A Map-Reduce-Like System for Programming and Optimizing Data-Intensive Computations on Emerging Parallel Architectures

Dissertation

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

By

Wei Jiang, B.S., M.S.

Graduate Program in Department of Computer Science and Engineering

The Ohio State University

2012

Dissertation Committee:

Gagan Agrawal, Advisor
P. Sadayappan
Radu Teodorescu
© Copyright by
Wei Jiang
2012
Parallel computing environments are ubiquitous nowadays, including traditional CPU clusters and the emergence of GPU clusters and CPU-GPU clusters because of their performance, cost and energy efficiency. With this trend, an important research issue is to effectively utilize the massive computing power in these architectures to accelerate data-intensive applications arising from commercial and scientific domains. Map-reduce and its Hadoop implementation have become popular for its high programming productivity but exhibits non-trivial performance losses for many classes of data-intensive applications. Also, there is no general map-reduce-like support up to date for programming heterogeneous systems like a CPU-GPU cluster. Besides, it is widely accepted that the existing fault tolerant techniques for high-end systems will not be feasible in the exascale era and novel solutions are clearly needed.

Our overall goal is to solve these programmability and performance issues by providing a map-reduce-like API with better performance efficiency as well as efficient fault tolerance support, targeting data-intensive applications and various new emerging parallel architectures. We believe that a map-reduce-like API can ease the programming difficulty in these parallel architectures, and more importantly improve the performance efficiency of parallelizing these data-intensive applications. Also, the use of a high-level programming model can greatly simplify fault-tolerance support, resulting in low overhead checkpointing and recovery.
We performed a comparative study showing that the map-reduce processing style could cause significant overheads for a set of data mining applications. Based on the observation, we developed a map-reduce system with an alternate API (MATE) using a user-declared reduction-object to be able to further improve the performance of map-reduce programs in multi-core environments. To address the limitation in MATE that the reduction object must fit in memory, we extended the MATE system to support the reduction object of arbitrary sizes in distributed environments and apply it to a set of graph mining applications, obtaining better performance than the original graph mining library based on map-reduce.

We then supported the generalized reduction API in a CPU-GPU cluster with the ability of automatic data distribution among CPUs and GPUs to achieve the best-possible heterogeneous execution of iterative applications. Finally, in our recent work, we extended the generalized reduction model with supporting low overhead fault tolerance for MPI programs in our FT-MATE system. Especially, we are able to deal with CPU/GPU failures in a cluster with low overhead checkpointing, and restart the computations from a different number of nodes. Through this work, we would like to provide useful insights for designing and implementing efficient fault tolerance solutions for the exascale systems in the future.
This is dedicated to the ones I love: my parents and my younger sister.
Acknowledgments

It would not have been possible to write this doctoral dissertation without the help and support of the kind people around me, to only some of whom it is possible to give particular mention here.

First of all, I would like to thank my advisor Prof. Gagan Agrawal. This dissertation would not have been possible without the help, support and patience of Prof. Gagan Agrawal, not to mention his advice and unsurpassed knowledge of parallel and distributed computing in computer science. He has taught me, both consciously and unconsciously, how good computer systems and software should be done. I appreciate all his contributions of time, ideas, and funding to make my Ph.D. life productive and stimulating. The joy, passion and enthusiasm he has for his research were contagious and motivational for me, even during tough times in the Ph.D. pursuit. I am also indebted to him for the excellent example he has provided as a successful computer scientist and professor.

Second, I would like to acknowledge the financial, academic and technical support of the Ohio State University, particularly in the award of a University Fellowship that provided the necessary financial support for my first year. The library facilities and computer facilities of the University, as well as the Computer Science and Engineering Department, have been indispensable. I also thank the Department of Computer Science and Engineering for their support and assistance since the start of my graduate associate work in 2008,
especially the head of department, Prof. Xiaodong Zhang. My work was also supported by the National Science Foundation.

Also, I would like to thank my colleagues in my lab. The members of the Data-Intensive and High Performance Computing Research Group have contributed immensely to my personal and professional time at Ohio State. The group has been a source of friendships as well as good advice and collaboration. I am especially grateful for the fun group of original members who stuck it out in graduate school with me all the way.

My time at Ohio State was made enjoyable in large part due to the many friends and groups that became part of my life. I am thankful for the pleasant days spent with my roommates and hangout buddies with our memorable time in the city of Columbus, for my academic friends who have helped me a lot in improving my programming and paper writing skills, and for many other people and memories of trips into mountains, rivers and lakes.

Special thanks are given to my family for their personal support and great patience at all times. My parents and my younger sister have given me their unequivocal support throughout, as always, for which my mere expression of thanks likewise does not suffice.

Last, but by no means least, I thank my friends in America, China and elsewhere for their support and encouragement throughout, some of whom have already been named.
Vita

October 29th, 1985 . . . . . . . . . . . . . . . . . . . . . . . . . . Born - Tianmen, China

2007 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . B.S. Software Engineering,
Nankai University, Tianjin, China

2011 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . M.S. Computer Science and Engineering,
The Ohio State University

Publications

Research Publications


Fields of Study

Major Field: Computer Science and Engineering
# Table of Contents

Abstract . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . ii

Dedication . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . iv

Acknowledgments . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . v

Vita . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . vii

List of Tables . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . xii

List of Figures . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . xiii

1. Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1

   1.1 High-Level APIs: MapReduce and Generalized Reduction . . . . . . . 3
   1.2 Parallel Architectures: CPUs and GPUs . . . . . . . . . . . . . . . . . . . 5
   1.3 Dissertation Contributions . . . . . . . . . . . . . . . . . . . . . . . . . 7
   1.4 Related Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
   1.5 Outline . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22

2. Comparing Map-Reduce and FREERIDE for Date-Intensive Applications . . . 24

   2.1 Hadoop and FREERIDE . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
      2.1.1 Hadoop Implementation . . . . . . . . . . . . . . . . . . . . . . . . . 25
      2.1.2 FREERIDE API and Runtime techniques . . . . . . . . . . . . . . . 26
   2.2 Comparing FREERIDE and Hadoop: Case Studies . . . . . . . . . . . . . 29
      2.2.1 K-means Clustering . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
      2.2.2 Word Count . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
   2.3 Experimental Evaluation . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
      2.3.1 Tuning Parameters in Hadoop . . . . . . . . . . . . . . . . . . . . . . 35
      2.3.2 Performance Comparison for Four Applications . . . . . . . . . . . 38
3. MATE: A MapReduce System with an Alternate API for Multi-core Environments ............................................. 49
   3.1 A Case Study ......................................................... 50
   3.2 System Design ....................................................... 52
      3.2.1 The System API ............................................... 52
      3.2.2 Runtime System ............................................. 54
   3.3 Experimental Results ............................................. 57
   3.4 Summary ............................................................. 63

4. Ex-MATE: Data-Intensive Computing with Large Reduction Objects and Its Application to Graph Mining ........................................ 68
   4.1 Ex-MATE Design ....................................................... 69
   4.2 Parallel Graph Mining Using Ex-MATE ............................... 74
      4.2.1 GIM-V and PageRank .......................................... 76
      4.2.2 Parallelization of GIM-V ..................................... 76
      4.2.3 Other Algorithms ............................................. 78
   4.3 Experimental Results ............................................. 79
   4.4 Summary ............................................................. 85

5. MATE-CG: A MapReduce-Like Framework for Accelerating Data-Intensive Computations on Heterogeneous Clusters .............................. 90
   5.1 System Design and Implementation .................................. 91
      5.1.1 Detailed System API .......................................... 91
      5.1.2 Runtime System ............................................. 93
   5.2 System-Tuning for Optimizing Runtime Execution ...................... 96
      5.2.1 Auto-Tuning for Optimizing Heterogeneous Execution .......... 97
      5.2.2 Determining the Settings for CPU/GPU Chunk Sizes ............. 100
   5.3 Experimental Results ............................................. 104
      5.3.1 Scalability Studies ........................................... 108
      5.3.2 Auto-Tuning and Performance with Heterogeneous Computing .... 110
      5.3.3 System Tuning with Setting CPU/GPU Chunk Sizes ............... 112
   5.4 Summary ............................................................. 113

6. FT-MATE: Supporting Efficient Fault Tolerance for MPI Programs in MATE Systems ........................................ 115
   6.1 Fault Tolerance Approaches ....................................... 116
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>The scheduler_args_t Data Structure Type</td>
</tr>
<tr>
<td>3.2</td>
<td>Descriptions of the Functions in the System API</td>
</tr>
<tr>
<td>5.1</td>
<td>Descriptions of Important Data Types and Functions in the System API</td>
</tr>
<tr>
<td>7.1</td>
<td>Comparison of Size of Checkpoint Information</td>
</tr>
</tbody>
</table>
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 Processing Structures</td>
<td>3</td>
</tr>
<tr>
<td>2.1 Varying Input Split Size</td>
<td>33</td>
</tr>
<tr>
<td>2.2 Varying No. of Concurrent Maps</td>
<td>34</td>
</tr>
<tr>
<td>2.3 K-means: Comparison between FREERIDE and Hadoop</td>
<td>35</td>
</tr>
<tr>
<td>2.4 Apriori: Comparison between FREERIDE and Hadoop</td>
<td>36</td>
</tr>
<tr>
<td>2.5 KNN: Comparison between FREERIDE and Hadoop</td>
<td>37</td>
</tr>
<tr>
<td>2.6 Wordcount: Comparison between FREERIDE and Hadoop</td>
<td>38</td>
</tr>
<tr>
<td>2.7 Scalability of FREERIDE and Hadoop: k-means</td>
<td>39</td>
</tr>
<tr>
<td>2.8 Scalability of FREERIDE and Hadoop: wordcount</td>
<td>40</td>
</tr>
<tr>
<td>2.9 K-means: Varying No. of k</td>
<td>41</td>
</tr>
<tr>
<td>2.10 K-means: Varying No. of Dimensions</td>
<td>42</td>
</tr>
<tr>
<td>2.11 Pseudo-code for k-means using FREERIDE API</td>
<td>45</td>
</tr>
<tr>
<td>2.12 Pseudo-code for k-means using Hadoop API</td>
<td>46</td>
</tr>
<tr>
<td>2.13 Pseudo-code for wordcount using FREERIDE API</td>
<td>47</td>
</tr>
<tr>
<td>2.14 Pseudo-code for wordcount using Hadoop API</td>
<td>48</td>
</tr>
</tbody>
</table>
3.1 One-Stage Execution Overview in MATE ........................................ 54
3.2 K-means: Comparison between MATE, Phoenix and Hadoop on 8 cores .......................... 58
3.3 K-means: Comparison between MATE, Phoenix and Hadoop on 16 cores ....................... 59
3.4 PCA: Comparison between MATE and Phoenix on 8 cores ........................................ 60
3.5 PCA: Comparison between MATE and Phoenix on 16 cores ........................................ 61
3.6 Apriori: Comparison between MATE, Phoenix and Hadoop on 8 cores ......................... 62
3.7 Apriori: Comparison between MATE, Phoenix and Hadoop on 16 cores ....................... 63
3.8 Pseudo-code for apriori using Generalized Reduction API ........................................ 64
3.9 Pseudo-code for apriori using MapReduce API ....................................................... 66
4.1 Execution Overview of the Extended MATE ....................................................... 70
4.2 Matrix-Vector Multiplication using checkerboard partitioning. B(i,j) represents a matrix block, I_V(j) represents an input vector split, and O_V(i) represents an output vector split. The matrix/vector multiplies are done block-wise, not element-wise. ........................................ 72
4.3 PageRank: Comparison between Ex-MATE and PEGASUS on 4, 8 and 16 nodes for a 16G matrix ....................................................... 80
4.4 HADI: Comparison between Ex-MATE and PEGASUS on 4, 8 and 16 nodes for a 16G matrix ....................................................... 81
4.5 HCC: Comparison between Ex-MATE and PEGASUS on 4, 8 and 16 nodes for a 16G matrix ....................................................... 82
4.6 HCC: Comparison between Ex-MATE and PEGASUS on 4, 8 and 16 nodes for a 8G matrix ....................................................... 83
4.7 HCC: Comparison between Ex-MATE and PEGASUS on 4, 8 and 16 nodes for a 32G matrix ....................................................... 84
7.7 Jacobi: Benefit of Checkpoint and Work Re-distribution During GPU Failure

7.8 PageRank: Benefit of Checkpoint and Work Re-distribution During GPU Failure

7.9 Gridding Kernel: Benefit of Checkpoint and Work Re-distribution During GPU Failure
Chapter 1: Introduction

Our overall research work in this dissertation has been motivated by four recent trends. First, the past decade has seen an unprecedentedly data growth as information is being continuously generated in digital format. This has sparked a new class of high-end applications, where there is a need to perform efficient data analysis on massive datasets. Such applications, with their associated data management and efficiency requirements, define the term *Data-Intensive SuperComputing* (DISC) [30]. Examples of these data-intensive applications arise from domains like science, engineering, medicine, and e-commerce.

The second trend is the wide-spread use of parallel computing environments, dominated by CPU clusters, GPU clusters, and CPU-GPU clusters. With CPU clusters as traditional HPC platforms, GPU clusters and CPU-GPU clusters became popular recently due to their cost-effectiveness and energy efficiency. Two of the important considerations with the use of any parallel system are *programmer productivity* and *performance efficiency*. Both GPU clusters and CPU-GPU clusters are lacking with respect to one or both of these factors, as compared to the traditional HPC platforms.

Third, the emergence of map-reduce paradigm [50] and its variants has coincided with the growing prominence of data-intensive applications. As a result, map-reduce has been used for implementing this class of applications in different parallel architectures. Especially, map-reduce was originally developed for large-scale clusters and has been easily
ported and extensively evaluated on multi-cores CPUs [44, 111, 141], a single GPU [34, 70, 72], and GPU clusters [57, 127]. Also, there are many efforts supporting and/or translating other parallel primitives and programming models to GPUs [31, 73, 57, 89, 91, 121], but there is no general support to date for data-intensive computing on a heterogeneous CPU-GPU cluster.

Fourth, fault tolerance has become a major challenge for large-scale HPC systems [32, 120, 87]. It is widely accepted that the existing MPI-based fault tolerant solutions with checkpointing will not be feasible in the exascale era [32], as with growing size of high-end systems and relatively lower I/O bandwidths, the time required to complete a check-point can exceed the Mean-Time To Failure (MTTF). Moreover, with energy consumption being the dominant consideration for exascale systems, and with data movement (including data movement for checkpoints) being one of the major energy consumers, novel solutions are clearly needed.

Therefore, our overall work focuses on the design of a map-reduce-like model with compiler and runtime support for different parallel architectures. In other words, we would like to provide better programming productivity by exposing a simple and high-level API and also achieve better performance efficiency by utilizing massive computing power delivered by different parallel platforms and provide efficient fault tolerance support on top of a high-level programming model.

In this chapter, first, we would like to introduce the background knowledge about the parallel programming models and parallel architectures we have been working on. Then, we will give the overview of our current work including a comparative study and three parallel systems we have developed for different platforms. Finally, we will introduce our
work in model-based fault tolerance support in our parallel systems and discuss related
efforts in all these areas.

1.1 High-Level APIs: MapReduce and Generalized Reduction

This section describes the existing popular API for data-intensive computing, map-
reduce, and a variant of it that we implemented in our systems and refer to as the general-
ized reduction model.

The pseudo-code for the map-reduce paradigm is shown in Figure 1.1(a), which can be
summarized as follows [50]. Map-reduce expresses the computation as two user-defined

```
// outer sequential loop
while () {
    // reduction loop
    for each (element e) {
        (i, val) := process(e);
    }
    sort (i, val) pairs over i
    reduce to compute each rObj(i)
}
```

(a) Map-Reduce

```
// outer sequential loop
while () {
    // reduction loop
    for each (element e) {
        (i, val) := process(e);
        rObj(i) := reduce(rObj(i), val);
    }
    global reduction to combine rObjs
}
```

(b) Generalized Reduction

Figure 1.1: Processing Structures
functions: map and reduce. The map function takes a set of input instances and generates a set of corresponding intermediate output \((key, value)\) pairs. The map-reduce library groups together all of the intermediate values associated with the same key and shuffles them to the reduce function. The reduce function, also written by the user, accepts a key and a set of values associated with that key. It merges together these values to form a possibly smaller set of values. Typically, just zero or one output value is produced per reduce invocation.

The generalized reduction API reduces memory requirements for intermediate data and appears to be more suited for a heterogeneous environment. This API is shown in Figure 1.1(b). The differences between this API and map-reduce are as follows. One difference involves the fact that our API allows developers to explicitly declare a reduction object and perform updates to its elements directly, whereas in map-reduce, the reduction object is implicit and not exposed to the programmer. Another important distinction is that, in map-reduce, all the data elements are processed in the Map step and the intermediate results are then later combined in the Reduce step. With the generalized reduction API, both Map and Reduce steps are combined into a single step where each data element is processed and reduced before next data element is processed. This choice of design avoids the overheads due to sorting, grouping, and shuffling, which can degrade performance in map-reduce implementations. The function, reduce(), is an associative and commutative function. Thus, the iterations of the for-each loop can be performed in any order. The data structure RObj, which is the reduction object, is updated after each iteration, then later combined with all other reduction objects to attain the final result.

The following are the required functions that must be provided by the application developer:
• **Reduction:** The reduction function specifies how, after processing one data instance, a *reduction object* (initially declared by the programmer) is updated. The result of this processing must be independent of the order in which data instances are processed on each processor. The order in which data instances are processed is determined by the runtime system.

• **Combination:** In this function, the final results from multiple copies of a *reduction object* are combined into a single reduction object. A user can choose from one of the several common combination functions already implemented in the system library (such as aggregation, concatenation, etc.), or they can provide one of their own.

It may appear that the generalized reduction API is simply making the *combine* function, which is an option available to developers in map-reduce, a requirement. However, this is not the case. The combine function of map-reduce can only reduce communication. The *(key, value)* pairs are still generated on each node and can result in high memory requirements, causing application slowdowns [80, 77]. With the generalized reduction API, *map, reduce* and possible *combine* are integrated together while processing each element, thus avoiding memory overheads.

### 1.2 Parallel Architectures: CPUs and GPUs

Many high-performance computing clusters are composed of multi-core CPUs and/or many-core GPUs. Recently, the emergence of CPU-GPU clusters has been a dominant HPC platform to the heterogeneous clusters, with the three of the four fastest supercomputers in the world falling in this category. These environments has been popular due to their cost-effectiveness and energy efficiency. As a result, the need for exploiting both the CPU and GPU on each node of such platforms has created a renewed interest in heterogeneous
Here, we especially introduce the parallel architecture of modern CPUs and GPUs we have been working on.

For a modern CPU or GPU, its architecture consists of two major components, i.e., a multi-threading processing component and a multi-layer memory hierarchy.

**Modern CPU Architecture:** Multi-core CPUs have been widely used to achieve high performance for many years. A modern multi-core CPU typically consists of a certain number of CPU cores as the multi-threading processing component, which features an out-of-order super-scalar micro-architecture. The CPU memory hierarchy has a memory controller that connects the CPU cores to the channels of DDR memory and it also provides the high-speed L1/L2/L3 data caches with different sizes on each level. For example, the Intel Core i7-960 CPU offers four cores on the same die running at a frequency of 3.2GHz with newly added 2-way hyper-threading. Each core has a separate 32KB L1 for both instructions and data, and a 256KB unified L2 data cache. All four cores share an 8MB L3 data cache.

**Modern GPU Architecture:** The processing component in a typical GPU is composed of an array of streaming multiprocessors (SMs). Each SM has a set of simple cores that perform in-order processing of the instructions. To improve performance and hide memory latency, the multi-threading support on GPUs allows a few tens of thousands of threads to be launched and hundreds to be active simultaneously. The hardware SIMD structure is exposed to programmers through *thread warps*. Each warp of threads are co-scheduled on the streaming multiprocessor and execute the same instruction in a given clock cycle (SIMD execution).

To alleviate memory bandwidth, the GPU memory hierarchy typically contains several layers. One is the *host memory*, which is available on the CPU main memory. This is
essential as any general purpose GPU computation can only be launched from the CPU. The second layer is the *device memory*, which resides on the GPU card and represents the global memory on a GPU. The device memory is accessible across all streaming multiprocessors and interconnected with the host through a PCI-Express card (version can vary depending upon the card). This interconnectivity enables DMA transfer of data between the host and device memory. The on-chip memories also include a local shared buffer, which is private to each streaming multiprocessor, programmable and supports high-speed access, This local shared buffer is termed as the *shared memory* on NVIDIA cards. Typically, the size of this shared memory was 16 KB.

Tesla 2050 (a.k.a. Fermi) is the latest series of graphic card products by NVIDIA. The architecture of Fermi features 14 or 16 streaming multiprocessors. Each SM has 32 scalar processing units running, each at 1.215GHz, and the memory clock is 3348MT/s. As compared to earlier products, Fermi has a much larger shared memory/L1 cache, which can be configured as 48KB shared memory and 16KB L1 cache, or 16KB shared memory and 48KB L1 cache. Unlike previous cards, an L2 cache is also available.

### 1.3 Dissertation Contributions

We will now give a summary of our previous work starting from a comparative study. **Comparing FREERIDE and Hadoop for Data-Intensive Applications:** Map-reduce has been famous for its simplicity and robustness since it was proposed by Google in 2004, however, the performance aspects of map-reduce less well understood at that time. So, in this work, we compared the programming APIs and performance of the Hadoop implementation of map-reduce with a parallel data-mining system named FREERIDE [81, 82, 83]
that was based on the generalized reduction model and developed earlier by Agrawal’s group at Ohio State.

The main observations from our results are as follows. First, the API and the functionality offered by FREERIDE has many similarities with the map-reduce API. However, there are some differences in the API. Moreover, while FREERIDE was motivated by data mining computations, map-reduce was motivated by searching, sorting, and related applications in a data-center.

Second, we studied three data mining applications, which are k-means clustering, apriori association mining, and k-nearest neighbor search. We have also included a simple data scanning application, word-count. For the three data mining applications we have considered, FREERIDE outperformed Hadoop by a factor of 5 or more. For word-count, Hadoop is better by a factor of up to 2. With increasing dataset sizes, the relative performance of Hadoop becomes better. Overall, it seems that Hadoop has significant overheads related to initialization, I/O, and sorting of $(key, value)$ pairs. Also, Hadoop does not fully utilize the overall resources in the cluster. Thus, despite an easy to program API, Hadoop’s map-reduce does not appear very suitable for data mining computations on modest-sized datasets.

**A MapReduce System with an Alternate API:** Although we found out that FREERIDE could outperform Hadoop for the three data mining algorithms, often by a factor of 5-10, this could be attributed to various reasons, like the difference in processing using Java (as in Hadoop) and C++ (as in FREERIDE), the inefficiency of resource utilization in Hadoop, or the overheads of the Hadoop distributed file system. This leaded to the implementation of MATE, which was created by introducing the generalized reduction API as part of the Phoenix system (a shared memory and C-based implementation of map-reduce [111]).
MATE allows a comparison of the APIs and The distinct API in MATE still offers a high-level interface, leading to much simpler implementations than alternate approaches like Pthreads or OpenMP. At the same time, it reduces the memory requirements associated with the large number of \((key, value)\) pairs that can be generated in the original map-reduce API for several applications, even as compared to the map-reduce implementations that use \textit{combine} functions.

Our implementation on top of the Phoenix system allows us to carefully evaluate the performance advantages of the new API. We have extensively evaluated our system using three data mining applications and compare it with both Phoenix (with original map-reduce API) and Hadoop on two shared-memory platforms. The three data mining applications are k-means clustering, apriori association mining, and principal components analysis (PCA). The experimental results show that our system outperforms Phoenix for all the three applications by an average factor of 50% and also leads to reasonable speedups in both environments. Hadoop is much slower than both the systems.

Overall, we observe that while map-reduce eases the burden of programmers, its inherent API structure may cause performance losses for some applications due to the rigid two-stage computation style. Our approach, which is based on the generalized reduction, offers an alternate API, potentially useful for several sub-classes of data-intensive applications.

\textbf{The Extended MATE with Large Reduction Object Support}: The motivation of this work arises from two observations. First, many challenging real-world problems can be modeled as graphs [29, 88, 110, 124] and graph analysis or mining can help solve these problems [38, 112, 42, 138, 132, 85, 71]. Processing such large-scale graphs has become
inevitable due to the increasing ubiquity and growing size of these graphs. Examples include the World Wide Web and social networks, such as those created by friendship links on Facebook. Because of the size of the graphs, it is natural to use parallel implementations to solve the problems. There has been much interest in the use of map-reduce for scalable graph mining. The work at CMU has developed a graph mining package based on the Hadoop implementation of map-reduce [86].

Second, a limitation of the explicit reduction object based approach in MATE has been that it can only work when the reduction object can be maintained in the main memory of each node. In comparison, map-reduce implementations do not have a similar restriction and can maintain (key, value) pairs on a disk. Many graph mining algorithms could not have been implemented in the original MATE system, because the reduction objects can be really large with typical problem sizes.

In this work, we present the design and evaluation of a system we refer to as Extended MATE (Ex-MATE) to address the above limitation with support of parallelizing graph mining applications. We have compared our Ex-MATE implementations with PEGASUS implementation. We used three algorithms originally implemented in PEGASUS, which are PageRank, Diameter Estimation, and finding Connected Components. For each of three applications, PEGASUS provides a naive implementation as well as an optimized block version. Our experimental results show that our system outperforms PEGASUS for all the three applications with factors of 9 to 35.

Overall, on a multi-core cluster of 16 nodes (128 cores), the results were as follows. For PageRank, our system is 35 times faster than its naive version on PEGASUS and 10 times faster than the block version. For Diameter Estimation, we could achieve a 27-time speedup than the naive version and an 11-time speedup compared with the block version.
For finding Connected Components, our system is 23 times as fast as the naive version on PEGASUS and 9 times as fast as the block version. Thus, we have demonstrated that the MATE approach can be used for applications involving very large reduction objects and can continue to outperform implementations based on the original map-reduce API.

