecessors before and after it performs its main
(a) before recycling  (b) after recycling

Figure 6.5: Eager recycle detection on an example task graph. (a) and (b) show system state before and after task B recycles task A’s data item D. C detects that A’s output has been recycled and initiates its re-execution.

computation. If the flag is set to true for any of them, the task resets itself and starts a re-execution procedure for its immediate predecessors whose output have been overwritten. Note that, during this time, no other task can recycle the data item that the task itself is recycling, since active_write of the corresponding data item would not be changed back to false until the task successfully finishes its own execution.

An example scenario for detection of a data item that has been recycled too soon is illustrated in Figure 6.5. Different colors correspond to the same task status as in Figure 6.4. (a) shows the runtime state before B recycles A’s output stored in D. After the recycling operation, with the corresponding state shown in (b), C tries to access the data produced by its immediate predecessor A. Since A’s output has been already recycled, C detects the recycle operation by checking A’s recycled flag and initiates re-execution of A.

In addition to providing correct execution regardless of the recycling function being used, the execution should also proceed in an efficient manner. If a recycling function is observed to lead to premature recycling or task re-executions, we switch to the recycling
function that results in the next largest number of recycles, so that we do not continue to pay re-execution costs by employing the same incorrect recycling function for rest of the execution.

6.5 Implementation

In this section, we demonstrate an implementation of the algorithms presented in Sections 6.3 and 6.4 in the context of an existing task graph scheduling framework: NABBIT [4]. NABBIT is a provably time-efficient task graph scheduler using work stealing. In this section, we first briefly describe task graph scheduling performed by the NABBIT framework and then explain the extensions to support our recycling algorithms.

6.5.1 Task graph scheduling using NABBIT

In NABBIT, each task in computation is referred by a task key, a unique identifier to relate different references to the same task without the need for a pre-allocated task object. User provides two functions to the runtime: one to return a list of immediate predecessors of a task given its key and the other to define the computation to be performed by each task.

The runtime stores a task descriptor for each task created during execution. A task descriptor mainly consists of a join counter to track the number of outstanding predecessors for a task, a notifyArray to store successors which are to be notified once the task finishes its execution, status to represent execution status of the task at the moment and data pointer that points to the memory region, which the task uses to produce its output.

Task graph execution is orchestrated by the runtime through a concurrent hash map. A task descriptor is inserted into the hash map for each task that is created during execution. Execution of task graph starts with the insertion of a sink task (task with no outgoing dependences) into the hash map. Each inserted task first traverses its predecessors and
checks their status. For each predecessor whose task descriptor does not exist in the hash map, a new task descriptor is created. This way, the task graph is expanded in a recursive fashion until a source task (task with no incoming dependences) is found. If a predecessor is to be computed by another thread, the task registers itself in the predecessor’s notifyArray, so that it can be notified once predecessor finishes execution. Each time a predecessor completes, join of the task is decremented. Tasks are scheduled for execution once their join counter reaches 0. Execution of a task graph terminates once the sink task completes its execution.

### 6.5.2 Extensions to the NABBIT Scheduler

The evaluation of recycling functions is implemented in the NABBIT scheduler by augmenting the relevant events as shown in Figure 6.2 and does not require any implementation details specific to the NABBIT scheduler. In order to detect incorrect recycle operations, we hold an additional boolean field in each task descriptor, referred to as recycled. Once a task’s output data item has been recycled, its recycled flag is raised. Successors of this task would observe the incorrect recycle operation by checking the value of this flag either when they first traverse the task, before they perform their respective computation, or after they finish their computation.

Upon observation, a successor initiates re-execution of the overwritten predecessor. First, the current task descriptor of the predecessor is replaced with a new one, where each field is assigned to its initial value. The predecessor to be re-executed traverses its own dependencies and transitively re-executes any predecessors whose output has also been recycled. Just like any other task, a re-executed task performs its computation when its join
counter reaches 0. Upon successful completion, it notifies the successors that observed the incorrect recycle operation in the first place.

