ADAPTATION OF THE MAPREDUCE PROGRAMMING FRAMEWORK TO COMPUTE-INTENSIVE DATA-ANALYTICS KERNELS

BY

REZA FARIVAR

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2012

Urbana, Illinois

Doctoral Committee:

Professor Roy H. Campbell, Chair
Professor Wen Mei Hwu
Professor William Sanders
Professor Anand Raghunathan, Purdue University

ABSTRACT

Compute-intensive data-analytics (CIDA) applications have become a major component of many different business domains, as well as scientific computing applications. These algorithms stem from domains as diverse as web analysis and social networks, machine learning and data mining, text analysis, bio-informatics, astronomy image analysis, business analytics, large scale graph algorithms, image/video processing and recognition, some high performance computing problems, quantitative finance and simulation among others. These computational problems deal with massive data sets, and require performing lots of computation per data element.

This thesis presents a vision of CIDA applications programmed in a MapReduce style framework and running on clusters of accelerators. Regardless of the type of accelerator, whether GPUs (NVIDIA or AMD), other manycore architectures (like Intel Larrabee or MIC) or even heterogeneous chips (AMD Fusion or IBM Cell processor), there is a fundamental condition imposed on the software, namely the increased sensitivity to locality. As a result, the common theme in this thesis is to increase the locality in CIDA applications.

We report on four research efforts to achieve this goal. The *Multiple independent threads on a heterogeneous resource architecture* (MITHRA) project integrates Hadoop MapReduce and GPUs together, where the *map()* functions execute. As a result, by moving the *map()* functions to GPUs we increase the locality of reference and gain better performance. We have shown that when the MITHRA model is applicable (for instance for Monte Carlo algorithms), each computing node can perform orders of magnitude more work in the same run-time.

Then we introduce *partitioned iterative convergence* (PIC) as an approach to realize iterative algorithms on clusters. We observed that conventional implementations of iterative algorithms using MapReduce are quite inefficient as a result of several factors. Complementary to prior work, we focused

on addressing the challenges of high network traffic due to frequent model updates and lack of parallelism across iterations. PIC has two phases. In the first phase, called the best-effort phase, it partitions the problem and runs the sub-problems in individual cluster nodes, where the locality can be exploited better. The results of this phase can be numerically inaccurate (about 3% based on experimental results), but can be computed much faster. The second phase of PIC, called the top-off phase, runs the original iterative algorithm a few more iterations (starting with the results of the best-effort phase) to compute an accurate answer.

Finally we introduce two GPU-based projects that try to increase the performance of MapReduce style functions in GPUs. The first is loop maximizing, a code transformation for GPUs that can eliminate code flow divergence (and hence serialization in GPUs) and result in better usage of GPU processing elements. Using this technique, we have achieved the highest reported speedups for gene alignment algorithms. The second GPU-based project is a library for dynamic shared memory allocation and access in GPUs assuming independent execution of the GPU threads, which happens in a MapReduce style environment among both *map()* and *reduce()* functions.

The two MapReduce adaptations (MITHRA and PIC), the GPU-based loop-maximizing optimization and the Plasma library together lay the plan for the goal of achieving good performance on locality-sensitive clusters. This thesis shows the feasibility of this approach, and describes how each of these projects contributes to the collective target.

I dedicate my thesis to my parents, for their perpetual love and support.

ACKNOWLEDGMENTS

Over the past seven years I have received support, encouragement and education from a great number of individuals. First and foremost, I would like to thank Professor Roy H. Campbell, who supported me in whatever field I liked to research. He taught me how to be independent in my research and how to find connections between seemingly unrelated topics. Finally, he gently pushed me (and his other students) toward the correct path without making it too obvious. I have learned a great deal from him and will be forever thankful for that.

I would like to thank Professor Anand Raghunathan and Doctor Srimath Chakradhar of the NEC laboratories of America for their help and support. I spent many hours discussing the ups and downs of system design with Professor Raghnathan, and much time discussing the high-level view of my research with Dr. Chakradhar. I also express my gratitude toward Professor Wen Mei Hwu, whose enthusiasm about GPU computing pushed me towards that topic. I studied his great lecture notes when no other source was available, and subsequently entered a research domain that eventually would prove to be a major industry revolution. I thank him for pushing through his vision and making us students interested in this field.

Professor Ravi Iyer in the ECE department helped me find my way in a new country and environment, and pushed me towards realizing my potential. I learned a great deal from him, and I am thankful for that. I am grateful toward Professor Ghasem Miremadi of the Sharif University of Technology, who taught me the alphabet of research: identifying the right problem to tackle and how to find interesting solutions through perseverance where none seems to exist at the first glance. I am thankful to Professor Hasan Katuzain of the AmirKabir University of Technology, who helped me among a group of young undergraduates to perform research where no one expected us to. He was great at motivating his students, and to me that was his greatest

achievement. He taught us that"we should learn to extract energy from deep within ourselves even when one feels he is at the bottom of a pit". He is a true motivation to his students.

I would also like to thank my colleagues and dear friends throughout the years, in particular Dr. Mahdi Fazeli, Dr. Ellick Chan, Dr. Abhishek Verma, Dr. Catello Di Martino, Mr. Mirko Montanari, Mr. Shivaram Venkataraman, Mr. Baeksan Oh and Mr. Harshit Kharbanda. I thank them for the many hours, day and night, spent with me on projects in the labs, and without their contributions the projects would never complete.

Finally I thank all the teachers, mentors and professors I have had throughout the years. I feel lucky to have had the privilege of sharing my academic journey with such great mentors and colleagues. There is no doubt that each one of these individuals has put a positive mark on me and made me who I am.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

API	Application Programming Interface
CIDA	Compute Intensive Data Analytics
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
EC2	(Amazon®) Elastic Compute Cloud
GPU	Graphics Processing Unit
HDFS	Hadoop Distributed File System
HPC	High Performance Computing
IC	Iterative Convergence
MITHRA	Multiple Independent Threads on a Heterogeneous Resource Architecture
PIC	Partitioned Iterative Convergence
Plasma	Parallel Library for Architecture-aware Shared Memory Allocation

CHAPTER 1

INTRODUCTION

Cloud computing is increasingly considered as a potential platform for deploying scalable data-analytics applications. These algorithms stem from domains as diverse as web analysis and social networks, machine learning and data mining, text analysis, bio-informatics, business analytics, large scale graph algorithms, image/video processing and recognition, some high performance computing problems, quantitative finance and simulation among others. These computational problems deal with massive data sets, and require doing lots of computation per data element. In this document, we call this class of computational problems as *compute-intensive data-analytics* problems, and use the acronym CIDA to represent them in the text.

We would like to distinguish CIDA algorithms from scientific HPC (high performance computing) problems. Scientific problems typically rely heavily on intercommunication between the parallel processing elements. On the other hand the communication requirement between different computing elements for CIDA problems is less pressing than traditional scientific HPC problems. For this reason, they can potentially run efficiently on cluster computers, and hence are a suitable match for cloud adoption.

With the growing size and importance of these applications and the data associated with them, scalable data analysis and management systems and programming frameworks form a crucial part of the cloud infrastructure. However, many of the current cloud enabling technologies were designed with a different mindset, one that is rooted in commercial enterprises, where computations are typically simple, and the bottleneck is usually the I/O subsystems.

As a popular and powerful programming model and framework, the MapReduce model (and its open source implementation Hadoop) clearly shows this set of assumptions. Hadoop can be very efficient in big-data processing tasks, where the majority of the job times are spent waiting for data I/O from disks

or network. On the other hand, when compute-intensive tasks are implemented using Hadoop, the results are not optimal [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]. Therefore, implementing CIDA and HPC applications in clouds has not gained the same popularity as enterprise big-data processing tasks [2].

The problem is due to the fact that MapReduce frameworks do not strive to utilize possible localities in the CIDA algorithms to their best advantage, as the parallelism model of MapReduce is quite rigid. On one hand, CIDA problems can allow for more localities than the framework can utilize, and as a result the MapReduce framework currently has unwanted overhead for CIDA algorithms. On the other hand, the emerging infrastructure hardware (such as accelerator clusters) can offer extra parallelism and locality-based run-time enhancements, but the current MapReduce frameworks ignore them.

The astute reader might then ask, Why bother to use MapReduce for programming CIDA applications in the first place? After all, when running a regular enterprise big-data application, these issues get masked by the I/O wait times, but when a CIDA application is implemented in MapReduce, they become bottlenecks and need to be addressed.

There is a twofold answer. First, MapReduce has gained massive popularity due to its success in enterprise cloud computing, and therefore the current open source frameworks are quite well-developed. As a result, many researchers have already started to look into Hadoop for HPC or CIDA purposes [3, 4, 10, 11, 12, 13, 14, 15]. The second reason, as we will show throughout this report, is that we *can* indeed increase its performance, while keeping its programmer-friendly semantics and all the benefits that come with cloud computing.

Thus, my thesis statement is as follows: **Exploiting a problem's potential for locality and better mapping to the available resources (e.g. GPUs) is the key to increase the performance of MapReduce programming frameworks for CIDA problems.**

From a high level, the way to do this is to introduce more locality in the CIDA algorithms (as we have done in PIC and loop maximizing projects), and to optimally utilize this locality from a systems point of view (as has been addressed in both MITHRA and PIC projects). We have tried to address this goal from three angles. First, when the opportunity for using locality is already present (as in MITHRA), we strive to better use the available hardware resources, in this case GPU accelerators, to utilize the localities. The

second aspect is by exploiting the problem domain properties and identifying the computational localities that can be extracted from CIDA problems, changing the programs to manifest them and to utilize them (as is done in the PIC project). Finally, we identify possible hindrances to their successful utilization (as is done in loop maximizing and the Plasma library).

This thesis reports on how we have tackled these three aspects. We introduce ***MITHRA*** [16], one of the first (and up to now among the very few) attempts to integrate GPUs into the Hadoop programming framework, thus trying to address the problem from the first angle mentioned above. MITHRA moves the *map()* functions into GPUs, while running the *reduce()* functions globally in the cluster.

We then introduce ***PIC*** [17], which is designed for iterative convergence computational problems. Many HPC and CIDA algorithms are based on iterative algorithms, and when applicable, PIC can significantly reduce the run-time. PIC addresses the problem from the second angle mentioned above, trying to work with the specific properties of iterative CIDA problems to exploit further parallelism. PIC is currently targeting a regular CPU cluster, but its final goal is to move both the *map()* and *reduce()* functions into GPUs.

Finally we present two research efforts towards achieving better performance when a MapReduce style environment (such as MITHRA or PIC) runs on GPU clusters. Because of the MapReduce formalized semantics, we can assume certain properties about the functions running on GPUs, and as a result develop solutions targeted for MapReduce style functions on GPUs. The first of these is **Loop Maximizing** [18], a novel code transformation that can eliminate control flow divergence for different threads. Control flow divergence is detrimental to the performance of GPU programs. As MITHRA currently runs *map()* functions in the GPUs and PIC is being extended to run both *map()* and *reduce()* in GPUs, we expect to gain a higher level of performance while removing the responsibility of writing GPU compliant code on the programmers. The second research effort is a GPU memory allocation library called **Plasma** [19] (**Parallel Library for Architecture-aware Shared Memory Allocation**), a library for shared-memory dynamic allocation and bank-conflict free accesses for NVIDIA GPUs, which allows for independent, parallel, run-time allocation and deallocation of small arrays, and ensuring bank-conflict free accesses. A major assumption in Plasma is independent execution of different threads in the GPU, which is exactly how

map() functions in MITHRA execute. The system allows order-of-magnitude faster pointer chasing operations for small and medium sized arrays.

CHAPTER 2

MOTIVATION: CIDA ADAPTATIONS OF THE MAPREDUCE MODEL

This chapter briefly introduces the MapReduce programming model, and then provides information on some related adaptations of it to some domain specific problem domains, beyond the original model it was designed for.

2.1 MapReduce Programming Model

MapReduce is a programming idiom developed by Google [20] for large-scale parallel computations. *map()* and *reduce()* are derived from functional programming where the *map()* function maps an input value to a list of intermediate key/value pairs $map(k_1, v_2) \rightarrow List(k_2, v_2)$, and *reduce()* produces a collection of values for pairs with the same key $reduce(k_2, list(v2)) \rightarrow list(v3)$.

The programming framework handles reading the input data and breaking it into key/value pairs. It then calls the user-provided *map()* function repeatedly and across a cluster, each time with a different input key/value pair. The framework is also responsible for collecting the intermediate key/value pairs that the *map()* function invocations produce throughout the cluster, grouping them together by keys, sorting them and finally feeding all the values associated with each key to an invocation of the *reduce()* function across the cluster. Finally, the framework collects key/value pairs generated by *reduce()* functions and stores them in a distributed file system, such as HDFS in case of Hadoop.

2.2 CIDA Adaptations

As mentioned in the previous section, there are two broad research directions that aim to increase the performance of MapReduce frameworks for CIDA

and to some extent HPC applications, as presented in the following two subsections.

2.2.1 GPU Based Adaptations of MapReduce

Algorithms that have enough data-independence in their parallel execution (and thus locality) are suitable targets for implementation on hardware accelerators such as GPUs. A great amount of research in the past few years have been spent on identifying such compute-intensive applications and porting them to GPUs, with different levels of ease and success.

The data independence requirement of GPU-friendly applications is similar to the data independence requirement of *map* tasks in MapReduce. As such, aside from typical cluster implementations [20, 21], other implementations of this framework have emerged for other platforms as well. In Section 9.1 we have listed a few such frameworks.

Fueled by all the research activity in the GPU field, in November of 2010 Amazon® took the first steps towards integrating GPUs in clouds by providing GPU-enabled EC2 instances. However, the current offering is only based on providing infrastructure-as-a-service, and does not provide any help to cluster programmers.

Later in this document we describe our MapReduce on GPU clusters framework, MITHRA, which even up to now remains one of the most integrated solutions in adapting GPUs to MapReduce infrastructure [22, 23].

2.2.2 Reducing MapRedue Overhead

Quite a few recent publications have shown the overheads of MapReduce in scientific HPC or CIDA computational problems. These reports have mainly focused on evaluation of MPI applications in cloud-based infrastructure and compare them with traditional HPC supercomputers, and have been mostly pessimistic. Walker [24] conducted the first study on HPC in Cloud by benchmarking Amazon EC2. Many others [5, 6] performed similar evaluation of Amazon EC2 for HPC applications using benchmarks. He et al. [25] extended previous research to three public cloud computing platforms and used a real application in addition to running classical benchmarks and com-

pared the results with that from dedicated HPC systems. Ekanayake et al. [4] compared applications with different communication and computation complexities and observed that latency-sensitive applications experience higher performance degradation than bandwidth-sensitive applications. Perhaps the most comprehensive evaluation to date was performed under the US Department of Energy (DoE) Magellan project [2, 7]. Jackson et al. [7] compared conventional HPC platforms to Amazon EC2 and used real applications representative of the workload at a typical DoE supercomputing center. They concluded that the interconnect and I/O performance on commercial cloud severely limits performance and causes significant variability. Napper and Bientinesi [8] performed cost evaluation and concluded that clouds cannot compete with supercomputers based on the metric $/GFLOPS, since memory and network performance is insufficient to compete with existing scalable HPC systems.

Gupta et al. [9] studied the performance-cost trade-offs of running different applications on supercomputer vs. cloud, and showed that cloud can be a cost-effective platform for *some* but not *all* applications. In addition, [9] demonstrated that the optimal platform for an HPC application depends upon application characteristics, performance requirements and user preferences.

On the other hand, only a few research projects have started to look into reducing the overhead of MapReduce frameworks in cloud environment for CIDA and HPC problems. One common direction among some of them is identifying *iterations* as a fundamental building block of many HPC and CIDA algorithms, and they have identified various overheads that a MapReduce implementation of an iterative algorithm imposes. Each of these projects has tried to address these issues slightly differently. In Section 9.2 we give a list of research efforts that try to tackle them in various ways.

2.2.3 GPU Optimizations

With the assumption of MapReduce style functions (whether one-pass execution model of MITHRA or iterative model of PIC), certain optimizations become possible to achieve better performance on GPU resources. We will

describe a code transformation technique developed to optimize the performance of parallel independent functions on a GPU. The specific bottleneck that we target is code flow divergence, which can result in code serialization. Using our technique, called *Loop Maximizing*, we perform a code-transformation that ensures uniform execution across all function invocations.

We will also describe a GPU memory allocation library called *Plasma*, which is optimized for use with MITHRA. Plasma allows easy dynamic allocation and deallocation of very fast but small memory arrays. Due to the specific architecture of GPUs, this is something that the compiler cannot perform at the moment. To achieve this goal, Plasma uses the *shared memory* resources of the GPUs, and provides a set of wrappers that can be used to easily access and use small arrays. It should be noted that Plasma is a stand alone CUDA library, and can be used for non-MapReduce style functions. However, the independent execution of threads should be present in any usage of this library.

CHAPTER 3

ADAPTATIONS OF MAPREDUCE TO MUD AND ITERATIVE CONVERGENCE DOMAINS

In this section we provide the motivations behind MITHRA and PIC, and describe their programming model. We also introduce a technique called loop maximizing which helps towards optimizing parallel execution in GPUs that are targeted by MITHRA, as well as the Plasma library for dynamic memory allocation in GPUs.

3.1 MITHRA: Running MUD Algorithms on a GPU Cluster

CIDA computing problems are typically bound by available computing cycles rather than I/O activity. MapReduce and Hadoop perform well in I/O-bound problems, for example *word count, inverse indexing* and *distributed grep*, where several simple operations are performed over large data sets. On the other hand, CIDA problems exhibit locality in many cases, and it would be advantageous to provide provisions in MapReduce frameworks to support in memory storage of intermediate key value pairs.

In 2004, Philipp Colella described seven scientific computing core kernels, the so-called seven *dwarfs* of parallel programming [26], that can represent a majority of the scientific computing algorithms. These dwarfs include the following algorithm classes: *Dense Linear Algebra, Sparse Linear Algebra, Spectral Methods (FFT), N-body methods, Structured Grids, Unstructured Grids and Monte Carlo.* The Monte Carlo method is often used when the model is complex, nonlinear, or involves more than just a couple uncertain parameters [27]. Monte Carlo methods are versatile enough to cover a wide problem domain including the simulation of galactic formation to business risk analysis to solving systems of linear equations [28].

In the MITHRA project, we show that the Monte Carlo algorithm class

can map very well onto a cluster of computers with GPUs, programmed using the MapReduce programming model. From a theoretical point of view, the Monte Carlo dwarf is equivalent to the MapReduce model. However, in practice, one often encounters secondary effects which might adversely affect performance on different architectures. MapReduce, as discussed in this section, can be deployed on a commodity cluster of workstation-class machines sporting moderately priced GPU cards. From our experiments, we demonstrate that this architecture can perform very well on compute-intensive Monte Carlo jobs.

The MITHRA project has the vision that MapReduce can be used beyond the typical big-data enterprise tasks, and that it can be used to ease the burden of parallel programming for CIDA computations. Furthermore, it targets clusters of GPUs as its execution environment to maximize its performance.

3.1.1 MUD: Massive Unordered Distributed Algorithms

Here we borrow a formalism provided in [29], which introduces a simple formal model for a class of algorithms referred to as *MUD* (massive unordered distributed) algorithms.

A MUD algorithm is formally defined as a 6-tuple mathematical structure $m = (\Phi, \oplus, \eta, \Sigma, (K, V), \Gamma)$. $\Phi, \oplus$ and η are functions that the framework programmer provides; $\Sigma, (K, V)$ and Γ are sets acting as the domain and image sets of the former functions. Members of the (K, V) set are tuples of the form (k_i, v_i).

The function $\Phi : \Sigma \rightarrow (K, V)$ can map an input item σ from its domain set Σ to a list of $(key, value)$ tuples, where the list can contain 0 or more items. The Φ function corresponds to the "map" function written in Google's MapReduce framework or Hadoop, and (K, V), being the image set of Φ, is the intermediate key/value domain.

The aggregator $\oplus : (K, V) \times (K, V) \rightarrow (K, V)$ is a binary operator that maps two items from the set (K, V) to a single item as follows: $(k, v_1) \oplus (k, v_2) \rightarrow (k, v_3)$. The $\oplus$ aggregator therefore "reduces" the intermediate key/value pairs to a single key/value pair for each key. Notice that we do not limit the $\oplus$ aggregator to be commutative and associative, unlike the formalism in list homomorphisms for the monoid used for reduction [30].

However, the output can depend on the order in which $\oplus$ is applied.

Let T be an arbitrary binary tree with n leaves. We use $m_T(x)$ to denote the $(k, v) \in (K, V)$ that results from applying $\oplus$ to the sequence $\Phi(x_1), ..., \Phi(x_n)$ along the topology of T with an arbitrary permutation of these inputs as its leaves. Notice that T is not part of the algorithm definition, but rather, the algorithm designer needs to make sure that $m_T(x)$ is independent of T. This is implied if $\oplus$ is associative and commutative; however, this is not necessary [29].

The optional post-processing operator $\eta : (K, V) \rightarrow \Gamma$ produces the final output. The overall output of the MUD algorithm is then $\eta(m_T(x))$, which is a function $\Sigma^n \rightarrow \Gamma$, where Σ^n means the Φ function is applied on n members of the Σ set. We say that a MUD algorithm computes a function f if $\eta(m(\cdot)) = f$ for all trees T; in other words, the intermediate key/value pair ordering for each key would not change the outcome of the computation.

MITHRA implements the Monte Carlo class of algorithms [26] as a case study for MUD algorithms, and shows that it can map very well onto a cluster of computers with GPUs, programmed using the MapReduce programming model. The Monte Carlo method is often used when the model is complex, nonlinear, or involves more than just a couple uncertain parameters [27]. Monte Carlo methods are versatile enough to cover a wide problem domain including the simulation of galactic formation to business risk analysis to solving systems of linear equations [28]. In Section 5.1.1, we describe how Monte Carlo simulations are implemented in MITHRA as a case study.