**A MapReduce-like Framework for the CPU-GPU clusters:** This work is driven by the growing prominence of *Data-Intensive SuperComputing* (DISC) [30] and the emergence of CPU-GPU clusters as a dominant HPC platform, with the three of the four fastest supercomputers in the world falling in this category. The reasons for the popularity of these environments include their cost-effectiveness and energy efficiency. However, heterogeneous platforms are currently lacking with respect to both of programming productivity and performance efficiency, as compared to the traditional HPC platforms.

Due to the popularity of map-reduce, there have been some efforts on supporting map-reduce on a single GPU [34, 70, 72], GPU clusters [57, 127], as well as many efforts supporting and/or translating other parallel primitives and programming models to GPUs [31, 73, 57, 89, 91, 121], However, there is no support to date for data-intensive computing on a heterogeneous CPU-GPU cluster.

To accelerate data-intensive computing on parallel heterogeneous environments, specifically, the hybrid CPU-GPU clusters, this work presents the MATE-CG (Map-reduce with an AlternaTE api for Cpu-Gpu clusters) system, which is a map-reduce-like framework to leverage the power of both CPUs and GPUs in a cluster. The MATE-CG system ports the *generalized reduction* model, which is similar to map-reduce. Considering the significant differences in the processing ability of the CPU cores and the GPU, resulting in the fact that different applications have very different relative performance on the GPUs, the MATE-CG runtime supports three schemes. The **CPU-only scheme** allows the use of one or more
cores on each node, the GPU-only scheme uses the many-core GPUs only, and finally, the CPU-n-GPU scheme can use the aggregate computing power of multi-core CPU and GPU(s) on each node. The performance of an application while executing in such an environment is impacted by a partitioning parameter, which controls the amount of data that is processed by the CPU and the GPU, respectively. The optimal choice of this parameter is highly application and dataset dependent. Our middleware includes a novel auto-tuning approach, where we exploit the iterative nature of many data-intensive applications to find the appropriate value of the parameter at runtime, with very low tuning overheads.

Overall, we have used our MATE-CG system to implement several data-intensive applications arising from various domains, including data mining, graph mining, and scientific computing. We show how the generalized reduction model, with support of programming clusters of multi-core CPUs and/or many-core GPUs, enables efficient and scalable implementation of data-intensive algorithms. We also highlight the importance of tuning some system parameters.

**Supporting Efficient Fault Tolerance for MPI Programs in the MATE System:** Fault tolerance has become a major challenge for large-scale HPC systems [32, 120, 87]. The most popular fault tolerance approach in HPC is checkpointing, where snapshots of the system are taken periodically. Checkpointing MPI programs and facilitating recovery after failure has been a topic of much investigation [15, 28, 52, 125, 74, 27, 47, 103, 67], consistent with the fact that MPI has been the most popular parallel programming model over the last two decades.

It is widely accepted that the existing MPI-based solutions will not be feasible in the exascale era [32], as with growing size of high-end systems and relatively lower I/O bandwidths, the time required to complete a check-point can exceed the Mean-Time To Failure
(MTTF). Moreover, with energy consumption being the dominant consideration for exascale systems, and with data movement (including data movement for checkpoints) being one of the major energy consumers, novel solutions are clearly needed.

Besides fault-tolerance, another important problem in high performance computing is *programmability*. With programmer productivity being a critical factor, and with increasing intra-node concurrency and diversity, there is a need for alternate programming models. Given the dual emphasis on programmability and fault-tolerance, it is natural to ask whether the design of a programming model can help facilitate lower-cost fault-tolerance and recovery.

The reduction object model has been shown to achieve fault tolerance with lower overheads than Hadoop for data mining and graph mining applications [26]. Therefore, in this work, we are trying to extend our ideas to support more efficient reduction object-based fault tolerance for MPI programs. Particularly, we show that designing a programming model (based on MATE systems) focused on an application’s underlying communication pattern can greatly simplify fault-tolerance support, resulting in at least an order of magnitude reduction in checkpointing overheads. The communication patterns we consider are similar to the notion of *dwarfs* in the Berkeley view on parallel processing \(^1\), where applications needing high performance are categorized into 13 dwarfs. While we believe that our approach can be applied to most or all of the dwarfs, we have so far focused on three patterns.

We show how simple APIs can support development of applications involving these patterns. Furthermore, we automatically save important state and can efficiently recover from failures, potentially using a different number of nodes than the original execution. The

\(^1\)Please see http://view.eecs.berkeley.edu/wiki/DwarfMine
main underlying idea in our approach is that when we focus on communication routines in a low-level library like MPI, the high-level information about the application is lost. Thus, implementing fault-tolerance through a higher layer, where the properties of the underlying computation are preserved, can be very beneficial. Moreover, by knowing the applications’ pattern, we can choose points for checkpointing and what data needs to be checkpointed, without involving the user or a compiler. Furthermore, a higher-level API can enable redistribution of work and data, potentially allowing recovery using a different number of nodes.

Our model-based approach has been implemented in the context of a precursor map-reduce-like system, MATE [80, 77, 78]. We have evaluated our approach using four applications, two each from the classes of dense grid computations and irregular reductions. Our results show that the checkpointing overheads of our model-based approach in absence of failures are much lower than those of a fault tolerant MPI library (MPICH2), especially with a high frequency of checkpointing. Furthermore, when a failure occurs, the failure recovery is very efficient and the only significant reason for any slowdown is because of the extra work re-distributed onto other nodes. Finally, our approach can tolerate both permanent node failures, where the failed node is excluded for the rest of the processing, and temporary node failures, where the failed node can be recovered and join the processing again.

**Dealing with GPU Failures in the MATE-CG System:** It is also increasingly important to provide fault tolerance in GPUs since GPU pipelines are becoming more popular and programmable, which has made GPUs more attractive to a wider audience [99]. However, pre-Fermi Nvidia GPUs do not provide fault tolerance. As GPUs have become more general purpose, more high performance computing applications that require fault tolerance are
being ported on GPUs. Thus, providing fault tolerance on GPUs is becoming increasingly important.

Therefore, in this work, we are trying to deal with GPU failures by incorporating fault tolerance support for GPU computations into the MATE-CG system. The MATE-CG runtime will be able to periodically checkpoint their work from GPUs into the host CPUs. In our implementations, we assume that only the GPU is unreliable (i.e., the CPU is reliable). If a fault has been detected on a GPU, then the runtime system can roll back to the last checkpoint on the host CPU-GPU node and continue executing in the recovery. Using four applications from stencil computations, data mining and scientific computing, our results on 8 CPU-GPU nodes show that the use of checkpointing for GPU results and evenly redistribution of the remaining work on the remaining resources in presence of GPU failures incur very low overheads in application runs.

1.4 Related Work

This section compares our overall work with related research efforts in data-intensive computing, computing using different parallel environments, especially heterogeneous systems, and fault tolerance in high-end systems.

Data-Intensive Computing: Recent years have seen a large number of efforts on map-reduce, its variants, and other abstractions. Efforts underway include projects that have used and evaluated map-reduce implementations for a variety of applications, as well as those that are trying to improve its performance and programmability in multi-core CPUs, GPUs, and heterogeneous systems.

Gillick et al. [64] evaluate Hadoop in a distributed environment with clusters of nodes using both a set of benchmark applications and machine learning algorithms. Chu et al. [44]
have used Google’s map-reduce for a variety of learning algorithms. GraphLab [93] was developed by Hellerstein’s group for machine learning algorithms and improves upon abstractions like map-reduce using a graph-based data model. Aluru et al. implemented a map-reduce-like framework to support searches and computations on tree structures [116].

CGL-MapReduce [54] uses streaming for all the communications, and thus improves the performance to some extent. Zaharia et al. [143] improved Hadoop response times by designing a new scheduling algorithm in a virtualized data center Seo et al. [122] proposed two optimization schemes, prefetching and pre-shuffling, to improve Hadoop’s overall performance in a shared map-reduce environment. MapReduce Online [45] extended Hadoop by supporting online aggregation and continuous queries for processing pipelined data. Verma et al. [135] and Kambatla et al. [84] reduced the job completion time by enforcing less-restrictive synchronization semantics between Map and Reduce phases for certain classes of applications in different ways. Each of these techniques results in substantial performance improvements and will likely benefit the graph mining implementation in PEGASUS also. However, the reported improvements are much smaller (less than a factor of 2) than the improvements reported from the Ex-MATE system for these applications.

Ranger et al. [111] have implemented a shared-memory map-reduce library named Phoenix in multi-core systems, and Yoo et al. [141] optimized Phoenix specifically for large-scale multi-core systems. Mars [70] was considered as the first attempt to harness GPU’s power for map-reduce applications. MapCG [72] presents a model based on MapReduce to support source code level portability between CPU and GPU. Catanzaro et al. [34] also built a framework around the map-reduce abstraction to support vector machine training as well as classification on GPUs.
None of these efforts have, however, considered a cluster of heterogeneous nodes as the target. The limited amount of work on map-reduce type processing on GPU clusters is from MITHRA [57] and IDAV [127] projects. MITHRA [57] was introduced by Farivar et al. as an architecture to integrate the Hadoop map-reduce with the power of GPGPUs in the heterogeneous environments, specific for Monte-Carlo simulations. IDAV [127] is a recent project to leverage the power of GPU clusters for large-scale computing by modifying the map-reduce paradigm. The key differences in our approach used in the MATE-CG system are: 1) support for exploiting both multi-core CPUs and GPUs simultaneously, and 2) processing of applications involving a very large reduction object.

Other programming abstractions for data-intensive computing have also been developed, either on top of map-reduce, or as an alternative to map-reduce. Disco [5] is an open source map-reduce runtime which is similar to Google’s and Hadoop’s architectures, but does not support a distributed file system. Microsoft has built Dryad [75], which is more flexible than map-reduce, since it allows execution of computations that can be expressed as DAGs. Yahoo has proposed Pig Latin [105] and Map-Reduce-Merge [43], both of which are extensions to Hadoop, aiming to support more high-level primitives and improve the performance. Facebook uses Hive [131] as the warehousing solution to support data summarization and ad-hoc querying on top of Hadoop. Google itself developed Sawzall [107] on top of map-reduce to provide a higher-level API for data processing. And for graphs, Google first proposed several graph operations using map-reduce but concluded that map-reduce was not quite suited to the graph operations due to the high overheads of map-reduce iterations and communication [46]. Pregel [96] was then developed recently as a new programming model and an optimized infrastructure for mining relationships from graphs.
**Programming Heterogeneous CPU-GPU Clusters:** With the emergence of this type of systems, there has been a lot of interest in programming models for heterogeneous systems. Merge framework [92] is a general purpose programming model for heterogeneous multi-core systems. StarPU [21] provides a unified execution model to parallelize numerical kernels on a multi-core CPU equipped with one GPU. Ravi et al. [115, 115] developed a compiler and runtime framework that can map a class of applications also using generalized reductions to a system with only one multi-core CPU and one GPU. Shirahata et al. proposed a hybrid scheduling prototype for running Hadoop tasks on both CPU cores and GPU devices [123]. But they only considered distributing part of the computations, i.e., the *Map* tasks across CPU cores and GPUs, while the *Reduce* tasks are always executed on CPUs. Our system, however, can distribute the overall computations onto both CPUs and GPUs based on the generalized reduction model.

EXOCHI [139] creates a unified view by providing dynamic management of both CPU and GPU resources. Harmony [51] involves an execution model and a runtime framework to schedule computations either on a CPU or on a GPU, based on the estimated kernel performance. Teodoro et al. [130] describe a runtime framework that selects either of a CPU core or the GPU for a particular task. In comparison to [51] and [130], we focus on data parallelism and exploit both the multi-core CPU and GPU simultaneously in the MATE-CG system. Qilin system [94] has been developed with an adaptable scheme for mapping the computation between CPU and GPU simultaneously. The adaptivity of the system is based on extensive offline training and does not provide mechanisms to minimize GPU device overheads.
CUBA [62] is an attempt to avoid data management between CPU and GPU by letting the GPU access data present in CPU memory directly and allows overlapping data communication and computation. NVIDIA just released CUDA 4.0 toolkit [10] integrated with the thrust library [14] to provide a unified memory space and also use unified virtual addressing to support faster multi-GPU programming.

CUDA [9] has been the most popular language for programming a single GPU. There is a very large volume of efforts on programming a single GPU using CUDA [129]. CUDASA project [126] exploits parallelism in multi-GPU systems and GPU-cluster environments by presenting an extension to the CUDA programming language. Lee et al. are trying to extend OpenMP to GPUs by providing a compiler framework for automatic translation and code optimization [91, 90], though targeting a single GPU. There have been other efforts at translating OpenMP to accelerators as well [37]. OpenCL [11] is a relatively low-level API for heterogeneous computing that runs on the CUDA architecture. Grewe et al. proposed a purely static approach to partition and map OpenCL programs on heterogeneous CPU-GPU systems based on predictive modeling and program features [66]. Our systems target a restricted class of applications, but offer a much higher-level API. We have currently targeted a reduction-style API, though we could consider an alternative interface like OpenMP with user-defined reductions [53].

Also, there is a large volume of work on auto-tuning [140, 142, 48]. We have focused on a novel application of auto-tuning, using a novel performance model in the MATE-CG system.
Checkpointing and Fault Tolerance: Fault-tolerance support for MPI applications has been investigated by several research groups. Coordinated checkpointing and uncoordinated checkpointing coupled with message logging have been used most often for supporting fault-tolerance [125, 52, 15, 47, 28, 74, 103, 67]. In addition, researchers have also developed fault-tolerant algorithms for specific problems [109, 23, 20, 40, 49], and others have proposed approaches based on replication [22, 13, 147].

Currently, coordinated checkpointing is widely used in almost all the popular MPI stacks [28, 12, 7, 8, 1]. In this work, we have compared our work against the coordinated checkpointing protocol used in MPICH2. OpenMPI [12] and MVAPICH2 [8] also support a similar coordinated checkpointing protocol and we can expect that their checkpointing times will be similar to those from MPICH2. Compared with coordinated checkpointing, uncoordinated checkpointing with message logging has similar or more checkpointing time but less absolute recovery time. Process migration, which is supported in MVAPICH2, is less expensive than traditional checkpointing, but it requires fault prediction and redundant nodes. In comparison, our approach does not rely on these assumptions.

OpenMPI [12] plans to support a variety of checkpointing protocols in addition to coordinated checkpointing in the future. MPICH-V [28], built on top of older versions of MPICH2, supported a different protocol based on uncoordinated checkpointing with message logging. A recent effort [67], implemented an uncoordinated checkpointing protocol for send-deterministic MPI applications [33] on top of MPICH2. Compared to coordinated checkpointing, they show that this method reduces the checkpoint time and the average number of processes to be rolled back also reduces. The structured grid computations we studied in this work are send-deterministic, but the irregular reductions are not. While the uncoordinated checkpointing with message logging has not been made available with the
latest MPICH2 version, our approach only needs to restart the same number of MPI processes as the number of failed nodes, without the need of rolling back other processes. The Scalable Checkpoint/Restart (SCR) library [13] tolerates a single node failure following a memory-distributed checkpointing scheme, and like our approach, could exploit both a reliable disk storage and the distributed memory.

Zheng et al. [146] have proposed an in-memory double checkpointing protocol for fault-tolerance. Without relying on any reliable storage, the checkpoint data, which is encapsulated by the programmer, is stored at two different processors. Also, the checkpoints are taken at a time when the application memory footprint is small. Another approach proposed by Marques et al. [97] dynamically partitions objects of the program into subheaps in memory. By specifying how the checkpoint mechanism treats objects in different subheaps as always_save, never_save and once_save, they reduce the checkpoint size at runtime. Our work has some similarities, but the key difference is our ability to use the reduction property to checkpoint the necessary computational states into the distributed memory and/or the reliable storage and also re-distribute the remaining work across other nodes.

Our approach can be viewed as being somewhat similar to the work from Cornell on application-level checkpointing for parallel programs [60, 61, 59]. Their approach investigates the use of compiler technology to instrument codes to enable self-checkpointing and self-restarting. Our approach does not require any compiler analysis, and can enable restart with a different number of nodes.

AMPI [1], which is based on Charm++ [3], also supports the use of both disk and memory to checkpoint MPI processes [35, 146, 144, 145]. AMPI also provides additional fault tolerance support via message logging [101, 102] and proactive fault avoidance [117, 36]. One advantage of this approach is that it can support restart using a different number of
processors. However, because high-level information about the application and the underlying communication patterns are not taken into account, we expect checkpointing times to be similar to the existing MPI approaches.

Partitioned Global Address Space (PGAS) models have recently started becoming popular for parallel programming, with Global Arrays being one of the prominent examples. Similar to MPI, various approaches have been proposed and investigated for Global Arrays, including algorithm-based fault tolerance [134, 18], redundant communication approaches [17], hybrid hardware-software solutions [136], and data-driven techniques for task-based work stealing computations [95]. Similar to AMPI approaches, this work can handle restart with a different number of nodes. Finally, there have been some recent efforts towards providing high-performance fault tolerance interface for GPU and GPU clusters [128, 99, 98, 24]. Their focus here is on general checkpointing of GPU programs. Our work instead is derived from a high-level programming model and aims to deal with failures on CPU or GPU failures in a cluster of nodes via checkpointing only the reduction objects.

1.5 Outline

The rest of the dissertation is organized as follows. In Chapter 2, we introduce Hadoop and FREERIDE and compare the performance of these two systems. In Chapter 3, we give an overview of the MATE system designed for shared-memory environments. The extended MATE system with support of large-sized reduction objects in distributed environments are introduced in Chapter 4. In Chapter 5, we present the design and implementation of the MATE-CG system with the support of programming the CPU-GPU clusters.
and a novel auto-tuning framework. Then we discuss our latest work about providing efficient fault tolerance in MATE systems in Chapter 6 and Chapter 7. Finally, we will discuss possible work for the future in Chapter 8 and conclude in Chapter 9.
Chapter 2: Comparing Map-Reduce and FREERIDE for Date-Intensive Applications

Map-reduce has been a topic of much interest in the recent years. While it is well accepted that the map-reduce APIs enable significantly easier programming, the performance aspects of the use of map-reduce are less well understood and studied, especially about the comparison with other paradigms. To better understand the performance issues of map-reduce, our first work is a comparative study that focuses on comparing Hadoop (a open-source version of the map-reduce) with a system that was developed earlier at Ohio State, FREERIDE (FRamework for Rapid Implementation of Dataminning Engines). The API and the functionality offered by FREERIDE has many similarities with the map-reduce API. However, there are some differences in the API. Moreover, while FREERIDE was motivated by data mining computations, map-reduce was motivated by searching, sorting, and related applications in a data-center. We compare the programming APIs and performance of the Hadoop implementation of map-reduce with FREERIDE. For our study, we have taken three data mining algorithms, which are k-means clustering, apriori association mining, and k-nearest neighbor search. We have also included a simple data scanning application, word-count.

The main observations from our results are as follows. For the three data mining applications we have considered, FREERIDE outperformed Hadoop by a factor of 5 or more.
For word-count, Hadoop is better by a factor of up to 2. With increasing dataset sizes, the relative performance of Hadoop becomes better. Overall, it seems that Hadoop has significant overheads related to initialization, I/O, sorting of $(key, value)$ pairs, and inefficiency of the resource utilization. Thus, despite an easy to program API, Hadoop’s map-reduce does not appear very suitable for data mining computations on modest-sized datasets.

2.1 Hadoop and FREERIDE

2.1.1 Hadoop Implementation

Map-reduce is a general idea for which there are different implementations. Hadoop is an open-source project attempting to reproduce Google’s implementation, and is hosted as a sub-project of the Apache software foundation’s Lucene search engine library [6].

Google’s implementation is based on a distributed file system called Google File System (GFS) [63]. Hadoop also uses a similar distributed file system, named Hadoop Distributed File System (HDFS)\(^2\). Hadoop is implemented in Java, and HDFS is a pure-Java file system. It stores input datasets as well as the intermediate results and outputs across multiple machines. HDFS replicates the data on multiple nodes so that a failure of nodes containing a part of the data will likely not impact the entire computation process. Also, data locality is exploited in the Hadoop implementation.

One limitation of HDFS, compared with GFS, is that it cannot be directly mounted on an existing operating system. Data needs to be copied in and out of HDFS before and after executing a job. This can be time-consuming for large datasets. A Filesystem in Userspace (FUSE)\(^3\) has been developed to address this problem, at least for Linux and Unix systems. In the Hadoop implementation, the intermediate pairs generated by the Maps

\(^2\)http://hadoop.apache.org/core/docs/current/hdfs_design.html

\(^3\)http://wiki.apache.org/hadoop/MountableHDFS
are first written to the local disks and then retrieved by the appropriate *Reduces*. In this intermediate stage, the results are *sorted* and *grouped* based on the key and the resulting groups are *shuffled* to the corresponding *reduces* by the scheduler. This strategy makes Hadoop more robust, but brings extra steps of sorting, grouping, and shuffling. This in turn introduces a communication overhead, which can be a limiting factor in performance of certain applications.

Fault-tolerance is also an important aspect of the Hadoop implementation. There are three components that support fault tolerance. They are the (i) *Name Node*, (ii) *Job Tracker*, and, (iii) *Task Tracker*. The *Name Node* functions as the server for HDFS and a *Secondary Name Node* is created along with it to avoid a single point of failure of the *Name Node*. Once HDFS is started and a client submits a job, the *Job Tracker* schedules the entire workload based on a number of available *Task Trackers* in the cluster nodes, to execute the *map/reduce* tasks. If a *Task Tracker* is aborted for some reason, this part of the job can be rescheduled. However, the failure of *Job Tracker* will cause the entire job to be lost, since it is the scheduler for all tasks.

### 2.1.2 FREERIDE API and Runtime techniques

Agrawal’s group at Ohio State has previously developed a middleware system for cluster-based data-intensive processing, which shared many similarities with the map-reduce framework. However, there are some subtle but important differences in the API offered by these two systems. One, FREERIDE allows developers to explicitly declare a reduction object, and perform updates to its elements directly, while in Hadoop/map-reduce, reduction object is implicit and not exposed to the application programmer. While explicit reduction object can help with performance gains, there is also a possibility of incorrect usage by application
developer. Other important distinction is that, in Hadoop/map-reduce, all the data elements are processed in the map step and the intermediate results are then combined in the reduce step, where as, in FREERIDE, both map and reduce steps are combined into a single step where each data element is processed and reduced before next data element is processed. This choice of design avoids the overhead due to sorting, grouping, and shuffling, which can be significant costs in a map-reduce implementation.

**FREERIDE Interface**

The FREERIDE system was motivated by the difficulties in implementing and performance tuning parallel versions of data mining algorithms. FREERIDE is based upon the observation that parallel versions of several well-known data mining techniques share a relatively similar structure, which is that of a generalized reduction. Though the map-reduce paradigm is built on a similar observation [50], it should be noted that the first work on FREERIDE was published in 2001 [81], prior to the map-reduce paper by Dean and Ghemawat in 2004.

The interface exploits the similarity among parallel versions of several data mining algorithms. The following functions need to be written by the application developer using the middleware.

**Reduction:** The data instances owned by a processor and belonging to the subset specified are read. A reduction function specifies how, after processing one data instance, a reduction object (initially declared by the programmer), is updated. The result of this processing must be independent of the order in which data instances are processed on each processor. The order in which data instances are read from the disks is determined by the runtime system.

**ProcessNextIteration:** This is an optional function that can be implemented by an application programmer. Typically, this function should include a program logic specific to an
application that would control the number of iterations for which the application should run. This adds more flexibility to the programmer.

**Finalize:** After final results from multiple nodes are combined into a single reduction object, the application programmer can read and perform a final manipulation on the reduction object to summarize the results specific to an application.

Throughout the execution of the application, the reduction object is maintained in main memory. After every iteration of processing all data instances, the results from multiple threads in a single node are combined locally depending on the shared memory technique chosen by the application developer. After local combination, the results produced by all nodes in a cluster are combined again to form the final result, which is the global combination phase. The global combination phase can be achieved by a simple all-to-one reduce algorithm. If the size of the reduction object is large, both local and global combination phases perform a parallel merge to speed up the process. The local combination and the communication involved in global combination phase are handled internally by the middleware and transparent to the application programmer.

**Shared Memory Parallelization**

The processing structure of FREERIDE enables us to develop general but efficient shared memory parallelization schemes. Consider again the loop in Figure 1.1(b). The main correctness challenge in parallelizing a loop like this on a shared memory machine arises because of possible race conditions (or *access conflicts*) when multiple processors update the same element of the reduction object. Two obvious approaches for avoiding race conditions are: *full replication* and *full locking*. In the full replication approach, each processor can update its own copy of the reduction object and these copies are then merged.
together later. In the full locking approach, one lock or latch is associated with each aggregated value. However, in our experience with mining algorithms, the memory hierarchy impact of using locking when the reduction object is large was significant [113].

To overcome these overheads, we have designed several other optimized schemes. One such technique is *optimized full locking*. This approach overcomes the large number of cache misses associated with full locking scheme by allocating a reduction element and the corresponding lock in consecutive memory locations. By appropriate alignment and padding, it can be ensured that the element and the lock are in the same cache block. Each update operation now results in at most one cold or capacity cache miss. The possibility of false sharing is also reduced.

### 2.2 Comparing FREERIDE and Hadoop: Case Studies

In the previous section, we described the Hadoop and FREERIDE systems. This section focuses on the API differences between these two systems. We study k-means clustering and word-count to compare the two systems’ APIs.