In order to provide a low overhead re-execution mechanism, we need to ensure that the producer task of a recycled, but still needed, data item is re-executed only once, and not by every observer of the incorrect recycle operation. We associate each task descriptor with a life number. When multiple consumers of a data item detect the incorrect recycle, each consumer checks if a re-execution procedure has been already initiated for the producer task with current life number. The first consumer that observes this condition performs the actual re-execution. To facilitate this, we maintain a separate hash table that associates a given task with its most recent life number that has been re-executed. Note that this hash table is initially empty and a record for a given key is inserted only if a task needs to be re-executed.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>N</th>
<th>B</th>
<th>T</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>LCS</td>
<td>256Kx256K</td>
<td>2Kx2K</td>
<td>16384</td>
<td>1</td>
</tr>
<tr>
<td>LU (d1)</td>
<td>5Kx5K</td>
<td>128x128</td>
<td>22140</td>
<td>4</td>
</tr>
<tr>
<td>LU (d2)</td>
<td>10Kx10K</td>
<td>128x128</td>
<td>173880</td>
<td>4</td>
</tr>
<tr>
<td>Cholesky (d1)</td>
<td>5Kx5K</td>
<td>128x128</td>
<td>11480</td>
<td>3</td>
</tr>
<tr>
<td>Cholesky (d2)</td>
<td>10Kx10K</td>
<td>128x128</td>
<td>88560</td>
<td>3</td>
</tr>
<tr>
<td>FW (d1)</td>
<td>2.5Kx2.5K</td>
<td>128x128</td>
<td>8000</td>
<td>4</td>
</tr>
<tr>
<td>FW (d2)</td>
<td>5Kx5K</td>
<td>128x128</td>
<td>64000</td>
<td>4</td>
</tr>
<tr>
<td>SW (d1)</td>
<td>3Kx3K</td>
<td>128x128</td>
<td>17576</td>
<td>2</td>
</tr>
<tr>
<td>SW (d2)</td>
<td>6.2Kx6.2K</td>
<td>128x128</td>
<td>132651</td>
<td>2</td>
</tr>
<tr>
<td>Heat2D (d1)</td>
<td>7.5Kx7.5K</td>
<td>128x512</td>
<td>9000</td>
<td>1</td>
</tr>
<tr>
<td>Heat2D (d2)</td>
<td>16Kx16K</td>
<td>128x512</td>
<td>40960</td>
<td>1</td>
</tr>
<tr>
<td>Rician (d1)</td>
<td>256x256x256</td>
<td>32x32x32</td>
<td>10240</td>
<td>2</td>
</tr>
<tr>
<td>Rician (d2)</td>
<td>512x512x512</td>
<td>32x32x32</td>
<td>81920</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 6.3: Matrix size (N), block size (B), total number of tasks (T) and number of task classes (C) for each benchmark. “(d1)” and “(d2)” represent different problem sizes.
6.6 Experimental Evaluation

We have extensively evaluated our recycling based approach and its automation through a series of experiments. We evaluate our approach on Intel Xeon Phi 7110P many-core coprocessor. It has 61-core running at 1.1GHz with 32KB L1 cache, 512KB L2 cache per core and 8GB device memory for all cores. Xeon Phi represents a system with a large number of cores and relatively modest memory per-core, two characteristics that are expected to be very common among nodes of future supercomputers, and thus was an ideal choice for our experiments. All benchmarks were compiled using the Intel icc-13.3.163 compiler with -O3 optimization and -no-vec flag. All experiments were executed in the native mode directly on many-core coprocessor without sharing any work with the host processor. All experimental data we report is the average of 5 separate executions. We also show the standard deviation as the error bar for the small fraction of cases where the standard deviation is non-trivial. In the rest of this section, we shall compare executions with (a) no data reuse (referred to as single-assignment), (b) memory reclamation based on information on the number of uses for each task’s output (referred to as use-count) (c) hand-optimized implementations in NABBIT (referred to as handwritten-recycle), and (d) our approach to automated recycling function selection (referred to as auto-recycle).

6.6.1 Benchmarks and Applicability of Use-Count Based Garbage Collection

We used seven benchmarks for our evaluation studies: LCS (longest common subsequence), Smith-Waterman (SW for short, local sequence alignment algorithm [95]), Floyd-Warshall (FW for short, all-pairs shortest path algorithm in a weighted graph), two dense linear algebra kernels, which are LU decomposition and Cholesky factorization, Heat2D (a
standard two-dimensional stencil) and Rician Denoising [27] (an image processing benchmark). The configuration for all benchmarks is shown in Table 6.3.