3.2 PIC: Partitioned Iterative Convergence

3.2.1 Introduction to PIC

The explosion of digital data in various forms (referred to as BigData) has led to a burning need for technologies that efficiently process and make sense of large amounts of data. Algorithms from a wide range of emerging domains, including data analytics, web search, social networks, and recognition, mining and synthesis (RMS) [31], build models from a large corpus of unstructured input data. To do so, they often use *iterative convergence* (IC) algorithms. These algorithms typically build and refine a model from

a large corpus of unstructured data. The model is computed by generating a sequence of increasingly accurate solutions, starting from an initial guess, until a convergence criterion is satisfied.

Due to the scale of data sets that they process, IC algorithms are frequently realized on clusters using high-level programming models like MapReduce [20]. For example, the Apache Mahout [13] project provides implementations of a wide range of IC algorithms using the Hadoop framework [21]. In these conventional implementations of IC algorithms on MapReduce, the data-parallel computations performed in each iteration are expressed as one or more MapReduce jobs. This approach offers a quick path to implementation and leverages the strengths of MapReduce such as ease of programming, load balancing, and fault tolerance. However, recent research has demonstrated that cluster implementations of iterative-convergence algorithms based on MapReduce suffer from performance degradation [1, 2, 3, 4, 5, 6, 7, 8, 9]. The performance degradation has been attributed to repeated reads of input data and repeated creation and termination of MapReduce jobs in each iteration [10, 11, 12]. These issues have been addressed by modifying the MapReduce framework to provide mechanisms for caching invariant data [10, 11, 12], long-running tasks [10], and by making the run-time scheduler loop aware [11], resulting in performance improvements.

Here we identify two key bottlenecks not addressed by previous work that limit the performance of cluster implementations of IC algorithms with MapReduce. The first bottleneck is the large volume of intermediate data (also referred to as the shuffle data) between map and reduce tasks in each iteration. Note that, due to the all-to-all nature of the shuffle traffic, it stresses the cluster bisection, a resource that is both scarce and difficult to scale [21, 32]. Second, in applications where the model itself is large, there is a large amount of network traffic due to model updates in each iteration. These bottlenecks cannot be alleviated by previous work [10, 11, 12], since the shuffle data and model are not constant and therefore cannot be cached.

We propose a new approach called *partitioned iterative convergence* (PIC) to express iterative-convergence algorithms for parallel execution, and describe a programming framework for PIC that greatly improves performance while preserving the other benefits of MapReduce. We exploit a key property of most IC algorithms - that they start with an arbitrary initial model (often chosen randomly) and produce an acceptable model at convergence.

However, the time to convergence depends on the specific choice of the initial model. The key insight in PIC is to view iterative convergence computations as a two-phase process. In the first phase, we use a transformed version of the original computation to quickly produce a good initial model. Then, in the second phase, we use this model as the starting point and further refine the model by using only a few iterations of the original computation.

In the first phase of PIC, which we refer to as the *best-effort phase*, we partition the problem (input data and model) into a number of smaller model-building sub-problems, by using a programmer-specified `partition` function. Although there are typically dependencies across sub-problems, we ignore these dependencies and perform independent iterative-convergence computations, in parallel, and without any synchronization or communication between them (we refer to these as local iterations). The models computed by the sub-problems are combined using a programmer-specified `merge` function into a single model. The above process is repeated with the new, single model as the starting point (we refer to this process as best-effort iterations), until the single model satisfies a best-effort convergence criterion. In the second phase of PIC, which we refer to as the *top-off phase*, we refine the model computed in the best-effort phase, by using iterations of the original computation (i.e., without partitioning or ignoring dependencies) until convergence.

Restructuring iterative-convergence computations as espoused by PIC addresses the performance bottlenecks in MapReduce due to intermediate and model updates, since (i) there is no traffic across partitions while the local iterations are executed in the best-effort phase, (ii) while cluster-wide communication is required once during each best-effort iteration in the best-effort phase, the number of best-effort iterations is typically quite small since most of the model refinement is accomplished by the local iterations, and (iii) the number of iterations required in the top-off phase is quite small in practice. Therefore, as shown in our results, PIC implementations achieve significant performance improvements over the state-of-the-art.

Our proposal requires no modifications to the underlying MapReduce runtime framework (we build on top of the MapReduce framework). Therefore, previous optimizations of MapReduce frameworks for iterative algorithms [10, 12, 11] can be fully leveraged. In addition, the effort required to migrate conventional implementations into the PIC framework is small since the original implementation is fully re-used to solve the sub-problems

in the best-effort phase, as well as to realize the top-off phase. Finally, we note that un-partitioned iteration execution (second phase of PIC) is merely a special case of partitioned execution (first phase of PIC) where the number of partitions is set to 1, and does not incur any duplication of programming effort.

To evaluate the benefits of our proposal, we have developed a library for PIC on top of the Hadoop MapReduce framework [21]. We have implemented five popular iterative-convergence algorithms (PageRank, K-Means clustering, neural network training, linear equation solver and image smoothing) using PIC. We compare the performance of PIC implementations to conventional MapReduce implementations that are already optimized to eliminate the overheads of repeated job creation and reads of input data [10, 12, 11]. Our results demonstrate that PIC achieves speedups of 2.5X-4X across clusters of 6-256 nodes.

3.2.2 Motivations for PIC

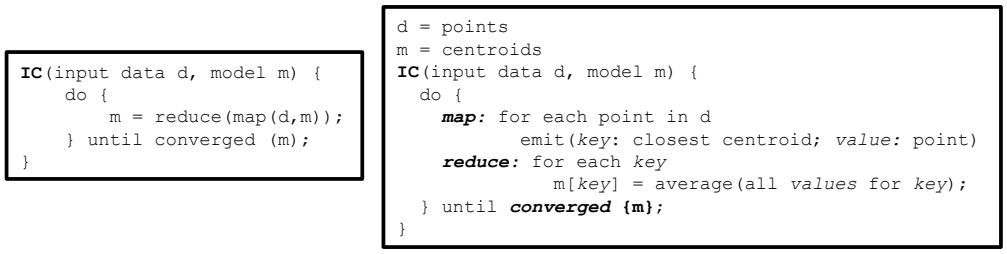

Figure 3.1: Conventional implementation of an iterative convergence algorithm using MapReduce: (a) generic template, and (b) K-means clustering example.

Figure 3.1(a) shows a generic template of a MapReduce implementation of an iterative convergence algorithm. The iterations of the do-until loop generate successive versions of the model. In each iteration, the input data (d) and the current model (m) are processed in a data parallel manner by a MapReduce job to update the model (in general, multiple chained MapReduce jobs may be used). The map functions produce intermediate data (key-value pairs). The reduce functions process the intermediate data to generate the

next update of the model. Although the input data does not change from one iteration to the next, the model data is updated after every iteration. A convergence criterion is evaluated on the model to decide when to terminate the iterations. Figure 3.1(b) shows how the popular K-means clustering algorithm [33] is realized by using the template of Figure 3.1(a). The clustering algorithm partitions a set of points into clusters where each cluster is represented by a *centroid*, which is the average of points in the cluster. The map computation associates each point with the centroid that it is closest to, while the reduce computation recomputes the values of the centroids. The computation is repeated until the centroids do not change by more than a threshold from the previous iteration, or if no more than a certain fraction of a points change clusters from the previous iteration.

We have observed that MapReduce based implementations of iterative-convergence algorithms suffer from performance degradation due to the following factors.

- **MapReduce intermediate data.** The intermediate key-value pairs have to be communicated across the cluster interconnect because of the all-to-all nature of this communication. Despite well-known optimizations (use of combiners and overlapping shuffle with the Map phase [20]), a large volume of intermediate data is quite common, with adverse impact on application performance.

- **Model updates.** Since we update the model in each iteration, it is necessary to synchronize at the end of each iteration and communicate data across the cluster interconnect to update the model. Note that the model is stored in the cluster file system with replicas (for fault tolerance), hence the performance impact of frequent model updates is significant, and severe for applications where the model is large.

Note that the above factors are over and beyond the issues addressed in previous work [10, 12, 11], such as repeated initialization of the MapReduce runtime and repeated reads of constant input data. Addressing these factors will therefore require new techniques that go beyond these prior proposals.

The key insight motivating the proposed work is that IC algorithms produce acceptable results while starting from an arbitrary model. However, the number of iterations necessary for convergence is strongly dependent on the

initial starting point. Therefore, we view iterative-convergence computation as a two-phase process that consists of (i) coming up with a good starting point (initial model), and (ii) refining the initial model to generate the final solution. Of course, determining a good initial model, in general, can be as difficult as finding the solution in the first place. Moreover, we do not want to burden the programmer with the task of coming up with a new algorithm to generate an initial model. In this work, we demonstrate that breaking up the original problem into sub-problems, and executing iterative convergence on the sub-problems in a loosely coupled manner can generate a very good initial model with drastic improvements in efficiency. The second phase starts off with the model produced by the first phase and runs the unmodified iterative convergence computation. However, this phase now requires far fewer iterations than a conventional implementation.

Figure 3.2 quantitatively demonstrates this insight for K-means clustering of 100 million points into 100 clusters. The graph on the left compares the execution time of the conventional MapReduce that follows Figure 3.1(b), with the implementation using the proposed PIC framework (described further in following sections). The execution time for the PIC case is broken down into the time consumed in each of the phases. The graph on the right shows a similar comparison for the total volume of intermediate data and model updates. The graphs show that (i) the best-effort phase executes in around one-fifth the time as the conventional implementation, primarily due to a drastic reduction in cluster interconnect traffic due to shuffle and model updates, and (ii) the top-off phase requires around one-sixth the number of iterations of the conventional implementation, resulting in proportionally lower execution time and traffic. Overall, the PIC implementation achieves around 3X speedup over the conventional implementation.

In the next section, we present a new programming framework called partitioned iterative convergence (PIC) that embodies the two-phase approach described above. We note that the PIC framework is built on top of MapReduce. Therefore, we leverage the inherent advantages of the MapReduce programming model, as well as all prior improvements [10, 12, 11] to the MapReduce run-times.

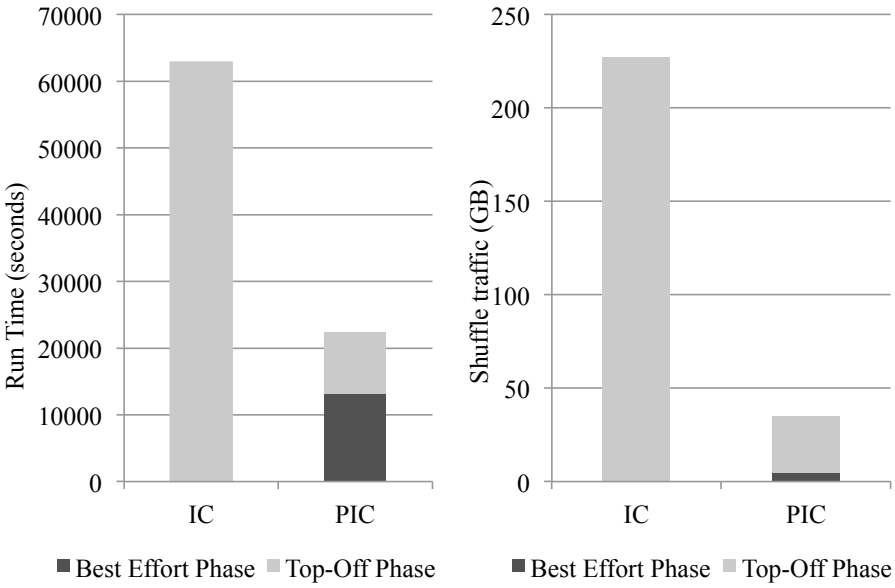

Figure 3.2: Run time and Shuffle traffic for K-means clustering (100 million points into 100 clusters, 64-nodes cluster).

3.2.3 PIC Model

```
Partitioned_IC(input data d, model m) {
    do {
      partition(d, m); //p1 … pz  partitions
        for each partition pi{
          IC(di,mi)
        }
      m = merge(m1, m2, … , mz);
    } until converged (m);
}
```

Figure 3.3: Partitioned iterative-convergence: A programming template.

Figure 3.3 presents a generic template for an algorithm expressed using the proposed partitioned iterative convergence (PIC) approach. Given an input data set (d), and an initial model (m), the best-effort phase partitions the input data and the model to create several, smaller model-building sub-problems. Dependencies across the sub-problems are initially ignored, and conventional iterative convergence methods are used to solve the sub-problems. The solutions to the sub-problems (i.e., updated models) are then merged using problem-specific merge functions to create a single, merged model. If the merged model does not meet a convergence criterion, we continue the best-effort phase. Otherwise, we end the best-effort phase and enter the *top-off phase*. This second phase further refines the model produced by

the best-effort phase by using conventional implementations of iterative convergence (i.e., without partitioning or ignoring dependencies). We end the top-off phase when the model satisfies the specified convergence criterion.

3.2.4 Best-Effort Phase

The best-effort phase is key to the performance improvements achieved by PIC, and is governed by three functions (`partition`, `merge`, and `BE_converged`). The application developer can either implement these functions, or use the default implementations in the PIC framework.

The `partition` function is useful to control the number and size of the sub-problems, as well as the degree of parallelism. For example, we can create more sub-problems than the number of nodes available in a cluster. Or, we can size a sub-problem so that a group of tightly-coupled nodes (e.g., a rack) can execute the sub-problem. However, more sub-problems of smaller size can increase the number of best-effort iterations that the best-effort phase may require to converge. For a given number of sub-problems, the partitioning function should try to reduce the dependencies between the sub-problems so that the number of best-effort iterations, as well as the number of iterations required in the top-off phase, are minimized.

The specific choice of the `partition` function is application-dependent, much like the contents of map and reduce in the MapReduce framework. In some problems (for example, the PageRank case study in Section 5.2.5), we partition both the input data and the model. In other cases (for example, the K-Means case study in Section 5.2.4), it is more appropriate to partition the input data, but create multiple copies of the model. The complexity of the `partition` function may range from simple techniques like randomly breaking up the input data and/or model (in which case the programmer can simply use the default partitioner classes provided by PIC), to sophisticated partitioning schemes such as min-cut graph partitioning.

The `merge` function depends on the strategy used by the `partition` function. For example, if the `partition` function divides the model into disjoint parts that are updated by the different sub-problems, then the `merge` function may simply piece them back together. On the other hand, if copies of the model are created by the `partition` function, then they may be "averaged"

or aggregated to construct the merged model. Similar to the `partition` function, the programmer can either specify an application-specific `merge` function, or use one of the defaults provided by PIC. For models that can be represented as vectors, the default merge functions can concatenate the vectors from sub-problems into a single vector, sum the vectors, or average the respective entries in the vectors.

Finally, PIC uses the `BE_converged` function to determine if the best-effort phase can be terminated. In principle, the application developer can use the same criterion that they specified using the `converged` function of the conventional implementation, or they can specify a much looser criterion to quickly terminate the best-effort phase.

In Chapter 5, we provide examples of how these three functions are specified for real applications.

Figure 3.4 compares the execution flow of the best-effort phase of PIC with the conventional implementation of Figure 3.1(a). By processing sub-problems while ignoring their dependencies, communication traffic on the cluster interconnect due to shuffle data and model updates is drastically reduced. As we show in Section 8, this leads to drastically improved performance in the best-effort phase, while providing a very good starting model for the top-off phase. From a different perspective, conventional MapReduce implementations can only exploit parallelism within each iteration. The best-effort phase of PIC introduces an additional degree of parallelism - sub-problems that can be solved independently - beyond the opportunity of exploiting parallelism within each iteration of a sub-problem. By increasing the amount of parallelism, the best-effort phase can scale more easily than the conventional implementation.

Finally, an important special case of the best-effort phase of PIC is worth noting. If the number of partitions is one, the `merge` function becomes the identity function (i.e., the `merge` function returns the only model it receives), and the `BE_converged` function terminates the best-effort process after only one iteration, the best-effort phase of PIC degenerates to the conventional implementation of Figure 3.1(a). This is important because it implies that PIC requires no additional programmer effort to realize the top-off phase.

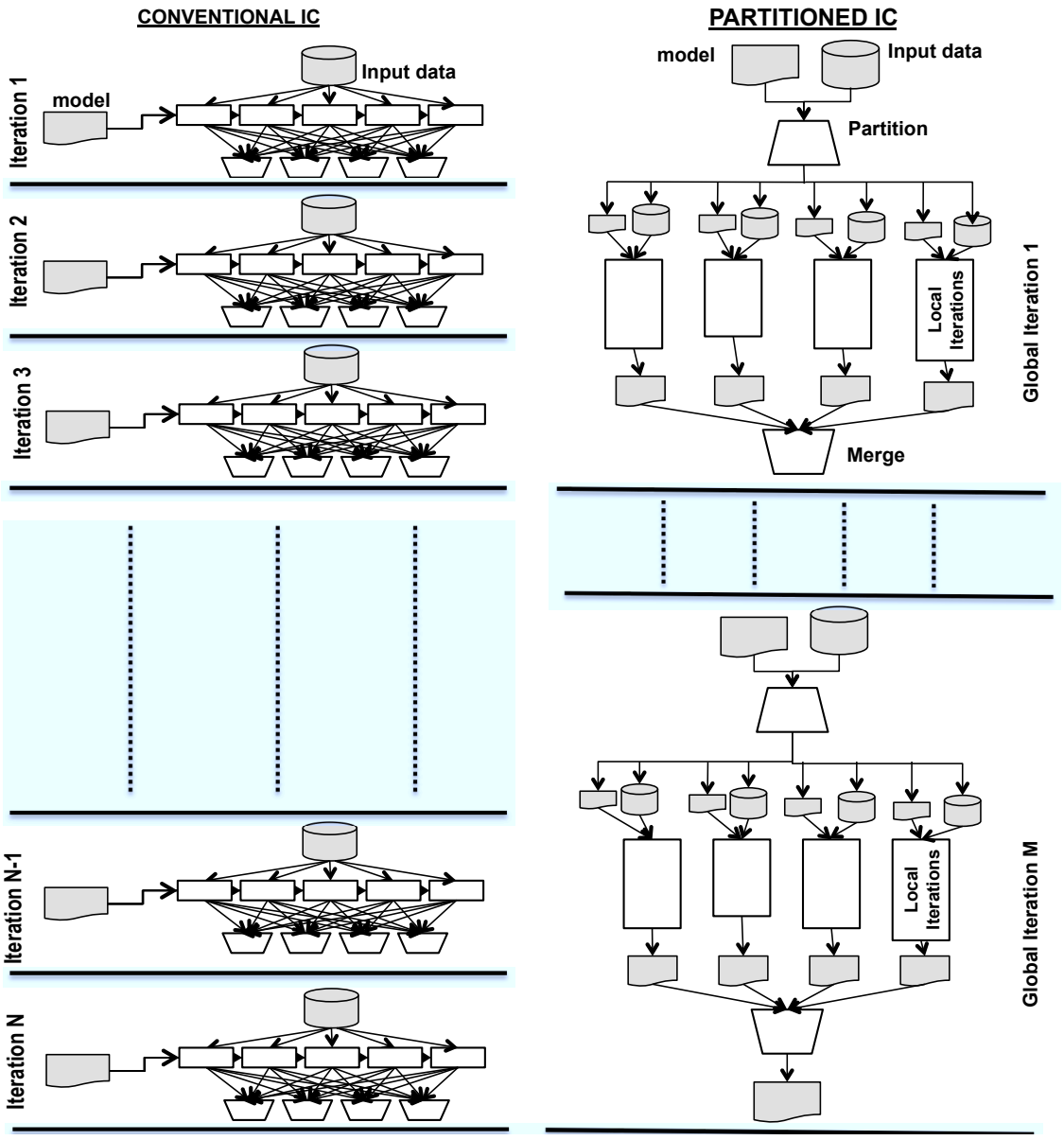

Figure 3.4: Comparing the conventional iterative convergence and the best-effort phase of PIC.

3.2.5 Programming with PIC

In MapReduce, the map function specifies the computation to be performed on one element of the input data, leaving the runtime framework to partition the input data set and invoke the map function on each data element. However, these semantics are insufficient for iterative convergence methods because they also have a need to:

- Pass the model, as well as the input data to map functions.
- Keep two copies of the model - current and previous - to evaluate the convergence criterion.
- Replace the model from a previous iteration with the model computed in the current iteration.

In addition, the best-effort phase of PIC also has a need to:

- Generate models for each sub-problem, starting from a single, unified model at the beginning of a best-effort iteration.
- Collect models from sub-problems to implement the `merge` function.

While the above operations on the model can be implemented by application code that uses a conventional MapReduce framework like Hadoop, this is a significant burden on the application developer.

The best-effort phase can further introduce a burden on the programmer. Many implementations of `partition` and `merge` functions require that the model be divided into elements that are uniquely identifiable and operable. Given two models that need to be merged, we may have to first establish the correspondence of elements in the two models. For example, consider the case of K-means. Assume that we have two sets of centroids (models) that were computed by two different sub-problems. If the `merge` function requires that corresponding centroids be averaged, then we have to keep track of which centroid in one model (centroid set) corresponds to which centroid in the other.

PIC only requires that the model be expressed in the form of key/value pairs to facilitate splitting a model into elements and identification of correspondence between elements. Representing the model as key/value pairs also allows the `merge` function itself to execute in a distributed fashion as a MapReduce job.

3.3 Loop Maximizing

It can be seen that a major motivating factor in the development of both MITHRA and PIC is to move the *map()* and if possible *reduce()* functions to run in GPUs. In MITHRA, we have shown how compute-intensive *map()* functions can be executed on GPUs. PIC has successfully created the foundations of moving the *reduce()* functions into GPUs as well, by running the local map and reduce functions on the same GPU a few iterations before a global merge phase. Given the typical *map()* function templates, the program structure has to be simple. For example, *map()* functions usually are not recursive or do not use complex dynamic function calls. As such they are almost suitable for GPU execution. The same holds for *reduce()* functions, which are now feasible to be ported to GPUs thanks to PIC.

However, when porting code to GPUs, there are other practical implications that one has to consider. A prominent challenge of porting multi-threaded programs to GPUs is diverging code flows in different threads. Using conditional branches or data-dependent loops would usually result in flow-divergent code. This is a serious problem for a SIMD machine such as a GPU. The whole reason for the massive parallelism potential of SIMD machines is that they sacrifice independent control logic circuity for more computational units. When faced with diverging flows in different threads, a GPU has to revert back to serial execution of each thread, reducing the utilization of hardware processing units by orders of magnitude. As such, diverging code flows would make use of the parallelism potential of GPUs impossible.