#### 2.2.1 K-means Clustering

The first application we describe is the k-means clustering, which is one of the commonly used data mining algorithms [76]. This was one of the earliest algorithms implemented in FREERIDE. The clustering problem is as follows. We consider transactions or data instances as representing points in a high-dimensional space. Proximity within this space is used as the criterion for classifying the points into clusters. Four steps in the sequential version of k-means clustering algorithm are as follows: 1) start with \(k\) given centers for clusters; 2) scan the data instances, for each data instance (point), find the center closest to it and assign this point to the corresponding cluster, 3) determine the \(k\) centroids
from the points assigned to the corresponding centers, and 4) repeat this process until the assignment of points to clusters does not change.

Figure 2.11 gives the pseudo-code of k-means using FREERIDE API. In FREERIDE, first, the programmer is responsible for creating and initializing the reduction object. The $k$ cluster centroids are initialized as well. For k-means, the reduction object is an array with $k \times (ndim + 2)$ elements, where $ndim$ is the number of dimensions for the coordinate space. Each cluster center occupies $(ndim + 2)$ consecutive elements in the reduction object array. Among the $(ndim + 2)$ elements for each cluster, the first $ndim$ elements are used to store the sum of data points assigned to each cluster. The other two elements represent the number of data points belonging to each cluster and the accumulated overall distances.

In each iteration, the reduction operation reads a block of data points, and for each point, it will compute the distance between the point and the $k$ cluster centers and find the closest. Then, we can obtain the object ID in the reduction object for the closest cluster and update the particular elements correspondingly in the reduction object. A global reduction phase is also required to merge the results from all nodes after the reduction operation is applied on all data points.

Figure 2.12 gives the pseudo-code of k-means implemented using Hadoop API. The input datasets are partitioned into a number of splits first. Then, the Map tasks will be launched. In each iteration, the Map task reads the existing cluster centroids from files and then processes a split of the input data points using the map function. For each point, map finds the closest cluster center, assigns the point to it, and then emits the cluster center id as the key and the point as the value. The Reduce task collects all the points with the same cluster center id which is the key, and computes their mean as the new centroid of
the cluster. Finally, the output of Reduce tasks will be written into files and taken as new cluster centers by next iteration.

The combine function is optional in the Hadoop API, but to improve the performance, we implemented this function. It is similar to the reduce function. It just accumulates the coordinates of data points belonging to the same cluster without computing the mean. The $k$ cluster centers are initialized and written into files and then copied to the HDFS before we run the program. Since k-means is iterative, a sequence of iterations is represented by the same number of map/reduce jobs. But due to the constraints of the Hadoop implementation, the configuration and initialization of map/reduce tasks for each job needs to occur every time a job is submitted. Hadoop also provides the run function, where the programmer can control the number of iterations to run k-means.

We can compare the codes shown and understand the similarities and differences between the two systems. In FREERIDE, the reduction object is used to store the sum of the data points belonging to the same cluster center. It has to be created and initialized first. Then, for each point, the reduction object is updated in the reduction operation, after we know which cluster it belongs to. When all data points are processed, the finalize operation will compute the new center vector for each cluster. In Hadoop, map computes the closest cluster center id for each point and emits it with the point value. The reduce function will gather the data points with the same cluster center id. So, to summarize, FREERIDE adds the coordinates of points belonging to each cluster point-by-point, and computes the mean after all points are read. Whereas, Hadoop computes the cluster center id point-by-point, without performing the sum. It performs the accumulation and computes the mean in the Reduce Phase. Besides, the intermediate pairs produced by the Map tasks are written into files and read by the Reduce tasks through HTTP requests.
One observation we can make is as follows. In FREERIDE, the \textit{reduction object}, which is allocated in memory, is used to communicate between the \textit{reduction} and the \textit{finalize} functions. In comparison, HDFS is used to connect the \textit{Map} and the \textit{Reduce} functions in Hadoop. So, Hadoop requires extra sorting, grouping, and shuffling for this purpose.

\subsection*{2.2.2 Word Count}

The second application we use is the word-count program, which is a simple application, often used to describe the map-reduce API \textsuperscript{4}.

Figure 2.13 shows the pseudo-code for this application using FREERIDE API. This application does not naturally fit into the FREERIDE framework. This is because we do not know the exact size of the \textit{reduction object}, as it depends on the number of distinct words in the dataset. So, the \textit{reduction object} for this application is allocated dynamically to record the frequency of each unique word and its reference \textit{id}. We used a \textit{word map} to store the string value of each unique word and its \textit{id}. The \textit{reduction} operation takes a block of words, and for each word, searches for it first in the \textit{word map}. If we can find the word, then retrieve its \textit{id} from the \textit{word map}, and update its frequency. Otherwise, this is a new word, and we need to insert one entry before updating the reduction object. A \textit{global reduction} phase is also needed to merge the results from all nodes.

Figure 2.14 gives the pseudo-code for this application using Hadoop API, which is smaller in code size. The \textit{map} function takes a line of words and for each word, emits the word and its associated frequency which is 1 in all cases. The \textit{reduce} function gathers all the counts for each unique word, and compute the sums before the final output.
2.3 Experimental Evaluation

We now present a detailed experimental study evaluating and analyzing the performance of Hadoop and FREERIDE.

Since our main focus is on data mining applications, we choose three popular data mining algorithms. They are, k-means clustering, apriori associating mining, and k-nearest neighbor search. We had described k-means clustering earlier in Section 2.2. Apriori is a well accepted algorithm for association mining, which also forms the basis for many newer algorithms [16]. Association rule mining is the process of analyzing a set of transactions to extract association rules and is a commonly used and well-studied data mining problem. Given a set of transactions, each of them being a set of items, the problem involves finding subsets of items that appear frequently in these transactions.

\[ \text{http://wiki.apache.org/hadoop/WordCount} \]

Figure 2.1: Varying Input Split Size
k-nearest neighbor classifier is based on learning by analogy [68]. The training samples are described by an n-dimensional numeric space. Given an unknown sample, the k-nearest neighbor classifier searches the pattern space for $k$ training samples that are closest, using the Euclidean distance, to the unknown sample. Among these three algorithms, k-means and apriori both require multiple passes over data, whereas KNN is a single pass algorithm.

To complete our experimental study, we also included a fourth application, which is quite different from the other three applications. Word-count is a simple search application, and can be viewed as a representative of the class of applications that motivated the development of the map-reduce framework. In comparison, as we had stated earlier, FREERIDE was motivated by data mining algorithms, including the three applications we are using in this study.

Our experiments were conducted on a cluster of multi-core machines. Each node in the cluster is an Intel Xeon CPU E5345, comprising two quad-core CPUs. Each core has
a clock frequency of 2.33GHz and each node has a 6 GB main memory. The nodes in the cluster are connected by Infiniband.

2.3.1 Tuning Parameters in Hadoop

The performance of an application on Hadoop can vary substantially based on the choice of various parameters. We carefully tuned these parameters, and the comparisons with FREERIDE reported are based on the best performance we could achieve with Hadoop. Using k-means clustering as an example, this section describes the performance tuning we performed. Such tuning was performed for each of our applications, and the results were quite similar, but not identical. For this study, k-means was executed with 2 datasets, which are of 1.6 GB and 6.4GB, respectively. The number of dimensions is 3 and the number of clusters ($k$) is 1000.
Figure 2.1 shows the average execution time per iteration of k-means when we vary the split-size. Split-size refers to the size of the portion of a dataset that each map task deals with. Results from both the datasets show that the split-size plays an important role in the performance. The size of an input split determines the number of map tasks. With a large split-size or a smaller number of map tasks, the initialization cost can be minimized, but the parallelism may be limited, and all cores in a multi-core node may not be utilized. At the same time, a very small split-size and a large number of map tasks also introduce a large overhead of spawning threads and merging their results. Thus, as we can see from Figure 2.1, there is an optimal split size for each dataset. The results also show that when the dataset size is increased, the split size should also be increased correspondingly to avoid unnecessary startup overheads.

Figure 2.4: Apriori: Comparison between FREERIDE and Hadoop
The second factor we consider is the number of concurrent maps, which refers to the maximum number of map tasks that can be executed concurrently per node. The results are shown in Figure 2.2. For both datasets, the best performance is achieved with 8 concurrent maps per node, which is also the number of cores available on each core. Clearly, we cannot exploit adequate parallelism with a smaller number of concurrent maps, whereas more tasks than the number of cores only increases scheduling overhead.

Another factor we tuned was the number of reduce tasks. For k-means, our tuning showed that the best performance is achieved with 4 tasks per node. We do not show the details here.
2.3.2 Performance Comparison for Four Applications

We now focus on comparing the two systems for the four applications we had described earlier. We executed these applications with FREERIDE and Hadoop, on 4, 8, and 16 nodes of our cluster. Thus, besides comparing the performance of the two systems, we also wanted to study the scalability of the applications with these frameworks.

For FREERIDE, we create 8 threads on each node. The shared memory technique we used for k-means, kNN, and apriori was full-replication, since it gave the best performance. For word-count, we used optimized full locking. This was because we could not use replication on word-count, (due to an implementation limitation), and among all locking schemes, optimized full locking seems the best choice. The datasets were disk-resident for FREERIDE, and were read from disks for each application. For Hadoop, we tuned the parameters as before, and in most cases, we used 8 concurrent maps per node, 4 reduce
tasks, and a 64 MB input split-size. We used 6.4 GB datasets for k-means, kNN, and word-count, and a 900 MB dataset for apriori. Later in this section, we compare performance of the two systems while varying the size of the dataset.

For k-means, the dataset is 3-dimensional, and the number of clusters, $k$, was set to be 1000. With apriori, the support and confidence levels were 3% and 9%, respectively, and the application ran for 2 passes. In kNN, the value of $k$ used in our experiments was 1000.

Figures 2.3 through 2.6 show the results for these 4 applications. For the three data mining applications, FREEERIDE outperforms Hadoop significantly. The differences in performances ranged from a factor of 2.5 to about 20. We can also see that the scalability on increasing number of nodes is almost linear with FREEERIDE. With Hadoop, speedups in going from 4 nodes to 16 is about 2.2 for k-means and word-count, and only about 1.1 for apriori. Overall, we can see that Hadoop has high overhead for data mining applications.
with a significant amount of computations. Reasons for such overheads are analyzed later in the section.

The results are very different for word-count. On 4 nodes, Hadoop is faster by a factor of about 2. With increasing number of nodes, FREERIDE is more competitive, though still slower by about 10% on 16 nodes. We see two possible reasons for this performance limitation. First, for such an algorithm, FREERIDE’s shared memory techniques are not most appropriate. Second, the specialized file system in Hadoop may be improving the performance for this application, where there is very little computation.

The experiments we have reported so far considered only 1 dataset size for each application. To further understand how the relative performance between the two systems is impacted by changing datasets sizes, we further experimented with k-means and word-count. We executed k-means and word-count both systems with 8 nodes, while considering
a number of different dataset sizes. The results are shown in Figure 2.7 and Figure 2.8. We can see that for both applications, the execution with FREERIDE is proportional to the size of the dataset. However, for Hadoop, the processing time increases sub-linearly. For example, when we increase the dataset size 4 times, i.e., from 800 MB to 3.2 GB, k-means execution time with Hadoop only increases by 50%. This shows that initial overheads are quite high for Hadoop. For k-means, even with 12.8 GB dataset, FREERIDE is about 9 times faster than Hadoop. Consistent with what we saw earlier, results are quite different for word-count. With a 12.8 GB dataset, Hadoop is faster by a factor of about 2. With the two smallest datasets, FREERIDE still outperforms Hadoop. Again, this confirms that initial overheads on Hadoop are quite significant.
2.3.3 Detailed Analysis of k-means

To further understand the reasons for slow-down in Hadoop, we did a detailed analysis with k-means. We can see that there are four primary components in the execution time of data-intensive applications. The first is the initialization cost. Hadoop can be slower because of the overheads associated with dynamic scheduling, data partitioning, and task spawning. The second is the I/O time, which should be mainly dependent on the size of dataset. Hadoop can be slower because of the HDFS, and also due to the fault tolerance techniques this system uses. The third is the time associated with sorting, grouping, and shuffling between the map phase and reduce phase. Because of the differences in API, this cost only exists in Hadoop. The fourth is the computation time to process the input data. Hadoop can be relatively slower because it uses Java, whereas FREERIDE uses C++.
However, it is not easy to determine how significant each of these factors is. We did two sets of experiments on k-means to try and isolate the impact of these four factors. In the first experiment, we choose a dataset of 6.4GB containing 3 dimensional coordinates. We first varied the value of $k$ to be 50, 200, and 1000. Since the dataset size is constant, the I/O time does not change. The computing time in k-means is linearly proportional to $k$, since computing the distances between every point in the dataset and the $k$ cluster centers is the dominant computation in every iteration. Figure 2.9 shows that the application execution time for k-means in FREERIDE is linear in terms of the value of $k$. This shows that the computing time is the dominant cost for k-means in FREERIDE. The total execution time for Hadoop also increases with increasing $k$, but at a slower rate. Also, for all values of $k$, Hadoop is much slower. This clearly shows that the one or more of the three factors, i.e., the initialization cost, I/O time, and sorting, are an important limiting factor in Hadoop’s performance.

The second experiment we conducted was as follows. We fixed the value of $k$ and varied the dimensionality of the data points, considering 3, 6, 48, 96, and 192 dimensions. When the dataset size is fixed and the dimensionality is increased, the number of data points to be processed decreases. This implies that fewer $(key, value)$ pairs are generated. This in turn implies that the sorting overheads should be lower. At the same time, the total I/O volume does not change. Furthermore, the impact of varying dimensionality on the computation is as follows. The distance computation time does not change, as the product of the dimensions and number of points does not change. However, for each point that is processed, k-means involves finding the closest center. This part should decrease as we process fewer points. However, with increasing dimensionality, this component becomes less significant.
Figure 2.10 shows that in FREERIDE, the execution time first drops and then remains stable when we increase the dimensionality. This seems consistent with the observations earlier, that processing time for k-means is proportional to the amount of computations. While in Hadoop, we can observe that the time does not change too much when we vary the dimensionality. Since the sorting time should be reduced when we increase the dimensionality, we can conclude that the sorting time is also not very dominant in Hadoop.

Coupled with earlier observations that with increasing dataset sizes, the relative difference between Hadoop and FREERIDE is lowered, we believe that the most significant difference between the two systems is the high initialization costs in Hadoop, and the limited I/O bandwidth of HDFS.

2.4 Summary

In the last few years, the growing importance of data-intensive computing has been closely coupled with the emergence and popularity of the map-reduce paradigm. While there is a huge focus on easy programmability aspect of map-reduce model, the performance aspects of map-reduce model have not yet been well understood. This work focuses on comparing the Hadoop, which is an implementation of map-reduce paradigm with a system that was developed earlier at Ohio State, FREERIDE (FRamework for Rapid Implementation of Datamining Engines). For our study, we have taken three data mining algorithms, which are k-means clustering, apriori association mining, and k-nearest neighbor search, and a simple data scanning application, word-count.

Some important observations from our results are as follows. FREERIDE outperforms hadoop for all the three datamining applications we have considered, by a factor of 5 or more. Hadoop outperforms FREERIDE for the word-count application by a factor of up
void Kmeans::reduction(void *block) {
    for each point ∈ block{
        for (i = 0; i < k; i++) {
            dis = distance(point, i);
            if (dis < min) {
                min = dis;
                min_index = i;
            }
        }
        objectID = clusterID[min_index];
        for (j = 0; j < ndim; j++)
            reductionobject− > Accumulate(objectID, j, point[j]);
        reductionobject− > Accumulate(objectID, ndim, 1);
        reductionobject− > Accumulate(objectID, ndim + 1, dis);
    }
}

int Kmeans::finalize() {
    for (i = 0; i < k; i++) {
        objectID = clusterID[i];
        count = (*reductionobject)(objectID, ndim);
        for (j = 0; j < ndim; j++)
            clusters[i][j] = (*reductionobject)(objectID, j) / (count + 1);
        totaldistance+ = (*reductionobject)(objectID, ndim + 1);
    }
}

Figure 2.11: Pseudo-code for k-means using FREERIDE API

Our results show that, Hadoop can be tuned to achieve peak performance based on the factors like split size, maximum number of concurrent maps per node and number of reduces. Also, the relative performance of Hadoop becomes better with the increasing size of dataset. Overall, Hadoop has significant overheads due to initialization, I/O (during storage of intermediate results and shuffling), and sorting of (key, value) pairs. Thus, despite
**HADOOP (k-means)**

```java
public void map(LongWritable key, Text point)
{
    minDistance = Double.MAXDISTANCE;
    for (i = 0; i < k; i++)
    {
        if (distance(point, clusters[i]) < minDistance)
        {
            minDistance = distance(point, clusters[i]);
            currentCluster = i;
        }
    }
    EmitIntermediate(currentCluster, point);
}

public void reduce(IntWritable key, Iterator<PointWritable> points)
{
    num = 0;
    while (points.hasNext())
    {
        PointWritable currentPoint = points.next();
        num += currentPoint.getNum();
        for (i = 0; i < dim; i++)
        {
            sum[i] += currentPoint.point[i];
        }
    }
    for (i = 0; i < dimension; i++)
    {
        mean[i] = sum[i]/num;
        Emit(key, mean);
    }
}
```

Figure 2.12: Pseudo-code for k-means using Hadoop API

an easy to program API support, Hadoop’s map-reduce does not appear very suitable for data mining computations on modest-sized datasets.
void Wordcount :: reduction(void * block) {
    for each word ∈ block{
        map < string, int > :: iterator iter = wordmap.begin();
        iter = wordmap.find(word);
        if (iter != wordmap.end()){
            wordID = iter -> second;
            reductionobject-> Accumulate(wordID, 1, 1);
        }
        else {
            newID = reductionobject-> alloc(2);
            wordmap.insert(pair < string, int >(word, newID));
            reductionobject-> Accumulate(newID, 0, newID);
            reductionobject-> Accumulate(newID, 1, 1);
        }
    }
}

int Wordcount :: finalize() {
    size = wordmap.size();
    for (i = 0; i < size; i + +){
        id = (*reductionobject)(i, 0);
        count = (*reductionobject)(i, 1);
        output(get_word(id), count);
    }
    return 0;
}
HADOOP (wordcount)

```java
public void map(LongWritable key, Text words) {
    for each word ∈ words
        EmitIntermediate(word, "1");
}

public void reduce(Text key, Iterator<IntWritable> values) {
    sum = 0;
    while (values.hasNext()) {
        sum += values.next.get();
    }
    Emit(key, new IntWritable(sum));
}
```

Figure 2.14: Pseudo-code for wordcount using Hadoop API
Chapter 3: MATE: A MapReduce System with an Alternate API for Multi-core Environments

Although the comparative study we conducted before demonstrated that FREERIDE outperformed Hadoop by a factor of 5-10 for a set of data mining applications, this could be attributed to various reasons, like the difference in processing using Java (as in Hadoop) and C++ (as in FREERIDE), the inefficient resource utilization in Hadoop, or the overheads of the Hadoop distributed file system. To allow further comparison of the two APIs with eliminating other factors, in this work, we present a system MATE (Map-reduce with an AlternaTE API), that provides a high-level, but distinct API. Particularly, our API includes a programmer-managed reduction object, which results in lower memory requirements at runtime for many data-intensive applications. MATE implements this API on top of the Phoenix system, a multi-core map-reduce implementation from Stanford.

We evaluate our system using three data mining applications and compare its performance to that of both Phoenix and Hadoop. Our results show that for all the three applications, MATE outperforms Phoenix and Hadoop. Despite achieving good scalability, MATE also maintains the easy-to-use API of map-reduce. Overall, we argue that, our approach, which is based on the generalized reduction structure, provides an alternate high-level API, leading to more efficient and scalable implementations.
3.1 A Case Study

Since our API is based on the generalized reduction model, as an example, we now first use the apriori association mining algorithm, to show the similarities and differences between map-reduce and the generalized reduction APIs. Apriori is a well known algorithm for association mining, which also forms the basis for many newer algorithms [16]. Association rule mining is a process of analyzing a set of transactions to extract association rules and is a commonly-used and well-studied data mining problem. Given a set of transactions, each of them being a set of items, the problem involves finding subsets of items that appear frequently in these transactions. Formally, let $L_i$ represent the set consisting of frequent itemsets of length $i$ and $C_i$ denote the set of candidate itemsets of length $i$. In iteration $i$, $C_i$ is generated at the beginning of this stage and then used to compute $L_i$. After that, $L_i$ is used as $C_{i+1}$ for iteration $(i + 1)$. This process will iterate until the candidate itemsets become empty.

Figure 3.8 shows the pseudo-code of apriori using the generalized reduction API. Using this API, in iteration $i$, first, the programmer is responsible for creating and initializing the reduction object. The reduction object is allocated to store the object ids for each frequent-$i$ itemset candidate with associated counts of support. Then, the reduction operation takes a block of input transactions, and for each transaction, it will scan through the frequent-$i$ itemset candidates, which are the frequent-$(i - 1)$ itemsets generated from last iteration. During the scan, if some frequent-$i$ itemset candidate is found in the transaction, the object id for this itemset candidate is retrieved and its corresponding count of support is incremented by one. After the reduction operation is applied on all the transactions, the update_frequent_candidates operation is invoked to remove the itemset candidates whose count of support is below the support level that is defined by the programmer.
Figure 3.9 gives the pseudo-code of apriori using the map-reduce API. The map function is quite similar to the reduction given in Figure 3.8 and the only difference is that the emit Intermediate function is used instead of the accumulate operation for updating the reduction object. In iteration $i$, the map function will produce the itemset candidate as the key and the count one as the value, if this itemset can be found in the transaction. After the Map phase is done, for each distinct itemset candidate, the reduce function will sum up all the one’s associated with this itemset. If its total count of occurrences is not less than the support level, this itemset and the total count will be emitted as the reduce output pair. The reduce output will be used in the update frequent candidates to compute $L_i$ for the current iteration and use it as the $C_{i+1}$ for next iteration, the same as in Generalized Reduction.

By comparing the implementations, we can make the following observations. In Generalized Reduction, the reduction object is explicitly declared to store the number of transactions that own each itemset candidate. During each iteration, the reduction object has to be created and initialized first as the candidate itemsets are dynamically growing. For each itemset candidate, the reduction object is updated in the reduction operation, if it exists in a transaction. When all transactions are processed, the update frequent candidates operation will compute the frequent itemsets and use it as the new candidate itemsets for next iteration.

In map-reduce, the map function checks the availability of each itemset candidate in a transaction and emits it with the count one if applicable. The reduce function will gather the counts of occurrences with the same itemset candidate and compare the total count with the support level. To summarize, for each distinct itemset candidate, the generalized reduction sums up the number of transactions it belongs to and checks whether the total number is not less than the support level. In comparison, map-reduce examines whether
the itemset candidate is in the transaction in the *Map* phase, without performing the sum. It then completes the accumulation and compares the total count with the support level in the *Reduce* phase. Therefore, the *reduction* can be seen as a combination of *map* and *reduce* connected by the *reduction object*. Besides, the *intermediate pairs* produced by the *Map* tasks, will be stored in the file system or main memory and accessed by the *Reduce* tasks. The number of such intermediate pairs can be very large. In apriori, processing each transaction can produce several such pairs. In comparison, the reduction object only has a single count for each distinct itemset. Therefore, it requires much less memory. In addition, map-reduce requires extra sorting, grouping, and shuffling of the intermediate pairs, since each reduce task accumulates pairs with the same *key* value. In comparison, the generalized reduction API does not have any of these costs.

### 3.2 System Design

In this section, we discuss the design and implementation of the MATE system. This system has been implemented on top of Phoenix, a shared memory implementation of map-reduce developed at Stanford [111].

#### 3.2.1 The System API

The current implementation of our system is written in C. The system includes one set of system functions that are transparent to the programmers, in addition to the two sets of APIs, which are summarized in Table 3.2. The first API set are the functions that are simply invoked in the application code to initialize the system, perform the computation, and produce the final output. The second API set includes the functions that have to be defined by the programmers, specific to an application.
Table 3.1: The scheduler\_args\_t Data Structure Type

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic Fields</strong></td>
<td></td>
</tr>
<tr>
<td>Input_data</td>
<td>Data pointer to the input dataset</td>
</tr>
<tr>
<td>Data_size</td>
<td>Size of the input dataset</td>
</tr>
<tr>
<td>Data_type</td>
<td>Type of the data instance</td>
</tr>
<tr>
<td>Stage_num</td>
<td>Computation-Stage number</td>
</tr>
<tr>
<td>Splitter</td>
<td>Pointer to the Splitter function</td>
</tr>
<tr>
<td>Reduction</td>
<td>Pointer to the Reduction function</td>
</tr>
<tr>
<td>Finalize</td>
<td>Pointer to the Finalize function</td>
</tr>
<tr>
<td><strong>Optional Fields</strong></td>
<td></td>
</tr>
<tr>
<td>Unit_size</td>
<td># of bytes of one data instance</td>
</tr>
<tr>
<td>L1_cache_size</td>
<td># of bytes of L1 data cache</td>
</tr>
<tr>
<td>Model</td>
<td>Shared-memory parallelization model</td>
</tr>
<tr>
<td>Num_threads</td>
<td>Max # of threads for Reduction workers</td>
</tr>
<tr>
<td>Num_procs</td>
<td>Max # of processors cores available</td>
</tr>
</tbody>
</table>

The most important operation is the reduction function that processes one split of the data and updates the reduction object. Besides, the programmers can also define application-specific splitter function to split the data before each stage. Note that the splitter, reduction, and combination functions can be different for each iteration, if necessary. In other words, multiple such functions may be defined. Also, to maintain the computation state, the reduction object after each iteration’s processing is stored as intermediate results and can be reused by calling the get\_intermediate\_result function in future stages. The descriptions of other functions are listed in Table 3.2. Note that C pointers can provide flexibility in handling different data types. Thus, the function arguments are declared as void pointers at several places to utilize this feature.