A candidate alternative approach for comparison against our method is the use-count based garbage collection scheme, a standard technique for memory-optimized execution of task graphs. Existing runtimes such as Intel CnC [25] present a tuning interface to programmers for specifying the number of uses associated with each data item. The main idea is to decrement the use count of a data item after each access on the data item by a `get()` call. Once the use count reaches 0, memory allocated for the data item is reclaimed by the runtime. The main advantage of our technique as compared to this approach is in terms of programmer productivity and general applicability of the technique. Thus, before reporting performance comparisons, we discuss the applicability and programming effort required on our target set of benchmarks.

It turns out that applying use-count based memory reclamation to the benchmarks we are evaluating demands significant additional effort from the programmer. For linear algebra kernels LU and Cholesky, specification of linear functions to determine the exact use count of each data item is required. Moreover, one needs to pay attention to any boundary conditions (particularly in stencils such as Heat2D) to avoid over-specification of use counts, which lead to memory leaks until the task graph completes execution. An existing study [26] to automate extraction of use counts in Intel CnC shows that this can be a significant challenge for programmers. Instead, our recycling approach totally eliminates the need for use counts, minimizing the involvement required of the programmer in the process.

In some cases, the exact number of uses of data items might be unknown at runtime. For example, after each iteration, Rician Denoising performs a global convergence check.
by a reduction on convergence status of all data items. The convergence check is sped up with a short-circuiting evaluation, which initiates the next iteration immediately if any of the data items have changed significantly. Introduction of short-circuiting for performance eliminates the need to access convergence status of all data items. However, it can lead to memory leaks, since the number of accesses on these elements can not be determined a priori. In contrast, our recycling approach is not based on use-count information and is able to reclaim assigned memory regions even when performance optimizations such as short-circuiting are applied.

### 6.6.2 Comparison Between Single-Assignment and Recycling

Our first experiment compares the performance of each benchmark with single-assignment (new memory allocation for each task execution) and memory recycling based implementations. The goal of this experiment is to demonstrate the memory efficiency of recycling-based implementations. For this comparison, we refer to Figure 6.6(a) through (f) and Figure 6.7(a) through (g). Single-assignment version allocates a new data item for each task in execution and hence leads to maximum possible memory consumption for each benchmark. In one of the memory recycling versions (labeled as “handwrite-recycle”), recycling operations are dictated directly by user in application code, and the minimum memory consumption possible is achieved with manual effort. We created this version as the best performing baseline against which our scheme can be evaluated. Except LCS, we run each benchmark with a small and a large problem size. These instances are distinguished by “(d1)” and “(d2)” labels following benchmark name in each subfigure. On large problem instances, single-assignment version can not even finish execution due to out of memory errors, so no bar is shown for this version in corresponding figures. On the
Figure 6.6: Execution time of LU, Cholesky and FW with single-assignment, use-count, optimized handwritten-recycle, and auto-recycle versions. Performance of all benchmarks are evaluated on two different problem sizes labeled as “(d1)” and “(d2)”. Execution of single-assignment versions in LU (d2), Cholesky (d2) and FW (d2) did not finish as they ran out of memory.
Figure 6.7: Results of the same experimental setup in Figure 6.6 for SW, Rician, Heat2D and LCS. Execution of single-assignment versions in SW (d2), Heat2D (d2) and Rician (d2) did not finish as they ran out of memory.
other hand, handwritten-recycle version completes execution successfully, and moreover, continues to scale well for almost all benchmarks. Specifically, it achieves (over 1 thread versions) 144.1x speedup for LU (244 threads), 144.3x for Cholesky (244 threads), 137.3x for FW (244 threads), 51x for SW (61 threads), 15.3x for Heat2D (244 threads) and 49.6x for Rician (244 threads). On smaller problem instances, performance of single-assignment version degrades with increasing number of threads. This trend can be clearly seen in LU (d1), Cholesky (d1), Heat2D (d1) and Rician (d1) instances at 61 threads and onward. After a cache behavior analysis, we observed that L1 cache misses and number of data references for single-assignment version increase considerably with increasing number of threads, leading to significant overheads. Clearly, memory recycling is a better alternative to the single-assignment scheme, in terms of better performance and the ability to handle larger problem sizes. Heat2D is inherently memory-bound and does not scale as well as other benchmarks using Nabbit’s dynamic scheduling approach.