It is possible to handle different code branches that are due to conditional operators such as "*if...else*" conditions, for example by using predicated execution of instructions. GPUs have had hardware support for predication execution for some time, and many compilers (including the Open64 compiler back-end that is used in the NVIDIA CUDA SDK) can transform conditional branches into predicated code. On the other hand, the situation is much harder when different threads run the same loop with different number of iterations. The situation is worse when the number of iterations is dependent on the input data values.

An example of this sort of algorithm can be found in some dynamic programming algorithms, where a matrix is used to memoize partial solutions.

Filling the matrix has a deterministic flow, by scanning the matrix cells from top-left to bottom-right. Each cell can be computed using the matrix values on its top, left and top-left diagonal cells. This homogeneous algorithmic pass is easy to implement in GPUs. However, once finished, the values of the matrix elements are used in other passes, where the execution flow is dependent on the values of the matrix. For instance in edit distance or gene alignment computations, the second pass is a traceback from the bottom-right element of the matrix to the top-left. The number of steps that the traceback requires depends on the matrix values. There have been many attempts to implement gene alignment on GPUs in the last few years [34, 35, 36, 37], but most of them have reported sub-optimal results due to this problem.

Loop maximizing is a novel code transformation technique that can potentially address the issue of flow-divergence [18]. The idea is simple: change the code so that every thread takes the maximum number of loop iterations. If a certain thread requires less than the maximum iterations, it will run dummy instructions. To do this, condition variables (as opposed to conditional statements) are defined and embedded into the arithmetic functions. This way, we can ensure that when the condition is set, the impact of the computations is nullified before affecting the variables inside the loop.

This process is depicted in Figure 3.6. At first, we have a *while* loop that is at most running for 20 iterations. In the first step, we change the data-dependent *while* loop to a data-independent *for* loop, by introducing a condition variable *cond* and a conditional *if* statement to use it. At the next step of the transformation, we eliminate this conditional statement by embedding the condition variable inside the arithmetic expression. For this latter transformation, we can generally substitute

$$\text{if } (cond) \text{ then } a = f(x) \text{ else } a = g(x)$$

with this equivalent transformation:

$$a = cond \cdot f(x) + !cond \cdot g(x)$$

This transformation is feasible since in the C language conditions can be evaluated as integer numerical values, and for instance are evaluated to zero when condition is FALSE.

To further describe describe loop maximizing, we consider the Needleman-Wunsch gene alignment algorithm described in [18], and shown here in Figure 3.5.

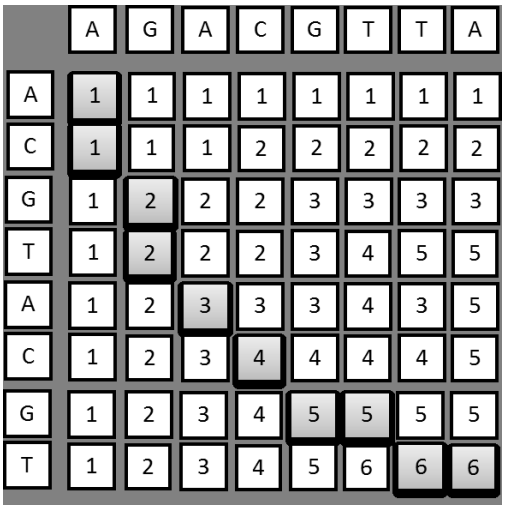

Figure 3.5: Global alignment using Needleman-Wunsch algorithm. The traceback path is identified in this picture with grey matrix elements. The traceback phase of this algorithm is data-dependent and therefore flow-divergent and not suitable for GPU implementation. Loop maximizing can solve this problem.

This dynamic programming algorithm consists of two passes. In the first pass, a matrix is filled by scanning it from top-left to bottom-right. During the traceback pass, the algorithm computes condition variables that evaluate to the zero value once the traceback reaches either the left or top boundary of the quadrant. These condition variables are then embedded in the expressions that find the next cell in the traceback path. In effect, once the traceback reaches either of the left or top boundaries, it gets *stuck* there until the $2 \cdot k - 1$ traceback steps are finished. Using this mechanism, the same set of instructions are executed in each thread, eliminating any flow divergence among different threads.

This solution ensures that any combination of input arguments take the *worst case path*. However, we utilize a GPU specific mechanism to help the situation. The SIMD execution model of NVIDIA GPUs runs different threads in warps of 32 threads in each SM. Therefore, once all 32 threads of the same warp get *stuck*, we can break the loop and move on to the next phase, instead of waiting for all of them to reach $2 \cdot k$ steps. NVIDIA

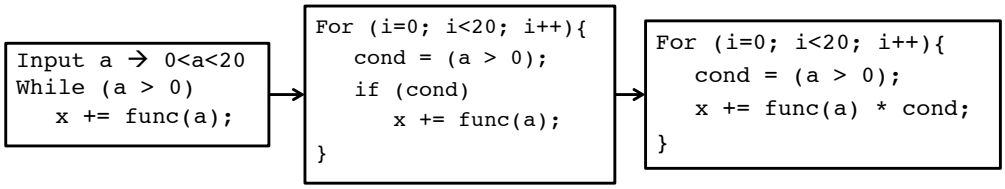

Figure 3.6: An example of loop maximizing in two steps.

GPUs of compute capability 1.3 or higher provide and intrinsic function *"__all(condition)"*, which evaluates to TRUE once all the individual *conditions* of the warp's threads become TRUE. Using this mechanism helps our algorithm to run faster.

Using the above techniques, our algorithm runs completely divergence-free, and not a single instruction is serialized during execution.

3.4 Plasma Memory Allocation Library

A few fast memory resources are available in GPUs: Registers, shared memory, caches and texture memory. However, not all of them can be used for dynamic allocation/deallocation, run-time indexable access and read/write.

Texture memory can be fast since it is cached, but is limited to read-only operations. Registers are the fastest on-chip storage, but they are not array addressable in general. Under certain circumstances, and if the compiler can identify every array index statically at the compile time, it can store the individual array elements in different registers. For example a simple loop such as this can be stored in the registers:

```
uint32_t array[8];
for (i=0; i<8; i++) sum+=array[i];
```

But as soon as the compiler cannot determine a static value for an index, it decides to put the whole array in the local memory, which physically resides outside the GPU chip and on the GDDR5 memory banks subject to bus bandwidth limits and coalescing requirements. Fortunately the Fermi architecture introduced on-GPU L1 and L2 caches. Each streaming multiprocessor (SM) on the chip has a minimum of 16KB L1, which can be increased to 48KB at the expense of shared memory. The results are much

better than the previous Tesla architecture, but as we will see in this thesis still not optimal.

This leaves us with **shared memory**, which is a very fast on-chip memory, and can be readily used as arrays. However, it is shared among all the threads running in the same block, which is not always desirable. If the program does not require actual information sharing, its shared nature makes the programming harder. Furthermore, the shared memory can observe decreased performance if bank conflicts happen; i.e. different threads try to read different addresses that happen to be placed in the same shared memory bank.

CHAPTER 4

API DESIGN: USING THE MODIFIED MAPREDUCE FRAMEWORKS

4.1 MITHRA

In MITHRA, the input values can either be loaded from the HDFS, or if possible they are generated on the fly. This case happens in the Monte Carlo simulation algorithms, since they only require a set of random numbers as their Σ set. In the latter case, we generate quasirandom numbers using the Niederreiter quasirandom generator [38] in the GPUs. A quasirandom or low discrepancy sequence, such as the Faure, Halton, Hammersley, Niederreiter or Sobol sequences [38, 39] is less random than a pseudorandom number sequence, but more useful for such tasks as approximation of integrals in higher dimensions and in global optimization. This is because low discrepancy sequences tend to sample space more uniformly than random numbers. Algorithms that use such sequences may have superior convergence [38]. Creating quasi or pseudo random numbers is a very time-consuming task, so it is best to create the random numbers in the GPUs where they will be consumed in a distributed manner to utilize all the processing elements available, in contrast to creating them directly in the CPU using a sequential algorithm and then transferring them to the GPU memory through the low bandwidth PCI-Express bus. We base this part of our work on an implementation of this algorithm from the CUDA SDK [40]. In the former case (where data is loaded from HDFS), the Hadoop framework distributes Σ among the nodes of the cluster. The Hadoop framework divides up the data determined to be processed on that node to smaller chunks of data (typically between 64MB to 256MB in our implementation, which is decided by HDFS configuration). These chunks are small enough to fit in the GPU global memory.

The *map()* function in MITHRA is designed as a two stage process. The first phase makes use of the Hadoop *map()* function. The Hadoop frame-

work distributes the user-supplied *map()* function across the cluster. However, the main functionality of the Φ function in the MUD formalism is not programmed in the Hadoop *map()*. Hadoop's *map()* merely acts as a distribution mechanism for the Φ workload across the nodes of the clusters.

Whether the inputs are loaded from HDFS or generated in the GPUs, by the time the user-provided *map()* functions that are written in CUDA are ready to run, the input dataset is loaded in the GPU global memory and ready for execution.

The Φ function in a MUD algorithm, being data independent in its multiple evaluations on its input data set Σ, is performed on GPUs. This is the second phase of the *map()* function design in MITHRA, in which the programmer writes a CUDA kernel to work on one $\sigma \in \Sigma$. The user supplied Φ function runs on the GPU, processes the input values and emits intermediate key/value pairs.

The next part of the framework involves intermediate key/value management. Focusing on the Monte Carlo simulation class of problems, we did not need to use more than a single key for the intermediate key/value pairs; therefore, the implementation of the key/value management scheme is not completely general and is limited to one intermediate key per input key/value pair. Once MITHRA collects the intermediate key/value pairs, it emits them back to the Hadoop framework, where they are gathered across the cluster, grouped by keys and are ready to be passed to *reduce()* Hadoop functions.

The last phase of a MUD algorithm is the application of the $\oplus$ aggregator on all the key/value pairs with the same key, and calculating the mean and variance of them. The reduction aggregator $\oplus$ is applied locally in each node through Hadoop's combiner functions, and the final round of $\oplus$ and $\eta(\cdot)$ application happens in Hadoop reducers.

4.2 Programming with PIC

Figure 4.1 summarizes the PIC programming model API that application developers can use to realize their iterative convergence applications. Note that, except for three functions (`partition`, `merge`, and `BE_converged`), all the other functions in the API are standard, in the sense that they are necessary to realize any iterative convergence application on a MapReduce

abstract class PIC_Main	This is the main entry class into the PIC library. The user has to sub-class it, create an object of its class and call the run() function to get a PIC job started.
Constructor (inputDataDir, inputModelDir, resultsDir, *partition*);	Identifies directories in HDFS where input data and model are stored, where the output models is to be stored, and whether the user is requesting the PIC library to partition the input data set. Otherwise users can partition the files themselves, (see PIC_Partitioner class).
BE_converged (Collector<T_k, T_v> oldModel, Collector<T_k, T_v> newModel);	provides the user with the old and new best-effort iteration models, easing the user's programming overhead.
abstract class PIC_Job	Mainly embeds the local computations.
map (T_k key, T_v value, Collector<T_k, T_v> model, Collector<T_k, T_v> output);	Similar semantics to Hadoop's map(). The current model is also provided to the map() function for ease of programming. Baseline IC implementations typically need extra programming to perform this functionality.
reduce(T_k key, Iterator<T_v> values, Collector<T_k, T_v> output);	Similar semantics to Hadoop's reduce().
converged (Collector<T_k T_v> oldModel, Collector<T_k, T_v> newModel);	Provides the user with the old and new local iteration models for a convenient convergence threshold decision.
merge (T_k key, Iterator<T_v> values, Collector<T_k, T_v> output);	Finds the corresponding elements of each sub-model, groups them together and passes them to the user to merge. This reduces the programmers task of identifying corresponding elements in different sub-problems.
abstract class PIC_Partitioner	This is intended for simple partitioning algorithms. The Library includes some implementation of this class, for example simple modulo partitioning and random partitioning. Alternatively the user can use more complex partitioners outside PIC and instruct PIC not to attempt partitioning.
partition (T_k key, T_v value, minPartitions);	should return the partition number for one input point.

Figure 4.1: The partitioned iterative convergence user-facing API. Most of the API is necessary for a MapReduce implementation of IC, so is not an extra addition for PIC. Only three extra functions, shown in *italics*, are needed for the best-effort phase. Also, our library is based on templates, and any Hadoop-provided data structure object (e.g. *Text*, *IntWritable*, *LongWritable*, etc.) can be used within PIC for key/value pairs.

framework. The `map`, `reduce` and `converged` functions of a conventional implementation (Figure 3.1(a)) correspond to similarly named functions in Figure 4.1.

In MapReduce, the map function specifies the computation to be performed on one element of the input data, leaving the runtime framework to partition the input data set and invoke the map function on each data element. However, these semantics are insufficient for iterative convergence methods because they also have a need to:

- Pass the model, as well as the input data to map functions.
- Keep two copies of the model - current and previous - to evaluate the convergence criterion.
- Replace the model from a previous iteration with the model computed in the current iteration.

In addition, the best-effort phase of PIC also has a need to:

- Generate models for each sub-problem, starting from a single, unified model at the beginning of a best-effort iteration.
- Collect models from sub-problems to implement the `merge` function.

While the above operations on the model can be implemented by application code that uses a conventional MapReduce framework like Hadoop, this is a significant burden on the application developer. For example, let us consider the requirement that the model should be passed to each invocation of the map function. Since the map function in MapReduce only accepts one input (the input data) and cannot have any best-effort state, inconvenient workarounds are typically used to pass the model to the map function. For example, programmers commonly read the model from a fixed location in the underlying cluster file system in the constructor of their mapper objects, so that by the time the run-time framework calls their `map` function, the model is available. This task requires directly accessing the distributed file system while keeping track of which files and directories contain the model from the last iteration, and reduces the simplicity of the MapReduce semantics.

The best-effort phase can further introduce a burden on the programmer. Many implementations of `partition` and `merge` functions require that the

model be divided into elements that are uniquely identifiable and operable. Given two models that need to be merged, we may have to first establish the correspondence of elements in the two models. For example, consider the case of K-means. Assume that we have two sets of centroids (models) that were computed by two different sub-problems. If the `merge` function requires that corresponding centroids be averaged, then we have to keep track of which centroid in one model (centroid set) corresponds to which centroid in the other.

The PIC library and API significantly ease the implementation burden on an application developer. For example, the programmer need not worry about storing the models to, and reading them from, the cluster file system. PIC only requires that the model be expressed in the form of key/value pairs to facilitate splitting a model into elements and identification of correspondence between elements. Representing the model as key/value pairs also allows the `merge` function itself to execute in a distributed fashion as a MapReduce job. PIC reads the models from the previous iteration and passes it to the map functions as a collection of key/value pairs. There is no limit on the number of the key/value pairs in a model. and it even allows duplicate keys.

Since PIC completely takes care of *model* management, we believe that developing iterative applications in PIC is arguably easier for an application developer than writing a conventional implementation in Hadoop.

There are three main methods of interacting with a *Model* object in PIC:

1. `Model.collect(key, value)` adds a key/value pair to the model.

2. `Model.keysIterator()` returns an iterator to all the keys stored in the model.

3. `Model.perKeyIterator(key)` returns an iterator to all the values which have the same key.

This design was chosen so that the programmer is not burdened with ensuring non-duplicate keys in their *model*. In practice we have found it an easy task to use a set of key/value pairs to represent models for different case studies.

A key objective in the design of PIC is to add minimal programmer overhead and maintain the ease of programming that is inherent to high-level

frameworks such as MapReduce. From a programmer's perspective, the process of porting a conventional MapReduce based implementation written in Hadoop to a PIC implementation involves two primary additional tasks - the specification of the `partition` and `merge` functions. Other than that, the `map`, `reduce` and `convergence` functions remain largely the same, as the PIC's programming syntax is designed to mimic that of Hadoop as much as possible.

As mentioned in the previous section, PIC takes care of *model* management, so developing iterative applications in PIC is arguably easier for a programmer than a baseline implementation in Hadoop. Therefore the map function semantics are enhanced to accommodate this functionality. Instead of `map(DataElement:key,value)`. PIC's map passes in the model object: `map(DataElement:key,value), Model)`. The `reduce` definition remains exactly the same.

The `partition` function creates data subsets from the input data set and model. The specific choice of this function is application-dependent, much like the contents of map and reduce in the MapReduce framework. In some problems, it is appropriate to break up both the input data and the model (e.g., PageRank, as described in Section 5.2.5); in other cases, it is more appropriate to break up the input data, but create multiple copies of the model (e.g., K-means clustering, as described in Section 5.2.4). The partitions may be strictly disjoint or contain some overlap (e.g., Image Smoother, as described in Section 5.2.1) in order to facilitate faster convergence. The complexity of the `partition` function may range from simple techniques like randomly breaking up the input data and/or model (in which case the programmer can simply use the default partitioner classes of PIC), to sophisticated partitioning schemes. For slightly more complex partitioning algorithms, the programmer can provide one function (by sub-classing a default PIC partitioner class) that gets one data element from PIC at a time, and in response the function should return the partition number for that element. For even more complex partitioning functions the programmers can either provide their own classes to the PIC library, or they can use external programs to partition data (for example METIS for graph partitioning) and instruct PIC to assume that the input is already partitioned and not to attempt partitioning. In general, the partitioning function should try to reduce the dependencies between the sub-problems for better results and so that the

number of global iterations required is minimized. Also, the sub-problems that are created should be of roughly the same size so that the computations on them are balanced. This is not essential if the number of partitions is considerably larger than the number of nodes in a cluster, since dynamic load-balancing techniques will compensate for variations in the computation time required per partition.

By default, PIC chooses to partition the input data only during the first global iteration since the input data is unchanged, and only the `merged` model is partitioned during subsequent global iterations. However as Figure 3.3 suggests, the programmer can request that the `partition` function executes in each global iteration, since some problems might benefit from repartitioning in every global iteration.

The `merge` function takes the models produced from different sub-problems and creates a unified model. The `merge` function depends on the application semantics as well as the choice of the `partition` function. For example, if the `partition` function divides the model into disjoint parts that are updated by the different sub-problems, the `merge` function may simply piece them back together. If copies of the model are created by the `partition` function, they may be "averaged" or otherwise aggregated to construct the unified model. In some problems (e.g., PageRank described in Section 5.2.5), the `merge` function is also used to perform additional computations that compensate for the effect of ignoring dependencies across partitions. In addition to improving the quality of the model, this may also facilitate faster convergence, thereby improving performance.

Representing the model as key/value pairs allows the `merge` function in PIC to be executed in a distributed fashion. Each instance of `merge` function receives one key and all the values associated with it. If the *models* are replicated versions of each other, they will have similar keys with different values at the end of a global iteration. In this case, corresponding elements of the models that have the same key will be passed to one instance of the `merge` function, which can aggregate them. On the other hand, if the model is partitioned such that each sub-model is representing separate parts of the whole model, they will be simply concatenated in the merge function. Specific examples of `partition` and `merge` functions are discussed in the next section for various example applications.

Finally, the local and global convergence functions in PIC are defined as

follows:

```
localConvergence (previousModel, currentModel)
globalConvergence (previousModel, currentModel)
```

In both cases, the PIC library takes care of model management, so that the programmer only needs to write the required logic to compare the two models and determine convergence.

4.3 Using the Plasma Library

The first function that the user needs to call is

```
int array = sm_malloc(size);
```

in which *size* is the number of bytes that the user requests for an array. The return value of this function is a pointer, which should be stored and used in the subsequent function calls.

The user can then use the `sv8` function to write an 8-bit *value* into the *index* location of *array*:

```
sv8(array, index, value);
```

Similarly, the `gv8` function can be used to read the value stored at index i of the *array*:

```
uint8_t value = gv8(array, index);
```

Finally, the user can free the allocated memory by calling the `sm_free(array)` function. The Plasma system can reclaim the freed memory and use it for further allocations.

Similar to the previous 8-bit access functions, Plasma provides their 32-bit counterparts as well:

```
int array = sm_malloc32(size);
sv32(array, index, value);
uint32_t value = gv32(array, index);
sm_free(array);
```

CHAPTER 5

CASE STUDIES

5.1 MITHRA: Monte Carlo Simulations

5.1.1 Monte Carlo Simulation

Monte Carlo methods can be loosely considered statistical numerical simulation methods where sequences of random numbers are used to perform the simulation. These statistical simulation methods differ from conventional numerical discretization methods which are used to analyze ordinary or partial differential equations that model physical or mathematical systems [41].

Monte Carlo simulation requires the system to be either described by probability distribution functions or a parametric model. Informally, this means that any implementation of a Monte Carlo needs to create volumes of high quality random numbers and use these numbers as inputs to a parametric model of the system. Simulations are performed over many trials and the desired result is filtered through an aggregation function which combines the output of the trials. Typically, one would like to use the average and variance functions to perform this aggregation and the number of samples is chosen to achieve a given accuracy level of the simulation.

The following steps define the required components of a typical Monte Carlo simulation [27]:

1. Create a parametric model, $y = f(x_1, x_2, ..., x_q)$.
2. Generate a set of random inputs, $x_{i1}, x_{i2}, ..., x_{iq}$.
3. Evaluate the model - and store the results as y_i.
4. Repeat - steps 2 and 3 for $i = 1$ to n.

5. Analyze the results - using histograms, summary statistics, etc.

6. Error estimation - an estimate of the statistical error (variance) as a function of the number of trials and other quantities can be determined as part of this step.

After the random trials are complete, the results must be aggregated over a combiner function to produce the desired result for the particular problem. However, the essence of the Monte Carlo method is the use of random sampling techniques to approximate the solution of a complex problem [41].

Monte Carlo Simulation as a MUD Algorithm

It is clear that a Monte Carlo algorithm can be easily described in the MUD formalism. The first step of the Monte Carlo algorithm described in the previous subsection is typically done in the design stage, and does not correspond to a certain algorithmic step. The important thing to note, however, is that the function f has a limited set of arguments, x_1 to x_n, and the input data set of each trial is independent of other trials. Described as a MUD algorithm, this application and domain specific function will form the function Φ. The next step of the typical Monte Carlo algorithm requires an efficient method of creating $n \cdot q$ random, psuedo-random or quasi-random numbers, in other words creating the Σ set. Formally, each $\sigma \in \Sigma$ is a vector of q elements: $\sigma = (x_1, x_2, ..., x_n)$, and Σ is the set of all random vectors of size q.