Apart from the API functions, the data structure used to communicate between the application code and the runtime system is of type scheduler\_args\_t. Table 3.1 shows the basic and optional fields of this structure. The basic fields should be appropriately set in the
application code since they provide the data pointers to both the input/output data buffers and the user-defined functions. The optional fields are used for scheduling and performance tuning.

### 3.2.2 Runtime System

As stated above, our system is built on Phoenix runtime, which in turn is based on P-threads [111]. We use the same scheduling strategy, where the data is partitioned and the splits are assigned to threads dynamically.

**Execution Overview:** Figure 3.1 gives an overview of the execution of an application. Some of the key aspects are as follows:

*Scheduler Initialization:* To initialize the scheduler, the programmer needs to specify all required data pointers and functions. Particularly, the programmer should pass the input data pointers and implement the *splitter*, *reduction*, and *combination* functions. Also, the programmer has to declare a *reduction object*. 

![One-Stage Execution Overview in MATE](image-url)
Reduction: After initialization, the scheduler checks the availability of processor cores, and for each core, it creates one worker thread. Before the Reduction phase starts, the splitter function is used to partition the data into equal-sized splits. Each worker thread will then invoke the splitter to retrieve one split and process it with the reduction function.

To avoid load imbalance, we try to assign Reduction tasks to worker threads dynamically instead of statically partitioning the data. Using this approach, each thread worker will repeatedly retrieve one split of the data and then process it as specified in the reduction function until no more splits are available. The split size, however, must be set appropriately to balance the lower overheads (few larger splits) and the load balancing (more smaller splits). To make use of temporary locality, by default, the runtime adjusts the split size such that the input data for a Reduction task can fit in the L1 cache. The programmer can also vary this parameter to achieve better performance given the knowledge of a specific application.

The reduction object is updated correspondingly in the reduction function as specified by the user code. Each thread will repeat this step until all data instances have been processed. At this point, the Reduction stage is over. The scheduler must wait for all reduction workers to finish before initializing the Combination stage.

Combination: The processing structure of Generalized Reduction enables us to develop general but efficient shared-memory parallelization schemes. Consider again the loop in Figure 1.1(b). The main correctness challenge in parallelizing a loop like this on a shared-memory machine arises because of possible race conditions (or access conflicts) when multiple processors update the same element of the reduction object. Two obvious approaches for avoiding race conditions are: full replication and full locking. In the full replication approach, each processor can update its own reduction object and these copies are merged
together later. In the full locking approach, one lock or latch is associated with each aggregated value.

Currently, we use the full replication approach to implement the parallelization. So a Combination phase is required to merge the reduction-objects of multiple threads when all data is processed. The full locking and its optimized schemes are being considered and will be supported in the future.

In the Combination phase, the scheduler will spawn a combination-worker thread to merge the reduction object copies of all Reduction threads. The result is a single copy of the final reduction object after every iteration. If the application involves multiple iterations, this object is stored by the scheduler and can be accessed by the future stages.

**Finalize:** This phase occurs after all other stages of the computation have been performed, i.e., each Combination stage has its own copy of the reduction object, which is combined from multiple threads. The programmer is then able to perform a manipulation on the reduction-objects to summarize the results specific to an application. The scheduler also needs to free the allocated space in this phase, but this is transparent to the programmer.

**Buffer Management:** The runtime system handles two types of temporary buffers. One is the reduction-object buffer allocated for each thread to do its own computation over different splits. The other is the combination buffer created for storing the intermediate output results of each stage. The combination buffer is of type reduction object in our implementation. The buffers are initially sized to a default value and then grown dynamically if needed.

**Fault Tolerance:** Fault detection and recovery have been an important aspect of map-reduce implementations, as data-intensive applications can be very long running, and/or may need to be executed on a large number of nodes or cores. Our current support for
fault-tolerance is based on the Phoenix system. It supports fault-recovery for the Reduction tasks. If a Reduction worker does not finish within a reasonable time limit, the scheduler assumes that a failure of this task has occurred. The failed task is then re-executed by the runtime. In our implementation, separate buffers are allocated for the new task to avoid data access conflicts.

The alternate API implemented in the MATE system also allows a different approach for fault-tolerance. This will be a low-cost checkpointing approach, where a copy of the reduction object can be cached at another location after processing of each split. If a thread fails while processing a particular split, we can restart computation using the cached reduction object. We only need to process the current split assigned to this thread, and not the splits that had been already processed. We plan to support this approach in future versions of the MATE system.

### 3.3 Experimental Results

In this section, we evaluate the MATE system on multi-core machines by comparing its performance against Phoenix and Hadoop.

Since our main focus is on data mining applications, we choose three popular data mining algorithms. They are k-means clustering, apriori association mining, and principal components analysis (PCA). Among these, we described apriori association mining earlier in Section 3.1. K-means clustering is one of the commonly used data mining algorithms [76]. The clustering problem is as follows. We consider data instances as representing points in a high-dimensional space. Proximity within this space is used as the criterion for classifying the points into clusters. Four steps in the sequential version of k-means clustering algorithm are as follows: 1) start with $k$ given centers for clusters; 2) scan the data instances, for each
data instance (point), find the center closest to it and assign this point to the corresponding cluster, 3) determine the $k$ centroids from the points assigned to the corresponding centers, and 4) repeat this process until the assignment of points to clusters does not change. PCA is a popular dimensionality reduction method that was developed by Pearson in 1901. Its goal is to compute the mean vector and the covariance matrix for a set of data points that are represented by a matrix.

The datasets and the application parameters we used are as follows. For k-means, the dataset size is 1.2 GB, and the points are 3-dimensional. The number of clusters, $k$, was set to be 100. With apriori, we used a dataset that has 1,000,000 transactions, with each one having at most 100 items, and the average number of items in each transaction is 10. The support and confidence levels were 3% and 9%, respectively, and the application ran for 2
Figure 3.3: K-means: Comparison between MATE, Phoenix and Hadoop on 16 cores

passes. In PCA, the number of rows and columns used in our experiments were 8,000 and 1,024, respectively.

Our experiments were conducted on two distinct multi-core machines. One system uses Intel Xeon CPU E5345, comprising two quad-core CPUs (8 cores in all). Each core has a clock frequency of 2.33GHz and the system has a 6 GB main memory. The other system uses AMD Opteron Processor 8350 with 4 quad-core CPUs (16 cores in all). Each core has a clock frequency of 1.00GHz and the system has a 16 GB main memory.

We executed the aforementioned three applications with both MATE and Phoenix systems on the two multi-core machines. For two of the applications, k-means and apriori, we also do a comparison with Hadoop. We could not compare the performance of PCA on Hadoop, as we did not have an implementation of PCA on Hadoop. Hadoop implementations of k-means and apriori were carefully tuned by optimizing various parameters [79].
Figures 3.2 through 3.7 show the comparison results for these three applications as we scale the number of cores used. Since our system uses the same scheduler as Phoenix, the main performance difference is likely due to the different processing structure, as we had discussed in earlier sections. Note that for Hadoop, we only report results from the maximum number of cores available on each machine (8 and 16). This is because the number of tasks in Hadoop cannot be completely controlled, as there was no mechanism available to make it use only a subset of the available cores.

Figures 3.2 and 3.3 show the comparison for k-means. Our system outperforms both Phoenix and Hadoop on both the machines. MATE is almost twice as fast as Phoenix with 8 threads on the 8-core machine, and almost thrice as fast with 16 threads on the 16-core machine. Hadoop is much slower than both MATE and Phoenix, which we believe is due to the high overheads in data initialization, file system implementation, and use
of Java for computations. As compared to MATE, it also has the overhead of grouping, sorting, and shuffling of intermediate pairs [79]. We also see a very good scalability with increasing number of threads or cores with MATE. On the 8-core machine, the speedup with MATE in going from 1 to 8 threads is about 7.8, whereas, Phoenix can only achieve a speedup of 5.4. The result is similar on the 16-core machine: our system can get a speedup of about 15.0 between 1 and 16 threads, whereas, it is 5.1 for Phoenix. With more than 100 million points in the k-means dataset, Phoenix was slower because of the high memory requirements associated with creating, storing, and accessing a large number of intermediate pairs. MATE, in comparison, directly accumulates values into the reduction object, which is of much smaller size.

The results from PCA are shown in Figures 3.4 and 3.5. PCA does not scale as well with increasing number of cores. This is because some segments of the program are either
not parallelized, or do not have regular parallelism. However, we can still see that, with 8 threads on the 8-core machine and 16 threads on the 16-core machine, our system was twice as fast as Phoenix. Besides the overheads associated with the intermediate pairs, the execution-time breakdown analysis for Phoenix shows that the \textit{Reduce} and \textit{Merge} phases account for a non-trivial fraction of the total time. This overhead is non-existent for the MATE system, because of the use of the \textit{reduction object}.

Finally, the results for apriori is shown in Figures 3.6 and 3.7. For 1 and 2 threads, MATE is around 1.5 times as fast as Phoenix on both the machines. With 4 threads and more, the performance of Phoenix was relatively close to MATE. One of the reasons is that, compared to k-means and PCA, the \textit{Reduce} and \textit{Merge} phases take only a small fraction of time with the apriori dataset. Thus, the dominant amount of time is spent on the \textit{Map} phase, leading to good scalability for Phoenix. For the MATE system, the \textit{reduction object}
can be of large size and needs to be grown and reallocated for all threads as the new candidate itemsets are generated during the processing. This, we believe, introduced significant overheads for our system. Hadoop was much slower than our system, by a factor of more than 25 on both the machines.

Overall, our experiments show that the MATE system has between reasonable to significant speedups for all the three datamining applications. In most of the cases, it outperforms Phoenix and Hadoop significantly, while slightly better in only one case.

3.4 Summary

This work has described a system MATE, which provides an alternate API to the original map-reduce model for developing data-intensive applications. Our API takes advantage of a user-declared reduction object, which substantially reduces an overhead arising
void reduction(void * reduction_data) {
    for each transaction ∈ reduction_data{
        for (i = 0; i < candidates.size; i + +) {
            match = false;
            itemset = candidates[i];
            match = itemset.exists(transaction, itemset);
            if (match == true) {
                object_id = itemset.object_id;
                accumulate(object_id, 0, 1);
            }
        }
    }
}

void update_frequent_candidates(int stage_num) {
    j = 0;
    for (i = 0; i < candidates.size; i + +) {
        object_id = candidates[i].object_id;
        count = get_intermediate_result(stage_num, object_id, 0);
        if (count >= (support_level * num_transactions)/100.0) {
            temp_candidates[j + +] = candidates[i];
        }
    }
    candidates = temp_candidates;
}

Figure 3.8: Pseudo-code for apriori using Generalized Reduction API

because of the original map-reduce API. These overheads are caused by high memory requirements of intermediate results and the need for data communication between Map and Reduce stages. MATE supports this alternate API on top of Phoenix, a multi-core map-reduce implementation from Stanford.

We have evaluated our system with three popular data mining algorithms, which are k-means clustering, apriori association mining, and principal component analysis. We also compared our system with Phoenix and Hadoop on two distinct multi-core platforms. Our
results show that MATE achieves good scalability and outperforms Phoenix and Hadoop for all three applications. Overall, our work suggests that, despite an easy to program API support, map-reduce may not fit naturally into some data-intensive applications and results in non-trivial performance losses. An alternate API, which is based on the generalized reduction, offers better performance and scalability while still maintaining easy programmability.
MAPREDUCE (Apriori)

```c
void map(void * map_data) {
    for each transaction ∈ map_data{
        for (i = 0; i < candidates_size; i++){
            match = false;
            itemset = candidates[i];
            if (match == itemset_exists(transaction, itemset)) {
                emit_intermediate(itemset, one);
            }
        }
    }
}
```

```c
void reduce(void * key, void ** vals, int vals_length) {
    count = 0;
    for (i = 0; i < vals_length; i++){
        count += vals[i];
    }
    if (count >= (support_level * num_transactions)/100.0) {
        emit(key, count);
    }
}
```

```c
void update_frequent_candidates(void * reduce_data_out) {
    j = 0;
    length = reduce_data_out− > length;
    for (i = 0; i < length; i++){
        temp_candidates[j + ] = reduce_data_out− > key;
    }
    candidates = temp_candidates;
}
```

Figure 3.9: Pseudo-code for apriori using MapReduce API
### Functions Provided by the Runtime

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int mate.init(scheduler_args_t*)</code></td>
<td>Initializes the runtime system. The arguments provides the needed functions and data pointers</td>
</tr>
<tr>
<td><code>int mate.scheduler(void*)</code></td>
<td>Schedules the Reduction tasks on the input dataset</td>
</tr>
<tr>
<td><code>int mate.finalize(void*)</code></td>
<td>Executes the Finalizing task to generate a final output if needed</td>
</tr>
<tr>
<td><code>void reduction_object.pre_init()</code></td>
<td>Allocates a default size of space for the reduction object</td>
</tr>
<tr>
<td><code>int reduction_object.alloc()</code></td>
<td>Assigns a unique object id representing a group of consecutive elements in the reduction object</td>
</tr>
<tr>
<td><code>void reduction_object.post_init()</code></td>
<td>Clones the reduction object for all threads in the full replication technique</td>
</tr>
<tr>
<td><code>void accumulate(int, int, void* value)</code></td>
<td>Updates a particular element of the reduction object by addition. The arguments are the object id, the offset, and the associated value</td>
</tr>
<tr>
<td><code>void reuse_reduction_object()</code></td>
<td>Reuses the reduction object for future stages. Memsets all fields of the reduction object to be zero</td>
</tr>
<tr>
<td><code>void process.next_iteration()</code></td>
<td>Notifies the runtime that the computation is proceeding to next stage</td>
</tr>
<tr>
<td><code>void* get_reduction_object(int)</code></td>
<td>Retrieves the reduction object copy belonging to one thread. The argument is the thread number</td>
</tr>
<tr>
<td><code>void* get_intermediate_result(int, int, int)</code></td>
<td>Retrieves a particular element of the reduction object in the combination buffer. The arguments are the stage number, the object id, and the offset</td>
</tr>
</tbody>
</table>

### Functions Defined by the User

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int (*splitter_j)(void*, int, reduction_args_t*)</code></td>
<td>Splits the input dataset across Reduction tasks</td>
</tr>
<tr>
<td><code>void (*reduction_f)(reduction_args_t*)</code></td>
<td>The Reduction function. Each Reduction task executes this function on its input split. This function must be implemented by the user</td>
</tr>
<tr>
<td><code>void (*combination_f)(void*)</code></td>
<td>The Combination function. In the full replication model, this function is used to combine results from all threads</td>
</tr>
<tr>
<td><code>void (*finalize_f)(void*)</code></td>
<td>The Finalize function. The runtime will execute this function when all computation is done</td>
</tr>
</tbody>
</table>
Chapter 4: Ex-MATE: Data-Intensive Computing with Large Reduction Objects and Its Application to Graph Mining

The APIs comparison between MATE and Phoenix has shown that an alternate API MATE, where a reduction object is explicitly maintained and updated, reduces memory requirements and can significantly improve performance for many applications. However, unlike the original API, support for the alternate API has been restricted to the cases where the reduction object can fit in the memory. This limits the applicability of the MATE approach. Particularly, one emerging class of applications that require support for large reduction objects are the graph mining applications.

To address such a limitation, in this work, we describes a system, Extended MATE or Ex-MATE, which supports this alternate API with reduction objects of arbitrary sizes. We develop support for managing disk-resident reduction objects and updating them efficiently. We evaluate our system using three graph mining applications and compare its performance to that of PEGASUS, a graph mining system implemented based on the original map-reduce API and its Hadoop implementation. Our results on a cluster with 128 cores show that for all three applications, our system outperforms PEGASUS, by factors ranging between 9 and 35.
4.1 Ex-MATE Design

As we discussed in the previous section, the MATE captures the computational state of the algorithms in the reduction object. For many applications, the reduction object has a much smaller size than the input dataset. For example, in k-means clustering algorithm, the reduction object size depends upon the number of clusters being computed and is independent of the input dataset size. Thus, even if the input is in tera-bytes, the reduction object may only be in kilo-bytes. For such applications, by maintaining the small-sized reduction object in the memory, the performance is greatly improved. However, there are also several other data-intensive applications where this is not true. Many graph mining algorithms involve Sparse Matrix-Vector Multiplication (SpMV) or its variant as the main computational step. Implementing SpMV in MATE will treat the output vector as the reduction object, which can actually be very large in size. The situation is also similar for some data pre-processing algorithms, where the output is not much smaller than the input.

This section describes the implementation of the Extended MATE or Ex-MATE system, which provides support for management of a large reduction-object in distributed environments.

**Overview of Extended MATE:** Figure 4.1 gives an overview of the execution flow of a typical application using Ex-MATE in a distributed environment. The middleware comprises of two components: a runtime system and a user program. The runtime is responsible for partitioning the input datasets, assigning splits to computing nodes, and scheduling the computation for all stages. The user program needs to specify important functions like **reduction**, **combination**, and/or **finalize**. The runtime will invoke the corresponding functions in different stages and these functions act as the interaction between the runtime and the
user code. The most important stage is the Reduction, where all the data instances are processed and the reduction objects are updated. After the Reduction, the global combination function will merge results from multiple nodes and form a final copy of reduction object. The user can choose to operate on the final reduction object in the finalize function.

While Ex-MATE allows large-sized reduction objects, the fact that they can be disk-resident is transparent to application developers. Thus, while the large-size reduction objects are allocated on local disks, the user can write applications manipulating them as if they are in memory. Updates to the disk-resident locations are handled by the runtime system.

Figure 4.1: Execution Overview of the Extended MATE
Another interesting aspect of Ex-MATE is the support for processing of very large datasets. First, given the input datasets, the runtime system will invoke a partitioning function to partition the inputs into a number of smaller splits. A default partitioning function is provided by the runtime, but the user can also specify a customized partitioning function. Then, as one of the scheduling tasks, the input splits are distributed and assigned to different nodes. Data locality is considered so that each node can read most of the input splits assigned to it from its own local disk. Customized partitioning functions enable better locality while updating a disk-resident reduction object.

**Management of A Large Reduction-Object:** To support applications that involve a large reduction object, we need to keep the entire reduction object on disks. Moreover, so that each node can update its own copy of the reduction object, a copy is maintained in each node’s local disk. The global combination phase later merges the disk-resident copies from all nodes and forms a final copy that is eventually distributed to all nodes.

Even with the reduction object on a local disk, the performance will be poor if there are frequent disk I/O operations. Thus, one of the obvious optimizations is to divide the entire reduction object into smaller parts or *splits* and only load one or more needed splits into memory based buffers. This can be done in a demand-driven fashion. Thus, if we find that the reduction function applied on an input element results in updates to element(s) from a particular split of the reduction object, this split is loaded into the memory. Such write operations are applied in the memory based buffers first and then written back to disk, before loading other splits of the reduction object into the same buffers. Another split of the reduction object is loaded when any element(s) from that split need to be updated. At this time, the previous split(s) could be written back to the disk.
Figure 4.2: Matrix-Vector Multiplication using checkerboard partitioning. $B(i,j)$ represents a matrix block, $I_V(j)$ represents an input vector split, and $O_V(i)$ represents an output vector split. The matrix/vector multiplies are done block-wise, not element-wise.

The key issue with this approach is finding input elements to process that involve updating a certain portion of the reduction object and processing them together. In the worst case, each pair of consecutive input elements may result in updates to different splits and the overhead of reading and writing the elements can be extremely high. Another option could be to load a split of the reduction object and then perform all processing associated with it. However, this may require that each portion of the input file is read and processed multiple times.

A much better solution can be developed for many algorithms, where the input can be partitioned in a way that processing each partition of the input results in updates to only one portion of the reduction object. The user-provided input partitioning method is used for this purpose. We will now use matrix-vector multiplication as an example to show how we
can make use of the data access patterns to improve performance. In the extended MATE implementation, the matrix as well as the initial vector would be used as the input, and the user would declare a reduction object containing the partial sums for the output vector. Both the inputs and the reduction object are stored on disk due to their large size. Let $m_{i,j}$ denote the $(i, j)$-th element of matrix $M$ and $v_i$ denote the $i$-th element of vector $V$. In each iteration, for $m_{i,j}$, the computation involves the multiplication of $m_{i,j}$ and $v_j$ of the input vector and the result will be accumulated for $v_i$ in the output vector by updating the reduction object. Generally, every multiplication needs to update an offset of the reduction object. After all computation is done, the reduction object will contain final sums and can be used as the input vector for the next iteration. Since the reduction object resides on disk, it is easy to see the high costs if the accumulations are done one by one on the disk. Considering the data access pattern of this multiplication, however, we can process the computation in a block fashion to reduce disk I/O.

As Figure 4.2 illustrates, using what is called the checkerboard partitioning, the input vector and the reduction object (output vector) would be partitioned into $n$ splits and the matrix would be partitioned into $n^2$ blocks. In this block partitioning scheme, one or more blocks of the matrix can be loaded into memory at the same time, as well as the corresponding input/output vector splits. For example, when we process the matrix block $B(0, 0)$, we only need the input vector split $I_{V}(0)$ and the output vector split $O_{V}(0)$. So, we check whether $O_{V}(0)$ has been loaded into memory first and read it if it is not the case. Then, the subsequent updates can be performed in memory.

As we stated above, this portion of the output continues to be allocated in the memory till it needs to be replaced by the next reduction object split. Thus, our goal is to reuse the reduction object split that already resides in memory, for which we can re-order the
computation sequence of matrix blocks to maximize the number of hits for reduction object splits. In this way, the disk I/O is significantly reduced, by updating the reduction object block by block, instead of updating it element by element.

One issue with this approach is setting the block/split size appropriately. With a smaller block/split size, we can make sure that at least one matrix block and two vector splits can fit into memory. With a larger one, the spatial locality could be better utilized by reading more data at one time from the disk. The block/split size can either be empirically tuned or could be pre-defined according to the memory constraints and the input dataset size. Overall, we can see that this block-partition scheme can take advantage of the data access patterns of the matrix-vector multiplications and reduces the number of times needed to access the disk significantly.

### 4.2 Parallel Graph Mining Using Ex-MATE

We now describe how the large reduction object support in Ex-MATE can be used for graph mining. Our work is based on the recent work from CMU [86], which views the main computations in several graph mining algorithms as Generalized Iterative Matrix-Vector Multiplication or GIM-V. In this section, we will briefly describe the GIM-V concept, show how a popular graph mining algorithm fits into the structure of GIM-V, and then focus on how to parallelize GIM-V using both map-reduce and the Ex-MATE API.

First, suppose we have a $n$ by $n$ matrix $M$ and a vector $V$ of size $n$. Let $m_{i,j}$ denote the $(i, j)$-th element of $M$ and $v_i$ denote the $i$-th element of $V$. Then, the normal matrix-vector multiplication is $V' = M \times V$ where, $v'_i = \sum_{j=1}^{n} m_{i,j} v_j$. It is easy to see that there are three operations involved in this computation:

- **combine**: multiply $m_{i,j}$ and $v_j$
• \textit{combineAll}_i: sum \( n \) partial multiplication results for the element \( i \)

• \textit{assign}: the previous value of \( v_i \) is overwritten.

\textit{GIM-V} keeps the overall structure, but the expression is denoted as \( V' = M \times_G V \), where \( \times_G \) is defined by the user. Particularly, we have

\[ v'_i = assign(v_i, \text{combineAll}_i(\{x_j | j = 1 \ldots n\})) \]

\[ x_j = \text{combine}(m_{i,j}, v_j) \]

The functions \textit{combine}, \textit{combineAll}, and \textit{assign} have the following generalizations:

• \textit{combine}(\( m_{i,j} \), \( v_j \)): \textit{combine} \( m_{i,j} \) and \( v_j \) (i.e. it does not have to be a multiplication.)

• \textit{combineAll}_i(\( x_1, \ldots, x_n \)): \textit{combine} \( n \) partial results for the element \( i \) (i.e. it does not have to be the sum.)

• \textit{assign}(\( v_i \), \( v_{\text{new}} \)): the previous value of \( v_i \) is updated by a new value.

Using \textit{GIM-V}, and particularly by providing different definitions of the two \textit{combine} functions, several key graph mining algorithms can be developed, including \textit{PageRank}, \textit{Random Walk with Restart}, \textit{Connected Components}, and \textit{Diameter Estimation}. Now, we will use PageRank as an example to show how it is an instantiation of \textit{GIM-V}. Later in this section, we also briefly discuss algorithms for Diameter Estimation and finding Connected Components. More details of these algorithms can be seen from the publications of the PEGASUS project [85, 86].
4.2.1 *GIM-V and PageRank*

PageRank [29] is a well-known algorithm used by Google to calculate the relative importance of web pages. The PageRank vector $p$ of $n$ web pages satisfies the following eigenvector equation:

$$p = (cE^T + (1 - c)U)p$$

where, $c$ is a damping factor, $E$ is a row-normalized adjacency matrix, and $U$ is a matrix with all elements initialized to $1/n$. To calculate $p$, the power method is used to multiply an initial vector with the matrix a number of times. Initially, all elements in the current PageRank vector $p^{\text{cur}}$ are set to be $1/n$. Then the next PageRank $p^{\text{next}}$ is calculated as

$$p^{\text{next}} = (cE^T + (1 - c)U)p^{\text{cur}}$$

This multiplication is repeated until $p$ converges.

In the representation of *GIM-V*, a matrix $M$ is constructed by column-normalized $E^T$ such that elements in every column of $M$ sum up to 1. Then, the next PageRank is calculated as $p^{\text{next}} = M \times_G p^{\text{cur}}$ where the three operations are defined as follows:

$$\text{combine}(m_{i,j}, v_j) = c \times m_{i,j} \times v_j$$

$$\text{combineAll}_i(x_1, \ldots, x_n) = \frac{1 - c}{n} + \sum_{j=1}^{n} x_j$$

$$\text{assign}(v_i, v_{\text{new}}) = v_{\text{new}}$$

4.2.2 Parallelization of GIM-V

*GIM-V* and its different instantiations have been implemented using map-reduce in PEGASUS [86], which is a graph mining package based on Hadoop. In this section, we
describe how GIM-V can be parallelized using the Ex-MATE API and compare the implementation with that of map-reduce.