We also evaluated use-count based garbage-collection and reported the results with bars labeled as “use-count” in Figure 6.6 and Figure 6.7. Results show that it performs no better than the manually optimized versions (“handwrite-recycle”). As previously emphasized, the key advantage of our method is its wider applicability and programmer productivity. However, in achieving these benefits, we do not have any performance loss over the use-count based garbage collection scheme.

6.6.3 Auto-recycling Overheads

Our previous experiment has demonstrated the advantage of memory recycling over single-assignment. The next question is the effectiveness of automated recycling versions. To this end, we measure performance of our scheme, which is shown by “auto-recycle” bars
Figure 6.8: Correlation between number of reuses (x-axis) in verification and observed memory consumption (in GB) (y-axis) in actual runs. Each point in the figures represents a pair of these two values for an automatically generated recycling function.

In Figure 6.6 and Figure 6.7, and calculate its overheads over handwritten-recycle versions. The auto-recycle versions for these experiments were generated by first exploring recycling functions automatically with a 2-hops traversal, and then employing the one which leads to the most number of simulated recycling operations. At 61 threads, our scheme leads to 4.6%, 1.8%, 0.3%, 0.2%, 2.6%, 5.5% and 3% overheads over handwritten-recycle version, respectively for LCS, LU (d2), Cholesky (d2), FW (d2), SW (d2), Heat2D (d2) and Rician (d2). These small overheads demonstrate that recycling approach can be effectively automated. The reasons for the small overheads are multi-fold: a small amount of synchronization to detect concurrent and premature recycling in automatic versions, and a more direct mapping and avoidance of indirect references in the handwritten versions.

To further demonstrate the effectiveness of our methods for automatically generating recycling functions, Table 6.4 reports memory consumption (in GB) of single-assignment,
handwritten-recycle and auto-recycle versions on LCS and large instances of remaining benchmarks. The table further splits memory consumption of auto-recycle versions into two as the consumption of auto-recycling functions that lead to minimum (“Min-Auto”) and maximum (“Max-Auto”) values. In most benchmarks, except FW, we are able to generate recycling functions that consume the same amount of memory as the handwritten-recycle versions. Recall that from the set of recycling functions that are found to be correct, we choose the version which has the most reuse in verification. We observed that, in all benchmarks, the minimum consumption is achieved by the chosen version. To further establish that this method works well, in Figure 6.8, we present the reuse count in verification step (x-axis) and the actual memory consumption of each generated recycling function (y-axis). Clearly, as the number of reuse counts increases, total memory consumption drops in actual runs revealing an inverse correlation pattern. We can see that while the function with highest estimated reuse leads to low memory requirements, other correct functions do have higher memory requirements. To summarize, the data presented in Figure 6.6, Figure 6.7, Figure 6.8 and Table 6.4 clearly demonstrate that automated generation of recycling function is working well – both in terms of memory requirements and execution time.

In FW, the hand-optimized implementation reduces memory usage using additional ordering constraints that are not implied by the dependence predecessors. In particular, the implementation begins with a two-dimensional matrix and explicitly orders tasks that perform subsequent updates to the same matrix block. Our approach is only based on data dependences and thus cannot match manually optimized FW’s memory overhead. Note that the memory usage is still considerably lower than the single-assignment version. The
auto-recycle version can match the hand-optimized version if the additional dependences are provided by the user when we explore candidate recycling functions.

### 6.6.4 Recycling Function Verification Costs

In Figure 6.9, we evaluate the costs associated with the verification of a set of candidate recycling functions. For simplicity, data is only reported for 61 thread execution. The reported costs mainly consist of vector clock calculation and verification of recycling functions with the help of vector clocks. Problem sizes used during verification are much smaller than the problem sizes in actual runs, since the goal is to determine the appropriate recycling functions with low overheads, and before large-scale runs.