The third and fourth steps of the Monte Carlo application are about evaluating the model. In other words, this is where the Φ function is applied in parallel to all the random input vectors. Recalling that in a Monte Carlo simulation all of the trials will contribute equally to the final summary statistic values of interest, and hence the requirement of considering all of the trials in calculating the final analysis results, we can simply use the same key for all the key/values pairs generated here in a MUD model. Formally, $\Phi(\sigma_i) = (k, v_i)$. It should be noted that the MUD formalism allows us to create a list of key/value pairs by Φ, but the typical Monte Carlo algorithm uses each input random vector once to create a single value. Also note that the input values to each round of function evaluation, conforming to MUD formalism requirements, are independent and thus allow the function evaluations to be performed in parallel.

The final step of the Monte Carlo simulation algorithms involves applying summary statistics to the n evaluated trials of the function Φ. Since in the previous step, all of the intermediate key/value pairs were generated with a single key, there is no intra-key parallelism. However, many of the summary statistics functions are either associative and commutative, or can be decomposed into two functions, Φ and η, so that Φ captures the main functionality while being associative and commutative, and η is not commutative or associative, performing the final operation. Examples of such functions include mean and variance calculation. Therefore, the summary statistics functions can utilize the inter-key parallelism of a MUD algorithm as mentioned earlier.

5.1.2 Black Scholes Option Pricing by Monte Carlo Simulation

A financial option [42] is the right, but not an obligation to purchase an asset at a future date (*expiration date*) at an exercise price. *Call options* grant the holder the right to purchase an underlying asset while *Put* options allow the holder to sell. Since the price of the underlying asset varies over time due to volatility, pricing an option is an extremely important process. Theory suggests that if options are traded in the free market, their price will converge to a *fair* market value. Black Scholes is one method of calculating this value by simulating many possible paths where the price drifts up and down along the time axis in accordance with a Gaussian distribution curve of end results.

In this project, we concern ourselves with European stock options which can only be exercised at the expiration date, and we do not consider transaction costs, dividends and restrictions on short selling. We assume that money can be borrowed freely at a risk-free rate. Let S denote the price of the stock, *V(S,t)* represent the price of the derivate as a function of time and stock price, *C(S,t)* be the price of a European *put*, K designate the strike price, r represent the annualized risk-free interest rate, μ be the drift rate of S, σ be the volatility of the asset and t represent the time in years.

Although we do not discuss the Black Scholes differential equation, we

briefly summarize the explicit formula for the price of a call option:

$$\mathbf{P(S,t) = K \cdot e^{-r(T-t)} \cdot N(-d_2) - S \cdot N(-d_1)}$$

N(x) represents a Gaussian-distributed input; d_1 and d_2 are inputs to our algorithm randomly chosen sample points from the Monte Carlo simulation. This formula is evaluated over all sample points in the *map* stage and the intuition is that the gain on the security is equal to the difference between the worth of the security and the price paid for it. The first half represents the strike price of the option adjusted for interest accrual by the exponential function, whereas the second portion represents the purchase price of the stock.

To compute the price of an option, we apply the aforementioned mathematical formula to a Monte Carlo simulation to estimate possible final option prices. Each trial computes the gain from a normally-distributed pool of points with a geometric Brownian motion. Given the random normal distribution of points sampled, taking the arithmetic mean and standard deviation of the distribution establishes the confidence interval, exercise price and strike price.

The algorithm is divided into three stages, as depicted in Figure 5.1: the Φ stage performs the map and evaluates the Black Scholes formula over random Gaussian sample points producing intermediate key/value pairs. Once the mappers finish executing, the framework collects all the intermediate key/value pairs with like keys and feeds them to the reducer $\oplus$. The final computation is produced by η which calculates the mean and standard deviation[1] of the samples. To evaluate the performance of our architecture across multiple parallel programming implementations, we implement the following pseudocode on all architectures.

[1]The calculation of standard deviation from sum of squared values is optimized as follows: $\sum_{i=1}^{N}(x_i - \bar{x})^2 = \left(\sum_{i=1}^{N} x_i^2\right) - N\bar{x}^2$

$\implies \sigma = \sqrt{\frac{1}{N}\left(\sum_{i=1}^{N} x_i^2\right) - \bar{x}^2}$

$\Phi(n)$:
  for $i = 1..n$
    $G \leftarrow$ generate a Gaussian random number
    $V \leftarrow S \cdot exp\left((r - \frac{\sigma^2}{2}) \cdot T + \sigma\sqrt{T} \cdot G\right)$
    value $\leftarrow exp(-r \cdot T) \cdot max\{V - E, 0\}$
    emitIntermediate$(1, value)$
    emitIntermediate$(2, value^2)$

$\oplus(key, values)$:
  sum[1] $\leftarrow$ 0
  sum[2] $\leftarrow$ 0
  while($values$.hasNext())
    sum[key] $\leftarrow$ sum[key] + $values$.next().get()

$\eta(sum, n)$:
  mean $\leftarrow \frac{sum[1]}{n}$
  variance $\leftarrow \sqrt{\frac{sum[2]}{n} - mean^2}$

Figure 5.1: Black Scholes as a MUD algorithm

5.2 PIC: Image Smoothing, System of Linear Equations, Neural Network Training, K-Means and PageRank

In this section, we describe five iterative-convergence algorithms — image smoothing, system of linear equations, K-means clustering, PageRank computation, and neural network training using back propagation — that are implemented using PIC. For two of these examples, we briefly describe the map and reduce computations and the application-specific functions (`partition`, `merge`, and `converged`) that we used for the PIC implementations.

5.2.1 Image Smoothing

Our first example was chosen for its simplicity, as it can clearly show how porting an algorithm to PIC works. Image smoothing is used to reduce the high contrast and noisy areas of a picture and is often a first step in image recognition algorithms. This algorithm iteratively replaces the value of a pixel with the average of all its 8 neighbors and itself. Typically this algorithm is executed iteratively a number of times until the resulting image becomes good enough for other image recognition algorithms. Therefore the

convergence criterion only counts the number of iterations performed.

The partitioning function divides the image into slightly overlapping cells, and each cell is used for a PIC sub-problem. Finally the merge function pieces the pixels next to each other. The overlapped pixels are averaged, while all other pixels are simply written into the final results. In this case study we run the same number of iterations as baseline locally, and use only one global iteration.

5.2.2 System of Linear Equations

In this example we consider a system of linear equations of the form $A.x = b$, with a weakly diagonal dominant matrix A. The weak diagonal dominance of A is a sufficient condition for the system to converge to a solution if one exists using serial Jacobi iterations or a parallel Gauss-Seidel iteration [43]. A sufficient condition for a PIC-like implementation to converge is that A also be a positive matrix [43].

Solving this problem requires iteratively computing

$$x = x - \gamma(A.x - b)$$

For scalability to solve very large problems, the matrix multiplication is itself implemented in a distributed fashion. Naturally the same Map and Reduce functions are used in PIC. The partitioning function ensures that all the elements of the matrix A which are on the same row are placed in the same sub-problem. The model in this case study, the vector x, is replicated and each sub-problem gets a local version of it. Each sub-problem in PIC only updates the rows of its local model (vector x) that correspond to its local row entries of matrix A. The local convergence function checks for the average movement of the centroids to be below a set threshold (similar to the baseline). Once sub-problems reach local convergence, the merge function puts the updated results from each sub-problem next to each other to form the merged resulting vector x. Finally the global convergence function checks for the average movement of centroids to be below a set threshold.

5.2.3 Neural Network Training Using Back-Propagation

The back-propagation algorithm is used to train a neural network. The algorithm looks for the minimum of the error function in weight space using the method of gradient descent. The combination of weights which minimizes the error function is considered to be a solution of the learning problem. The algorithm is iterative in nature, and after each iteration the algorithm is expected to move towards the desired value of the weights. The iterations are performed until the error falls below a specified threshold value. This is the stage when the neural network is said to be trained.

The partition function used for the back-propagation algorithm was a random partitioning function. Random partitioning allows the sub-data sets to be non-similar to each other, thereby allowing global convergences to help the whole dataset converge. In the IC implementation and the PIC implementation, we train the neural network using a batch update rather than an online update. The batch update is more suited for the parallel implementation of the algorithm [44].

In the batch-mode variant of back-propagation, the mean square error is determined for each input pattern. This error is used to find the gradients of the hidden and final layers. The Δw for hidden and final layers are then calculated according to

$$\Delta w_{ij} = -\gamma * o_i * \delta_j$$

where δ is the gradient for the hidden or the final layer. This process is repeated for all the input patterns and the cumulative Δw is determined. This cumulative Δw is then used to adjust the weights of the hidden and final layers.

In a global iteration of the PIC implementation, every sub-problem performs as many local iterations as necessary to converge the sub-data sets. The convergence criterion we used is the same as the criterion used in the IC implementation. Every iteration of the IC implementation starts with a model which is the set of weights in the neural network. The model is represented as a key-value pair with the weight position in the network being the key and the weight being the value. At the end of one global iteration we have a refined model which satisfies convergence of the sub-datasets. In the global merge function, we average all the weight sets (models) obtained from different sub-problems. This averaged model is then given to another

map-reduce task which takes the dataset as the input and gives out the number of vectors which the network is able to detect with the new model. We iterate if the number of vectors being detected is less than the total number of vectors in the dataset.

5.2.4 K-Means

```
kmeans_PIC(input data d, model m) {
  // best-effort phase
  do {
    // partition input data points,
    // copy m to each partition
    p1 … pz = partition (d,m)
    map: for each partition pi {
           kmeans_IC(di, mi);
           emit (key=1; value=mi)
         }
    // Average closest centroids from z sub-problems
    // as new centroid in merge
    reduce: m = merge(values m1 … mz for key=1 )
  } until BE_converged (m);

  // top-off phase
  kmeans_IC(d,m);
}
```

Figure 5.2: PIC implementation of K-means (for IC implementation, see Figure 3.1(b)

K-means is an iterative convergence algorithm designed to create a representative model (k centroids) from a data set (a body of "points" in a cartesian space of n dimensions).

Figure 5.2 shows the PIC implementation of K-means. We fully re-use the IC implementation shown in Figure 3.1(b).

We used a simple random `partition` function for K-means. In a best-effort iteration of the PIC implementation, each sub-problem performs as many local iterations as necessary to obtain a *converged* partial model. The convergence criterion we used is the same as the criterion used in the IC implementation. Every iteration of the IC implementation starts with a model (i.e., set of proposed values for the K centroids). At the end of the iteration, we have a refined model (i.e., new set of values for the K centroids). If the change in the value of all the K centroids is within a pre-specified threshold, we conclude that the model has converged. We used the same convergence criterion for every sub-problem in PIC, and for detecting best-effort convergence in PIC. Each sub-problem computes a partial model (set

of K centroid values). So, for each centroid, we have a value from every sub-problem. Our `merge` function identifies corresponding centroid values from each partition and averages them to compute the centroid values for the unified model.

5.2.5 Page Rank

The PageRank algorithm is used to obtain a global ordering of a set of web pages. The input to the algorithm is a web graph. There is one vertex in the graph for each web page (or URL). A directed edge from a vertex to another vertex implies that the source web page has a hyperlink to the destination web page. The PageRank value for a vertex is a function of the number of web pages that refer to the vertex, either directly or through other web pages. The solution (or model) created by the PageRank algorithm is a set of PageRanks for all vertices in the web graph. The PageRank algorithm also assigns a score to every edge in the graph. This information is not reported as a solution, but edge scores are used to compute the vertex PageRanks. Therefore, in the PIC implementation, we consider the set of edge scores as part of the model in our implementation.

As shown in Figure 5.3, every iteration in the PageRank algorithm consists of two phases: aggregation and propagation (this reflects the implementation in the Nutch [45] open source search engine). In the aggregation phase, we update the PageRank of every vertex by aggregating the scores of its incoming edges. The PageRank of vertex i is computed from the scores of its incoming edges, as follows:

$$PageRank_i = (1 - c) + c * \sum_j edge_{ji}$$

In the above formula, c is a pre-specified constant (damping factor that is typically 0.85), and $edge_{ji}$ is a directed edge from vertex j to the vertex i. In the propagation phase, we update the score of every edge. The score of $edge_{ji}$ is the ratio of the PageRank of vertex j to the number of outgoing edges of vertex j.

Figure 5.4 shows the PIC implementation of PageRank. We partition the web graph into sub-graphs, by splitting the vertices into disjoint groups.

```
G(V,E): vertex represents a URL;
        edge represents a link to another URL
mV : page rank values for vertices
mE : scores for edges
c : constant

PageRank_IC(input data G(V,E), model mV, model mE) {
    do {
        //aggregate
      map: for each edge e (from vertex u to vertex v)
             emit (key = v, value = mE[e]);
      reduce: for each key {
                 total = sum of values of key;
                 mV[key] = (1-c) + c * total;
          }
        //propagate
      map: for each edge e (from vertex u to vertex v)
             emit (key = u, value = e);
      reduce: for each key {
                count = number of values for the key;
                for each value
                  mE[value] = mV[key] / count;
               }
    } until converged (mV);
}
```

Figure 5.3: IC implementation of PageRank

Vertices and the edges that are fully contained in a group (*i.e.*, the two vertices of the edge are in the same vertex group) form a sub-graph that is assigned to a partition. The convergence criterion for a sub-problem is the same as the criterion used in the conventional IC implementation. The PageRank implementation in Nutch automatically terminates after a pre-specified set of iterations, independent of the quality of the solution. For the PIC implementation, we also terminate the local and best-effort iterations after a pre-set iteration limit.

During the local iterations, every sub-problem updates the PageRank or edge scores of vertices and edges included in its partition. At the end of the local iterations, the various sub-problems have computed PageRanks for all vertices. However, edge scores have been computed only for edges that are fully inside a partition. In particular, no edge scores have been computed for edges that are between partitions. The `merge` function first computes the scores for all outgoing edges from a partition (i.e., source vertex of the edge is in the partition but the destination vertex is in another partition). Then, the `merge` function also updates the PageRanks of the destination vertices of all outgoing edges. This is the only mechanism we have used to factor in the dependencies between the sub-problems. Like the K-means case, one can develop more complex mechanisms to consider the effect of other sub-problems.

```
G(V,E): vertex represents a URL;
        edge represents a link to another URL
mV : page rank values for vertices
mE : scores for edges

PageRank_PIC(input data G(V,E), model mV, model mE) {
  //best-effort phase
  do {
    //partition vertices and include all
       edges contained in partition
    p1 … pz = partition (G(V,E), mV, mE)
    map: for each partition pi {//solve each partition
           PageRank_IC(Gi(V,E), mVi, mEi);
           emit (key = 1, value = mVi )
         }
    //Update scores of edges between partitions;
       update pageranks of vertices for these edges
    reduce: mV = merge(values for key=1)
  } until BE_converged (mV);

  //top-off phase
  PageRank_IC(G, mV, mE);
}
```

Figure 5.4: PIC implementation of PageRank

5.3 Loop Maximizing: Needleman-Wunch Back-Trace

Calculation of edit distances between two sequences and aligning the two sequences based on their edit distances is a very compute-intensive process. This problem becomes n-fold when the number of sequences on which these operations have to be done are huge. Many fields have hence refrained from using edit-distance as a possible solution to solve problems. This section contains an explanation of the use of this algorithm in a larger context in which this algorithm plays a part, and then describes the challenges in implementing the Needleman-Wunsch algorithm in SIMD architectures, specifically GPUs.

5.3.1 Pairwise Alignment in a Larger Context

Pairwise alignment can be used to compute the edit distance between two sequences and then align these sequences with an allowed fixed number of insertions, deletions and mismatches. Edit distance computation between two sequences also finds its applications in other domains. Reference [46] uses an edit-distance algorithm to detect correlated attacks in distributed systems. Reference [47] Uses the edit distance technique to identify the type of intrusion. A very common use of edit distance is to find how close two strings are to each other and auto-check the spelling of the word accordingly.

Similar approaches could also be used to suggest search strings in search engines. Edit distance is used in speech [48] and evaluating optical character recognition [49]. All of these approaches involve calculation of edit distances of millions of sequences and then alignment of the sequences.

An interesting application of edit distance and alignment of two sequences is in bio-computation. The machines which generate the DNA data have progressed at a much more rapid pace than the techniques to analyse this data [50]. This is a profound problem in the bio-computation domain. Today biologists have terabytes of data but do not have enough computational power and techniques to work on this data. In many ways this problem is similar to the big-data problem that the computer industry has been facing for some time now. There is a need for techniques which could be used to analyse this data rapidly. Accurately aligning the genes against each other allows the comparison of DNA strings and gives researchers the ability to draw inferences from these alignments. Our tool allows fast alignment of a large number of short reads quickly.

5.3.2 A Short Description of the Needleman-Wunsch Global Alignment Algorithm

The Needleman-Wunsch global alignment algorithm [51] is a dynamic programming algorithm that is used for global alignment, meaning that it can be used to find the best alignment possible between two different strings. This is in comparison to local alignment algorithms (for example the Smith-Waterman algorithm), where the best alignment is among smaller segments of the two strings. In [51], to find the alignment of two strings x and y, a two-dimensional m by n score matrix is allocated, where m and n are the lengths of the two strings x and y. In the first phase of the algorithm, the (i, j)'th entry of the matrix contains the optimal score for the alignment of the first i characters in x and the first j characters in y.

The contents of each cell $a_{i,j}$ is computed based on the i'th character in x, the j'th character in y and the contents of the cells on its left ($a_{i,j-1}$), top ($a_{i-1,j}$) and top-left ($a_{i-1,j-1}$) that are already computed and stored in the matrix. The value of the cell $a_{i,j}$ is computed using the following expression: $a_{i,j} = Max\{a_{i,j-1} + g, a_{i-1,j} + g, a_{i-1,j-1} + S[x[i], y[j]]\}$ where g is the gap

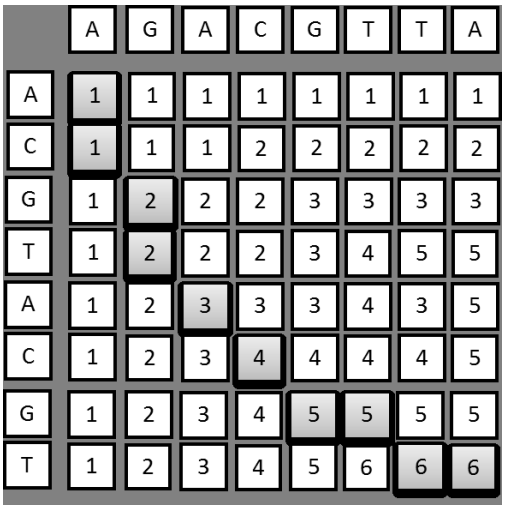

Figure 5.5: Global alignment using Needleman-Wunsch algorithm

penalty value and $S[x[i], y[j]]$ represents a similarity matrix (a look-up table that represents the penalty of changing one character to another). Figure 5.5 depicts the contents of the matrix a.

Once the matrix is computed and stored in memory ($O(m.n)$ memory requirement and $O(m.n)$ time), the algorithm starts the backtracing phase from the last location of the matrix ($a_{m,n}$) backwards. At each step, the algorithm picks the cell whose value was used in the previous phase of the algorithm, and goes backwards to the first cell $a_{0,0}$. Each movement upwards means an insertion in string x, and a movement leftwards means an insertion in string y (or a deletion in string x). A diagonal movement would mean either a match or a mismatch. The backtracing phase requires the whole matrix in memory ($O(m.n)$), and takes $O(m + n)$ time to execute.

5.3.3 Needleman-Wunsch Back-Trace Challenge and Loop Maximizing

Implementing the Needleman-Wunsch algorithm on SIMD machines, and more specifically GPUs in this research, is challenging. As mentioned in the last discussion, the second phase of the Needleman-Wunsch algorithm is inherently data-dependent. In the previous section, we described how this data-dependence affects the memory access patterns if the score matrix is stored in global memory. Another manifestation of this problem is in diverg-

ing code flows. Unlike the first phase of the algorithm, in which the same code flow takes place regardless of the data contents, the backtracing phase is completely data-dependent. The backtracing might take anywhere between $max(m, n)$ (when the backtracing always takes the diagonal route) and $m+n$ steps (for example when backtracing first completely moves upwards in m steps and then takes n steps left).

This is a serious problem for a SIMD machine such as a GPU. The whole reason for the massive parallelism potential of SIMD machines is that they sacrifice independent control logic circuitry for more computational units. Diverging code flows would make use of the parallelism potential of GPUs impossible. This challenge is one of the main reasons that many previous attempts to implement either the Needleman-Wunsch or the Smith-Waterman algorithm in GPUs have reported sub-optimal performance figures.

To solve this problem, the trace-back algorithm is modified using the loop maximizing technique, and is written such that the trace-back in each quadrant (for a definition of quadrant-based Needleman-Wunsch please refer to [18]) takes exactly $2 \cdot k$ steps. To achieve this, the loop maximizing technique adds condition variables that evaluate to the zero value once the trace-back pointer reaches either the left or top boundary of the quadrant. These condition variables are then embedded in the expressions that find the next cell in the traceback path. In effect, once the trace-back reaches either of the left or top boundary, it becomes stuck there until the $2 \cdot k$ trace-back steps are finished. As a result, the same set of instructions are executed in each thread, eliminating any flow divergence among different threads.

The same problem should also be tackled for loading different quadrants (refer to [18]), and is addressed in a similar way. Each thread loads the boundaries for exactly $\frac{2 \cdot m}{k}$ quadrants and performs trace-back in each of them. However, once a quadrant on the first row or column of the quadrants grid is loaded, a similar mechanism is utilized to make the computation stuck, while every thread keeps executing similar instructions.