Figure 4.9 and Figure 4.10 show the map-reduce code for the two stages of GIM-V. The inputs are an edge file and a vector file. Each line of the edge file contains one \((id_{\text{src}}, id_{\text{dst}}, mval)\) tuple, which corresponds to a non-zero cell in the adjacency matrix \(M\). Similarly, each line of the vector file contains one \((id, vval)\) pair, which corresponds to an element in the vector \(V\). Stage 1 performs the combine operation by combining columns of the matrix \((id_{\text{dst}} \text{ of } M)\) with the corresponding values of the vector \((id \text{ of } V)\). The output of the first stage are \((key, value)\) pairs where the key is the source node id of the matrix (and the value is the partially combined result \(\text{combine}(mval, vval)\)). The output of the first stage becomes the input of the second stage, which now combines all partial results from first stage and creates the new vector.

In contrast, the Ex-MATE API based implementation is more straightforward and only one stage is needed. Figure 4.11 shows the outline of the implementation. The inputs are the same as in map-reduce. The user now needs to declare a reduction object containing the partial sums for the output vector. The reduction function would simply combine \((id_{\text{src}}, id_{\text{dst}}, mval)\) with \((id_{\text{dst}}, vval)\) from the given inputs. The partially combined results from \(\text{combine}(mval, vval)\) are used to update the node \((id_{\text{src}})\) of the output vector.

By comparing the two implementations, we can make the following observations. With the Ex-MATE API, the reduction object is explicitly declared to store the partial results of the output vector. For each partially combined result computed from \((mval, vval)\) pair, the corresponding element of the reduction object is updated in the reduction function. When all reduction tasks are finished, the reduction object will contain the final results for the output vector. In map-reduce, we need two stages. In the first stage, the map function
outputs the \((key, value)\) pairs with \(id_{dst}\) as the key. The reduce function would simply reduce these partially combined results for the corresponding element of the output vector. In the second stage, the map function will simply emit the input \((key, value)\) pairs without any computation and these pairs will be grouped by \(id_{src}\). In the reduce function that follows, all partial vectors could be combined together as the final results for each row.

4.2.3 Other Algorithms

We now briefly describe the other two applications. Diameter Estimation is used to estimate the diameter and the set of radii for all nodes in a given graph. The diameter of a graph is defined as the maximal length from the set of shortest paths between all pair of nodes. The radius of a node \(v_i\) refers to the number of hops for the node \(v_i\) to reach its farthest peer. HADI [85] is an algorithm to calculate such parameters. In particular, the number of neighbors within the hop \(h\) reachable from node \(v_i\) is encoded in a probabilistic bit-string \(b^h_i\). Thus, the main idea of HADI is to update \(b^h_i\) as \(h\) increases for each iteration. Like PageRank, HADI can also be mapped to GIM-V using the expression \(b^{h+1} = M \times G b^h\), where \(M\) is an adjacency matrix. \(b^{h+1}\) is a vector of length \(n\), which is updated by

\[b^{h+1}_i = \text{assign}(b^h_i, \text{combineAll}_i(\{x_j|j = 1 \ldots n\}))\]

\[x_j = \text{combine}(m_{i,j}, b^h_j)\]

And its three customized operations are:

\[\text{combine}(m_{i,j}, v_j) = m_{i,j} \times v_j\]

\[\text{combineAll}_i(x_1, \ldots x_n) = \text{BITWISE} - \text{OR}\{x_j|j = 1 \ldots n\}\]

\[\text{assign}(v_i, v_{new}) = \text{BITWISE} - \text{OR}(v_i, v_{new})\]
Similarly, HCC [86] is a new algorithm to find the connected components of large graphs. For each node $v_i$, we use $c^h_i$ to represent the minimum node id within $h$ hops from $v_i$. We start from $c^0_i$ with its initial value being its own node id $i$. In each iteration, $c^h_i$ is updated by finding the minimum value among its current component id and the component ids received from its neighbors. The important observation is that this process can also be applied to GIM-V with the expression $c^{h+1} = M \times_G c^h$, where $M$ is an adjacency matrix. $c^{h+1}$ is a vector of length $n$, which is updated as

$$c^{h+1}_i = assign(c^h_i, combineAll_i(\{x_j | j = 1...n\}))$$

$$x_j = combine(m_{i,j}, c^h_j)$$

And its three customized operations are:

$$combine(m_{i,j}, v_j) = m_{i,j} \times v_j$$

$$combineAll_i(x_1...x_n) = \text{MIN}\{x_j | j = 1...n\}$$

$$assign(v_i, v_{\text{new}}) = \text{MIN}(v_i, v_{\text{new}})$$

4.3 Experimental Results

In this section, we present results from an evaluation study comparing the Ex-MATE system with Hadoop implementation of map-reduce on a cluster of multi-core machines. Particularly, we have compared graph mining implementations on Ex-MATE against the implementation that were part of the PEGASUS system, focusing on PageRank, Diameter Estimation, and finding Connected Components, which were all described in the previous section.
Our experiments were conducted on a cluster of multi-core machines. The system uses Intel Xeon CPU E5345, comprising of two quad-core CPUs (8 cores in all). Each core has a clock frequency of 2.33GHz and the main memory is of size 6 GB. We have used up to 128 cores for our study.

PEGASUS provides a naive implementation of GIM-V in map-reduce and also an optimized block-version of multiplication. We have included the results for both the naive and the block implementations, denoting these versions as P-Naive and P-Block. We executed each application for a number of iterations and take the average time of each iteration as the comparison criteria. The P-Block versions involve a non-trivial amount of time for pre-processing the input datasets. Because this additional pre-processing step need to be done only once, whereas the output of pre-processing can be used by every iteration, we do
Figure 4.4: HADI: Comparison between Ex-MATE and PEGASUS on 4, 8 and 16 nodes for a 16G matrix

not include the pre-processing times while reporting our results. We also carefully tuned Hadoop to achieve the best possible performance.

Our initial study used a graph with 256 million nodes and 1 billion edges, represented by an adjacency matrix of 16 GB bytes, for all three algorithms. Along with the matrix, we have an input vector with its length being the number of nodes and a size of 2 GB bytes. The initial value of the input vector is algorithm-specific.

The output vector is represented by the reduction object, which has the same size as the input vector. The 6 GB main memory on each node needs to be shared by input buffers, reduction object(s), and other application and kernel memory requirements. Thus, it is clearly not possible to have a private allocated copy of the 2 GB reduction object for each worker thread. The split size of the reduction object is set to 128 MB for our experiments.
Figures 4.3 through 4.5 show the comparison results for these three applications on the 16 G matrix as we scale the number of nodes. As we can see in Figure 4.3, for PageRank, our system outperforms both P-Naive and P-Block on 4, 8, and 16 nodes. Especially, Ex-MATE is 35 times faster than P-Naive and 10 times faster than P-Block on 16 nodes. Ex-MATE and PEGASUS shows similar scalability as we increase the number of nodes. The speedup is around 2.2 when the number of nodes is increased from 4 to 16 for the two systems. This is due to some fixed overheads in both systems, related to communication of a large reduction object (or the number of \((key, value)\) pairs after combine function). P-Block is faster than P-Naive because the block version can reduce the size of intermediate results and also compress the input dataset size. But it is still slower than Ex-MATE in all cases.
Figure 4.6: HCC: Comparison between Ex-MATE and PEGASUS on 4, 8 and 16 nodes for a 8G matrix

We have similar observations from Figure 4.4 (for HADI) and Figure 4.5 (for HCC). For HADI on 16 nodes, Ex-MATE is 27 times as fast as P-Naive and 11 times as fast as P-Block. The speedup from 4 nodes to 16 nodes is 1.9 on Ex-MATE, 1.8 on P-Naive, and 1.7 on P-Block respectively. For HCC, Ex-MATE is 23 times faster than P-Naive and 9 times faster than P-Block with 16 nodes. Ex-MATE has a speedup of 1.8 from 4 nodes to 16 nodes, P-Naive has around 1.9 and P-Block is around 1.7. The similar trends in performance difference are consistent with the fact that all the three applications are implemented using the same underlying GIM-V method.

We also vary the dataset size to evaluate the scalability of two systems. We run the HCC algorithm on three additional graphs, which are represented by three matrices of the size 8 GB, 32 GB, and 64 GB, respectively. The size of matrices is tuned by varying the number of edges while keeping the same number of nodes in the graph, so the size of the input
Figure 4.7: HCC: Comparison between Ex-MATE and PEGASUS on 4, 8 and 16 nodes for a 32G matrix

vector as well as that of the reduction object remain 2 GB. From Figures 4.6 through 4.8, we can observe a similar trend - Ex-MATE is still much faster than P-Naive and P-Block for the three additional dataset sizes. As we increase the dataset size, we can see that the Ex-MATE has better speedups from 4 to 16 nodes due to an increased amount of computation and also better utilization of the file system throughput. For a smaller dataset (8 GB), we have 1.9 speedup from 4 to 16 nodes, whereas the corresponding speedup is 2.8 with the 64 GB dataset. For PEGASUS, the speedup from 4 to 16 nodes is around 2 for the three additional datasets. As the trends with the other 2 algorithms were very similar, detailed results from other datasets are not presented here.

Overall, our experiments show that the Ex-MATE system can reduce the completion time significantly compared with PEGASUS. The Ex-MATE also has reasonable speedups
Figure 4.8: HCC: Comparison between Ex-MATE and PEGASUS on 4, 8 and 16 nodes for a 64G matrix

for all the three applications on four different datasets and relative speedups seem to improve for larger datasets.

4.4 Summary

This work has described the implementation and evaluation of Ex-MATE, which is based on an alternate high-level API for developing data-intensive applications. Unlike our previous work on the MATE system, Ex-MATE has support for managing disk-resident reduction objects, which are required for many data-intensive applications. We have evaluated the Ex-MATE system with three popular graph mining algorithms, which are PageRank, Diameter Estimation, and finding Connected Components. We also compared our system with PEGASUS in an environment with up to 128 cores. Our experimental results
show that the extended MATE achieves much better performance than Hadoop based PE-GASUS. The scalability of our system is better with larger-sized datasets. Our API also results in simpler code than the use of the original map-reduce API.

In summary, our approach offers a promising alternative for developing efficient data-intensive applications. While our approach is based on a distinct API, it is still a very high-level API and should be easy to learn for those familiar with map-reduce.
MapReduce Stage-1 (GIM − V)

**INPUT**: Matrix \( M = \{ (id_{src}, (id_{dst}, m_{val})) \} \)
Vector \( V = \{ (id, v_{val}) \} \)

**OUTPUT**: Partial Vector
\( V = \{ (id_{src}, combine2(m_{val}, v_{val})) \} \)

```java
1 void map(Key k, Value v) {
2     if (k, v) is of type V{
3         emit_intermediate(id, v_{val}); {
4     }
5     if (k, v) is of type M{
6         emit_intermediate(id_{dst}, (id_{src}, m_{val}));
7     } }
8 }

9 void reduce(Key k, Values vals) {
10    matrix_column = \{ \};
11    vector_{val} = 0;
12    for each v ∈ vals{\n13        if (k, v) is of type V{
14            vector_{val} = v;
15            emit(k, ("v", vector_{val}));
16        } \n17        if (k) is of type M{
18            matrix_column.add(id_{src}, m_{val});
19        } \n20     }
21    for each (id_{src}, m_{val}) ∈ matrix_column{
22        result = combine2(m_{val}, vector_{val});
23        emit(id_{src}, ("m", result));
24    }
25 }
```

Figure 4.9: MapReduce Stage-1 Algorithm for GIM-V
**MapReduce Stage-2 (GIM - V)**

**INPUT**: Partial Vector $P_{\text{V}} = \{(\text{id}_{\text{src}}, p_{\text{V}}\text{val})\}$

**OUTPUT**: Final Vector $V = \{(\text{id}_{\text{src}}, \text{new}_V\text{val})\}$

```java
1 void map(key k, value v) {
2     emit_intermediate(k, v);
3 }

4 void reduce(key k, values vals) {
5     matrix_row = {};
6     vector_val = 0;
7     for each v in vals{
8         (tag, result) = v;
9         if tag == "v"{
10            vector_val = result;
11        }
12         if tag == "m"{
13             matrix_row.add(result);
14         }
15     }
16     final_result = combineAll(matrix_row);
17     emit(k, assign(vector_val, final_result));
18 }
```

Figure 4.10: MapReduce Stage-2 Algorithm for GIM-V
**GENERALIZED REDUCTION (GIM − V)**

**INPUT**: Matrix $M = \{(id_{src}, (id_{dst}, m_{val}))\}$  
Vector $V = \{(id, v_{val})\}$  

**OUTPUT**: Final Vector $V = \{(id_{src}, new_v_{val})\}$

1. `void reduction(void *reduction_data) {`
2.   `partial_result = 0;`
3.   `vector_val = 0;`
4.   `matrix_split = reduction_data −> matrix;`
5.   `vector_split = reduction_data −> vector;`
6.   `for each (id_{src}, id_{dst}, m_{val}) ∈ matrix_split{`
7.     `vector_val = vector_split.get(id_{dst});`
8.     `partial_result = combine2(m_{val}, vector_val);`
9.     `//combineAll() is done inside accumulate()`
10.    `accumulate(0, id_{src}, partial_result);`
11.   `}`
12. `}`

13. `void update_output_vector(void *) {`
14.   `final_result = 0;`
15.  `for each vector_id ∈ V{`
16.    `final_result = reduction_object.get(0, vector_id);`
17.    `assign(vector.get(vector_id), final_result);`
18.  `}`
19. `}

Figure 4.11: Generalized Reduction Algorithm for GIM-V
Chapter 5: MATE-CG: A MapReduce-Like Framework for Accelerating Data-Intensive Computations on Heterogeneous Clusters

Both MATE and Ex-MATE systems target the support of programming traditional CPU-only platforms. Nowadays, clusters of GPUs have rapidly emerged as the means for achieving extreme-scale, cost-effective, and power-efficient high performance computing. While people have been trying to port map-reduce on a single GPU or GPU clusters, developing map-reduce-like models for programming a heterogeneous CPU-GPU cluster remains an open challenge.

So, in this work, we present the MATE-CG system, which is a map-reduce-like framework based on the generalized reduction API. We develop support for enabling scalable and efficient implementation of data-intensive applications in a heterogeneous cluster of multi-core CPUs and many-core GPUs. Our contributions are three folds: 1) we port the generalized reduction model on clusters of modern GPUs with a map-reduce-like API, dealing with very large datasets; 2) we further propose three schemes to better utilize the computing power of CPUs and/or GPUs and develop an auto-tuning strategy to achieve the best-possible heterogeneous configuration for iterative applications; 3) we provide several useful insights into programmability and performance of CPU-GPU clusters by highlighting application-specific optimizations.
We evaluate our system using three representative data-intensive applications and report results on a heterogeneous cluster of 128 CPU cores and 16 GPUs (7168 GPU cores). We show an average speedup of $87\times$ on this cluster over execution with 2 CPU-cores. Our applications also achieve an improvement of 25% by using CPU cores and GPUs simultaneously, over the best performance achieved from using only one of the types of resources in the cluster.

5.1 System Design and Implementation

In this section, we discuss the design and implementation of the MATE-CG system.

5.1.1 Detailed System API

The goal of the API is to enable the specification of the reduction functions and to support efficient execution on a heterogeneous cluster. To this end, some additional information is required from the application implementor. Particularly, we need to be able to partition the data for processing among the nodes, further divide them among the CPU cores and the GPU, and then generate communication between the host and the GPU, as well from the GPU to the host.

The API for the programmers is summarized in Table 5.1. $R$ means that the functions must be defined by the user and $O$ refers to the optional functions where default implementations are provided. Other functionalities like system initialization, data loading, and task scheduling, are taken care of by the runtime. The runtime declares two new data types, named \textit{input\_space} and \textit{reduction\_object}. The \textit{input\_space} contains the input data, organized in blocks or chunks while the \textit{reduction\_object} represents the output data, which
is a two-dimensional array in the form of \((key, set of values)\). The specification of the reduction functions builds on top of these data types. Moreover, these data type declarations help generate data movement functions.

Two of the most important user-defined functions are the \texttt{cpu\_reduction} and \texttt{gpu\_reduction}. The \texttt{cpu\_reduction} processes data in the \texttt{input space} assigned to the CPU and updates the \texttt{reduction object} in the CPU host memory. The \texttt{gpu\_reduction} contains the CUDA kernels to perform similar computation on GPUs. Since the GPU card has its own device memory, we need to generate a set of data-copying functions to move data in the \texttt{input space} and

<table>
<thead>
<tr>
<th>Data Type Declarations</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{typedef struct {size, length; void * data; } input_space_t;}</td>
</tr>
<tr>
<td>\texttt{typedef struct {size, length; void * *reduction_array; } reduction_object_t;}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Functions Defined by the User</th>
<th>R/O</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{void (<em>cpu_reduction_f)(input_space_t</em>, reduction_object_t*)}</td>
<td>R</td>
</tr>
<tr>
<td>The \texttt{cpu_reduction} function. CPU workers execute this function on the input and update the host reduction object.</td>
<td></td>
</tr>
<tr>
<td>\texttt{void (<em>gpu_reduction_f)(input_space_t</em>)}</td>
<td>R</td>
</tr>
<tr>
<td>The \texttt{gpu_reduction} function. GPU executes the CUDA kernel functions on the input in the device memory.</td>
<td></td>
</tr>
<tr>
<td>\texttt{void (<em>gpu_update_host_f)(reduction_object_t</em>)}</td>
<td>R</td>
</tr>
<tr>
<td>The \texttt{gpu_update_host} function. Update the host reduction object based on results copied back from GPU.</td>
<td></td>
</tr>
<tr>
<td>\texttt{void (*combination_f)(reduction_object_t **)}</td>
<td>O</td>
</tr>
<tr>
<td>The combination function. Combine partial results from all computing nodes to form a final copy of reduction object.</td>
<td></td>
</tr>
<tr>
<td>\texttt{void (<em>process_next_iteration)(void</em>)}</td>
<td>O</td>
</tr>
<tr>
<td>Notifies the runtime that the computation is proceeding to next iteration.</td>
<td></td>
</tr>
<tr>
<td>\texttt{int (*partitioner_f)(char * input_data, char * meta_data)}</td>
<td>O</td>
</tr>
<tr>
<td>The data partitioning function. Partitions the input datasets among computing nodes.</td>
<td></td>
</tr>
<tr>
<td>\texttt{int (<em>splitter_f)(input_space_t</em> blocks, int req_units, input_space_t* chunks)}</td>
<td>O</td>
</tr>
<tr>
<td>The data splitting function. Splits data blocks in memory into data chunks among cpu/gpu reduction workers.</td>
<td></td>
</tr>
</tbody>
</table>
the *reduction object* between the CPU host memory and the GPU device memory. After the GPU computation and data movement are done, *gpu_update_host* is invoked to perform updates of the reduction object in the CPU host memory based on the results copied back from the GPU.

Besides, the programmers can also define the application-specific *partitioner* function and *splitter* function to perform their own data partitioning. The *partitioner* function is used to partition the disk-resident input datasets among all computing nodes. The *splitter* function is used to split the data blocks into smaller chunks and provide them for processing to the CPUs and the GPUs, respectively. Note that, to maintain the computation state, a final copy of the reduction object will be stored after each iteration and can be retrieved in the future.

To communicate between the user code and the runtime, the MATE-CG system defines a data structure named *scheduler_args*. The basic fields in this structure should be set appropriately in the user code since they pass necessary and specific information to the runtime, such as the paths to store the input data and the function pointers to user-defined operations. The optional fields are used for scheduling computation on CPUs and GPUs and also tuning the performance. For example, the partitioning parameter controls the percentage of data to be processed on CPUs and GPUs respectively.

### 5.1.2 Runtime System

Figure 5.1 gives an overview of the execution work-flow for a typical iterative application using MATE-CG in a heterogeneous computing environment. The runtime is responsible for partitioning the input datasets among computing nodes, further assigning data
chunks to CPUs and GPUs, and scheduling the tasks for all stages. Note that the advantage of GPU over CPU is primarily computation, so the CPU handles the data partitioning, task scheduling and other data management tasks, and the GPU is mainly responsible for accelerating computation.

The runtime system invokes the user-defined functions in different phases and these functions act as the interaction between the runtime and the user code. The most important phase is the Reduction phase, where all the data instances are processed and the reduction
objects are updated. After the Reduction, the combination function will merge results from multiple nodes and form a final copy of the reduction object. The interprocess communication is automatically invoked by the runtime system. The user can choose to operate on the final reduction object in the finalize function to produce outputs on disk, which could possibly be utilized for next iteration. This processing style can continue for multiple passes until the application finishes.

One interesting aspect of MATE-CG is the support of multi-level data partitioning for the workload distribution. This enables processing of very large datasets in a distributed heterogeneous environment. First, given the disk-resident input datasets, the runtime system will invoke a partitioning function to partition the inputs into a number of data blocks. A default partitioner function is provided by the runtime, but the user can also specify a customized partitioner function. Then, as one of the scheduling tasks, the input blocks are distributed and assigned to different nodes. Data locality is considered so that each node can read most of the input blocks assigned to it from its own local disk. Customized partitioning functions enable better locality while updating a disk-resident reduction object. It could also help to reduce network communication of possible large-sized intermediate data generated in some applications.

After each node has its own set of data blocks, it can load one or more blocks from disk into memory. According to the partitioning parameter, each data block in the main memory is cut into two parts, one for the CPU and the other for the GPU. Furthermore, for the CPU or GPU, its part of each data block is split into smaller data chunks and these data chunks are dynamically retrieved by the CPU or GPU. In this way, the CPU and GPU can work on different data chunks concurrently to support the heterogeneous computing. The further splitting is needed to support better load balancing for the CPU and address the device
memory limitation on the GPU. Also, studies have shown that the CPU and GPU should have different settings for the chunk size [115, 114]. A small chunk size for the CPU could help to achieve better load balancing for its multi-core threading while a potentially large chunk size for the GPU could lower latency in data movement between the host memory and the device memory.

To fully utilize the potential in heterogeneous systems, the work-load should be distributed to different processing elements carefully and this data mapping can be done either dynamically [94, 115, 114], or statically [139, 92, 66]. The dynamic way automates the mapping process and saves coding labor but introduces extra overheads and the static way achieves better performance but requires searching efforts for the best-possible configuration. The work distribution method used in our system involves a new auto-tuning approach that helps to achieve the best-possible configuration automatically for iterative applications. This is discussed in the next section.

Management of reduction objects, including the possibility that they can be disk-resident, is transparent to application developers. Thus, while the large-sized reduction objects are allocated on local disks, the user can write applications manipulating them as if they are in memory. Updates to the disk-resident locations are handled by the runtime system. Especially, the data access patterns observed in many applications are used to improved the disk I/O performance for updating the reduction objects. Note that these optimizations are done in the runtime, and thus invisible to the users.

5.2 System-Tuning for Optimizing Runtime Execution

In this section, we show how applications are optimized to help improve runtime execution. In particular, we studied three important parameters to be tuned. One setting is
the CPU-GPU data distribution fraction, which is auto-tuned based on an analytical model. Two others are the chunk sizes for the CPUs and the GPUs, respectively. While the values of these two parameters can be supplied by the application developer, our system provides default values based on analytical models and off-line training.

5.2.1 Auto-Tuning for Optimizing Heterogeneous Execution

One of the features of our runtime support is auto-tuning for optimizing a data-intensive application for heterogeneous execution. The auto-tuning parameter which needs to be tuned for performance is the fraction of the data that has to be processed on the CPU’s cores (with the remaining data to be processed by the GPU). Because of the difference in the capabilities of CPU cores and GPUs, the optimal value of this parameter is dependent upon the nature of the computations performed by an application.

Our overall approach is based on developing an analytical model of the performance of the application with the use of heterogeneous resources. We focus on iterative data intensive applications, which perform very similar computation over a number of iterations. For such applications, the values of the key terms needed by our model can be learnt over the first few iterations. Then, the optimal value of the parameter based on this model can be obtained. The advantage of our approach is that no compile-time search or tuning is needed. Moreover, the only runtime overhead is the non-optimal execution of the first few iterations.

The analytical model we use is as follows. Let $T_{cg}$ represent the overall execution time on each node with the use of heterogeneous resources. The parameter to be tuned, $p$, is the fraction of the input data (for this node) that has to be processed by the CPU cores. We formulate an equation to relate $T_{cg}$ and $p$. As Figure 5.1 shows, on each node, the
overall running time for each iteration consists of two main components. The first is the
data processing time incurred on the CPU and/or the GPU. Its value varies with the choice
of the value of the parameter $p$. The second term is the overhead incurred on the CPU,
such as input data loading time and the combination/finalize time, and is independent of
the value of $p$.

Let $T_o$ represent these fixed times on the CPU. If we use $T_c$ to denote the overall running
time on each node when only CPUs are used for data processing, we have the following
expression:

$$T_c = T'_c + T_o$$

where $T'_c$ is the data processing time on the CPU.

Similarly, if we use $T_g$ to denote the overall running time on each node when only GPUs
are used for data processing, we have:

$$T_g = T'_g + T_o$$

where $T'_g$ is the data copying and processing times on the GPU.

Therefore, when both CPUs and GPUs are enabled to process different portions of data
chunks, $T_{cg}$ for each node can be represented as:

$$T_{cg} = \text{MAX}(T'_c, T'_g) + T_o$$

(5.1)

where $T'_c$ is the CPU’s time for processing the fraction $p$ of all data, and $T'_g$ is the GPU’s
data movement and processing time for the remaining fraction of the data.