In the figure, the first bar ("single-assignment") represents a baseline execution, without any recycling, vector clock calculations, or function verification. Remaining bars show respectively the time it takes to calculate vector clocks on top of single-assignment version, the time it takes to verify a single correct recycling function provided by user ("1_func_check") and to verify recycling functions explored automatically by 1 ("1_hop"), 2 ("2_hop"), and 3 ("3_hop") hops. Note that the last four bars for each benchmark also

Table 6.4: Memory consumption (in GB) of single-assignment ("Single"), handwritten-recycle ("Hand") and auto-recycle versions that lead to minimum ("Min-Auto") and maximum ("Max-Auto") consumption.

<table>
<thead>
<tr>
<th></th>
<th>Single</th>
<th>Hand</th>
<th>Min-Auto</th>
<th>Max-Auto</th>
</tr>
</thead>
<tbody>
<tr>
<td>LCS</td>
<td>0.26</td>
<td>0.004</td>
<td>0.004</td>
<td>0.008</td>
</tr>
<tr>
<td>LU (d2)</td>
<td>22.79</td>
<td>0.83</td>
<td>0.83</td>
<td>1.65</td>
</tr>
<tr>
<td>Cholesky (d2)</td>
<td>11.60</td>
<td>0.42</td>
<td>0.42</td>
<td>0.84</td>
</tr>
<tr>
<td>FW (d2)</td>
<td>8.38</td>
<td>0.21</td>
<td>0.61</td>
<td>1.19</td>
</tr>
<tr>
<td>SW (d2)</td>
<td>8.69</td>
<td>0.17</td>
<td>0.17</td>
<td>0.34</td>
</tr>
<tr>
<td>Heat2D (d2)</td>
<td>21.47</td>
<td>4.29</td>
<td>4.29</td>
<td>4.83</td>
</tr>
<tr>
<td>Rician (d2)</td>
<td>21.47</td>
<td>3.22</td>
<td>3.22</td>
<td>4.36</td>
</tr>
</tbody>
</table>
include the cost for vector clock calculation, which is an integral part of the verification scheme. Table 6.5 reports how many recycling functions are checked for correctness by “1_hop”, “2_hop” and “3_hop” versions. Results show that vector clock calculation does not incur any visible costs over baseline. 1 correct user-provided recycling function is checked for correctness efficiently without any overheads and the time it takes to execute the application while verifying functions generated with 1 and 2 hops are very close to baseline. For the 3 hops case, there is an observable increase in the execution time, especially in LU, FW and Rician where more than 70000 distinct recycling functions are checked for correctness. Though it is a high relative cost for execution with a small problem size, in absolute terms, it is still small (about 40 seconds for Rician). The entire verification phase takes less than 1 second for most benchmarks. Note that the generation of recycling functions is performed sequentially before verification starts, but the time ranges from a few microseconds to 0.035 seconds.

Figure 6.9: Costs associated with verification of recycling functions at 61 threads.
Our analysis on verified recycling functions revealed that in addition to efficiency, our technique is also able to filter out incorrect recycling functions with perfect recall and coverage, i.e., there are no false negatives or false positives in the entire process.

### 6.6.5 Re-execution Overheads

For all the benchmarks considered, we observed that all recycling functions verified for the smaller problem size continue to be valid and efficient for larger problem sizes. We therefore evaluate the performance impact when using one of the recycling functions identified as invalid.

For each benchmark, we evaluate the impact of all recycling functions that are explored with 2 hops and determined to be incorrect. For Rician Denoising, we picked 1500 samples from the 16,639 incorrect recycling functions identified within 2 hops. We start the production phase with one incorrect recycling function and switch to a correct one once an incorrect recycle operation is detected at runtime. We divide possible recovery overheads into 9 bins as <1%, 1-5%, 5-10%, ..., 50-60%. Figure 6.10 shows the number of