This solution ensures that any combination of input arguments take the worst case path. However, we utilize a GPU-specific mechanism to help the situation. The SIMD execution model of NVIDIA GPUs runs different threads in *warps* of 32 threads in each SM. Therefore, once all 32 threads of the same warp get stuck, we can break the loop and move on to the next phase, instead of waiting for all of them to reach $2 \cdot k$ steps. Fortunately

NVIDIA GPUs of compute capability 1.3 or higher provide and intrinsic function *"__all(condition)"*, which evaluates to TRUE once all the individual *conditions* of the warp's threads become TRUE. Using this mechanism helps our algorithm to run faster.

5.3.4 Using Plasma in Needleman-Wunch Case Study

The modified Needleman-Wunsch algorithm described in the previous section processes a pair of strings in each kernel function invocation, and therefore different GPU threads are completely independent from each other. This is the first requirement for the usage of the Plasma library. Moreover, the algorithm requires frequent use of fast scratch memory arrays, and a fast dynamic array would greatly benefit the performance of it. As a result, this algorithm is a prime candidate for using Plasma. The experimental results shown in Section 8.3 are reported with the Plasma library in use.

CHAPTER 6

MIDDLEWARE DESIGN AND IMPLEMENTATION

6.1 MITHRA Implementation

MITHRA is designed to excel at executing massive, data independent computing tasks. We leverage heterogeneous computing resources in the architecture design. In other words, the "correct" computing resources for each phase of a MUD algorithm can be different, thereby offering the potential to leverage different hardware classes. The MITHRA architecture takes the mathematical formalism described in previous sections and reifies the results into an efficient organization of hardware to exploit the properties exposed by our formalism. In this section, we describe the architecture in detail and explain how it achieves the parallelism goals set out earlier.

6.1.1 MITHRA System Design

Clusters of GPU have been utilized in recent years to solve different classes of problems mentioned earlier. Similarly, MapReduce or MUD algorithms have been used for both massive log analysis and to a lesser extent for scientific computing. The novelty of MITHRA, however, is the adoption of the MUD formalism to model a broad range of scientific computing problems, and adapting it to run on a low-cost commodity cluster of computers. In its basic design, MITHRA is a cluster of COTS computing nodes. Each node contains a mid-range CPU and is connected to other nodes through a gigabit-class Ethernet network. Graphics processing cards (GPUs) installed on each processing node make MITHRA suitable for running MUD algorithms.

Figure 6.1 depicts the architecture of MITHRA. The MITHRA framework is based on the open source Hadoop project, which is an implementation of Google MapReduce. It inherits the merits of fault tolerance and scalability

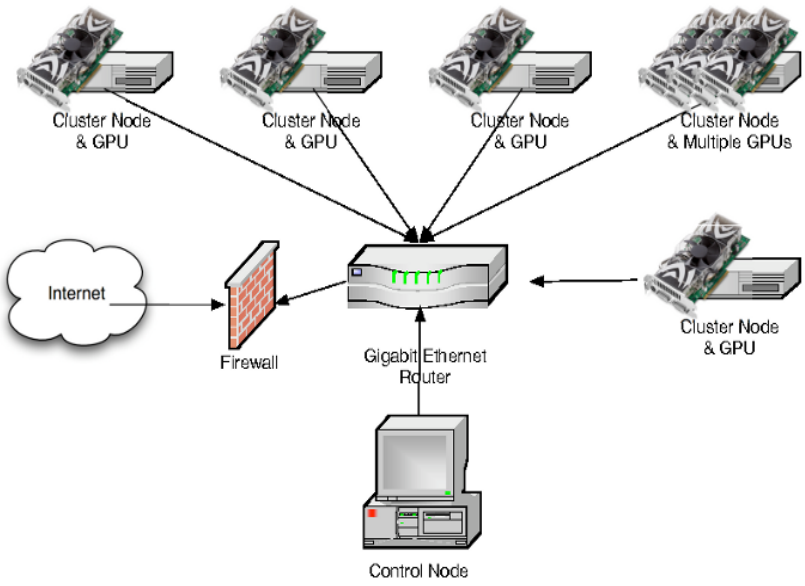

Figure 6.1: Architecture of the MITHRA system.

from Hadoop and speed from GPUs. It should be noted that in a cluster of COTS components, the probability of component failures is considerably higher than in a cluster of specialized high performance and highly reliable components. However, the adoption of the MapReduce programming model implies software fault tolerance features that guarantee the continued execution of computation jobs even under less than ideal conditions, without the programmer having to deal with them.

The map function in MITHRA is designed as a two-stage process. The first phase makes use of the Hadoop map function. The Hadoop framework distributes the user-supplied map function across the cluster. However, the main functionality of the Φ function in the MUD formalism is not programmed in the Hadoop map. Hadoop's map merely acts as a distribution mechanism for the Φ workload across the nodes of the clusters.

The Φ function in a MUD algorithm, being data independent in its multiple evaluations on its input data set Σ, can be performed on GPUs. This is the second phase of the map function design in MITHRA, in which the programmer writes a CUDA kernel to work on one $\sigma \in \Sigma$. In fact, the data independence property of the Φ function in the MUD formalism is similar and matches the data-independent compute intensive SIMD [52] kernel model of CUDA [53]. Unlike a traditional map function in the MapReduce model,

MITHRA does not promote having k Maps with each map evaluating the Φ function $\frac{N}{k}$ times, where N is the number of data items in Σ and k the number of nodes in the cluster. Instead, the map function written in CUDA corresponds directly to the function Φ, working on a single item from Σ.

The Hadoop framework distributes Σ among the nodes of the cluster. The Hadoop map task divides up the amount of data determined to be processed on that node to chunks of data small enough to fit in the GPU memory (typically 64MB to 256MB in our implementation). It then loads the data in the GPU memory and calls the CUDA kernel.

The next part of the framework involves intermediate key/value management. Focusing on the Monte Carlo simulation class of problems, we did not need to use more than a single key for the intermediate key/value pairs; therefore the implementation of the key/value management scheme is not completely general. As this is an important feature for the application to more general problems, it is one of our first future work milestones. In our current implementation, after the execution of the map functions, the intermediate key/value pairs are assumed to have been grouped by *the key*.

The last phase of a MUD algorithm is the application of the $\oplus$ aggregator on all the key/value pairs with the same key, which in a Monte Carlo simulation means all of the intermediate values across the cluster nodes. In a Monte Carlo simulation, the Φ function calculates the mean and variance of the trial runs of ϕ as indicator summary statistics values. As discussed earlier, even though these functions are not associative and commutative, they can be decomposed into separate functions, mostly because the division operator used in mean and variance is distributive. Therefore, the reduction aggregator $\oplus$ can be applied locally and the pre-final values can be sent to the head node for the final round of $\oplus$ and $\eta(\cdot)$ application.

6.1.2 Practical Implications of Adopting MUD Programming Model on MITHRA

Recalling the formalism presented earlier and the discussion of the inherent parallelism in MUD algorithms, this subsection shows how each of the four parallelism opportunities can be exploited for better performance in

MITHRA. We formally show that all Monte Carlo simulations can be performed in an efficient manner in our MITHRA architecture. Steps of the Monte Carlo algorithm correspond to a phase of the MUD model, or match an inherent parallelism opportunity of MUD.

Input Data Set

The input data set can either be pre-stored and distributed on hard disks of the individual machines in the cluster, streamed in from the Internet or generated for immediate consumption. When the data is already distributed among the hard disks, all of the Hadoop map functions can read the input data in parallel. Assuming k nodes are available in the MITHRA cluster, the theoretical aggregate I/O bandwidth becomes k times the I/O bandwidth of each hard disk.

When streaming from the Internet, each node receives its own specific data set. In theory, this task might not be highly parallelizable since the incoming network connection bandwidth becomes the bottleneck. However for practical purposes, it can have enough bandwidth to fill all the node's input data bandwidth specification, and thus the task can be parallelized.

Finally, our best parallelism can be achieved when the application needs or can be changed to use run-time generated values as its input set. This is because even though the generation of the Σ set takes time, it will probably take less time than to load them from hard disks or read them through slow network connections. This is the case for the Monte Carlo simulation algorithms, since they only require a set of random numbers as their Σ set.

We generate quasirandom numbers using the Niederreiter quasirandom generator [38] in the GPUs. A quasirandom or low discrepancy sequence, such as the Faure, Halton, Hammersley, Niederreiter or Sobol sequences [38, 39] is *less random* than a pseudorandom number sequence, but more useful for such tasks as approximation of integrals in higher dimensions and in global optimization. This is because low discrepancy sequences tend to sample space *more uniformly* than random numbers. Algorithms that use such sequences may have superior convergence [38]. Creating quasi or psuedo random numbers is a very time consuming task; therefore, it is best to create the random numbers in the GPUs where they will be consumed in a distributed manner to utilize all the processing elements available, in contrast to creating

them directly in the CPU using a sequential algorithm and then transferring them to the GPU memory through the low bandwidth PCI-Express bus. We base this part of our work on an implementation of this algorithm from the CUDA SDK [40].

Another requirement for the Monte Carlo simulation algorithms is the application of a normal distribution PDF to the random numbers. For the inverse cumulative normal function $z = N'(p)$, there are several numerical implementations providing different degrees of accuracy and efficiency [54]. A very fast and accurate approximation is the one given by Boris Moro in [55]. This algorithm also runs on the GPU.

Data Independent Φ

As mentioned earlier, Φ is data independent across executions, and can match the SIMD execution model of GPUs. For our experiments we chose the **Black Scholes** option pricing equation, which conforms to the requirements of the MUD formalism for Φ.

Inter-key Parallelism of $\oplus$

This parallelism opportunity can be exploited when the application uses more than one key in its intermediate key/value pairs. The inter-key parallelism is usually a side-effect of the algorithm used to solve a problem correctly. However, the programmer can use it as a measure to force parallelism across nodes of a cluster, especially when the $\oplus$ aggregator is not associative or commutative.

For a Monte Carlo algorithm, the intermediate keys need not be different, and the reduction aggregator can be made into an associative and commutative function, and therefore the intra-key parallelism (described next) can exploit all the inherent parallelism available in the problem.

Intra-key Parallelism of $\oplus$

The last step of a Monte Carlo algorithm is the calculation of required indicator summary statistics. We have provided examples and details of two summary statistics functions used in our implementation: mean and variance.

Both of these functions are decomposed in our implementation, so that an associative and commutative function (addition) is used as the aggregator. Therefore, this aggregator function can be applied inside the GPUs, and then across multiple nodes of the cluster in the Hadoop reduction function (which acts as the $\eta(\cdot)$ function of the MUD formalism).

6.2 PIC Implementation

The PIC library is implemented on top of the Hadoop MapReduce framework. PIC consists of a set of Java classes, some of which are abstract. The programmer overrides the abstract classes and adds their `map`, `reduce`, `localConvergence`, `globalConvergence`, `partition` and `merge` functions. Each function has a predefined signature, through which the PIC library passes appropriate arguments (such as an input data element and *model*) into the programmer's function.

PIC executes local map-reduce-convergence loops completely inside one Hadoop mapper. It reads models from files and stores them in an appropriate *Model* data structure. It then reads the input data subset and parses data elements and passes them one by one to the user provided `map` function, in conjunction with the *Model*. It also saves the intermediate key/value pairs that the `map` function emits. It then groups the intermediate key/values according to MapReduce semantics, and passes one key along with an iterator of values to the user-specified `reduce` function. The results of the `reduce` function are collected into another *model* instance, and along with the *model* from the last iteration are passed to the user provided `localConvergence` function. If this function returns `false`, PIC repeats this loop. Otherwise, it moves on to the next phase, which includes merging and global convergence test.

The merge functionality is implemented in Hadoop's reducers. The results are then stored by the PIC library in a file in HDFS (transparent to the programmer), which represents the merged output of a global iteration. The global convergence test is the only function in PIC that is executed in one node of the cluster, namely the head node. However, typically the task of comparing the results from one iteration to another is a lightweight computation and does not affect the total execution time too much.

6.3 Plasma Library Details

Two assumptions are considered in the design of this library.

- **Thread independence:** The individual threads are independent and do NOT need to share their values.
- **Thread linear identification:** Plasma requires a local thread identifier computed and stored in a specific variable: `local_tid`.

The Plasma library is a implemented as a collection of macros (using the # define syntax). Under the hood, these macros perform carefully designed address translation to avoid bank-conflicts.

The core of the address translation is based on the following principle:
`index --> (local_index * THREADS) + tid`
which translates a user-supplied `local_index` array offset into a real `index` offset into shared memory.

The Plasma library is internally based on 32-bit accesses. As a result, if the user requires access to 8-bit arrays, the individual 8-bit values are artificially constructed using masking operators. As an example, the following macro sets the value of an 8-bit array element, indexed by local_index:

```
#define sv8(local_index, value) {shared_mem[((local_index)>>2) *
THREADS_PER_BLOCK + tid] = (shared_mem[((local_index)>>2) *
THREADS_PER_BLOCK + tid] &  (0x000000ff <<
(((local_index) & 0x3 ) <<3 ))) |
((0x000000ff & ((uint32_t)(value)))
<<(((local_index) &0x3 ) <<3));}
```

This macro first reads the 32-bit value, masks it, replaces the 8-bit value in the masked subset, and finally stores the whole 32-bit word back in the shared memory array. This complex combination of logical operators is necessary to ensure optimal memory usage.

6.3.1 Shared Memory Usage vs. Occupancy

Clearly the more shared memory each thread needs, the smaller number of threads can run at the same time in each SM, which affects the occupancy of a CUDA kernel. Table 6.1 shows some possible run-time configurations of

Table 6.1: The different run-time configurations of Plasma, designed to achieve best occupancy for different memory requirements

Sh Mem	Thrd/Block	Occupancy	Regs	Thrd/SM	Warps/SM
63	96,128	50%	39	768	24
73	672	44%	48	672	21
95	128,256	33%	63	512	16
109	224	29%	64	448	14
127	64,128,192	25%	64	384	12
153	160,320	21%	64	320	10
170	96	19%	64	288	9
219	224	15%	64	224	7
255	96,192	13%	64	192	6
306	160	10%	64	160	5
383	128	8%	64	128	4
511	96	6%	64	96	3
767	64	4%	64	64	2

Plasma, assuming a Fermi GPU with 48KB of shared memory configuration. Each row is tuned to achieve the highest GPU occupancy possible.

Finally note that the CUDA run-time uses a small amount of shared memory for passing the kernel launch arguments. Plasma leaves 64 bytes for system usage, and uses the rest.

6.3.2 Bank-Conflicts, 8-bit Accesses and Allocation/Deallocation

Shared memory's performance can be hampered by bank conflicts. To ensure no bank conflict takes place, the Plasma library internally changes the index addresses:

```
index --> (local_index * THREADS) + tid
```

where *THREADS* contains the number of threads in a block, and *tid* is the local thread id in the current running block (which can be as simple as `threadIdx.x`).

To ensure no bank-conflict happens and maximum use of the shared memory is achieved, Plasma works on a 32-bit basis. Therefore, when there is a request to read or write an 8-bit value, a sequence of masking and shifting operations are performed to ensure only the specified 8-bit portion of a 32-bit word is affected.

Plasma allows for dynamic allocation and freeing. Currently, the system

is designed around a stack-based scheme. For proper usage, only the latest allocated array can be freed and reclaimed at any execution point. This simple scheme guarantees no memory fragmentation, but imposes limits on the program logic. A full implementation including a compaction operation is currently underway.

CHAPTER 7

FOUNDATIONS OF MITHRA AND PIC

This chapter delves deeper into the foundations of MITHRA and PIC. We first discuss the parallelism opportunities that MITHRA can exploit in section 7.1. Then we shift our attention to PIC, where we analyze its best-effort phase as a Schwartz preconditioner.

7.1 Parallelism in MITHRA

In Section 5.1.1 we showed a correspondence between the MUD model that MITHRA implements and the Monte Carlo class of simulations, thus establishing a direct relationship between them. Now we focus on how MITHRA can utilize the parallelism offered by MUD.

The main advantage of the MUD formalism is the broad range of problems that can be described and modeled using it, and the potential for parallelism. There are four distinct parallelism opportunities in a MUD algorithm, which are as follows:

1. Input data set creation
2. Data independent execution of the Φ function
3. Intra-key parallelism of $\oplus$
4. Inter-key parallelism of $\oplus$

We defined the input values of a MUD algorithm to be members of the set Σ. However, in real applications, its members should either be created at run-time, streamed into the machine executing the algorithm or stored on the hard disks distributed across the nodes of a cluster. Each of these possible cases allow for a parallelism and hence better performance. Considering the

typical I/O latency that dominates the latter two classes, the best parallelism opportunities can be found when the data can be generated right before consumption. It should be noted that the input data set creation can also be another MUD algorithm, starting with a simpler input data set (for example constant values).

The second class of parallelism is derived from the definition of the Φ function. Note that the input to each evaluation of Φ is an independent, single member of Σ (note that each $\sigma \in \Sigma$ can be a vector of values). Therefore, there is no dependency among multiple executions of Φ, and it can be parallelized among whatever number of processing elements that are available on the physical machine executing the algorithm.

The latter two types of parallelism are application dependent, contingent on the problem being solved and the mathematical properties of $\oplus$. Intra-key parallelism can be used when the programmer used multiple keys in the intermediate key/value pairs. Recalling from the definition of $\oplus$ function, $(k, v_1) \oplus (k, v_2) \rightarrow (k, v_3)$. Therefore $\oplus$ works on multiple values of key/value pairs with the same key, and generates another key/value pair with that key. If the programmer defines the program in such a way to have multiple keys, each execution of $\oplus$ is limited to the key/values with a single key. This parallelism can be used to gain performance in a cluster setting even if there is no inter-key parallelism to be found. A simple yet effective approach would be to have the same amount of keys as the nodes in the cluster, and distributing the total members of Σ among them with different keys.

The last type of parallelism potential, inter-key parallelism, can be exploited if the $\oplus$ aggregator is associative and commutative. In this case, the evaluation of $\oplus$ on a set of key/value pairs (with the same key) can be performed as a binary tree reduction. This operation can therefore theoretically be finished in $O(log(n))$ time, if the number of processing elements is at least $\frac{n}{2}$. If the number of processing elements is less than that ($k < \frac{n}{2}$), the dominant execution time will be the time to evaluate $\oplus$ function for pairs of leaves $\frac{n}{2}$ times, which can be parallelized across k processing elements. Therefore, the execution time can be reduced to $O(\frac{n}{2 \cdot k})$.

Sometimes, even if the $\oplus$ aggregator is not commutative and associative, one can decompose it into the composition of two functions: one that is associative and commutative, and one that is not. Then, it becomes possible to integrate the non-commutative or non-associative part with the η post pro-

cessing function, and therefore change $\oplus$ into a well-behaved binary operator. As an example, consider the mean aggregator $*$ as $a * b = \frac{a+b}{2}$. We know that this function is not associative, since $(a * b) * c = \frac{\frac{a+b}{2}+c}{2} \neq \frac{a+\frac{b+c}{2}}{2} = a * (b * c)$. However, considering the distributive property of the division operator, we can decompose the averaging function and leave the division for the last step, performing it in the $\eta(\cdot)$ function.

7.2 A Theoretic Treatment of the Partitioned Iterative Convergence Methods

Implementing an iterative algorithm in PIC can greatly reduce the run-time of the computation. In our experiments we have observed a typical factor of 4x to 6x speedup. On the other hand, the PIC framework in effect changes the algorithms. As such, we have to address the accuracy issues of PIC in more detail.

Through experimental studies we have shown the PIC system computes results that are either equal in numerical value to the baseline implementations, or are very close (we have observed up to 3% difference in numerical values). The question we try to address in this document is whether we can extend these expectations to a larger class of problems.

We first describe the three issues involved in the discussion of PIC results accuracy in Section 7.2.1. We then establish the similarity between PIC and domain decomposition methods in Section 7.2.2, which allows us to consider PIC as a domain decomposition method. Section 7.2.3 analyzes PIC as a preconditioner, and shows that it indeed converges to the same set of answers with a convergence rate related to the convergence rate of the original algorithm.

7.2.1 Accuracy of Solutions

There are multiple aspects to the accuracy issues that PIC has to address. These accuracy issues are enumerated in this section.

Accuracy Requirements of the Application Domain

The first aspect of accuracy comes from the application domain. Sometimes the application domain itself has a forgiving nature, and can withstand slightly inaccurate results from the computational kernels and algorithms that solve a certain computational problem. Note that from this aspect, we are differentiating between a target application and algorithms that solve computational problems towards that application. To illuminate this point of view, we provide the following examples:

- Web search: As long as the order of the web pages are the same, the numerical page rank values of each web page is irrelevant.
- Graphics: It is acceptable to have the values of a few pixels be slightly inaccurate. For example, in computer games, since each frame has a life time of at most $\frac{1}{30} seconds$, a few pixels can be (and in many current games are) inaccurate. Similarly, JPEG or MPEG compressions rely on the fact that human eye cannot discern slight inaccuracies in the pixel values.
- Data mining: Many of the algorithms in data mining are heuristic, and even in their original form at most provide a sub-optimal answer (e.g. K-means clustering).

Final Numerical Accuracy

Let us temporarily assume that the our observations in the experimental results are general, i.e. for a large class of problems, one can expect much faster run-times with a final inaccuracy of less than 3%. In other words, we assume the trajectory of the PIC results is towards the correct answer (also see section 7.2.1). In case the application domain is forgiving enough, it can utilize the computed values, making PIC a suitable choice. Furthermore, even if the application domain requires accurate answers (that result from the original algorithms), we can still benefit from PIC as a pre-conditioner. Basically, we can let PIC quickly reach its final accuracy level, and then switch back to the original algorithm. But now we can use the results computed from the PIC run as initial values for the original algorithm. From our experimental

results, we see that this hybrid approach can still drastically out-perform a regular iterative implementation.

Preconditioners are frequently used to *prime* differential equation solvers, and are otherwise known as domain decomposition methods. In the rest of this document, we will consider PIC as a method based on additive Schwarz preconditioners.

Trajectory of Accuracy

The final aspect of the accuracy issue considers the trajectory of the results created by PIC. In other words, how do we know that an algorithm implemented in PIC will not diverge or create drastically different results. The rest of this document addresses this issue.