Then, for a particular value of $p$, we can have the following two expressions to calculate
$T'_c$ and $T'_g$, respectively, based on $T_c$ and $T_g$ for each node:

$$T'_c = p \times T_c$$

(5.2)
Figure 5.2: An Illustration of Expression (5.1): $T_{cg}$ is a function of $p$

and

$$T'_{g,o} = (1 - p) \times (T_{g,o} - T_{g,o}) + T_{g,o} \quad (5.3)$$

where $T_{g,o}$ represents the time for moving data structures between the CPU and the GPU.

Note that Expressions (5.2) and (5.3) can be applied here if the applications exhibit good data-parallelism and there is no data-skew. For other applications, either more sophisticated modeling or off-line training may be needed.

Given Expressions (5.1), (5.2), and (5.3), we can establish a relationship between $T_{cg}$ and $T_p$ as illustrated through Figure 5.2. Here, the dashed-dotted-solid line represents Expression (5.2) and the solid-dotted line represents Expression (5.3). Expression (5.1) is shown as the solid curve.
We can see that the minimal value of $T_{cg}$ is reached at the intersection of two lines. In other words, the optimal $p$ can be calculated as follows:

$$p \times T_{cp} + T_o = (1 - p) \times (T_{gp} - T_{go}) + T_{go} + T_o$$

$$p = \frac{T_{gp}}{T_{gp} + T_{cp} - T_{go}}$$

(4)

As we stated above, for applications that need a number of iterations of very similar computation to finish, we can calculate the optimal predicted $p$ in the first few iterations. In practice, we use the following simple heuristic. We run the first iteration using CPUs only (i.e., $p$ is set to be 1 for each node) and obtain the value for $T_{cp}$. In the second iteration, we only use the GPUs instead (i.e., $p$ is set to be 0 for each node), to obtain the values for both $T_{gp}$ and $T_{go}$. Then, in the third iteration, we can calculate an initial $p$ based on Expression (4). We execute the third iteration with this value of $p$. For later iterations, $p$ could be further adjusted based on the results attained from the previous iteration, since in practice there can be some variation in the measured values. For applications that do not have such iterative structure, off-line training is needed.

5.2.2 Determining the Settings for CPU/GPU Chunk Sizes

As discussed in Section 5.1.2, in the runtime, after the CPU and the GPU obtain their part of a data block in the host memory based on the data distribution, a further splitting is needed to support the load balancing for the CPU cores and lower the latency for the data transfer for the GPU. More importantly, the CPU and the GPU should have different settings for their chunk sizes. Intuitively, a small chunk size for the CPU could achieve better load balancing among the multiple CPU threads, while a large chunk size for the GPU could minimize the impact of the data transfer latency on the execution time.
We now describe how we can use analytical models and/or off-line training to study how to set a proper chunk size to achieve the best-possible execution time for the CPU and the GPU respectively. First, on the CPU, the data chunks in a data block are assigned to the CPU threads in a dynamic way. To process a data chunk, each CPU thread will need to invoke an internal splitter function where the next available chunk is computed and assigned to the CPU thread. Note that the splitter function is shared by all CPU threads and protected by a mutex. At runtime, each CPU thread needs to lock the splitter mutex first, then invoke the splitter function, and finally unlock the mutex before moving on to the computations. In this way, all CPU threads can work on the retrieved chunks simultaneously. This locking/unlocking process is required before processing a data chunk. Thus, besides computations on the data chunk, there are overheads associated with the locking/unlocking process. These overheads, in turn, are related to the number of chunks to be processed by each CPU thread. Clearly, a extremely small chunk size will imply better load balancing but will result in more locking/unlocking overheads. Therefore, we need to achieve a trade-off between load balancing and locking/unlocking overheads.

Due to the dynamic splitting, the data chunks cannot be fully equally distributed to all the CPU threads in an ideal way. Since each CPU core in a multi-core CPU has the same processing power, the load imbalance $l$ in a given dynamic splitting for a CPU data block can be represented by the ratio of the largest number of data chunks for a CPU thread over the average number of chunks among all CPU threads. Formally,

$$ l = \frac{n \times \max_{i=1}^{n} C_i}{\sum_{i=1}^{n} C_i} \quad (l \geq 1) $$

where $n$ is the number of CPU threads and $C_i$ refers to the number of chunks assigned to the $i$th CPU thread.
Let $b_{cpu}$ be the CPU block size (the number of elements in a CPU data block) and $c_{cpu}$ be the CPU chunk size (the number of elements in a CPU data chunk), the overall data processing time $T_{cpu}$ for a data block on the CPU can be defined as follows:

$$T_{cpu} = \left( \max_{i=1}^{n} C_i \right) \times \left( c_{cpu} \times T_{c,comp} + T_{locking} \right)$$

where $T_{c,comp}$ represents the average data computation time per element in a data chunk for a CPU thread, and $T_{locking}$ represents the average overheads associated with the locking/unlocking process during retrieving a data chunk. Thus, we have:

$$\sum_{i=1}^{n} C_i = \left\lceil \frac{b_{cpu}}{c_{cpu}} \right\rceil$$

Based on the above three expressions, we can have the following expression:

$$T_{cpu} = \frac{b_{cpu}}{n} \times \left( l \times \left\lceil \frac{b_{cpu}}{c_{cpu}} \right\rceil \times (c_{cpu} \times T_{c,comp} + T_{locking}) \right)$$

$$= \frac{b_{cpu}}{n} \times (l \times T_{c,comp} + \frac{l}{c_{cpu}} \times T_{locking}) \quad (5)$$

Clearly, in Expression (5), given a block size and a number of CPU threads, the execution time $T_{cpu}$ is mainly impacted by the load imbalance $l$ and the CPU chunk size $c_{cpu}$. To further understand how the CPU chunk size affects the load imbalance, we have obtained a set of empirical results using off-line training.

Figure 5.3 illustrates the overall trend for the average load imbalance ratio with respect to the ratio between the CPU chunk size and the CPU block size. As comparison, we include the load imbalance ratio for an ideal splitting where each CPU thread will be assigned exactly the same number of data chunks. From this, we can make the following observations: given a number of CPU threads $n$ and a block size $b$, when the chunk size is relatively large, e.g., the same as the block size, the load imbalance ratio is quite high. When the chunk size becomes smaller and reaches $b/n$, the average load imbalance ratio
Figure 5.3: The Avg. Load Imbalance Ratio with Varying the CPU Chunk Size: $n$ is the number of CPU threads and $b$ is the CPU block size ($b_{cpu}$).

drops significantly close to 1. Overall, Figure 5.3 shows that the off-line training can be used to construct the relationship between the load imbalance and the CPU chunk size.

Therefore, based on Expression (5) and the off-line training, we can summarize an approach to set the CPU chunk size. To find the optimal CPU chunk size so as to minimize $T_{cpu}$ in Expression (5), we could apply a heuristic search in all possible CPU chunk sizes. This search process, together with the required off-line training, could introduce extra overheads to our runtime system. Therefore, currently, the CPU chunk size in our system is set to be $b_{cpu}/(16 \times n)$ by default. A more aggressive optimization module could still perform a search, and alternatively, an experienced application developer may supply a different value based on their experience.

As we stated earlier, the GPU chunk size is another factor that can be important for the system performance. On the GPU, since the data movement overheads are the dominant consideration, we can use the following expression to calculate the data processing time.
where \( b_{gpu} \) is the GPU block size (the number of elements in a GPU data block), \( c_{gpu} \) is the GPU chunk size (the number of elements in a GPU data chunk), \( T_s \) is the average start-up time for the data transfer, \( T_w \) is the average data copying time per element in a GPU data chunk, and \( T_{g\text{ comp}} \) is the average GPU computational time per element in a GPU data chunk.

From Expression (6), it is easy to see that the GPU chunk size should be set to be as large as the GPU block size if the GPU device memory has enough space to store a data block. In this case, the start-up costs for data movement can be minimized. In our system, the GPU chunk size is set to be the same as the GPU block size by default. In some cases during the runtime, when the GPU device is short of memory, the GPU chunk size would be gradually decreased to fit into the available GPU device memory.

### 5.3 Experimental Results

In this section, we first introduce the three applications we focus on and then report results from a detailed evaluation of the MATE-CG system. Every node on the cluster we used comprises of two Intel Xeon E5520 quad-core CPUs (8 cores in all), connected with a NVIDIA Tesla C2050 (Fermi) GPU with 448 cores in all (14 streaming multiprocessors each with 32 cores). Each CPU core has a clock frequency of 2.27 GHz and the machine has a 48 GB main memory. The GPU has a clock frequency of 1.15 GHz each core and a 2.68 GB device memory. We have used up to 128 CPU cores and 7168 GPU cores (on 16 nodes) for our study.
Our evaluation was based on the three data-intensive applications we described in the previous section. For each application, we run it in four modes in the cluster:

1. the CPU-1 mode (i.e., use only 1 CPU core per node as the baseline)
2. the CPU-8 mode (i.e., use all 8 CPU cores per node)
3. the GPU-only mode (i.e., use only the GPU per node)
4. the CPU-8-n-GPU mode (i.e., use both 8 CPU cores and GPU per node)

In our experiments, we focused on the ensuing aspects:

- We first evaluate the performance of the three applications by scaling each application with the number of GPUs (i.e., the number of nodes) used in the cluster. As
performance comparison, we run each application in the first three configurations (CPU-1, CPU-8, and GPU-only) as we scale the number of nodes.

- We study the performance impact of varying $p$ in CPU-8-n-GPU and show its performance improvement over CPU-8 and GPU-only. We also evaluate the auto-tuning strategy we proposed in Section 5.2.1 and show its effectiveness for iterative applications.

- We further show the performance impact of setting different CPU/GPU chunk sizes for particular applications, to validate the analytical models in Section 5.2.2 and also use them as examples of tuning important system parameters towards automatic optimizations in the future.
Figure 5.6: EM: Comparison between CPU-1, CPU-8, and GPU-only on 2, 4, 8, and 16 nodes for a 32 GB dataset

The datasets used in our performance studies are as follows: For PageRank, we chose a graph with 1 billion nodes and 4 billion edges, represented by an adjacency matrix of 64 GB and an input vector of 8 GB. The output vector represented by the reduction object is also of 8 GB. For the gridding kernel, we used a collection of around 800 million visibilities, represented by a dataset of 32 GB and the output grid was set to be 6.4 GB. For EM, we used a dataset of 32 GB, containing 1 billion points to be clustered. Note that the 48 GB main memory on each node needs to be shared by input/output buffers, reduction object(s), and other application and kernel memory requirements. Thus, for PageRank and the gridding kernel, we had to put the reduction objects on disk since it is clearly not possible to have a private allocated copy of the reduction object for each worker thread. For EM, since the reduction object is relatively small, it could be maintained in the main memory.
Also, for the use of GPUs, we did not apply data pipelining to try and overlap data transfers and computations on the GPUs. The reason is that pipelining works best (≤50% improvement) in the scenario where the data transfer and computational times are close to each other. For all three applications in this work, however, either the data transfer times are much more than the computations times (10-20 times for PageRank), or the computations are much more dominant than the data movements (70-100 times for EM and Gridding Kernel). Thus, pipelining was not used for these applications.

5.3.1 Scalability Studies

We first run the applications in the first three modes: CPU-1, CPU-8, and GPU-only. Figures 5.4 through 5.6 show the comparison results for the three applications as we scale the number of nodes.
As we can see in Figure 5.4, for PageRank, GPU-only version outperforms both CPU-1 version and CPU-8 version on 2, 4, 8, and 16 nodes. For example, on 16 nodes, GPU-only version is 5 times faster than CPU-1 version and further improves CPU-8 version by 16%. This is because the nature of processing in this application is well suited for acceleration using a GPU. Also, we can see that the MATE-CG system shows good scalability as we increase the number of nodes. The relative speedup of the GPU-only version is 6.3 when the number of GPUs is increased from 2 to 16. The system also has relative speedups of 7.0 and 6.8 for CPU-1 and CPU-8 versions, respectively, from 2 nodes to 16 nodes.

We have similar observations from Figure 5.5 (for the gridding kernel) and Figure 5.6 (for EM). For the gridding kernel, on 16 nodes, GPU-only version is around 6.5 times as fast as CPU-1 version and outperforms CPU-8 version by 25%. The relative speedups are 7.5, 7.2, and 6.9 for CPU-1, CPU-8, and GPU-only versions, respectively, as we increase the number of nodes from 2 to 16. For EM, GPU-only version is 15 times as fast as CPU-1 version.
version and 3 times faster than CPU-8 version with 16 nodes. The improvements using GPUs in EM are larger than those in other two applications because we could put some of small-sized reusable data structures, such as the vector of parameters, into the shared memory on GPUs in the EM kernels. Also, EM achieves relative speedups of 7.6, 6.8, and 5.0 for CPU-1, CPU-8, and GPU-only versions, respectively, from 2 nodes to 16 nodes.

5.3.2 Auto-Tuning and Performance with Heterogeneous Computing

We now focus on the CPU-8-n-GPU mode, which involves acceleration using a heterogeneous configuration, i.e., using both the multi-core CPUs and the GPUs. For PageRank and EM, which are both iterative applications, we show both how our auto-tuning method
Figure 5.10: Gridding Kernel: Varying CPU_Chunk_Size for CPU-8 and GPU_Chunk_Size for GPU-only on 16 nodes

is effective and the final speedups obtained from other (homogeneous) modes of execution. For the gridding kernel, which is a single pass algorithm, we demonstrate just the performance obtained with different values of the CPU processing parameter.

In our experiments, we executed PageRank and EM for a number of iterations while enabling the auto-tuning framework. As we can see from Figures 5.7 and 5.8, the best-possible choice of the CPU processing fraction could be obtained through the auto-tuning method. Figure 5.7 shows that the running time of PageRank in one iteration decreased and became quite stable starting from the fifth iteration. For EM in Figure 5.8, the running time for both the E-step and the M-step became stable after three iterations. PageRank took more iterations to become stable because the first calculated initial $p$ was not close enough to its best-possible value and needed further adjustments. This is due to some other
overheads in PageRank that are not considered in Expression (4). Also, the best-possible CPU-8-n-GPU version for PageRank further improves the better of its GPU-8 and GPU-only versions by around 7% with the \( p \) tuned to be 0.30 and the improvements for EM are around 29% with a \( p \) of 0.31 for the E-step and around 24% with a \( p \) of 0.27 for the M-step. Moreover, we can see that if the application is executed for a large number of iterations, the runtime overhead of auto-tuning will be quite small.

Finally, for the gridding kernel, Figure 5.9 shows the performance curve as we vary \( p \) (the percentage of data assigned to CPUs) in the CPU-8-n-GPU mode, when we run the application on 16 nodes. Compared with CPU-8 version and GPU-only version, we observe that the CPU-8-n-GPU version could further reduce the total running time by appropriately choosing \( p \). As in Figure 5.9, the best-possible \( p \) should be able to improve CPU-8 version by more than 56% and GPU-only version by more than 42%. It is easy to see that the performance curve in Figure 5.9 matches the trends shown in Figure 5.2.

### 5.3.3 System Tuning with Setting CPU/GPU Chunk Sizes

The last set of experiments are about the performance impact of varying the `cpu_chunk_size` and `gpu_chunk_size`. First, we would like to demonstrate that the experimental results are consistent with the rule-based settings about how to set the CPU/GPU chunk sizes as we discussed in Section 5.2.2. Second, these can be viewed as examples of system tuning that could be fully automated in the future. It would be quite promising to provide automatic optimizations in terms of a set of system configurations that are important to the overall performance.

Typically, we observe that a smaller chunk size for the CPU achieves better load balancing among multiple threads and a larger chunk size for the GPU improves data copying
efficiency. So, we first executed the gridding kernel with the CPU-8 and GPU-only modes, and varied the CPU/GPU chunk size between 4 MB and 512 MB, while keeping a block size of 512MB. As Figure 5.10 shows, the trends are consistent with the above observation. In particular, the 16 MB CPU chunk size obtains the best trade-off between data splitting (locking/unlocking) overheads and task load balancing. Note that since the block size is 512 MB, a chunk size of 512MB for CPU would actually cause only one thread performing the computation and thus degrade the performance significantly. For GPUs, the largest possible chunk size of 512 MB achieves the best running time as long as it can fit in the available device memory. Note that smaller GPU chunk sizes do not increase the total running time by a significant margin, because the gridding kernel is compute-intensive.

We had similar observations for the two steps in the EM algorithm; and for PageRank, where the ratio of data movement times is higher, we found out that larger chunk sizes appeared to provide much better results than smaller chunk sizes. The detailed results are not shown here because of space constraints.

5.4 Summary

This work has described the MATE-CG system, which supports a map-reduce-like API on a heterogeneous environment of CPUs and GPUs. One of the key novel contributions of this work is a novel auto-tuning approach for choosing the fraction of the data that should be processed by CPU cores.

We have evaluated our system using three representative data-intensive applications in three domains, which are PageRank from graph mining, EM from data mining, and the gridding kernel in Convolutional Resampling from scientific computing. We used up to 128 CPU cores and 7168 GPU cores for our performance studies in a heterogeneous cluster.
We shown scalability with increasing number of nodes and cores, as well as with combined use of CPU cores and GPUs. Our auto-tuning approach has been effective in choosing the appropriate fraction within a few iterations. We also studied how the CPU/GPU chunk sizes should be set appropriately to achieve the best-possible performance. We derived rules for choosing these important parameters and demonstrated that the CPU chunk size we choose leads to good performance.

In the future, we would like to introduce a more diverse set of applications including I/O-bound applications, data-skew work-flows, and applications suitable for data-pipelining on GPUs, and explore new methods for improving their performance. The current auto-tuning model will also be enhanced to address the challenges associated with the applications that have a data skew, as well as from the use of the data-pipelining technique on GPUs.
Chapter 6: FT-MATE: Supporting Efficient Fault Tolerance for MPI Programs in MATE Systems

It is widely accepted that the existing MPI-based fault-tolerance solutions will not be applicable in the exascale era, as with growing level of concurrency and relatively lower I/O bandwidths, the time required to complete a check-point can exceed the Mean-Time To Failure (MTTF). Moreover, with energy consumption being the dominant consideration for exascale systems, and with data movement (including data movement for checkpoints) being one of the major energy consumers, novel solutions are clearly needed.

In one of our earlier works [26], we have shown that the reduction object model can achieve fault tolerance with lower overheads than Hadoop for data mining and graph mining applications. Therefore, in this work, we are trying to extend our ideas to support more efficient fault tolerance for MPI programs. Specifically, on top of the MATE systems developed in earlier chapters, we show that designing a programming model that explicitly and succinctly exposes an application’s underlying communication pattern can greatly simplify fault-tolerance support, resulting in at least an order of magnitude reduction in checkpointing overheads over the current MPI-based solutions.

The communication patterns we consider are similar to the notion of dwarfs in the Berkeley view on parallel processing. We show the design of a programming system that explicitly exposes these patterns, and based on this information, is able to checkpoint the
state. Our approach can tolerate both permanent node failures, where the failed node is excluded for the rest of the processing, as well as temporary node failures, where the failed node can recover and join the execution again. We have evaluated our approach using four applications, two each from the classes of dense grid computations and irregular reductions. Our results show that the checkpointing overheads of our approach are very low, and when a failure occurs, the failure recovery is very efficient.

6.1 Fault Tolerance Approaches

In this section, we first introduce the popular fault-tolerance methods that are currently used. We then show how an alternate fault tolerance approach can be designed based on a high-level (BSP-like) programming model, and how applications of different types can be implemented using this model.

6.1.1 Checkpointing

The de facto practice of fault tolerance involves checkpoint and restart (C/R) [108, 55, 56, 67]. Checkpointing supports a rollback-based recovery. A program periodically outputs process and system state to persistent storage, and if there is a failure, a program can restart from the most recent point where the checkpoint was taken. If redundant nodes and fault prediction abilities are available, job/process migration [117, 36, 137], which is a pro-active fault tolerance mechanism, has also been proposed as a complement to C/R.

For message-passing distributed environments, there are two main families of checkpointing protocols, which are coordinated and uncoordinated checkpointing [55, 67]. Coordinated checkpointing is the most popular approach in HPC systems. Here, processes must coordinate to produce a consistent global system state, which consists of states of all processes and channels among the processes. If a failure occurs, all processes just rollback
to their last checkpoint and restart from there. It has the advantage of supporting simple recovery and efficient garbage collection, but the synchronization time for creating a consistent global system state and writing their checkpoints to the persistent storage could be expensive. It also causes bursty writes on the I/O system, which adversely impact the application performance.

Uncoordinated checkpointing can be used to address the synchronization problem by allowing asynchronous checkpointing. Uncoordinated checkpointing is usually combined with message logging to limit the number of processes to rollback in the event of a failure. Various message logging protocols have been proposed with different associated overheads of logging application messages [19].

Most MPI-based scientific programs are suitable for synchronous checkpointing because it is simpler and it fits the MPI’s SPMD (Single-Program Multiple-Data) execution mode. As a result, almost all popular MPI implementations now support fault tolerance using the coordinated checkpointing protocol in their library stacks, including OpenMPI [12], MPICH2 [7], MVAPICH [8], AMPI [1], and MPICH-V [28], among others. Most of them are based on OS-level checkpoint libraries such as BLCR [2] and Condor [4], except for AMPI, which is based on the Charm++ runtime system [3].

6.1.2 Our Approach: High-level Ideas

The approach we are proposing is based on the following observation. MPI is a very general programming model, and when checkpointing is performed at the level of MPI, the properties of the application are not taken into account. Most applications, on the other hand, have an iterative structure and loop with certain types of communication. If the
programming model explicitly captures the nature of the application, checkpointing can be much more efficient and recovery can be more flexible.

The communication patterns we explicitly capture are along the lines of dwarfs in the Berkeley view on parallel processing \(^1\). Thus, our premise is that if the dwarf an application belongs to is exposed explicitly, both programmer productivity and fault-tolerance support can be improved. While we believe that this approach can be used for all (or almost all) dwarfs, our current work is limited to handling only three dwarfs. They are, the applications involving generalized reductions, the applications involving a dense grid (i.e., stencil computations), and applications involving irregular meshes. Our earlier work has already shown that specialized and efficient fault-tolerance support can be enabled in a system that only supports generalized reductions \([26]\). Thus, the contributions of this work are in applying this idea to stencil computations and irregular reductions. We show how a high-level programming construct, which we refer to as the extended generalized reduction model, can support development of these classes of applications in a scalable fashion, and allow much more efficient support for resilience.

### 6.1.3 The Extended Generalized Reduction (EGR) Model

The original generalized reduction model exposes two functions that must be provided by the application developer. 1) **Reduction**: The reduction function specifies how, after processing one data instance, a reduction object (initially declared by the programmer) is updated. The result of this processing must be independent of the order in which data instances are processed on each processor. 2) **Combination**: In this function, the final results from multiple copies of a reduction object are combined into a single reduction object. A user can choose from one of the several common combination functions already...
implemented in the system library (such as aggregation, concatenation, etc.), or they can provide one of their own.

Note that the function, Reduction(), is an associative and commutative function since the data instances can be processed in any order. Overall, the reduction function can be viewed as an integration of map/combine/reduce in the MapReduce model, eliminates the sorting/grouping/shuffling overheads, and significantly reduces memory requirements for the intermediate data [79, 80].

As we can see, in the generalized reduction model, the reduction objects act as the output space, and can capture the state of the computation. This observation can be used to support a very efficient checkpoint [26].

The notion of output space and an explicit communication phase is seen in many other programming models, such as the Bulk-Synchronous Parallel (BSP) model implementations [133, 65]. Thus, the natural question is, can we use it for applications that involve other types of communication patterns. It turns out that with a small extension, other applications can also be supported.

Consider grid computations. The limitation of the simple reduction based model is that it will require large-sized reduction object. Thus, to reduce a large-sized reduction object, the combination phase will certainly become a bottleneck for scalable implementations. To address this issue for grid applications, we extend the original generalized reduction model by further dividing the reduction object into two disjoint parts. The first part represents the part of the output space that could be accessed/updated by more than one node, and is thus referred to as the global reduction object. The second part represents the remaining part of the output space that only needs to be accessed/updated by its host node, referred to as the local reduction objects. Typically, all the input blocks assigned to one node share a
**Dense Grid Computations** \((A, C_0, B)\)

<table>
<thead>
<tr>
<th>Input: Matrix A and coefficiency (C_0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output: Matrix B</td>
</tr>
</tbody>
</table>

**for each** non-boundary position \((i, j) \in \text{Matrix } B\) **do**

```plaintext
\[
\begin{align*}
\text{center} &= A(i, j) \\
\text{north} &= A(i - 1, j) \\
\text{south} &= A(i + 1, j) \\
\text{west} &= A(i, j - 1) \\
\text{east} &= A(i, j + 1)
\end{align*}
\]
```

\((i, j) = C_0 \ast (\text{center} + \text{north} + \text{south} + \text{west} + \text{east})\)

---

Figure 6.1: PseudoCode: Dense Grid Computations

global reduction object and each input block has a local reduction object. In other words, the global reduction object must participate in the global combination phase scheduled by the runtime, while the local reduction objects do not need to. For a specific application, one could involve only the global reduction object, or only a set of local reduction objects, or both of them.

### 6.1.4 Application Examples

Now, we will show how this two-level reduction object concept can result in efficient and scalable implementations for both dense and sparse grid applications.

**Dense Grid Computations:** Stencil kernels such as Jacobi and Sobel Filter are instances that represent *structured* or *dense* grid computations. This class of applications is mostly implemented using an iterative finite-difference technique, over a spatially dense grid, and involves performing some pattern of *nearest neighbor computations*. The input problem and the output are represented in the form of matrices, where each point in the matrix
is updated with weighted contributions from its neighbors. A simple example of a two-
dimensional, 5-point stencil kernel with uniform weight contribution from its neighbors is
shown as Pseudo-Code 6.1.