<table>
<thead>
<tr>
<th></th>
<th>1-hop</th>
<th>2-hops</th>
<th>3-hops</th>
</tr>
</thead>
<tbody>
<tr>
<td>LCS</td>
<td>1/3</td>
<td>8/12</td>
<td>33/39</td>
</tr>
<tr>
<td>LU</td>
<td>1/12</td>
<td>16/1440</td>
<td>192/70400</td>
</tr>
<tr>
<td>Cholesky</td>
<td>1/6</td>
<td>12/240</td>
<td>114/4400</td>
</tr>
<tr>
<td>FW</td>
<td>1/12</td>
<td>16/1440</td>
<td>192/70400</td>
</tr>
<tr>
<td>SW</td>
<td>1/4</td>
<td>6/36</td>
<td>18/196</td>
</tr>
<tr>
<td>Heat2D</td>
<td>0/5</td>
<td>5/30</td>
<td>88/155</td>
</tr>
<tr>
<td>Rician</td>
<td>0/98</td>
<td>266/16905</td>
<td>3857/100000</td>
</tr>
</tbody>
</table>

Table 6.5: Number of recycling functions that are checked and actually verified as correct, when recycling functions are explored automatically with 1, 2, and 3 hops. Each pair \( n_1/n_2 \) denotes that \( n_1 \) functions are determined to be correct after considering \( n_2 \) functions.
Figure 6.10: Distribution of incorrect recycling functions identified for each benchmark in terms of their re-execution overheads (in percentage of original execution time). 4, 1424, 228, 1424, 30, 25 and 1500 functions are evaluated for LCS, LU, Cholesky, FW, SW, Heat2D and Rician, respectively.

incorrect recycling functions in each overhead range as a percentage of total number of incorrect recycling functions evaluated – all executions are with 61 threads. Results show that number of incorrect recycling functions that cause less than 1% overhead turns out to be 50%, 100%, 84%, 16% and 50% of the total number of incorrect functions for LCS, LU, Cholesky, FW, and SW, respectively. Remaining portion of evaluated incorrect functions for the first four benchmarks result in overheads between 1 and 5%, whereas it spans almost the entire range of bins for SW.

For the last experiment, we now focus on one representative incorrect recycling function, and evaluate how the costs vary with different number of threads. Chosen incorrect functions map each task in execution to an immediate predecessor, whose output has the largest number of usages compared to the other immediate predecessors of the same task. Similar to previous experiment, once an incorrect recycle operation is detected by runtime,
we switch to a correct recycling function for rest of the execution. Figure 6.11 shows re-execution overheads over a baseline run, in which chosen recycling function does not lead to any re-executions. Recovery overheads on up to 61 threads are below 1% for most of the benchmarks with the exception of SW at 32 threads. We observed that overheads for SW have a large standard deviation, which is shown in the figure as an error bar. In practice, impact of an incorrect recycling function on execution is non-deterministic. Depending on the schedule resulting from the underlying task graph scheduler in each run, a data item might be recycled too soon before any task finishes its usage on it. In such extreme cases, this leads to significant portion of task graph to wait on reproduction of recycled data item, and can hamper the available parallelism in application significantly. With more threads, this trend becomes more visible, as higher recovery overheads at 122 or 244 threads show. Nevertheless, even at 122 cores, overheads for LCS, LU, Cholesky are below 2%, whereas it is around 5% in SW.

Figure 6.11: Re-execution overheads (in percentage) with a representative incorrect recycling function with varying number of threads.
Heat2D is a representative of a class of benchmarks that incur significant re-execution overheads. Such benchmarks, including stencil codes, involve overwrites of a larger fraction of the data items. When we attempt to re-execute a task, we often find that the task’s predecessors have also been overwritten. This transitively leads to re-execution of a large fraction of the total tasks. Therefore the penalty of an incorrect recycling function is much higher than in other benchmarks.

### 6.7 Related Work

Data-flow models have existed for four decades now [44, 9]. A key concept in data-flow is the notion of task graphs and task dependences. Data flow models are also considered closely related to functional languages. Scheduling of static task graphs on multiprocessors is one of the most widely studied topics in computer science, and the early work has been summarized in an excellent survey [89]. Recent interest in task-graph or data-flow like models for parallel computing can be seen from projects such as Concurrent Collections or CnC [25], OMPSS [50] and its variants, and others [99]. In the meantime, scheduling of dynamic task graphs has been an active topic [4, 139].