7.2.2 Applicability of Domain Decomposition Analysis Techniques to PIC

Domain decomposition methods have received a great deal of attention lately. These techniques are designed for differential equations, and rely on the fact that the computation pattern of the problem domain is local, or more precisely stencil-based. To compute the value of the target function (described by the difference equations) at each point of a grid, the algorithm only needs to know the values of its neighbors. The order of the differential equations dictates how many layers of neighbors information are required. For instance, a differential equation of order one requires only the immediate neighbors of a certain point. An order of two equation requires the values of neighboring nodes and their neighbors.

Domain decomposition methods first partition the grid into separate domains and then solve the numerical difference equations in each domain separately. A domain decomposition method can choose to compensate for the boundary values of the partitions after each iterations (as multiplicative Schwarz methods do), or they may ignore them (as additive Schwarz methods do) [56, 57]. In either case, the trajectory of the solutions is convergent towards the correct solution (also see Section 7.2.3.)

As mentioned, the reason domain decomposition techniques are successful is that the computations within each partition (or domain) is truly local. This

data locality can be represented in a tridiagonal dependency matrix [58] for differential equations of order one. Higher order differential equations can be modeled with Block tridiagonal matrices.

Additive Schwarz methods basically treat the block tridiagonal matrix as if it was a block diagonal matrix, ignoring the dependencies among different blocks. For a computational problem to be a suitable match for implementation in PIC, it should have similar dependency patterns.

A computational problem with a high degree of dependence among the data elements is not a good match for PIC (after all PIC stands for *partitioned iterative convergence*). However, many application domains have the *nearly-uncoupled* property (please refer to Chapter 9.3 for related references). Whenever a computational problem has such tendencies it is a potential match for implementation in PIC. Furthermore, if the application has a forgiving nature, we can increase the nearly-uncoupled property by temporarily ignoring some of the dependencies (using the local iterations mechanism).

As an example, consider the PageRank algorithm. In a parallel implementation, each node requires information from the adjacent nodes and edges for the computation. If the web was a complete graph with $\frac{n.n-1}{2}$ edges it would not be a good match for PIC implementation. Fortunately the web graph is typically local, and by properly partitioning it (for example using the METIS package), the connectivity matrix of the graph becomes nearly uncoupled.

Similar arguments apply to the K-means clustering. The impact of far-away points on a centroid is much smaller than the impact of close points to that centroid. As such, a rough first-pass partitioning can ensure that each sub-problem mostly relies on the points inside that partition.

The image smoothing algorithm is stencil based and clearly the dependencies are local. In the case of the linear system of equations, the *weak diagonal dominant* matrix of the equations guarantees the nearly-uncoupled property. In fact, the weak diagonal dominance property is powerful enough to ensure even asynchronous convergence [43, pp. 10–15].

7.2.3 Conditions for Convergence of PIC

PIC as an Additive Schwarz Preconditioner

We start by showing that a regular MapReduce implementation of an iterative algorithm (called *baseline* from here on) is equivalent to a block-Jackobi preconditioner. In a baseline implementation the input dataset is partitioned into HDFS-defined chunks, then the *map()* function is applied on each input record within a chunk on one cluster node. The intermediate results from the application of *map()* function are then all gathered before being passed to the local combiner function. As such, since the information used in the combiner are all from the previous iteration and none of them are updated, the information flow is similar to a Jacobi iteration (as opposed to a Gaus-Seidel iteration where the result of each *map()* would impact the values used in the other *map()* functions). The partitioning of data by HDFS resembles the *block* aspect of the block-Jacobi method. In turn, a block-Jacobi operator is equivalent to a simple form of additive Schwarz operator where there is no overlap in the partitions [57].

An additive Schwarz preconditioner is defined as

$$P^{ad} = \sum_{i=0}^{n} P_i,$$

where each P_i is an orthogonal projection onto the sub-space spanned by the elements in the sub-problem i [57]. Formally:

$$P_i = B_i A = (R_i^T (R_i A R_i^T)^{-1} R_i) A$$

where A represents (in matrix form) the mathematical operation happing in each iteration ($y = Ax$), R_i is the *restriction* operator from the original problem into a sub-problem (and is merely a formalization of the partitioning process in PIC), and R_i^T the *interpolation* represents mapping the sub-results back into the main problem space (and is the formalization of the merge operator in PIC). The matrix B_i in effect restricts the problem to one sub-domain, solves the problem locally in that sub-domain and generates a local correction vector, and finally extends the correction back to the entire domain [57].

We would like to emphasize again that the R_i and ${R_i}^T$ matrices are never formed in practice; they are simply introduced to express several different

types of partitioning and merging in a similar concise manner [57]. In implementation it is often convenient to represent R_i as an integer array containing the global sub-domain membership of each data element.

Since P_i is an orthogonal projection, the local correction vector $e_i = P_i e = B_i(f - Au^n)$ has a most attractive property: It is the closest vector to e in the sub-space spanned by the members of P_i.

Thus, the basic additive Schwarz method operates by projecting the error simultaneously on the sub-spaces V_i, and uses these projections as local corrections to the approximate solution in each iteration. Similarly, PIC finds the local correction vector ($e_i = P_i e$) multiple times. This can be written as:

$$e_i = P_i^{PIC} e = P_i P_i P_i ... P_i e = P_i^k e$$

Which in turn means the PIC method can be written as an additive Schwarz method itself, where:

$$P^{PIC} = \sum_{i=0}^{n} P_i^{PIC}$$

Sufficient Conditions for the Stability of an Additive Schwarz Preconditioner

We start by assuming the baseline algorithm itself has convergence guarantees. If a set of conditions are sufficient for convergence of an additive Schwarz preconditioner, they are also sufficient for convergence of the block-Jacobi schemes, and in turn sufficient conditions for the convergence of the baseline algorithm.

As such, we can safely assume that the baseline algorithm being targeted for PIC implementation has sufficient conditions that ensure its convergence, specifically the sufficient conditions for the convergence of additive Schwarz preconditioners. We use the set of sufficient conditions presented in [56] for this analysis.

The following three conditions are sufficient for an additive Schwarz preconditioner to be convergent [57, 56]. These three conditions identify the amount of interaction of the sub-spaces (or sub-problems) and the approximation properties of the iterative operations.

1. Stable Decomposition: this condition is simply satisfied if there exists a decomposition of the problem space, such that composition of

the decomposed sub-problems can successfully represent the original problem.

2. Strengthened Cauchy-Schwarz Inequality: This condition puts a bound on the orthogonality of the sub-spaces.
3. Local Stability: Ensures that the local operators are coercive and gives a measure of the approximation properties of the iterative operators.

In the next three subsections, we start from the assumption that the baseline implementation is an additive Schwarz preconditioner that satisfies a certain convergence property, and we show that implementation of that algorithm in PIC results in another additive Schwarz operator that converges to the same answer and has a convergence rate that relates to the baseline algorithm. We would like to urge the interested reviewer to consider the following material in the context of the abstract theory of Schwarz methods, specifically pages 35-46 of [56].

Stable Decomposition

Both the stable decomposition condition and the strengthened Cauchy-Schwarz inequality condition (in the next section) are mostly independent of the choice of the preconditioner function. Instead, they impose limits on the partitioning function.

The stable decomposition condition simply ensures that the sub-spaces are able to represent all the elements in the problem input space. Formally, there exists a constant C_0, such that every $u \in V$ admits a decomposition

$$u = \sum_{i=0}^{P} {R_i}^T u_i, \qquad \{u_i \in V_i,\ 0 \leq i \leq P\}$$

that satisfies the following condition

$$\sum_{i=0}^{P} \widetilde{a}_i(u_i, u_i) \leq {C_0}^2 a(u, u)$$

The C_0 parameter puts a lower bound on the eigenvalues of an additive Schwarz method.

In PIC, this condition clearly holds. The left-hand side of the inequality corresponds to merging of the results related to a certain data point across the partitions, while the right-hand side refers to the application of the baseline

operator on that data item. If the partitioning was performed such that one data item would be split across more than one sub-problem, this could be an issue to consider. However, in PIC the default partitioners divide the input data points such that each point belongs to at most one sub-problem. Formally, if the input element u is placed in partition k we have the following:

$$\widetilde{a}_i(u_i, u_i) = 0 \ \ \forall\, i, i \neq k$$
$$\Rightarrow \textstyle\sum_{i=0}^{P} \widetilde{a}_i(u_i, u_i) = \widetilde{a}(u_k, u_k)$$

The condition then boils down to

$$\exists C_0 \ \mid \ \widetilde{a}_k(u_k, u_k) \leq {C_0}^2 a(u, u)$$

Therefore, we only need to ensure that none of the local solvers create a value at the end of their local iterations that is infinitely larger than the baseline. In PIC and using the default partitioners, $C_0 = 1$.

Strengthened Cauchy-Schwarz Inequality

The strengthened Cauchy-Schwarz condition puts a bound on the interdependence of the sub-problems. This condition results in a matrix ϵ, whose elements enumerate the interdependence of any two sub-problems. As a result, the spectral radius of this matrix will appear in an upper bound for the largest eigenvalue of an additive Schwarz preconditioner, which in turn puts a bound on the condition number and convergence rate of the preconditioner.

Formally, there exist constants ϵ_{ij} (elements of the inter-dependence matrix ϵ), where

$$0 \leq \epsilon_{ij} \leq 1, \ 1 \leq i, j \leq P,$$

where P is the number of partitions, and

$$|a({R_i}^T u_i, {R_j}^T u_j)| \leq \epsilon_{ij} a({R_i}^T u_i, {R_i}^T u_i)^{1/2} a({R_j}^T u_j, {R_j}^T u_j)^{1/2},$$

for $u_i \in V_i$ and $u_j \in V_j$. The spectral radius of the inter-dependence matrix ϵ is denoted by $\rho(\epsilon)$.

If two sub-problems V_i and V_j are completely dependent on each other, the strengthened Cauchy-Schwarz inequality reverts back to the trivial case of the regular Cauchy-Schwarz inequality. If this is the case for all i and j, we

have $\rho(\epsilon) = P$, which would give a poor upper bound and convergence rate. On the other hand, the best bound is achieved for completely orthogonal sub-spaces $\{{R_i}^T {V_i}^T\}$ (e.g. block diagonal matrices), in which case $\epsilon_{ij} = 0$ for $i \neq j$ and $\rho(\epsilon) = 1$.

In reality, $1 \leq \rho(\epsilon) \leq P$. If the sub-problems have small amounts of inter-dependence on each other, the $\rho(\epsilon)$ will be closer to 1. As mentioned earlier, PIC is designed to target problems that can be successfully partitioned. A block diagonal matrix ensures $\rho(\epsilon) = 1$. A block tridiagonal matrix has a $\rho(\epsilon)$ much closer to 1 than N. In any case, the natural decomposability of the problems targeted by PIC (or other additive Schwarz preconditioners) ensures a proper value of $\rho(\epsilon)$.

Local Stability

The last assumption is related to the stability of local operators. Basically we need to find a lower bound (ω) on the required scaling factor of the local solutions to ensure that for any data point the scaled result will be larger than the original solver. Formally, we need to show that:

$$\exists\, \omega \in [1, 2) \mid a({R_i}^T u_i, {R_i}^T u_i) \leq \omega . \widetilde{a}(u_i, u_i), \quad u_i \in V_i \ , \ 0 \leq i \leq P$$

where ${R_i}^T$ is the interpolation operator, $a(.,.)$ is a symmetric, positive definite bilinear form representing the computation, $\widetilde{a}(.,.)$ represents the local computations, V_i is a sub-problem and P is the number of sub-problems. In theory if exact local solvers are used, i.e. if $a_i(.,.) = a(.,.)$, then $\omega = 1$. However, the local computations are not exact for two reasons:

- We have omitted the impact of some input points that lie outside a partition.
- Over-tightening of the local solutions can result in over-strengthened local solutions and slower global convergence; therefore, we usually do not run the local solvers until local convergence.

For these reasons the local computations are represented by $\widetilde{a}(.,.)$ instead.

As mentioned earlier, PIC is equivalent to the baseline if it is run for only 1 local iteration (and keeping the partitioning and merging phases). In this case, PIC is completely equivalent to an additive Schwarz method, and we

can assume the local stability assumption holds for it. In other words, there exists an $\omega \in [1, 2)$ which satisfies the stability condition. We will now look at what happens when the number of local iterations is increased to k:

$$a({R_i}^T u_i, {R_i}^T u_i) \leq \omega.\widetilde{a}(u_i, u_i) \Rightarrow$$
$$a({R_i}^T u_i, {R_i}^T u_i)^k \leq \omega^k.\widetilde{a}(u_i, u_i)^k \qquad \text{(EQ. 1)}$$

The bilinear form is positive definite, so the inequality can be raised to the power of k. This is equivalent to application of each computation k times. The left-hand side represents k iterations of the baseline algorithm, while the right-hand side represents k local iterations.
We first work on the left-hand side of the inequality:

$$a({R_i}^T u_i, {R_i}^T u_i)^k = ({R_i}^T u)^T A({R_i}^T u)^k$$

where A is the stiffness matrix associated with the bilinear form $a(.,.)$, representing the computation in a matrix format. Continuing:

$$= (u^T {R_i}^{T^T} A\ {R_i}^T u)^k = (u^T R_i A {R_i}^T u)^k$$
$$= u^T R_i A {R_i}^T u u^T R_i A {R_i}^T u u^T R_i A {R_i}^T u \ ... \ u^T R_i A {R_i}^T u \quad (k \text{ times})$$

Note that u and u^T are one-dimensional vectors; therefore, their multiplication results in a real number α, representing the dot product of the input vector u with itself, or the squared length of u:

$$= u^T R_i A {R_i}^T \alpha R_i A {R_i}^T \alpha R_i A {R_i}^T \ ... \ \alpha R_i A {R_i}^T u \quad (k \text{ times})$$
$$= \alpha^{k-1}(u^T R_i A {R_i}^T R_i A {R_i}^T R_i A {R_i}^T \ ... \ R_i A {R_i}^T u) \quad (k \text{ times})$$

As mentioned earlier, the restriction and interpolation operators (R_i and ${R_i}^T$) are just formalizations of the partition and merge functions. We can assume the partitioning and merging operations by themselves are lossless; i.e., running one after another represents a round of partitioning and merging that cancel each other out with no effect on the computations. Therefore, we set ${R_i}^T R_i = 1$ and get the following:

$$= \alpha^{k-1}(u^T R_i AAA \ ... \ A {R_i}^T u) \quad (k \text{ times})$$
$$= \alpha^{k-1}(u^T R_i A^k {R_i}^T u)$$

Therefore, the left hand side of EQ. 1 is shown to be a scaled version of the regular baseline with k.

Now we work on the right side of the EQ. 1, using the local stiffness matrix representation $\widetilde{A}_i$:

$$\omega^k.\widetilde{a}(u_i,u_i)^k = \omega^k.({u_i}^T\widetilde{A}_i u_i)^k$$
$$= \omega^k.({u_i}^T\widetilde{A}_i u_i {u_i}^T\widetilde{A}_i u_i {u_i}^T\widetilde{A}_i u_i ... {u_i}^T\widetilde{A}_i u_i) \quad (k \text{ times})$$

Note that u_i and ${u_i}^T$ are one-dimensional vectors; therefore, their multiplication results is a real numbers β_i, representing the dot product of u_i with itself or the squared length of the partitioned input vector:

$$= \omega^k.{\beta_i}^{k-1}({u_i}^T\widetilde{A}_i\widetilde{A}_i\widetilde{A}_i ... \widetilde{A}_i u_i) \quad (k \text{ times})$$

which is the definition of the PIC additive Schwarz preconditioner. Replacing the left and right-hand sides back in EQ. 1, we get:

$$\alpha^{k-1}a'({R_i}^T u, {R_i}^T u) \leq \omega^k.{\beta_i}^{k-1}{a_i}^{PIC}(u_i,u_i)$$

where $a'(.,.)$ represents k baseline iterations. Therefore:

$$a'({R_i}^T u, {R_i}^T u) \leq (\tfrac{\omega.\beta_i}{\alpha})^{k-1}\omega{a_i}^{PIC}(u_i,u_i)$$

As a result, compared to k iterations of the baseline, the PIC preconditioner with k local iterations in the $i'th$ subproblem has a scaling factor of $(\frac{\omega.\beta_i}{\alpha})^{k-1}$ in its local stability condition (which also verifies our assumption that in case of only 1 local iteration, PIC and baseline have the same local stability). To arrive at the scaling factor that the PIC method imposes on top of the baseline convergence rate, we can replace local β_i with the maximum of β_i's, which we denote by $\beta = max(\beta_i), \forall i$. Note that as mentioned earlier, α represents the length of the input vector u and β_i (and in turn β) represent the length of the partitioned local input vectoru_i. Therefore, the ratio of $\frac{\beta}{\alpha}$ is a number between 0 and 1. The more we partition a problem, the smaller this ratio will get, and as one can expect, the rate of convergence will become slower.

As a final step, to be able to use the condition number formula of an additive Schwarz preconditioner [56, 57] we need to find a scaling factor that is comparable to one iteration of the baseline. Therefore, we have to use the k^{th} root of this scaling factor

$$\sqrt[k]{(\frac{\omega.\beta}{\alpha})^{k-1}} = (\frac{\omega.\beta}{\alpha})^{\frac{k-1}{k}}$$

in the following formula to compute the condition number of the PIC compared to the baseline. In general, the condition number of an additive Schwarz preconditioner satisfies [57, 56]:

$$\kappa(P_{ad}) \leq {C_0}^2 \omega(\rho(\epsilon) + 1)$$

Therefore, the condition number of the PIC method, using the parameters computed so far, is as follows:

$$\kappa(P_{ad}) \leq {C_0}^2 (\frac{\omega.\beta}{\alpha})^{\frac{k-1}{k}} \omega(\rho(\epsilon) + 1) \qquad \square$$

7.2.4 Conclusions

Through this document a few major insights have become clear.

- In the problems that PIC targets, the interdependency matrix should have a semi-block-tridiagonal matrix. The amount of discrepancy between the PIC solution and the baseline depends on the impact of the triangular blocks between the major blocks.

- The previous insight can be quantified by the ϵ matrix from the strengthened Cauchy-Schwarz inequality condition. $\rho(\epsilon)$, representing the largest eigenvalue of the ϵ matrix, puts a bound on the "decomposability" of the problem and the quality of the partitioning function at the same time.

- Compared to the baseline algorithm's convergence rate, the PIC implementations converge to the same answer with a scaling factor in their convergence rate. The scaling factor is $(\frac{\omega.\beta}{\alpha})^{\frac{k-1}{k}}$. Therefore the convergence rate decrease is proportional to a function of the number of local iterations, the convergence power of the algorithm represented by ω, and the ratio of the length of non-partitioned input vectors to their partitioned counterparts.

CHAPTER 8

EXPERIMENTAL RESULTS

8.1 MITHRA

8.1.1 Threaded Implementation

In this implementation, we use standard POSIX threads to compute the option price on an dual quad-core Intel E5410 2.33 GHz machine with 4 GB memory. To do so, we apply a four-step process:

1. Generate a pool of sample points for evaluation.
2. Break the pool of sample points into n sub-pools, one for each thread.
3. Each thread evaluates the Black Scholes formula over all the sample points.
4. Mathematically, we average the means and sum the variances to compute the overall mean and standard deviation of the equal-sized buckets.

One issue encountered for this implementation was the use of the standard C-library *rand()* function because the function is not re-entrant and thus locking is required to maintain state consistency. Instead, we use the re-entrant *rand_r()* function to avoid contention issues.

Figure 8.1 depicts the performance of the threaded version of Black Scholes. We simulate the single-threaded version of the program by simply running the algorithm with a single thread. In each case, the map portion dominates the computation while the reduce accounts for less than one percent of the total execution time.

The performance improves nearly linearly until four cores are used and then performance increases sub-linearly afterward. At four threads, we notice that

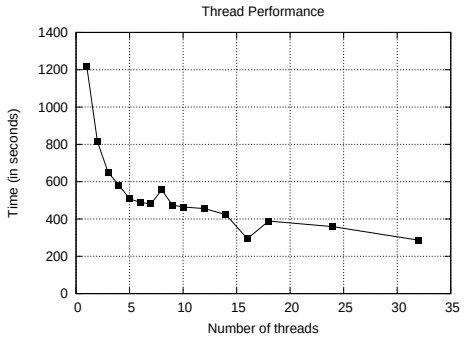

Figure 8.1: Threaded execution performance.

the performance increases start to taper off. We suspect that this is due to the hardware organization of our machine which has four cores per processor. Running the other processor is likely to diminish our returns since the L2 caches are not shared. Furthermore, there are anomalous points at eight and 16 threads. Since our machines are hyper-threaded, it is not surprising to see that the CPUs perform best at 16 threads since that is the number of virtual threads available. Past that point, however, the thread scheduling contention severely limits further performance gains.

To explain the anomaly at eight threads, we created CPU usages graphs depicted in Figures 8.2 and 8.3. It is clearly evident that the processor is not fully utilized in the seven thread case, and perhaps it is waiting on memory fetches. However, in the eight thread case, the processor is running at full steam. The bottleneck in one case is the physical limitation of the processor where all threads run in synchrony in the eight thread case while the seven thread case has some dispersion.

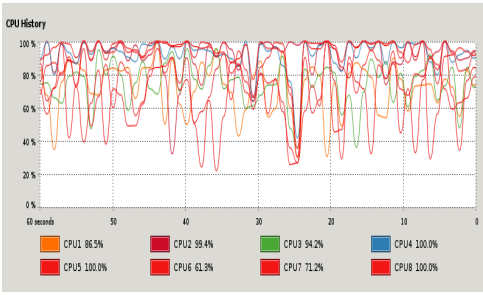

Figure 8.2: Seven thread CPU graph.

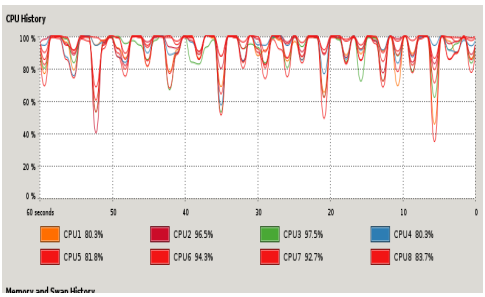

Figure 8.3: Eight thread CPU graph.