Using the extended generalized reduction API, there are two ways to implement stencil
computations. A simple mechanism for parallelizing this application is based on the output
matrix partitioning. Suppose the input and output matrices are partitioned along the rows
first. Now, the elements that reside on the border of two neighboring input partitions need
to be communicated/shared among a pair of output partitions. In this case, output partition
$i - 1$ and output partition $i$, output partition $i$ and output partition $i + 1$, and so on are the
communicating/sharing pairs of output partitions, respectively.

Alternatively, the stencil computations can be rewritten in a data-driven processing
structure, i.e., for each point in the input matrix, we determine the corresponding points to
be updated in the output matrix. Based on the input space partitioning, two neighboring
input partitions need to update two rows shared by two corresponding output matrix parti-
tions. And we can denote all this type of boundary rows in the output matrix as ghost output
rows. Then the output matrix can be represented using the reduction objects containing a
global reduction object (ghost output rows) and a set of local reduction objects (the other
output rows). Therefore, for each input space partition, the computations involve both local updates and global updates. The local updates are applied to the local reduction object
within the input partition while the global updates are performed on the global reduction
object. Although the size of the output matrix is the same as that of the input matrix, the
global reduction object can be very small. For example, for 8 partitions, the global reduc-
tion object has only 14 ($(8 - 1) \times 2$) rows, irrespective of the total number of rows in the
Irregular Reductions $(M, IA, P)$

**Input**: Mesh $M$ and Indirection Array $IA$  
**Output**: Mesh $P$

for each edge $e \in \text{mesh } P$

- Let $p_{x}$ and $p_{y}$ be indices for the two end points of $e$ in $P$
- Let $m_{x}$ and $m_{y}$ be indices for the two end points of $e$ in $M$

  $m_{x} = IA[p_{x}]$
  $m_{y} = IA[p_{y}]$

  Update $e$ based on $M[m_{x}]$ and $M[m_{y}]$

Figure 6.2: PseudoCode: Irregular Reductions

input matrix. At the end of the computation, each input partition updates its elements based on the global reduction object and its local reduction object using `memmove`.

Our extended generalized reduction model supports both the two implementations. For the first method, each output partition is a local reduction object and there is no use of a global reduction object. For the second method, the ghost output rows are the global reduction object and the other output rows are the local reduction objects.

**Irregular Reductions**: The representative application we use for the irregular reductions dwarf is Euler Solver [100], which is based on Computational Fluid Dynamics (CFD). It takes description of the connectivity of a mesh and calculates quantities like velocities at each mesh point. A canonical irregular reduction loop is shown as Pseudo-Code 6.2. Each edge in the mesh is updated through an *indirection array* $IA$ which stores indices instead of concrete quantities.

Now, let us consider how to parallelize such a loop using the extended generalized reduction model. To consider the partitioning choice, we consider the impact of choosing
Figure 6.3: Partitioning on Reduction Space for Irregular Reductions
input space partitioning. In these applications, the contents of the indirection arrays are not known at the compile time, and thus, any element in the output space could be updated by two or more input partitions. As a result, we cannot really distinguish the global reduction object from the local reduction objects. Thus, to maintain the correctness, the input space partitioning would need to treat the entire output space as the global reduction object. Our preliminary results show that compared with the computation time, the global combination time in such an implementation would be dominant and it will result in poor scalability.

Therefore, the partitioning we perform is the reduction space (output space) partitioning as shown in Figure 6.3(a). In this case, the reduction space is partitioned into a number of disjoint parts. For each part, we organize the corresponding input partition that also contains neighboring elements from other output partitions (we refer them to as external cross-edge elements). Then we can perform reduction on each input partition to produce the corresponding update on its portion of output space. The biggest advantage is that this partitioning scheme does not need the use of potentially large-sized global reduction object, and thus, eliminates the global combination time. Each reduction space partition can be treated as a local reduction object and updated independently.

In a distributed environment, after each reduction space partition is computed, each input partition needs to update its external cross-edge elements based on the local reduction objects from its neighboring input partitions since these external cross-edge elements were updated on other output partitions. Particularly, as in Figure 6.3(b), if two neighboring input partitions are assigned to the same node, this data exchange can be done using `memmove` (e.g., Partitions 1 and 2 on Node 1). Otherwise, two partitions computed on two nodes need to communicate with each other using a set of non-blocking operations (e.g., Partition 2 on Node 1 and Partition 3 on Node 2).
6.1.5 Fault Tolerance Approach Based on EGR Model

We now describe the fault-tolerance approach that has been implemented for the applications developed using EGR model described earlier in Section 6.1.3. Like the existing fault-tolerance solutions for MPI programs, the approach described here is automatic, i.e., it requires no user code modification, or compiler analysis.

Our fault tolerance approach exploits the properties of the reduction processing structure we target. Based on EGR model, an efficient and scalable implementation for typical grid computations should involve a small-sized global reduction object and/or a set of possibly large-sized local reduction objects. Let us consider again the processing structure we support. In this model, the output values are stored in the reduction object, which could have two parts now. Each node maintains a global reduction object that is used to store global updates to this object, as well as local reduction objects that contain local updates.

Thus, we specifically target applications written with the extended generalized reduction model. Each super-step, or more commonly called, an iteration, is composed of Reduction, during which the application is doing useful work, Combination, Checkpoint, and possible Restarts. Also, though there are many types of failures in terms of both processors and applications [118, 119], in this work, we focus on tolerating fail-stop node failures only. In these failures, the failed node stops working and does not induce correlated failures of others. This type of failures is most common in today’s large-scale computing systems such as high-end clusters with thousands of nodes and computational grids with dynamic computing resources [41]. Finally, we also assume that fail-stop failures can be detected and located. This may be through the aid of a parallel programming environment such as MPI, which will be used as an underlying library for implementation of our high-level API.
To checkpoint the overall reduction object, we can deal with the global and local reduction objects in different ways. From each node, after processing a certain number of elements, the global reduction object is copied into the memory of another node. The local reduction objects, on the other hand, are copied onto the persistent storage. We choose this design for several reasons. First, the global reduction object is typically small in size and only stores partial results during the computation. Thus, saving them on the persistent storage would introduce extra overheads for global combination as well as during a possible recovery phase. Caching them into the memory of other nodes can be quite efficient, and moreover, we are able to overlap this step with computations. Second, for the local reduction objects, because of their possibly large size, caching them into other nodes could cause significant communication costs and also consume a lot of memory of other nodes. Therefore, instead of using the distributed memory, we can save them on a persistent storage with large capacities, such as a parallel file system usually supported in a HPC environment. Since each node has their local reduction objects, this checkpointing process can be done in parallel for all nodes to make use of the parallel I/O capacities provided in the persistent storage. In this way, we are trying to utilize both the aggregate memory and the parallel file system to save the overall computational state.

Explicit declaration of reduction objects and their checkpointing has an important implication, as we elaborate below. Suppose we have \( n \) nodes executing an application and a node \( i \) fails. Let \( E_i \) be the set of elements that were supposed to be processed by the node \( i \), and let \( E'_i \) be the set of elements processed before the global reduction object was last copied. Now, the set of elements \( E_i - E'_i \) can be distributed and processed by the remaining set of nodes. Thus, another node \( j \) will process a subset of the elements \( E_i - E'_i \), along with the set of elements \( E_j \) that it was originally supposed to process. The global reduction
objects after such processing on the remaining $n - 1$ nodes can be combined together with the cached global reduction object from the failed node to produce the final results of the global reduction object.

Since the reduction operation is associative and communicative, by the argument above, we know that this will create the same final results as the original processing for the global reduction object. Also, the local reduction objects updated by the failed node before failing has been checkpointed onto the persistent storage and are visible to all remaining nodes. In this way, the overall computational states on each node can be saved in the memory of other nodes and/or on the reliable storage.

Although our system design can be viewed as a checkpoint-based approach, it has significant differences. First, unlike snapshots of the checkpoint-based systems, the size of the reduction object captures the necessary computational states that are smaller for most of the reduction-based applications. Therefore, the overhead of storing the reduction object to another location is smaller than storing the snapshots of the entire system. Furthermore, we do not have to wait for the failed process to restart, or for a replacement node to be found. Instead, by using the properties of the processing structure, the work from the failed node can be re-distributed to other nodes, and the checkpointed results of the finished work on the fail node can be recovered.

Now, we comment on the relationship of our approach to algorithm based solutions and application level checkpointing. First, while our approach may not be as general as the MPI checkpointing, it is clearly more general than solution for a particular algorithm. Moreover, as long as the application is developed using our programming model, no additional effort is needed from the programmer to achieve fault-tolerance. In comparison to application level checkpointing [60, 61, 59], the following observations can be made. First, because

127
application-level checkpointing needs to handle all possible MPI features, the protocols become complicated. Unlike our approach, it is also non-trivial to apply this approach to other programming models, including intra-node multi-core and many-core programming models. Second, application-level checkpointing requires that either the programmer or a compiler chooses the points for recovery. Similarly, compiler analysis or the user needs to determine what state would need to be checkpointed. In comparison, this information is obvious with our approach, once our programming model is used. Finally, when the failed node does not recover and a replacement node may not be available, our approach allows recovery using fewer processes than the original number.

6.2 System Design and Implementation

In this section, we discuss the design and implementation details of our fault tolerance approach, and how it is integrated with a runtime system that supports the EGR model. This runtime system has been implemented as an extension to the MATE system [80, 77, 78].

6.2.1 EGR Model Implementation in MATE System

We first describe the MATE runtime system, which has been used to implement the EGR model and our fault tolerance approach. The runtime is responsible for partitioning the input datasets among computing nodes, further assigning data blocks/chunks to available processing units (e.g., CPUs, and for one particular implementation [78], also the GPUs), and scheduling the tasks for all stages. At the runtime, the system invokes the above user-defined functions in different phases. The most important phase is the Reduction phase, where all the data instances are processed and the reduction objects are updated. After the Reduction, a combination function merges results from multiple nodes and forms
a final copy of the reduction object. The interprocess communication is automatically in-
voked by the runtime system. The user can choose to operate on the final reduction object
in the finalize function to produce outputs on disk, which could possibly be utilized for
next iteration. This processing style can continue for multiple passes until the application
finishes. Our system is based on two new data types, named input_space and reduc-
tion_object. The input_space contains the input data, organized in blocks or chunks
while the reduction_object represents the output space, which is a two-dimensional array
in the form of (key, set of values). Note that the output space of one input block could have
two parts, one global reduction object and one local reduction object. But only the global
reduction object needs to be combined in the combination phase. Typically, the global re-
duction object should be of small size to provide scalable implementations of applications
due to the combination overheads among all computing nodes. For a particular application,
if the global reduction object is not used or cannot represent all the necessary computational
states, the local reduction objects can be used together to store the overall computational
states.

6.2.2 Fault Tolerant Runtime System

Pseudo-Code 6.4 gives an overview of the fault tolerance algorithm used in the MATE
system in a parallel computing environment. The original MATE runtime is responsible
for distributing the input datasets among computing nodes and parallelizing the computa-
tions. The fault tolerance component deals with tolerating and recovering failures, includ-
ing checkpointing the computational states, detecting possible node failures, re-distributing
input data blocks on failing nodes, and scheduling the recovery tasks on remaining nodes
(and the failing nodes in case of a self-repair).
**ProcessingWithFaultTolerance (Input)**

<table>
<thead>
<tr>
<th>Input: data blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output: reduction objects</td>
</tr>
</tbody>
</table>

**while not finished**

**do**

**for each** data_block ∈ Input

- Process the data_block in reduction
- Update reduction objects (global and/or local)

---

**Upon Checkpoint***

- **MATETFTSetup(...)**
- **MATEMemCheckpoint(global_reduction_object)**
- **MATEDiskCheckpoint(local_reduction_object)**

**if** **MATEDetectFailure()**

- **then** **MATERecoverFailure(...)**

Perform global combination
Finalize the output

---

**Figure 6.4: PseudoCode: Processing with Fault Tolerance**

As in Pseudo-Code 6.4, we modified the processing structure of MATE. Overall, the fault tolerance runtime system can be summarized as follows:

**Configuration:** Before computing nodes start their data blocks processing and checkpointing, certain parameters should be provided by the programmer to the fault tolerance system (though default values can be used by the system). This includes the **checkpoint interval** of the global and local reduction objects, the destination directory where the local reduction objects will be saved, the number of connection trials before the node is assigned as failed in the system, the exchange method for the global reduction objects, and the output method for the local reduction objects. The checkpoint interval determines how frequently the reduction object is checkpointed in the system. This frequency is be controlled by two system parameters, the number of iterations to run and the number of input data blocks to process in the checkpointing iteration before a checkpoint. A larger value reduces the overheads of
copying, communicating, and/or storing the reduction object at another location. However, it can also lead to more repeated work in the case of a failure.

**Checkpointing:** On each node, upon checkpoint, the global reduction object is replicated into the memory of other nodes in the same *group*. A group means a disjoint subset of all nodes. The local reduction objects are saved on the persistent storage.

**Detecting Failures:** The fault tolerance runtime invokes peer-to-peer communication among the nodes in which the global reduction object will be stored. These connections are kept alive during the execution. Whenever a node fails, connections to the failed nodes become corrupt. If the node is still not reachable by any of its peers after a specific number of connection trials, it is considered as a failed node in the system. With the aid of MPI libraries, this failure detection process is accurate and instant.

**Recovering Failures:** The computing nodes query if there is a failed node in the system before they enter the global combination phase. If a failed node is determined, then the recovery process begins. The failure recovery process has three parts: data re-distribution, output space recovery for finished data blocks on failing nodes, and processing of unfinished data blocks on failing nodes. Data locality is considered during re-assigning the data blocks in order to minimize the data retrieval times. Note that, in the following iterations after recovery, if the failed node comes back, it can join the computing nodes again and the runtime will switch to the original data distribution as if no failures occurred.

Now we further explain how our approach is integrated with the processing structure in MATE (Figure 6.4). Specifically, each retrieved data block from the input is processed by the local reduction function, and then it is accumulated into the global reduction object and/or the local reduction objects. The function *MATE_FTSetup* resets the state values in the fault tolerant system. These values include the numbers representing the processed data.
blocks and the current iteration number. The $MATE_{\text{MemCheckpoint}}$ function starts transferring the global reduction object to other nodes’ memory and the $MATE_{\text{DiskCheckpoint}}$ function invokes the programmer’s API and starts checkpointing the local reduction objects on persistent storage.

Currently, our system supports two types of exchange method for the global reduction object. The first is synchronous data exchange, which blocks the application until the fault tolerant system finishes the global reduction object exchange. The second method is asynchronous data exchange. If the programmer enables this method in the program, the communication protocol of our fault tolerant system does not block the system’s execution. More specifically, whenever MATE invokes the $MATE_{\text{MemCheckpoint}}$ function, the fault tolerant system copies the reduction object into its own context and creates a background thread. This background thread starts the reduction object exchange. However, if a thread is in progress when the $MATE_{\text{MemCheckpoint}}$ function is called, then the system is blocked until it finishes its data exchange. Since the data processing times are usually longer than the small-sized global reduction object exchange time, the overhead of caching the global reduction object becomes minimal. Our preliminary experiments demonstrated that the asynchronous method clearly outperformed the synchronous method. Thus, the asynchronous method is set to be the default method.

Also, our system supports two types of data output method for the local reduction objects. The first is single-thread data output, which invokes only one I/O thread to output the entire local reduction object of a finished input data block into a single checkpoint file on the disk. The second method is named as multi-thread data output, which invokes multiple I/O threads and each thread outputs a split of the entire finished local reduction object into a separate sub-file. Multiple I/O threads can make the best use of I/O bandwidths for the
persistent storage, and introduces some extra overheads of opening/closing more files. This could also require an additional merge process in the MATE_RecoverCheckpoint function during loading the checkpointed sub-file and could increase the recovery overheads.

Our preliminary experiments demonstrated that the multi-thread data output had better performance than the single-thread data output for medium to large data sizes. Since the local reduction object usually represents a large-sized output space for grid computations, the multi-thread data output is set to be default. Note that in the multi-thread data output method, all processing units on one node will be used and the I/O threads are blocking operations and must finish before the node moves onto processing the next data blocks. Moreover, in practice, when a program checkpoints, it cannot simply overwrite the previous checkpoint file because a failure can occur during checkpoint. So in our implementation, we write to a different file and do not delete the previous checkpoint until the current checkpoint is completed.

If a node finishes processing all of its input data blocks, then it is ready to enter the global reduction phase. Before it enters this phase, it calls MATE_DetectFailure function which is responsible for detecting the failed nodes. If there is no failure, all the nodes enter the global combination phase without any interruption. Otherwise, if a node is assigned as failed in the system, MATE_RecoverFailure function is invoked. This function initiates the recovery process. All the data blocks originally assigned to the failed node are re-distributed to the alive nodes in the system. For each of such data blocks re-assigned to the remaining nodes, the input space is first created and the next step is to check whether the data block has been processed by the failed node. If a data block has been processed and the corresponding reduction object has been cached in other nodes and/or saved to the persistent storage, there is no need to re-process such a finished data block. Instead, the
alive node can simply retrieve the checkpointed global and/or local reduction object to re-
construct the output space of that data block. Otherwise, the input space of an unprocessed
data block will be simply passed to the user-defined reduction function to obtain the cor-
responding output space, like processing other data blocks previously assigned to the alive
node.

This process continues until all the data blocks are processed. Finally, a global combi-
nation is performed using the global reduction objects from the remaining nodes, and the
cached global reduction objects from failed nodes. So after the global combination phase,
each node has a complete copy of the global reduction object as well as the saved local
reduction objects for all the input data blocks on the persistent storage.
Figure 6.5 further illustrates how the fault recovery is performed for a particular irregular application which requires only local reduction objects. Similar to Figure 6.3, the input mesh in Figure 6.5 is partitioned based on the output space and each output partition corresponds to an input partition. All the input partitions are then distributed onto a set of computing nodes. In Figure 6.5, we give an example where the input mesh is partitioned into 16 disjoint parts and we have four computing nodes. Initially, on the left of Figure 6.5, each node has four partitions and it will process each partition in its work queue and take checkpoints after computations correspondingly. Now, suppose node 4 fails after it processes partition 13 and saves its checkpoint to the persistent storage. The system detects the failure of node 4, and starts the data re-distribution among the other three nodes in a round-robin fashion. In this case, the scheduler could assign the partitions 13 and 16 to node 1. Partitions 14 and 15 are then re-distributed to node 2 and 3, respectively. Each node will then update its work queue with extra partitions and process all of them one after another. Note that partition 13 has been processed on node 4 and its checkpoint has been saved, so node 1 will simply load the checkpoint file for partition 13 and re-construct the input/output partition without the need of re-computation. After all output partitions are updated, the communication phase will be started to exchange the updates needed for next iteration among all the partitions. Overall, the input/output partitions of the failed node are recovered by either loading the checkpoints or computing unprocessed input partitions as normal and the correctness can be ensured in this way.

6.3 Experimental Results

In this section, we report results from a detailed evaluation of the fault tolerance approach we have presented and implemented as part of the MATE system.
6.3.1 Evaluation Methodology

Our evaluation is based on four applications: Jacobi, Sobel Filter (Sobel), Euler Solver (Euler), and Molecular Dynamics (Moldyn). The first two applications involve stencil computations and the last two involve irregular reductions. To evaluate our fault tolerance approach, we also implemented the four applications using MPI. We studied the checkpoint/restart costs of using the MPICH2 library (version 1.4.1p), which currently supports fault tolerance based on the BLCR library. For convenience, we use MPICH2-BLCR to refer to the MPICH2 library with fault tolerance, and FT-MATE is used to refer to the MATE-based fault tolerance.

Overall, in our experiments, we focused on the ensuing aspects: 1) Evaluating the scalability of the four applications using FT-MATE system, i.e., we determine the suitability of the extended generalized reduction (EGR) model for efficient parallel programming, 2) Studying the overheads for checkpointing the reduction objects for the four applications, and comparing them against the MPICH2-BLCR implementations, and 3) Understanding the fault recovery costs for our system in case of a node failure in a cluster. We support the recovery of the four applications considering two cases, with and without the rebirth of the failed node. We also study the recovery costs from MPICH2-BLCR.

The experiments were conducted on a cluster, where every node comprises of two Intel Xeon E5520 quad-core CPUs (8 cores in all). Each CPU core has a clock frequency of 2.27 GHz and the machine has a 12 GB main memory. We have used up to 256 CPU cores (on 32 nodes) for our study. The datasets used in our performance studies are as follows: For Jacobi and Sobel, we chose a grid structure containing nearly a billion elements, represented by a matrix of 8 GB. For Euler Solver, we used a mesh of around 12 million nodes and 77 million edges, stored in text files with a total size of 5.8 GB. For Molecular
Dynamics, the mesh contains 4 million nodes and 500 million edges, with the size of 7.4 GB. We run each application for up to to 1000 iterations.

### 6.3.2 Scalability Studies

First, assuming there are no node failures, we run the applications implemented using the FT-MATE system without enabling the fault tolerance support. Figures 6.6 and 6.7 show the average running time per iteration for the four applications as we scale the number of nodes. Specifically, we run each application in two modes: CPU-1 (use only 1 CPU core per node) and CPU-8 (use all 8 CPU cores per node).

As we can see in Figure 6.7, the relative speedup for Jacobi, with the CPU-8 version, scaled from 2 nodes to 16 nodes, is around 5.5. The relative speedup is 7.6 for the CPU-1 version as in Figure 6.6, showing good scalability when the total number of cores is smaller. From Figures 6.6 and 6.7, we can see that Sobel achieves relative speedups of 7.8 and 6.8.
Figure 6.7: Scalability with CPU-8 versions on 2, 4, 8, and 16 nodes for each of the four applications

for CPU-1 and CPU-8 versions, respectively, from 2 nodes to 16 nodes. The scalability results for Sobel are better than Jacobi because Sobel involves more computations (9-point stencil computations) than Jacobi (5-point stencil computations) while the combination time remains the same for the same sized dataset.

For the two irregular applications, when we increase the number of nodes from 2 to 16, for Euler, the system achieves relative speedups of 7.5 and 4.2 for CPU-1 and CPU-8 versions, as also shown in Figures 6.6 and 6.7. The relative speedups for Moldyn are 7.1 and 6.4 for CPU-1 and CPU-8 versions, respectively. Overall, we can see good scalability up to a certain number of cores.

We can also observe good speedups from CPU-1 versions in Figure 6.6 to CPU-8 versions in Figure 6.7. In particular, on 2 nodes, the system achieves a relative speedup of 7.6 from CPU-1 to CPU-8 Jacobi and 7.2 for Sobel. For Euler and Moldyn, the relative
speedups are 5.2 and 4.5, respectively, which is due to the necessary second-level partitioning overheads for parallelizing irregular reductions with an increasing number of threads per node.

### 6.3.3 Overheads of Supporting Fault Tolerance

In this set of experiments, we evaluate the checkpointing overheads of our approach. We consider execution of each application for 1000 iterations, enable fault tolerance with different checkpoint frequencies, and measure the overheads when there are no failures. We normalized the checkpointing costs based on the absolute execution times. In our following experiments, we run each application in the CPU-8 mode to achieve the best-possible performance for FT-MATE. As comparison, we also include the results from the MPICH2-BLCR library for which we launch 8 MPI processes per node.
Figure 6.9: Euler and Moldyn: Relative Checkpointing Overheads with Varying Checkpoint Intervals (on 8 nodes)

Figure 6.8 shows the results for Jacobi and Sobel for an 8 GB dataset on 8 nodes as we vary the checkpoint interval to be 1, 10, 100, and 200 iterations. (Checkpoint-Interval 1 implies that checkpointing is performed in each iteration, and so on). In MPICH2-BLCR, each checkpointing requires all processes to synchronize and save images to disk simultaneously, while our system allows each process to save its reduction objects into the distributed memory and/or the disk. From Figure 6.8, we can clearly see that FT-MATE outperforms MPICH2-BLCR by almost two orders of magnitude for Jacobi. If we take checkpoints in each iteration, our system only incurs around 15% overhead, whereas the MPICH2 fault tolerance library had more than 30 times slowdown. Both systems have less checkpointing overhead as we reduce the checkpointing frequency. Our system has close to zero normalized overheads when we set the checkpoint interval to be 100 and 200, whereas MPICH2-BLCR still has nearly 30% and 20% normalized overheads, respectively, in these two cases.
Figure 6.8 also demonstrates similar results for the Sobel application. Since Sobel involves more computations than Jacobi and the reduction object size remains the same, the normalized checkpointing overheads are lower. However, two orders of magnitude performance improvement is still seen. The results for the two irregular applications are shown in Figure 6.9. As in Figure 6.9, with the checkpoint interval set to be 1, the normalized checkpointing overheads for Euler is around 17.9% in our system while the MPICH2-BLCR has a factor of 15 times slowdown. For Moldyn, our system only has 3.0% normalized checkpointing costs and the MPICH2-BLCR incurs 765% normalized checkpointing costs. There are two main reasons why our approach outperforms MPICH2-BLCR. First, our programming model clearly exposes the iterative structure of the application as well as the dependence between processes. Thus, we only checkpoint the necessary computational states, which are captured in the reduction objects. So the snapshot size we take is much smaller than the size of the checkpoint images saved in MPICH2-BLCR, which may contain all the input data, output data, temporary buffers, communication channels, and even shared libraries. For example, when a checkpoint is taken during running Moldyn with the 7.4GB input data, the system snapshot for MPICH2-BLCR is occupying up to 9.3GB disk space while the checkpoint files in our system are only around 48MB. Second, we make use of both the memory of the distributed system and the persistent storage, while MPICH2-BLCR only uses the persistent storage.