Memory optimization for data-flow(-like) programming systems is a well-studied problem. To the best of our knowledge, it was first studied as the problem of eliminating array copies (formally referred to as the aggregate update problem) by Hudak and Bloss [77]. As also summarized by Gerard et al. [63], the solutions can be of three types. The first group involves reference counts, an approach that has been used with dynamic task scheduling [87], but can limit the type of programs that can be expressed, and even otherwise, puts a large burden on the programmer. Other approaches include compile-time solutions [124, 63], which are not applicable when the task graph is dynamically generated,
and solutions based on typing the operators, which again are not applicable in our case (or would require additional burden on the programmer if added to the programming model). Another recent effort with a similar focus is from Abu-Mahmeed et al. [2], and involves scheduling to avoid unnecessary copies. Again, the work cannot support a programming model where the task graph is dynamically generated.

The issue of memory reuse in dynamic task-graph based systems has received limited attention. One proposal involves user-provided folding functions (similar to our recycling functions) for CnC [123]. This work requires the user to explicitly map the data items (or single-assignment tags) to physical memory, guaranteeing a memory bound. However, folding functions for different memory bounds might be completely different, imposing additional burden on the user to pick reasonable bounds and corresponding folding functions. We believe this was the motivation for a more recent solution involving an inspector-executor like approach [122] that explicitly expands the full task graph and determines a schedule satisfying a memory bound. In addition to explicit expansion of the task graph, this approach requires expensive analysis for each problem size. In comparison, we can derive recycling functions using execution on a smaller problem size, and can efficiently detect (and recover from) an incorrect recycling function when using them in larger-scale runs.

6.8 Summary

In this work, we addressed a different type of failures which can be introduced into execution by programmers during the efforts for performance optimizations. As a representative scenario of such efforts, we first presented a novel approach for optimizing memory usage when dynamically scheduling single-assignment task graphs. The practical approach
is based on memory recycling hints specified by user and combines the evaluation of various recycling strategies and dynamic recovery from incorrect recycling specifications. The approach allows iterative exploration of the space of recycling functions by the user, incrementally optimizing the implementation while retaining performance portability. We believe that the presented approach would be ideal as a component of the tuning language employed in prevalent dynamic task graph schedulers.
Chapter 7: Conclusion & Future Work

7.1 Conclusion

Fault-tolerance on parallel systems has become one of the most critical challenges in HPC and it has drawn a lot of attention of the community. With the advances in hardware throughout the last decade (emergence of multi-core and today many-core machines), the number of components in large computing-clusters has increased significantly. Such growth rate in computing power came with the expense of decreasing Mean Time Between Failures. The increasing frequency of soft errors due to chip manufacturing trends and power optimization efforts also contributed to the overall failure rate of large HPC systems. All of these factors made failures a norm rather than an exception. Hence, there is a pressing need to address this issue at the software level as well as at the hardware level.

In this dissertation, we proposed fault-tolerance solutions against the three of failures observed frequently; fail-stop failures, soft errors and programmer induced errors. We first directed our attention to big data processing environments and designed a fault-tolerant computing framework that processes queries and data analysis tasks on large scientific datasets (Chapter 2). Next, we presented two fault-tolerant parallel programming models based on different computing paradigms; single program multiple data in Chapter 3, 4 and task-graph execution model in Chapter 5. These programming models provide
fault-tolerance techniques spanning a wide spectrum of complementary solutions, including application-level checkpointing, replica execution and task re-execution, against both traditional fail/stop failures and silent data corruptions. Furthermore, in Chapter 6, we presented how errors induced by programmer specifications to tune application performance can be detected and recovered efficiently in the context of a memory-efficient dynamic task graph scheduler. Our specific contributions with this dissertation can be summarized as follows;

1. We showed how a query processing and data-analysis framework operating on massive scientific datasets can achieve high efficiency of execution, and at the same time can handle single machine and rack failures with only a small increase in execution times. Specifically, for fault-tolerance support, we proposed new data-replication schemes referred as subchunk and subpartition replication that preserves load-balance in the system even after a failure. Moreover, for range-queries on large spatial datasets, we presented a way of using an indexing structure both as a filtering structure and a failure management tool at the same time. For aggregation queries on point datasets, we provided an analytic model that guides distribution of data replicas to minimize communication volume as well as to maintain load balance.