8.1.2 Phoenix Implementation

Phoenix [59] is a MapReduce framework for multicore/multiprocessor machines. MapReduce takes a set of inputs, splits it into batches, then processes each input using a mapper. The mapper then emits key value pairs into intermediate storage and the reduce function then sorts all key value pairs and runs each pair with identical keys into a reducer. At the end of the process, a result is emitted. In Phoenix, as in Hadoop (discussed in the next section), the splitter first breaks up the batch of sample points over n mappers. Each mapper then evaluates the sampling function at each sample point and collects these intermediate outputs into a key value pair as per the MapReduce framework. The reducers then take the intermediate outputs and process them into the final output.

In our implementation, we use the same dual quad-core machine and vary the number of mappers. In this case, however, each mapper generates the sample points independently rather than reading from an input file, as our implementation is processor intensive rather than I/O intensive. Therefore, we made the conscious choice to generate the sample points at each mapper rather than incur the I/O costs related to reading the inputs from a mechanical disk.

Figure 8.4 shows the performance of the Phoenix engine plateauing as early as four cores and doubling the processing power seems to have diminishing returns. Nonetheless, the single-threaded version of Black Scholes on Phoenix runs slightly faster than our single threaded version of Black Scholes, probably due to the use of multiple threads in the runtime. Phoenix outperforms the multithreaded implementation with two to four cores where both implementations catch up. When operating at full speed, Phoenix beats the

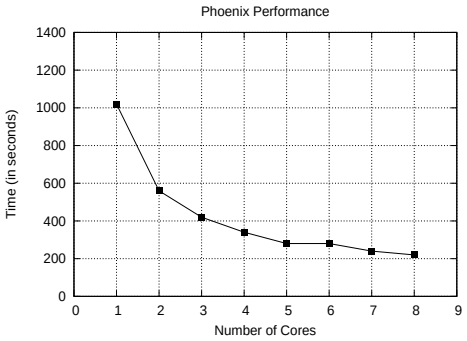

Figure 8.4: Phoenix performance.

multithreaded implementation by about 100 seconds. Presumably, this may be due to the extra parallelism in the MapReduce version of the algorithm.

8.1.3 Hadoop

We implemented Black Scholes on Hadoop (0.19) and ran it on our 496 core (62 nodes) Hadoop cluster.[1] In our design, each mapper is responsible for generating random sample points and evaluating the sampling function at each point. Upon completion, each mapper outputs two $(key, value)$ pairs as intermediate output: $(0, sumValues)$ and $(1, sumSquaresValues)$. We use the Combiner optimization mentioned in [20], in which we perform local reductions on each node and then emit $(0, sumLocalValues)$ and $(1, sumSquaresLocalValue)$. In the end, we use a single reducer to compute the mean and standard deviation over all the evaluated sample points. However, we did not aggressively optimize the implementation.

The cloud configuration dictates 4 mappers and 4 reducers per node. Each node is a dual quad core with 8 GB of RAM and 292GB of disk space. We run the MapReduce program with 32, 64, 128, 248 and 256 mappers. In each experiment, we perform a total of 4 billion iterations of the Black Scholes algorithm. We observed an increase in the run time when the number of mappers increases from 248 to 256 due to the increase in the overhead of the Hadoop framework and when there are more map tasks than the number of available map slots (62 * 4 = 248).

[1] http://cloud.cs.illinois.edu

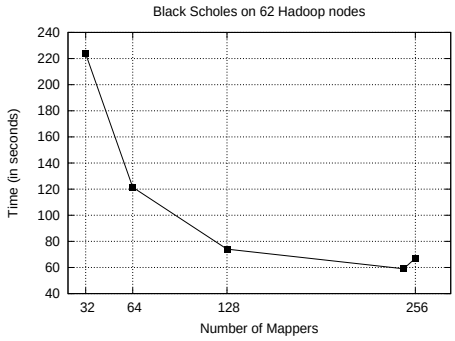

Figure 8.5: Hadoop performance.

8.1.4 CUDA Implementation

We also implemented the same Black Scholes calculations on NVIDIA's CUDA [53] general purpose GPU architecture. Rather than using the MARS [60] GPU MapReduce framework, we opted to design our system directly using NVIDIA's framework because it afforded us more control over execution. To ensure a fair comparison, we followed the MapReduce model introduced throughout this paper, where each computing element performs a single Black Scholes calculation in the Map phase. As mentioned earlier in the paper, the required random numbers are created in the GPU using a Niederreiter quasirandom generator [38].

We ran our experiments on two different NVIDIA GPUs. The first card is a 9800 GX2 with 2 GPU chips onboard, each having 128 processing elements running on a core clock of 600 MHz, and 512 MBytes of 256-bit bus GDDR3 for each chip. The second GPU card is a Quadro FX570 with 16 processing elements running on a core clock of 460 MHz and sporting 256 MBytes of 128 bit DDR2 RAM. The reduction is also programmed to run in the GPU using a binary tree reduction approach.

The first experiment was run on the 9800 GX2. We ran the Black Scholes computation for 4 billion iterations. Since such an array would be larger than the physical memory of our cards, we split the data set into smaller segments, and ran them in sequence. Each segment processes a data set of 64 MBytes, which counting for additional intermediate arrays takes less than 512 MBytes of memory and therefore fits in our GPU card memory. In total, the experiment ran for 24.51 seconds where the map stage finished in 8.34 seconds while the reduce took 16.2 seconds. By leveraging the parallelism

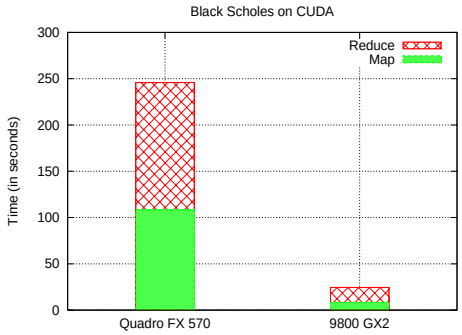

Figure 8.6: CUDA performance on two different NVIDIA GPUs.

inherent in CUDA, the map finishes in a short amount of time while the reduce was inherently less parallel. This is a stark contrast to the multi-core experiments, where the map stage took the majority of run-time. For the Quadro 570 with 16 cores, we reduce the size of the immediate array to 16 MBytes to match the size of the video memory. In this configuration, the total runtime is 245.77 seconds, where the map consumed 108.18 seconds and reduce took 137.59 seconds. The results are shown in Figure 8.6.

8.1.5 MITHRA

MITHRA combines the benefits of Hadoop's distributed computing with CUDA's raw processing power. Our technique requires no change to the Hadoop scheduler or any part of its core. Hadoop streaming [61] allows programmers to specify native binaries as mappers and reducers. Our mapper is a native Linux process which utilizes the unmodified NVIDIA CUDA version of Black Scholes mentioned earlier. Each node then utilizes the full power of its GPUs and performs the mapping process at full steam. Once the mappers complete, the local reducers (like the combiner optimization in Hadoop) begin to process the aggregate results using the binary tree reduction optimization. These intermediate values are then passed through the Hadoop framework to a single reducer which performs the final computation of means and standard deviation over all the evaluated sample points. As shown in Figure 8.7, we perform four experiments that vary the number of GPUs used running a total of 4 billion iterations of the Black-Scholes algorithm. In the first experiment, only a single node is used in Hadoop and only one CUDA

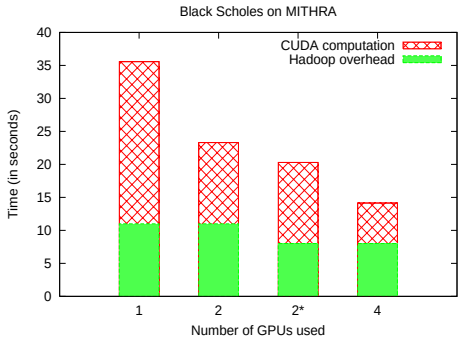

Figure 8.7: Mithra performance.

card is utilized. The Hadoop overhead is estimated by using an identity mapper and an identity reducer, which shows the amount of time taken by the Hadoop framework to fetch the inputs, start the program and write the output. Since the 9800 GX2 cards are actually 2 cards in one package (they even have two separate PCBs), and since most of the computation of both Map and Reduce are performed in GPUs, in the next experiment we utilize these 2 GPUs on the same machine. Our assumption here is that since the host CPU is not involved in either Map or Reduce, we can treat each physical host as two nodes of the Mithra cluster. Of course Hadoop has its own overhead, therefore this results in slower completion than if a single GPU is used on 2 machines (shown as 2*). Finally, we run our experiment utilizing both the CUDA cards on both machines.

Comparing the result with those of Hadoop, we see that a MITHRA cluster of 4 nodes runs the total computation in 14.4 seconds, while the Hadoop cluster takes 59 seconds. This means that for each node, the MITHRA architecture has $(59s \cdot 4map \cdot 62nodes)/(14.4s \cdot 4GPUs) = 254$ times performance improvement.

8.2 PIC Experimental Results

In this section, we report the results of executing five applications using PIC on three clusters of different sizes - a small research cluster, a medium sized production cluster and a large virtual cluster hosted on Amazon's Elastic MapReduce service. Section 8.2.1 describes the experimental setup in detail. The speedup of PIC implementations compared to the baseline IC imple-

mentations (which already employ known optimizations such as combiners, and elimination of repeated initialization overheads and input reads) are reported in Sections 8.2.2 and 8.2.3. In Section 8.2.4 we present insights into the speedups obtained by PIC. We show that the amount of communication traffic on the cluster interconnect due to MapReduce intermediate data and model updates are significantly reduced over the baseline implementations.

8.2.1 Experimental Setup

We have implemented the PIC library on top of the Apache Hadoop framework [21]. We ported conventional IC implementations of five applications (K-means clustering, PageRank, neural network training, linear equation solver, and image smoothing) into PIC. We compare IC (our baseline implementation, which was developed using Hadoop directly without PIC) and the PIC implementations. We ran experiments on three different clusters, which we refer to as *small*, *medium* and *large*, to demonstrate the use of the PIC library, and the benefits that accrue by using PIC at different cluster scales. The small cluster is a 6-node research testbed that uses Gigabit Ethernet as the cluster interconnect. Each node has two quad-core E5520 Xeon processors running at 2.27GHz (8 physical cores and hyper-threading support), with 48 GB of RAM. From Hadoop's point of view there are a total of 24 map and 24 reduce task slots on this cluster. The medium testbed is a 64-node shared production cluster. Each node has two quad-core E5430 Xeon processors running at 2.66GHz, with 16 GB of RAM. The medium cluster occupies 6 racks, and the nodes are connected to each other by using a Gigabit Ethernet switch. We used 330 map and 110 reduce task slots. Finally, the large testbed consists of 256 Amazon Elastic MapReduce extra large instances. Each instance has 15 GB of memory and 8 EC2 Compute Units (4 virtual cores with 2 EC2 Compute Units each).

Although recent proposals [10, 12, 11] have managed to reduce the overheads for repeatedly launching Map and Reduce tasks in each iteration, these optimizations are not yet available in Hadoop. Therefore, we took the following approach in order to remove the effect of these optimizations before evaluating PIC. We recorded the number of iterations that the baseline algorithm executed. We then ran a program that looped for the same number

of iterations, and in each iteration created a MapReduce job that reads the input data but does not perform any processing. The execution time of this job was subtracted from the baseline to account for the performance improvements from elimination of repeated job initialization and repeated input reads.

8.2.2 Speedups

Figure 8.8 shows the speedup experienced in the small cluster for three different applications: K-means, PageRank and linear equation solver. The K-means experiment was designed to cluster 5 million points into 100 clusters. The IC implementation of the PageRank algorithm was taken from the Nutch open source search engine (version 1.1). The Nutch implementation considers the results of the PageRank acceptable after 10 iterations. As the input web graph, we use the wikipedia.org website that contains 1.8 million documents. To implement the PageRank algorithm using PIC, our partitioning function randomly divides the web graph into 18 partitions, each having about 100,000 vertices. The cross-edges between partitions are also grouped into $18^2 = 324$ sets. For the linear equation solver, we used an example of a linear system of 100 variables with a weakly diagonal dominant matrix. In each case, the problem size was chosen to ensure that the baseline execution took about 1 hour on the cluster (for practical reasons). Such a large time window was chosen to minimize the impact of Hadoop job starting and finishing overhead (which is in the order of seconds). The results presented in Figure 8.8 show that PIC results in 2.5X-4X performance improvement over the baseline IC implementations.

Figure 8.9 shows the speedups for a medium sized cluster (64 nodes) for K-means, neural network training, and image smoothing. This time, a larger data set was used for the K-means experiment, namely 10 million points distributed in a 3-dimensional space. The neural network training application used a dataset of about 210,000 optical character recognition (OCR) training vectors. Finally a large 40 megapixel image was used as the dataset for the image smoother. Once again, the problem sizes were chosen such that the baseline execution took about one hour. We see that PIC still manages to outperform the baseline IC implementation by a factor of between 2.5X and

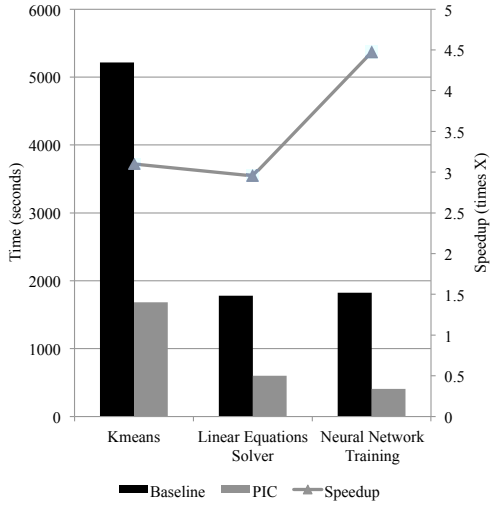

Figure 8.8: Performance of PIC and baseline IC on a small (6 node) cluster

4X.

The dataset sizes of the K-means and neural network training were increased when moving from the small cluster into the medium sized cluster. The main reason for this was to both ensure that there is enough work to utilize the whole cluster fully. These results demonstrate weak scalability of the PIC library.

8.2.3 Speedup: Strong Scaling on Large Clusters

To measure the impact of PIC on strong scalability, we performed experiments on Amazon Elastic MapReduce using 256 extra large instances (256 nodes, 8 cores each). In these experiments, the dataset size was fixed, while we scaled the number of nodes from 64 to 128 to 192 and finally to 256 nodes. Figure 8.10 shows the results of our experiments for the image smoothing application, shows that, for up to 256 nodes, the speedups of PIC over the baseline implementation are maintained. Furthermore, we can conclude that the PIC library does not have any negative impact on the scalability of Hadoop.

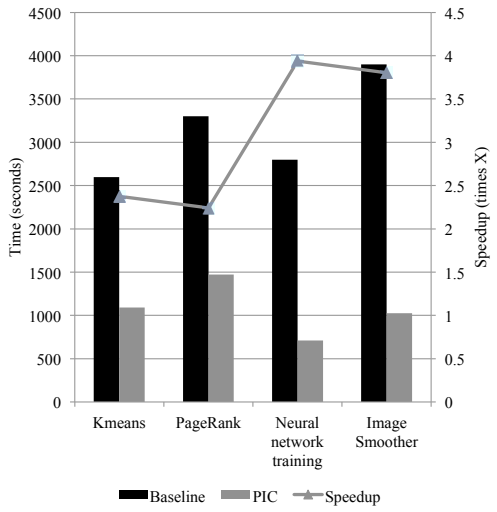

Figure 8.9: Performance of PIC vs. baseline IC on a medium (64 node) cluster.

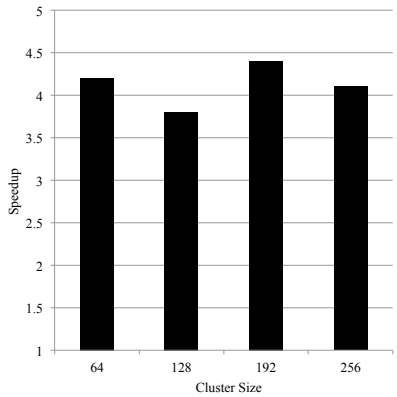

Figure 8.10: Strong scalability of the PIC speedup over IC baseline for the image smoothing application. The horizontal axis shows the number of computing nodes (each node has 8 processing cores) and the vertical axis shows the speedup of PIC vs. baseline IC.

Table 8.1: Iterations required for IC and PIC implementation of K-Means

DataSet Size	0.5M					5M				50M			500M		
Baseline IC Iterations	32					32				31			31		
Global Iterations (PIC)	5					4				3			3		
Local Iterations (PIC)	34	3	3	2	2	34	3	2	2	33	2	2	33	2	2

8.2.4 Analysis of Speedups

The speedups obtained by PIC in practice are due to two key factors: (i) in the best-effort phase the number of best-effort iterations is very small (relative to the number of iterations executed by a conventional implementation). This is critical since cluster-wide communication is incurred once per best-effort iteration. (ii) The number of iterations executed in the top-off phase is also small. In this section, we present data to quantitatively demonstrate these factors.

For K-means, Table 8.1 shows the number of best-effort iterations and the number of local iterations required in each best-effort iteration. Note that, except for the first best-effort iteration, only 2-3 local iterations are necessary in any best-effort iteration. Similar results were observed for all other applications.

Table 8.2: Breakdown of data read or generated during K-means clustering of 500 million points

	1 Baseline It. (IC)	Total Baseline (IC)	Total PIC
Intermediate data	9.21 GB	285.68 GB	80.9 KB
Model updates	30 KB	959.03 KB	92.23 KB

Table 8.2 shows the volume of intermediate data (mapper output) and model updates in the K-means application while clustering 500 million data points using the IC and PIC schemes on the small cluster (6 nodes, 8 processors each). The first column shows the breakdown for a single iteration of the baseline IC implementation. The second column shows the cumulative results for all the iterations required by the IC implementation. The third column corresponds to the PIC implementation. We can clearly see that the PIC implementation drastically reduces intermediate data and model updates. This disparity is in spite of the fact that all our baseline implementations utilize combiner optimizations.

In summary, our results indicate that reduction in communication traffic

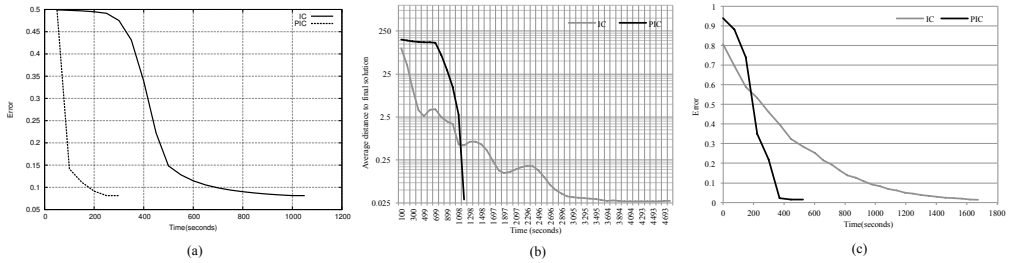

Figure 8.11: Accuracy of results vs. time for (a) neural network training, (b) K-means clustering and (c) solving a system of linear equations.

is a primary contributing factor to the speedups achieved by PIC. Since communication only becomes a more severe bottleneck for larger clusters, we believe that the benefits of PIC will be sustained if not enhanced with an increase in cluster size.

8.2.5 Effectiveness of PIC's Best-Effort Phase

The performance improvements obtained by PIC are in large part due to the effectiveness of the best-effort phase in computing a high-quality model, and the fact that it does so in a much shorter time than the original IC computation itself. For all our applications, we observed that the results produced by the best-effort phase are very close in quality to the final solution (in some cases, even identical), necessitating very few iterations in the top-off phase.

In this section we empirically evaluate the quality of the model generated by PIC's best-effort phase by comparing it to the results of the baseline IC implementation. We also briefly mention some analytical insights to explain our results. As a special case, we show that when the computations in an iteration are linear and the problem has the "nearly uncoupled" property, the best-effort phase of PIC can be analytically shown to converge to the same solution as the baseline IC implementation.

8.2.6 Error vs. Time

To illustrate the effectiveness of the best-effort phase, we evaluated the error of the models computed in each best-effort iteration, and plotted error vs. time at which each best-effort iteration completes. We compare these trajectories to the ones obtained from conventional IC implementations.

We note that the definition of quality or error is application-specific, therefore we consider each application separately.

For neural network training, the error was evaluated during training by applying the model to a validation data set and measuring the fraction of data points that are mis-classified (model error). Figure 8.11(a) shows the trajectory of the model error *vs.* time for the neural network training algorithm. The figure shows that the PIC implementation manages to reach a model error that is virtually identical to what the baseline implementation eventually achieves, but in less than a quarter of the time.

For K-means clustering, we computed the final solution (centroids) produced by a sequential implementation and used the distance to this reference solution as the error metric. Figure 8.11(b) plots the centroid displacement from iteration to iteration in the best-effort phase of PIC as well as the baseline implementation. Again, we see that the centroids converge much faster in the best-effort phase of PIC.

We also used the Jagota index [62], a popular metric to evaluate the quality of clustering algorithms, to compare the model computed by the best-effort phase of PIC with the model computed by the original IC implementation. The Jagota index measures the tightness or homogeneity of points within the clusters and is defined as follows:

$$Q = \sum_{i=1}^{k} \frac{1}{|C_i|} \sum_{x \in C_i} d(x, \mu_i)$$

where $d(x, \mu_i)$ is the distance between a point x and the centroid μ_i that it belongs to, and $|C_i|$ is the number of points in the i-th cluster. The results for two data sets, shown in Table 8.3, suggest that the best-effort phase of PIC is able to produce a solution that is within 3% of the quality of the baseline IC implementation, enabling the top-off phase to terminate in a small number of iterations.

For the system of linear equations, there exists a unique golden solution.

Table 8.3: Quality of best-effort phase of PIC in terms of Jagota index (K-means)

	Dataset 1	Dataset 2
IC K-means	2.109	2.146
PIC BE Phase K-means	2.112	2.205
Difference(%)	0.14%	2.75%

We use the distance to this solution as the error metric, and plot error *vs.* time for the best-effort phase of PIC and the conventional IC implementation in Figure 8.11(c). Again, we see that the best-effort phase of PIC produces comparable quality to the conventional IC implementation in one-third the time.