6.3.4 Overheads of Fault Recovery

In the last set of experiments, we evaluate the recovery overheads of the two approaches. We consider failure of a single node, occurring at different failure points, with application running on 32 nodes. We run each application in the CPU-1 mode for FT-MATE and we
Figure 6.10: Jacobi: Comparison between MPICH2-BLCR and FT-MATE with Varying Failure Points (8GB dataset on 32 nodes and checkpoint interval 100/1000)

launch 1 MPI process per node for MPICH2-BLCR. In the evaluation, we consider two cases, one in which the failed node never recovers and no spare node is available, and the second where either the failed node recovers quickly, or a spare node is available. We use W/O Rebirth and W/ Rebirth to denote these two scenarios. In the latest stable MPICH2-BLCR library (version 1.4.1p), failure recovery is feasible only using the same number of nodes as in the original execution. Therefore, results for MPICH2-BLCR we are able to report are for the W/ Rebirth mode.

We run each application for 1000 iterations and checkpoint an application every 100 iterations. Three failures points are considered, 25%, 50%, and 75% (A failure point 25% means that one node fails after executing 250 iterations, and so on). Since the checkpoint interval is 100, for failure points 25% and 75%, the recovery process introduces redundant computations for both the systems. In comparison, the failure point 50% only incurs the absolute recovery overheads. For our system, they include data re-distribution, input
space creation, and output space computation/recovery in the failed iteration, whereas for MPICH2-BLCR, they are dominated by loading all the checkpoint files and restarting all processes. Note that to focus on the recovery overheads, we disabled checkpointing after recovery in both the systems.

Figure 6.10 shows the comparison results between FT-MATE and MPICH2-BLCR for Jacobi. In the W/Rebirth mode and with the failure point 50%, our system has a very small absolute recovery overhead, i.e., 0.02% of the execution time. MPICH2-BLCR library has 9.94% absolute recovery overheads. This is because the main cost is loading the checkpoint files, and they tend to be very small for our system. With the failure points 25% and 75% in the W/Rebirth mode, the recovery times are instead dominated by the redundant computations in both the systems. Our system, however, is still much better than MPICH2-BLCR with redundant computations. Specifically, for the failure point 25%, our system results in 5.02% recovery costs while MPICH2-BLCR has around 14.94% overheads. Similarly, for
the failure point 75%, our system causes 5.15% recovery costs and MPICH2-BLCR incurs 15.12% overheads.

In the W/O Rebirth mode, the normalized overall recovery overheads are relatively higher, because we have a fewer number of nodes to perform the rest of the execution.

Similar results are observed for Sobel as shown in Figure 6.11. Our system has close to zero (around 0.03%) absolute recovery overheads in the W/ Rebirth mode with a failure point 50% while MPICH2-BLCR has around 3.11% absolute recovery overheads. For the two irregular applications, we again have similar observations as shown in Figures 6.12 and 6.13. Especially, the normalized absolute recovery overheads are 0.19% and 0.25% for Euler and Moldyn in our system, much less than the corresponding values 4.76% and 2.27% in MPICH2-BLCR.
Figure 6.13: Moldyn: Comparison between MPICH2-BLCR and FT-MATE with Varying Failure Points (7.4GB dataset on 32 nodes and checkpoint interval 100/1000)

6.4 Summary

In this work, we have described a programming model that can support many different applications classes, while enabling fault-tolerance with lower overheads and greater flexibility in recovery. Using four applications, we compared our approach with the MPICH2 library on a cluster of multi-core CPUs. Overall, our approach has at least one order of magnitude lower checkpointing overheads than MPICH2 for all the four applications. Besides, our approach can support continued execution without rebirth of the failed nodes and has close to zero absolute fault recovery costs in case of a node failure.

The approach we have described in this work requires development of applications using a new programming model. In the future, we will like to develop compiler analysis that can extract underlying communication patterns and data distribution from existing MPI
applications automatically, and apply the approach described in this work to support fault-tolerance.
Chapter 7: Dealing with GPU Failures in the MATE-CG System

In FT-MATE, we only considered CPU failures in a cluster of multi-core CPUs. However, it is also increasingly important to provide fault tolerance in GPUs since GPU pipelines are becoming more popular and programmable, which has made GPUs more attractive to a wider audience. However, pre-Fermi Nvidia GPUs do not provide fault tolerance. Since GPUs are now often used in high performance computing and other general purpose application domains where data integrity is important, providing fault tolerance on GPUs is becoming increasingly important.

Therefore, in this work, we are trying to deal with GPU failures in a heterogeneous CPU-GPU cluster. Especially, on top of our MATE-CG system, we support fault tolerance and recovery for GPU computations through checkpointing and work re-distribution in a hybrid CPU-GPU cluster. Based on the generalized reduction API provided by the MATE-CG system, the CPU-GPU algorithms will be able to periodically checkpoint their work from GPUs in the host CPUs. In our problem, we assume that only the GPU is unreliable (i.e., the CPU is reliable). So we are particularly dealing with the checkpointing of GPU computations. If a fault has been detected on a GPU, then the runtime system can roll back to the last checkpoint on the host CPU-GPU node and continue executing.
7.1 Our GPU Fault Tolerance Technique

In this section, we first describe the traditional method for checkpointing and then discuss our GPU fault tolerance approach based on MATE-CG.

7.1.1 GPU Checkpointing

One common method to checkpoint data from any process (or application) is to take a complete snapshot of the entire process (or application variables and data structures). While this is a conservative and a general approach that is applicable to any process (or application), it can be a very expensive method for certain classes of applications. In other words, the amount (size) of checkpoint information can be extremely large for certain classes of applications that may eventually slowdown the applications’ performance significantly. Thus, our first goal is to take advantage of our knowledge of the underlying programming model and reduce the size of checkpoint information.

Using MATE-CG, we focus on the class of applications that can be implemented using the generalized reduction model. Especially, we choose applications that involve generalized reductions and stencil computations. For generalized reductions, similar to that of FT-MATE, we just simply exploit the reduction objects exposed in the generalized reduction model. As described in Chapter 5, the input data blocks on each node are partitioned into two parts, and one part goes to the CPU and the other part goes to the GPU. On the GPU, the data blocks are further split into GPU chunks and then processed one by one. The corresponding reduction objects are updated in the GPU device memory. Since the reduction objects act as the output space and captures the necessary computational states from the GPU, we only need to save them on the host CPU via checkpointing.
For stencil kernels, the computations are performed on matrix, which is the primary data structure. Stencil kernels are implemented using an iterative finite-difference technique, over a spatially dense grid, and involves performing some pattern of nearest neighbor computations. The input problem and the output are represented in the form of matrices, where each point in the matrix is updated with weighted contributions from its neighbors. A simple example of a 2-D, 5-point stencil kernel with uniform weight contribution from its neighbors is shown in Figure 7.1. Typically, there can be a set of input data matrices and output data matrices. As a common case, these input and output matrices will be different from each other, however, in some cases, they can be the same. In this work, we focus on the common case, where input and output matrices are different.

Based on the generalized reduction API, a simple mechanism for parallelizing this application is based on the output matrix partitioning. Suppose the input and output matrices are partitioned along the rows first. Now, the elements that reside on the border of two neighboring input partitions need to be communicated/shared among a pair of output partitions. In this case, output partition $i - 1$ and output partition $i$, output partition $i$ and output partition $i + 1$, and so on are the communicating/sharing pairs of output partitions.

```c
for(int i = 1 to num_rows/chunklet) {
    for(int j = 1 to y-1) {
    }
}
```

Figure 7.1: Structured Grid Pattern
respectively. In this way, each output partition is treated as a reduction object that will be updated locally on each CPU/GPU worker.

In order to deal with the failures of stencil computations, it is enough if we checkpoint only the rows of the output matrix. However, the computation of every row in the output matrix is bound to the corresponding set of row(s) in the input matrix. Thus, in case of failure and restart, it is important to know which rows of input/output matrix were computed by which processor. This can be obtained by just maintaining the index of the work assignment to the available processors. Thus, the entire input matrix need not be stored as checkpoint information. Note that index information will much smaller when compared to the entire input matrix size. Also, stencil computations can be easily implemented using the generalized reduction model as explained in Chapter 6.1.4.

Specifically, we are focusing on checkpointing with CUDA runtime APIs. In a typical case (when there will be no failure), it is preferred to transfer the entire data that needs to be computed on the GPU at once, finish the computation, and then, transfer the results back to the CPU. This is done to reduce the latency component of the entire data transfer cost. However, this method of computation is not useful for checkpointing. This is because, once the computation starts, the output data cannot be checkpointed back to CPU until this computation is over. Thus, we divide the entire input/output data into a fixed number of chunks. Each input/output chunk is transferred to GPU, computed, and checkpointed. For checkpointing, we use the CUDA Streams and CudaMemcpyAsync() to improve the performance. For instance, the computation of the \(i+1\)th chunk will be overlapped with the checkpointing of the \(i\)th chunk that the GPU computed.
7.1.2 CPU-GPU Work Distribution upon GPU Failure

We now describe our method for dynamically distributing the input/output chunks between CPU and GPU. We use a classical Master-Worker model. We divide the input/output data into a fixed number of chunks. The master takes care of distributing the work (chunks) to worker (CPU or GPU) processors. For stencil computations as in Figure 7.1, there is dependency in the input matrix as there are ghost rows (border rows). Thus, we do the following. Whenever a chunk from output matrix (B matrix) is given to a worker, a corresponding chunk from the input matrix (A matrix) is given to the processor. Note that, in addition we also give the ghost rows from the input matrix (A matrix), so that all the rows of the chunk in the output matrix (B matrix) can be computed correctly. For generalized reductions, we just divide the input data into chunks and distribute chunks of input data and the entire output data (reduction objects) to CPU/GPU workers.

Now, whenever there is a GPU failure, the runtime will distribute the chunks evenly across other alive processors. The remaining chunks can be distributed to any of the other processors in the cluster. The rationale is that when a GPU fails, the other processors end up computing additional chunks. Since the data is divided into chunks, the load balancing will be more fine-grained.

Specifically, upon a GPU failure, the runtime will roll back to the last checkpoint for the GPU. Then the recovery process begins. In the recovery process, we support three schemes of continuing execution. The first is Restart w/o Checkpoint, which will simply redo the work assigned to the GPU before failure using the CPU on the same node. In this simple scheme, the work is not distributed to other nodes in the cluster. Another scheme is called Checkpoint w/o Re-distribution. In this scheme, we use the latest checkpoint from the GPU to avoid redoing the already computed work on the corresponding CPU. However, it does
not re-distribute the remaining work after failure evenly across all nodes and only performs the recovery on the CPU within the same node. The third scheme is Checkpoint with Redistribution. In this scheme, we allow both checkpointing and evenly re-distribution of the remaining work on the other resources in the entire cluster of nodes. By default, we enabled the third scheme in our system.

7.2 Experimental Results

In this section, we report results from a detailed evaluation of the fault tolerance approach we have implemented and incorporated into the MATE-CG system.

7.2.1 Evaluation Methodology

Our evaluation is based on four applications implemented in the MATE-CG system: Jacobi, Sobel Filter (Sobel), PageRank and Gridding Kernel. The first two applications involve stencil computations and the last two are chosen from data mining and scientific computing, and involve generalized reductions. The algorithmic details of the four applications and how to implement them using the generalized reduction model can be seen from Chapters 5 and 6.

Overall, in our experiments, we focused on the ensuing aspects: 1) Evaluating the scalability of the four applications using MATE-CG system. i.e., we determine the suitability of the generalized reduction model for efficient parallel programming using both CPUs and GPUs. 2) Studying the overheads for checkpointing and fault recovery costs for our system in case of a GPU failure in a cluster. We support the recovery of the four applications in three different schemes.

The experiments were conducted on a heterogeneous CPU-GPU cluster, where every node comprises of two Intel Xeon E5520 quad-core CPUs (8 cores in all) and an Nvidia
Tesla C2050 (Fermi) GPU card. Each CPU core has a clock frequency of 2.27 GHz and the machine has a 48 GB main memory. The GPU has a clock frequency of 1.15 GHz each core and a 2.68 GB device memory. We have used up to 64 CPU cores and 3584 GPU cores (on 8 nodes) for our study.

### 7.2.2 Scalability of Using CPU and GPU Simultaneously

In the first set of experiments, we evaluate the performance benefits that can be obtained from simultaneously utilizing the multi-core CPU and GPU in the cluster of nodes. To understand this benefit, we create three versions of executions - CPU-only, GPU-only, and GPU+GPU. In CPU-only execution, we use all the 8 cores in the multi-core CPU and the entire computation is performed on the CPU. In GPU-only version, the entire computation is performed on the GPU while running a host thread to communicate with the CPU on a single core. For CPU+GPU version, the computations are distributed evenly between the multi-core and the GPU using our work distribution scheme. The results from Sobel Filter
and Jacobi are shown in Figures 7.2 and 7.3 and the results from PageRank and Gridding Kernel are presented in Figures 7.4 and 7.5. For both applications from stencil computations, we used an input data size of 800 MB. For PageRank, we used a dataset of 3.6 GB and the data size for the Gridding Kernel is around 512 MB.

On the X-axis, we vary the number of nodes used from 1 to 8 to understand the scalability across nodes. All presented results are relative to the single-core CPU performance. From the results, it can be seen that for both Sobel and Jacobi, we achieve a scalability of about 40.4 and 41 times from the use of CPU+GPU from 8 nodes. The scalability numbers are 28.8 and 96.5 for PageRank and Gridding Kernel. Moreover, for Sobel, we are able to achieve anywhere between 15.3% to 22.8% from the use of CPU+GPU against the best of CPU-only and GPU-only on different number of nodes. Also from Jacobi, we can achieve up to 21.2% from the simultaneously use of CPU and GPU within 8 nodes. Similarly, for PageRank and Gridding Kernel, the further improvements of CPU+GPU versions over the
best of CPU-only and GPU-only are 8.2% and 32.2% on 8 nodes, respectively. This efficiency is attributed to the work distribution strategies within this framework to make use of CPUs and GPUs at the same time.

It is interesting to note that the relative speedup obtained from the use of CPU-only and GPU-only is different for all the four applications. For instance in Sobel and PageRank, both CPU and GPU perform comparable to each other, and hence contributing almost equally during the CPU+GPU execution. On the other hand in Jacobi, CPU-only is almost twice faster than the GPU-only and hence GPU contributes relatively less in the CPU+GPU execution. While in Gridding Kernel, GPU-only is much faster than CPU-only and GPU contributes relatively more in the hybrid execution. This observation has impact on the performance during the GPU failures which we study later.
Overall, in this experiment, we show that significant performance improvements can be achieved from the aggregate use of CPU and GPU, and thus, it is necessary to provide fault-tolerance mechanisms for GPU failures.

### 7.2.3 Benefits of Checkpoint and Work Re-distribution Upon GPU Failure

In the second set of experiments, we evaluate the benefits of checkpointing and the work re-distribution in the case of GPU failures. The results are shown in Figures 7.6 through 7.9. All the experiments were run on 8 nodes and both stencil applications use an input data size of 3.2GB while the input data sizes for the other two applications are 3.6GB and 512MB, respectively. We present the execution overheads due to GPU failures relative to the execution case where there is no failure. To analyze further and understand better, we use three schemes of execution as described in Section 7.1.2, denoted as *Restart w/o Chkpt*, *Chkpt w/o Re-Dist*, and *Chkpt with Re-Dist*, respectively.
Figure 7.6: Sobel: Benefit of Checkpoint and Work Re-distribution During GPU Failure

From Figures 7.6 through 7.9, we can analyze the performance degradation due to GPU failures for all the four applications. To understand better, we create three failure points (as shown in X-axis). For instance, 50% indicates that a GPU failure within a node happened after completion of the 50% of the entire work. A common observation is that Restart w/o Chkpt has the worst degradation of all the four applications at all the failure points. This is because, this scheme simply redoes all the GPU work on the corresponding CPU without using any checkpoint or re-distribution. For instance, performance degradation can be up to 230% using this simple scheme for Gridding Kernel since GPU obtains more data to process. Next, it can be seen that using Chkpt w/o Re-Dist, we can improve the degradation between 187% and 10% at different failure points. Thus, checkpointing is beneficial and clearly takes advantage of the failure points. Finally, using Chkpt with Re-Dist, we can further reduce the performance degradation significantly. It makes use of both failure points information (through checkpoints) as well as evenly re-distributes the work
across all other alive resources in the cluster. In fact, the degradation ranges only between 11.4% and 1.91% at different failure points for all the four applications. We also made some interesting observations with Jacobi and Gridding Kernel. In the case for Jacobi as in Figure 7.7, the performance degradation due to the simple scheme like Restart w/o Chkpt is not very large. This is attributed to the fact that the relative performance between CPU and GPU is skewed towards CPU and hence only a small portion of work is computed by the GPU. Thus, upon GPU failures, significant degradation is not observed. But for Gridding Kernel as in Figure 7.9, the simple scheme has significant performance slowdowns (around 2.5 times slower), which is because a large part of work is assigned to GPUs in Gridding Kernel and a GPU failure causes severe slowdown for the overall running time on the node.

We now compare the effectiveness of our GPU checkpointing method compared to the existing methods. In the existing checkpointing methods used in CheCUDA [128], all the memory variables and data structures are checkpointed. In our checkpointing method, we
take advantage of the knowledge of the computation pattern. Thus, we reduce the size of checkpoint data. We now compare the size of checkpoint data in our method, to the CheCUDA method. The results are shown in the Table 7.1. The results are presented for the worst case. Thus, in the worst case, the existing checkpointing methods like CheCUDA can use at least twice more memory for checkpoint purpose. The situation can be worse for those stencil kernels which use more than one input matrix for its computation. This is because, our checkpoint method does not need to checkpoint the input data, which CheCUDA will checkpoint every variable and input data in the computation.

<table>
<thead>
<tr>
<th>Applications</th>
<th>Input Size</th>
<th>Output Size</th>
<th>Our Checkpoint Method</th>
<th>CheCUDA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sobel</td>
<td>3.2GB</td>
<td>3.2GB</td>
<td>3.2GB</td>
<td>6.4GB</td>
</tr>
<tr>
<td>Jacobi</td>
<td>3.2GB</td>
<td>3.2GB</td>
<td>3.2GB</td>
<td>6.4GB</td>
</tr>
<tr>
<td>PageRank</td>
<td>3.6GB</td>
<td>0.5GB</td>
<td>0.5GB</td>
<td>4.1GB</td>
</tr>
<tr>
<td>Gridding Kernel</td>
<td>512MB</td>
<td>30MB</td>
<td>30MB</td>
<td>543MB</td>
</tr>
</tbody>
</table>

Table 7.1: Comparison of Size of Checkpoint Information
7.3 Summary

In this work, we supported fault tolerance and recovery in the MATE-CG system, especially dealing with GPU failures in a heterogeneous CPU-GPU cluster. Using four applications from stencil computations, data mining and scientific computing, we observed good scalabilities on up to 8 nodes for the four applications’ implementations by using CPUs and GPUs at the same time. Also, we evaluated our fault tolerant approach by comparing it against two other failure recovery schemes. Overall, the combined use of checkpointing and work re-distribution incurred very low recovery overheads in case of a GPU failure in a cluster. Also, our checkpoint size is much smaller than that of the CheCUDA method.
Chapter 8: Future Work

While our MATE systems have achieved better performance than the original MapReduce, we didn’t really improve the programming productivity. In order to use our MATE systems, programmers need to modify the existing MapReduce programs so that they can be running with our MATE runtime. This restricts the use of our systems since people need to spend extra time and efforts for rewriting the code to have better performance. It is even worse for researchers in other areas like statistics since they tend to be familiar with SQL or script languages such as Matlab and R, not C or C++. Apparently, our MATE systems lack the support of higher-level languages support.

Therefore, in the future, we plan to improve our systems by providing a higher-level API than the current generalized reduction API. In this way, with MATE systems as the backend and a higher-level API in the front end, we can have better programming productivity and also make our system easy to use by researchers inside/outside computer science.

8.1 Higher-Level APIs for MapReduce

Similar research works have been conducted on top of the original MapReduce to provide SQL-like support.
This origins from a contentious debate back to the 1970s in the database research community as to choose between the relational advocates and the Codasyl advocates [25]. According to the paper [106], the key issue of the discussion was whether a program to access data in a parallel database management system (DBMS) should be written either by:

- Expressing what you want for how to get data using declarative languages (Relational)
- Designing an algorithm for data access (Codasyl)

At the end, the relational database systems prevailed and the last several decades have been a testament to the value of them. As mentioned in the paper [106], “Programs in higher-level languages, such as SQL, are easier to write, easier to modify, and easier for a new person to understand. Codasyl was criticized for being ‘the assembly language of DBMS access’. We argue that MapReduce programming is somewhat analogous to Codasyl programming.” Therefore, research works from the MapReduce and Hadoop community suggest that there could be widespread sharing of MapReduce code snippets to do some common tasks, such as the join operation of different data sets. To further ease the burden of having to re-implement repetitive tasks, the MapReduce/Hadoop community is migrating higher-level languages on top of the current MapReduce interface to move such functionality into the runtime. Projects like Pig-Latin [105] from Yahoo!, Hive [131] from Facebook, and Sawzall [107] from Google are among the notable software systems in this direction.

### 8.2 Higher-Level APIs for Generalized Reduction

Since much work has been done to provide higher-level language support for MapReduce, a natural question would be that can we do similar improvements for our generalized
reduction model by providing higher-level languages support? In this way, our systems could be more useful and attractive for more people in different research communities.

Therefore, toward this goal, we could provide a higher-level API for our generalized reductions in the future. Especially, we can achieve this goal in the ensuing two directions.

First, a simple way would be that, we can keep the original MapReduce API and move the optimizations into the MapReduce runtime with the implicit use of the reduction objects. This idea has been demonstrated to be effective and efficient in a recent research project from our research group [39]. Based on this work, we would be able to easily reuse the existing higher-level API support for MapReduce since we can have the same MapReduce API while we are still able to improve the MapReduce performance in the runtime at the same time. This idea is attractive because it allows us to reuse the existing work from MapReduce.

Second, an alternative is that we could also directly migrate the SQL or script languages support on top of our programming model so that people can also use their familiar languages to program the MATE systems on different architectures. This idea is also promising because our MATE systems have been developed for programming different parallel architectures like multi-core CPUs, many-core GPUs, and heterogeneous CPU-CPU clusters while there is still limited or no general MapReduce-like support for these parallel architectures. Thus, it would be quite useful and interesting to have such features on top of the MATE systems so the users can easily manipulate a higher-level API to make use of a diverse set of parallel environments for accelerating their programs.

Overall, both ideas can be used as possible work for the future in order to further improve the programming productivity of our systems and also benefit more people from different communities.
This is a summary of the contributions of this dissertation.

9.1 Contributions

The use of map-reduce has gain high programming productivity in large-scale data centers but its performance efficiency is not satisfactory for high performance computing on emerging parallel architectures. To provide a map-reduce-like API with better performance and supporting different parallel architectures, starting from a comparative study showing that the original map-reduce API was not suitable for a set of data-intensive applications, we have developed and proposed different systems with the following contributions.

- We developed a map-reduce system with an alternate API (MATE) for multi-core environments. In the alternate API, the programmer takes advantage of a user-declared reduction object, which significantly reduces the overheads arising because of the original map-reduce API. These overheads are caused by high memory requirements of intermediate results and the need for data communication between Map and Reduce stages. In the experiments, our results show that this alternate API achieves good scalability and outperforms the original map-reduce for three data mining applications in two distinct multi-core platforms.
• We extended the MATE system to provide support for managing disk-resident reduction objects, which can be arbitrary-sized and are also required for many data-intensive applications. We have evaluated the Ex-MATE system with three popular graph mining algorithms. In the experiments, we were able to achieve better performance than Hadoop-based PEGASUS in an environment with up to 128 cores (16 nodes). Also, our API also results in simpler code than the use of the original map-reduce API for these graph mining applications.

• We further designed a map-reduce-like framework named MATE-CG for programming a heterogeneous environment of CPUs and GPUs. The MATE-CG system is built on top of MATE and Ex-MATE and one of the key novel contributions of this new system is a novel auto-tuning approach for identifying the suitable data distribution between CPUs and GPUs. In the experiments, we used up to 128 CPU cores and 7168 GPU cores for our performance studies in a heterogeneous cluster. We shown scalability with increasing number of nodes and cores, as well as with combined use of CPU cores and GPUs. Our auto-tuning approach has been effective in choosing the appropriate fraction with low runtime overheads for iterative applications.

• It is widely accepted that the existing solutions for providing fault tolerance in MPI programs will not be feasible for the exascale high-end systems in the future. The reduction object model used in our MATE systems has been demonstrated to have lower overheads for fault tolerance support than Hadoop for data mining and graph mining applications. Therefore, to provide more efficient fault tolerance for MPI programs in the exascale era, we extended the generalized reduction programming model in the MATE systems so that it can support more MPI-based applications.
classes, while enabling fault-tolerance with lower overheads and greater flexibility in recovery. Using four applications, we compared our approach with the MPICH2 library on a cluster of multi-core CPUs. Overall, our approach has at least one order of magnitude lower checkpointing overheads than MPICH2 for all the four applications. Besides, our approach can support continued execution without rebirth of the failed nodes and has close to zero absolute fault recovery costs in case of a node failure.

- As GPUs became popular for HPC and other general purpose programs, we also added fault tolerance support to deal with GPU failures based on the MATE-CG system. Using four applications from different domains, we evaluated the benefits of checkpointing and the work re-distribution in case of GPU failures in a CPU-GPU cluster. Our results show that, based on our programming model in the MATE-CG system, checkpointing GPU results with re-distributing the remaining work has very low overheads in the failure recovery process.

According to an article by Derrick Harris [69], “The great thing about big data is that there is still plenty of room for new blood, especially for companies that want to leave infrastructure in the rear-view mirror. The current data-infrastructure space, including Hadoop, is well-funded and nearly saturated, but it also needs help.”

Therefore, by focusing on these ideas and trying to provide higher-level API support in the future, we would like to enhance the programming productivity and improve performance efficiency of the current systems, and more importantly provide useful methods and beneficial insights for achieving high performance computing with similar systems for the future.
Bibliography


168


174