2. We proposed a new parallel programming model that exploits common characteristics of iterative scientific applications involving stencil computations, unstructured grids, or N-body simulations. The programming model provides three major benefits compared to the other existing programming models. First, by a set of abstractions, it hides the details of inter-process communication and work partitioning from the programmers. Also, it provides efficient execution in presence of heterogeneous processing cores by re-partitioning the work on-the-fly automatically. Second,
by capturing the main execution state through a programming abstraction, it supports *application-level checkpointing* for fault-tolerance. The checkpointing scheme requires as little programmer effort as possible and is shown to be superior than the popular system-level checkpointing approach. Lastly, the programming model detects soft error occurrences that impact the main execution state through a low-overhead partial replication strategy which redundantly executes only the most critical instructions in an application.

3. Fault-Tolerance solutions have been mostly developed against fail-stop failures in the context of single program multiple data computing model. Hence, we directed our attention to a different programming paradigm and explored fault-tolerance approaches against soft errors in task graph execution model. We presented a dynamic task graph scheduler which transparently recovers from soft errors that impact data blocks and task descriptors with minimum time and space overheads. Recovery is achieved through a selective and non-collective task re-execution algorithm, which only recovers the corrupted portion of a task graph and does not interfere with the other threads not impacted by a fault. We showed that the overhead of our approach during normal execution is negligible. The cost of recovery from failures is proportional to the work lost in general failure scenarios, whereas failure of a small number of tasks can be recovered with no statistically significant overhead. The proposed scheme can be used in conjunction with traditional checkpoint/restart solutions to decrease checkpointing frequency, hence to reduce synchronization overheads and restart costs.
4. In addition to fail-stop failures and soft errors, we focused on faults due to human factor and showed how errors that are incorporated to execution during performance tuning efforts can be dealt efficiently. Specifically, we presented an approach to memory-efficient task graph scheduling using the novel notion of store recycling functions. We gave definitions for the characteristics of correct and efficient store recycling functions and described an algorithm to efficiently evaluate correctness of a given, or a set of given, user-defined or automatically derived, store recycling functions. While our memory management scheme reduces memory consumption of static task graph programs substantially, it also deals with task graphs with more dynamic dependence relations. Lastly, and more importantly, the proposed scheme is able to dynamically detect and/or recover from incorrect choice of user-provided or automatically derived store recycling functions.

7.2 Future Work

In the future, we would like to extend our contributions in the following directions.

1. In Chapter 3, we presented DISC which is a new parallel programming model for iterative scientific applications. Currently, the overall execution of a DISC program is driven by a processing structure which invokes a set of functions in a specific order to carry out the main computational tasks in each iteration. One drawback of the proposed model is that this processing structure still maintains a critical design of bulk synchronous communications which hamper applications moving towards more levels of parallelism and concurrency. For this reason, we would like to allow a more asynchronous nature for DISC programs by using a finer grained expression for communication and computation operations. Additionally, we would like to extend the
interaction patterns that DISC model currently supports so that new computation patterns such as linear algebra kernels, which are extremely common in HPC systems, are also supported efficiently. We believe that these improvements will make DISC model a stronger candidate to be used in future systems.

2. Chapter 5 presents a fault-tolerant dynamic task graph scheduler algorithm. We would like to extend this work in several ways. Currently, our scheduling algorithm focuses only on the recovery path, i.e. how soft errors can be recovered efficiently without incurring significant overheads once they are detected in a timely fashion. For a more holistic approach, we first would like to incorporate one of the existing approaches in the literature into the scheduler or propose an entirely new low-overhead error detector. Second, our task scheduling currently targets shared memory architectures. To increase the applicability of our fault-tolerance approach, we would like to apply our recovery algorithm on a task scheduler targeting distributed memory architectures as well. We believe that extending the current algorithm in this direction will reveal new and interesting challenges such as considering locality of data blocks associated with the corrupted portion of task graph to the threads participating in recovery process. As another promising direction, we would like to combine our task-reexecution algorithm with traditional checkpoint/restart technique. This way, we will be able to limit the number of task re-executions and explore how our scheme interplays with alternative solutions.
Bibliography


182