Similar accuracy results were obtained for the other case studies, validating our hypothesis that the best-effort phase of PIC is effective in generating a high-quality model in a much shorter time than conventional IC implementations.

8.2.7 Analysis as a Preconditioner

We provide some analytical insights to explain the effectiveness of the best-effort phase of PIC. In other words: How can we know that an algorithm implemented in PIC's best-effort phase will not diverge or create drastically different results than the baseline?

In Section 7.2 we detailed a mathematical framework by drawing an analogy between the best-effort phase of PIC for the case of algorithms that perform linear computations in each iteration (e.g., PageRank, linear equation solver, image smoothing) and additive Schwarz preconditioners from the field of domain Decomposition. Without repeating the analysis in this section, we summarize the key insights.

We believe that the effectiveness of the best-effort phase of PIC is explained by the nature of applications that PIC targets, namely applications that are nearly uncoupled. The dependency patterns of such applications can be approximated using nearly block diagonal matrices, as depicted in Figure 8.12.

Additive Schwarz preconditioners deal with similar problems. Note that in differential equations (where domain decomposition techniques are tradi-

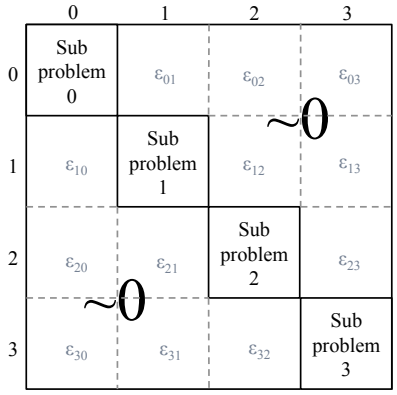

Figure 8.12: The ideal dependency matrix of an application that PIC can successfully target. The dependency between different partitions, shown by $\epsilon_{ij}, i \neq j$, should be minimal (symbolized by ~ 0) for PIC to be effective.

tionally used), the stencil based computation ensures the desired dependency patterns. However, stencil based algorithms are not the only algorithms with the desired dependency patterns.

As an example, consider the PageRank algorithm. In a parallel implementation, each node requires information from the adjacent nodes and edges for the computation. If the web was a complete graph with $\frac{n.n-1}{2}$ edges it would not be a good match for PIC implementation. Fortunately the web graph is typically local, and by properly partitioning it (for example using the METIS package), the connectivity matrix of the graph becomes nearly uncoupled.

Similar arguments apply to the K-means clustering. The impact of far-away points on a centroid is much smaller than the impact of close points to that centroid. As such, a rough first-pass partitioning can ensure that each sub-problem mostly relies on the points inside that partition.

The image smoothing algorithm is stencil based and clearly the dependencies are local. In the case of the linear system of equations, a *weak diagonal dominant* matrix property guarantees the *nearly-uncoupled* property. In fact, the weak diagonal dominance property is powerful enough to ensure even asynchronous convergence [43].

Section 7.2 shows that convergence rate of the best-effort phase can be found from the convergence rate of the baseline algorithm with a scaling factor, as follows:

$$(\omega\frac{\beta}{\alpha})^{\frac{k-1}{k}}$$

where $\frac{\beta}{\alpha}$ is the ratio of the maximum length of input partitioned vectors to the length of the unpartitioned vector. ω is a measure of the converging power of the iterative function, and can be derived from the "local stability" condition introduced in section 7.2, and k is the number of local iterations.

It is evident that more partitions translate to a slower convergence rate in the best-effort phase, but as we have seen earlier, the increased locality in the problems allows much faster local iterations by reducing network traffic, and performing computations locally.

8.3 Loop Maximizing

The baseline system has an Intel(R) Core i7 920 quad-core CPU with a clock frequency of 2.67GHz, a 8MB L3 cache and hyperthreading support (it is represented with 8 virtual cores to the operating system). The GPU used was the NVIDIA(R) GeForce GTX 275 with 895MB of global memory. Our baseline program uses OpenMP to utilize multi-core CPUs, and in all of our experiments the baseline is compiled with -O3 optimization level.

A range of input data sizes was utilized in this experiment to show the impact of the number of sequences to be aligned on the speedup. The data line in Figure 8.13 labeled as "GPU Registers" shows the speedup of the GPU algorithm with the loop maximizing transformation applied compared to the baseline multi-threaded implementation running on the testbed system (8 hyper threaded cores). The results show a speedup of up to **9.3 times** compared to the optimized (compiled with -O3 option) multi-threaded implementation on a quad-core CPU for 12582720 input pairs. The multi threaded CPU implementation takes about 65 seconds to finish aligning the 12582720 input pairs, while our GPU implementation goes through them in only 6.95 seconds.

As can be seen from Figure 8.13, the speedup of our GPU algorithm increases as the input size (number of pairs to be compared) increases. This effect is not consistent for the CPU. This can be attributed due to the fact that for smaller number of sequences, all the sequences fit in the cache of the CPU, whereas when the size of the sequences increases beyond a certain

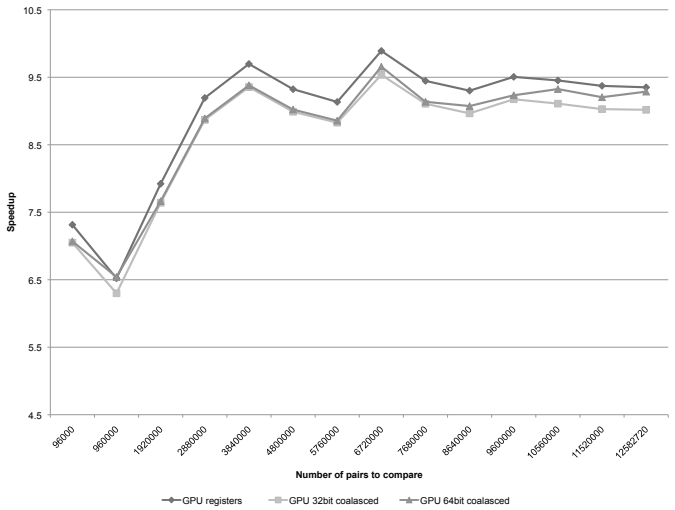

Figure 8.13: Speedup of our proposed algorithm implemented in a GPU compared to an optimized multi-threaded implementation in a quad-core CPU. Three versions of the GPU implementation are depicted above. At the largest input data set of over 12 million pairs, our shared memory algorithm runs 9.3 times faster.

threshold, the CPU starts incurring cache misses and hence the performance decreases. The increase in the number of sequences does not affect the time taken by the GPU to process the genes which, has a linear relationship with the number of sequences. This is because the GPU implementation keeps all the sequences to be aligned in the global memory and accesses them in a coalesced manner. The rest of the calculation (the Needleman Wunsch matrix calculation and the backtracing) are all done in the shared memory and the per-thread memory (registers). This calculation is independent of the number of sequences and hence does not affect the GPU performance like it does the CPU performance.

Please note that the achieved speedup is gained by allowing the GPU to run regular non-divergent code through loop-maximizing, even if it ends up running more instructions. We did run the results of loop maximizing transformation in a single-threaded CPU, and observed a 2.7 times *higher* runtime. Then again, the impressive speedups presented earlier are the result of running our algorithm on a SIMD architecture such as GPUs. Looking at these findings, we can argue that researchers have to think of different code optimizations for SIMD architectures such as GPUs, which is in contrast to the real-world practice of industry, where the legacy code optimization

techniques of CPUs are commonly applied to GPU programs.

8.4 Plasma Experimental Evaluation

To investigate the performance benefits of Plasma, we have developed a simple pointer chasing kernel. Each thread is required to allocate an array, fill it with random integer numbers between 0 and *SIZE - 1* (random numbers are provided by the host), and then perform a loop on this array. At each iteration, the value of the current element of the array will determine which element to be read next.

The baseline kernel is implemented as follows:

```
uint8_t array[SIZE];
//1.Fill the array with random numbers
//2.Now the pointer chasing loop:
for (i = 0; i < loops ; i++)
  pointer = array[pointer];
```

A similar pointer chasing program was written with plasma:

```
uint16_t array = sm_malloc(SIZE);
//1.Fill the array with random numbers
//2.Now the pointer chasing loop:
for (i = 0; i < loops ; i++)
  pointer = gv8(array, pointer);
```

Each kernel was run with 4 different loop iterations (10, 100, 1000 and 10000), and with different sizes for the array using an NVIDIA GTX480 GPU. The plasma kernel uses 48KB of shared memory while the baseline is configured to use 48KB of L1 cache. The results are depicted in Figure 8.14, and make clear that Plasma can provide a significant speedup with not much difficulty for the programmer under certain operating conditions. .

It is clear from Figure 8.14 that an optimal operation region exists where Plasma drastically outperforms the baseline. This region takes place when the array size is between 48 and 512 bytes, and a large number of array accesses are performed in the kernel. The reason for the deteriorating speedup in larger array sizes can be attributed to the kernel occupancy.

In Fermi, a 50% occupancy can fully mask all pipeline latencies, and even an occupancy of 20% can keep the instruction throughput and memory band-

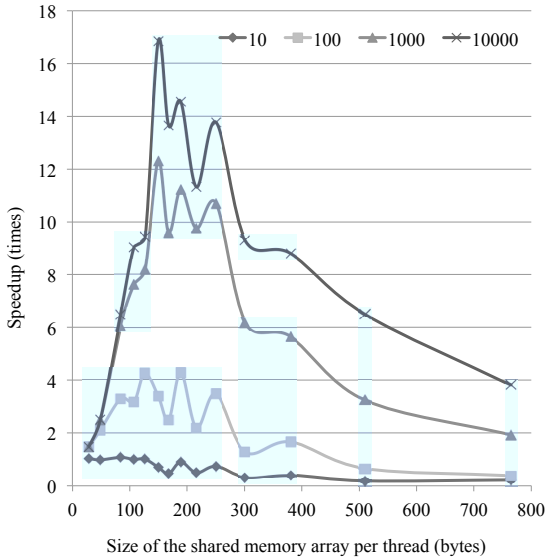

Figure 8.14: The speedup of Plasma over the baseline for pointer chasing operations. There is an optimal operation window in which Plasma drastically out-performs the baseline.

width above 90% of the respective maximums [63]. However, at 10% occupancy (memory size of 306 bytes according to the Table 6.1), only 50% of peak performance can be attainable. At 767 bytes with an occupancy of 4%, the GPU will be limited to 20% of its peak performance. The shared memory pressure in Plasma reduces the occupancy, while the baseline kernel typically runs at around 50% or more. The combination of these factors results in a reduced speedup in Figure 8.14 for large arrays.

Figure 8.14 also shows that when the number of array accesses in the kernel is small (around 10), the overhead of the plasma library makes its use ineffective. To be efficient, the kernel should perform lots of array accesses, as the best speedup curve is achieved with 10,000 loops. Nevertheless, the Plasma system can provide ample speedup with proper operating conditions. Similar to other tools, it is up to a user to fully utilize its potential by using it where it can be most effective.

CHAPTER 9

RELATED WORK

9.1 MITHRA

MITHRA is based on several well researched ideas and projects. It is built upon the Hadoop [21] open source MapReduce framework, and extends it with the ability to run GPGPU kernels [53]. Even though MapReduce [20] and Hadoop are recent developments, their basic mathematical underpinnings were discussed as early as 1987 [64], and have been well studied afterwards [65, 66, 67, 68]. List homomorphisms research is typically more theory oriented, however, and to the best of our knowledge has not been utilized in practical applications and architectures thus far. Other formalism also exists for MapReduce types of computation [29, 69]. Our formalism is an extension of that presented in [29] with some modifications to support the specifics of MapReduce.

The simplicity of the MapReduce model has long been one of its most attractive features, resulting in several implementations of various flavors and widespread practical use. Aside from typical cluster implementations [20, 21], other implementations of this framework exists for other platforms as well. Mars [60] is another attempt at bringing the MapReduce paradigm to a non-distributed system, implemented on a single NVIDIA G80 GPU. It aims to hide the complexities of GPU programming, while still achieving strong performance. However, reference [60] does not provide a solution for scalability. Reference [70] brought the MapReduce paradigm to heterogeneous multi-core systems. By taking a high-level library-based approach, they turn the usually ad-hoc and exhaustive approach to programming for heterogeneous parallel systems into a more manageable one, giving strong scalability with minimal programmer effort.

There has been lots of recent work on cluster of GPUs. Fan et al. [71] pro-

posed using cluster of GPUs for scientific computing. Their work preceded the availability of general-purpose GPU programming platforms; therefore they use GPGPU techniques to masquerade their computation as a graphics application. Zippy [72] abstracts GPU cluster programming with a two-level parallelism hierarchy and a non-uniform memory access model. They adapt the Global Arrays programming model to the GPU cluster model and combine it with the stream-processing model. A scalable parallel framework based on MapReduce has been built by Tu et al. [73] for analyzing tera-scale molecular dynamics simulation trajectories. Similarly, Phillips et al. [74] describe strategies for the decomposition and scheduling of computation among CPU cores and GPUs, and techniques for overlapping communication and CPU computation with GPU for kernel execution for NAMD. A similar approach is recently used by Herero-Lopez [15] for accelerating SVMs by integrating GPUs into MapReduce clusters. Finally, a similar attempt to MITHRA [16] was published in 2010 by Shirahata et al. [75].

9.2 PIC

High-level programming frameworks [20, 21, 76, 77, 78] have emerged as the preferred programming model for developing and deploying applications on shared-nothing clusters of unreliable machines. Iterative-convergence algorithms are commonly implemented using these frameworks; yet, none of these frameworks provide explicit support or optimizations for iterative-convergence algorithms. For example, the Apache Mahout [13] project builds machine learning libraries on top of Hadoop. Most machine learning algorithms are iterative, and Mahout uses a driver program to launch new MapReduce jobs in each iteration, resulting in the inefficiencies described in earlier sections.

Recently, several noteworthy attempts have been made to augment MapReduce frameworks for iterative computations [10, 11, 12]. Twister [10] is a stream-based framework that extends the basic MapReduce framework by explicitly avoiding repeated mapper data loading from disks. Twister also uses mappers and reducers that are long running with distributed caches. The SPARK [12] framework targets data intensive applications that reuse a working set of data across multiple parallel operations. They introduce

the concept of resilient distributed datasets, which refer to a read-only collection of objects partitioned across a set of machines that can be rebuilt if a particular partition is lost. HaLoop [11] extends Hadoop by introducing iterations into the programming model and several optimizations that include loop aware task scheduling, loop-invariant data caching and strategies for efficient fixed point verification. A recent effort reported in [14] demonstrates how asynchronous algorithms [43] can be realized using MapReduce with improved efficiency. A common attribute of our work and [14] is that both do not preserve numerical equivalence with a sequential implementation. However, unlike asynchronous algorithms, where the communication between parallel computations does not occur at pre-determined synchronization points and can lead to results depending on execution timing, PIC is fully synchronous and deterministic (i.e., for a given input data set and number of partitions, the computed model will be the same across multiple runs on the same cluster, clusters with nodes of different speeds, or different sized clusters.) iMapReduce [79] is a recent effort that combines some of the above techniques. A common thread across all these attempts is to push the support for iterative programs into the MapReduce programming model and runtime framework to improve the performance of iterative-convergence applications. In the context of graph-based computations, programming frameworks such as Pregel have demonstrated the potential for improved performance over MapReduce [80]. Although we realize the PageRank algorithm using PIC, we node that PIC targets any iterative-convergence algorithm, not just graph-based algorithms.

PIC exposes nested parallelism (across partitions and within a partition), which can also be achieved using frameworks such as NESL [81]. However, PIC goes well beyond nested parallelism frameworks by focusing on iterative-convergence algorithms and by leveraging the inherent forgiving nature of applications in its best effort phase.

An open-source effort that is gaining ground is the Apache Mahout [13] project, which builds machine learning libraries on top of Hadoop. Most machine learning algorithms are iterative, and Mahout uses a driver program to launch new MapReduce jobs in each iteration. Unlike HaLoop or Twister, Mahout does not modify the basic MapReduce framework.

None of the prior systems explicitly address the significant challenges of (a) reducing disk and network traffic due to MapReduce intermediate data

and model updates or (b) introducing parallelism across partitions. Augmenting the basic MapReduce framework to support iterations is useful, but cannot overcome these challenges. By leveraging the unique forgiving nature of applications that employ iterative-convergence, we are able to achieve significant performance and scalability advantage that is beyond most existing proposals. Moreover, PIC does not require any changes to the MapReduce framework, and it is possible to reuse an existing MapReduce based implementation when developing a PIC implementation, reducing programmer effort.

The forgiving nature of iterative-convergence computations has been recently exploited to improve performance and scalability on multi-cores and GPUs [82, 83, 84, 85], but not in the context of clusters. Meng et al. present a best-effort parallel execution framework for Recognition and Mining applications on multi-core platforms, based on the properties of iterative-convergence algorithms [82, 83]. They propose a variety of *best-effort* computing strategies.

None of the prior systems explicitly address the significant challenges of (a) reducing disk and network traffic due to MapReduce intermediate data and model updates or (b) introducing parallelism across partitions. Augmenting the basic MapReduce framework to support iterations is useful, but cannot overcome these challenges. By leveraging the unique forgiving nature of applications that employ iterative-convergence, we are able to achieve significant performance and scalability advantage that is beyond most existing proposals. Moreover, PIC does not require any changes to the MapReduce framework, and it is possible to reuse an existing MapReduce based implementation when developing a PIC implementation, reducing programmer effort.

9.3 Related Theoretical Works

A number of compile-time optimizations are proposed in [86], including some that do not preserve the accuracy of the final results. Similar approaches have been applied in the context of stencil computations [85, 87]. These techniques significantly differ from our proposal, and are in fact of limited use for clusters. For example, some of these techniques reduce the amount of

work performed whereas our proposal may actually increase the amount of work in order to improve parallel scalability. Moreover, the communication traffic due to MapReduce intermediate data and model updates is not a significant concern in multi-core platforms where hardware-supported coherence and shared memory are present. In contrast, PIC is specifically targeted at clusters, where communication overheads are at a totally different scale and parallelism must be exploited at a much coarser granularity.

Similar problems have been studied in the Operations Research community [88, 89] for nearly uncoupled problems, also known as nearly completely decomposable problems. [89] shows that an aggregation/disaggregation technique (including techniques such as Koury, Mcallister, Stewart, Takahashi and Vantilborgh) can compute Markov chain steady state solutions through a process similar to PIC's best-effort phase. Furthermore, they converge to the accurate solution with a certain error bound if a set of regulatory conditions are satisfied for the Markov chain matrix. These regulatory conditions dictate the level of decomposability of the problem, and in essence are equivalent to the conditions that a domain decomposition technique needs to satisfy, which were mentioned in section 7.2. Takahashi et al. [88] extend this notion to a general class of problems, namely the fixed point of non-expansive mappings (a sub-class of contraction mappings). It is shown by Bertsekas in [43] that many interesting non-linear (and linear) operators such as Jacobi and Gauss-Seidel operators and their block variants, as well as optimization techniques including gradient projection, are indeed contraction mappings, and as such the Takahashi aggregation/disaggregation iterative method can be applicable on them. In a related effort, [43] analyzes many contraction mapping operators used for iterative algorithms. It proposes to go a step beyond the aggregation/disaggregation method of Takahashi by synchronizing (aggregating) the sub-problems asynchronously. Similar to the techniques in [56, 57, 88, 89], the conditions on the decomposability of the problem guarantee the convergence. The conditions imposed by [43] are more restrictive than the ones proposed by Takahashi [88, 89]. In turn, the conditions imposed by additive Schwarz domain decomposition methods [56, 57] are slightly less restrictive than the Takahashi methods, since these methods are considered as preconditioners for further application of Krylov subspace methods (e.g. conjugate gradient methods), and therefore can withstand more inaccurate results [57]. Since the PIC best effort phase has the exact

same goals, we analyze the PIC methods as an additive Schwarz domain decomposition technique. However, the findings of [88] apply to PIC as well.

We would like to emphasize again that the difference between these analysis methods boils down to their decomposability requirements on the problem domain, and the levels of *contraction mapping* power in the function.

CHAPTER 10

FUTURE WORKS

So far in this thesis we reported on four projects (MITHRA, PIC, loop maximizing and the plasma library), all related to one common vision: enabling *compute-intensive data-analytics* (CIDA) applications on modern COTS clusters, specifically clusters of GPUs. As possible future works, the following come to mind.

1. **Integrate MITHRA and PIC with APARAPI**: As mentioned earlier, MITHRA used a user-supplied CUDA kernel as its *map()* function, while the reducers are written in java. It would be much more beneficial to allow the Hadoop users to write all their code in their familiar java language. Fortunately a new run-time translation tool has recently become available from AMD, called *Aparapi* [90]. Aparapi allows run-time translation of Java byte-codes into OpenCL and executing on GPUs. If for any reason Aparapi can not execute on the GPU it will execute in a Java thread pool. We envision integrating both MITHRA and PIC with Aparapi, and allowing the users a truly unified MapReduce model that runs on GPUs.

2. **Incorporate Loop-Maximizing as a code refactoring technique**: Currently Loop maximizing technique is a performed manually. We are working on integrating it with the Eclipse IDE as a code refactoring tool.

3. **Load balancing**: In PIC and to some extent in MITHRA, each computing node is assigned a partition of input data to work with for some time. If we can assume the amount of work is dependent on each partition's size, then all we need to do is to partition the data in roughly equal sizes. However, in some workloads, the amount of processing is related to the content of the data. We experienced this effect in [18],

where the human genome data is divided and processed locally, and based on the contents of a data split the processing can take shorter or longer. This problem cannot be easily handled by Hadoop's straggler reassignment mechanism. We believe this problem merits further study, but perhaps is out of the scope of this thesis.

CHAPTER 11

CONCLUSIONS

This research project aims to advance the performance of *Compute-Intensive Data-Analytics* (CIDA) applications on cloud computing infrastructure, specifically for CIDA applications implemented on the MapReduce programming framework.

We reported four research efforts to achieve this goal. The MITHRA project integrates Hadoop MapReduce and GPUs together, where the *map()* functions execute. We have shown that when MITHRA model is applicable (for instance for Monte Carlo algorithms), each computing node can perform orders of magnitude more work in the same run-time.

We then proposed *partitioned iterative convergence* (PIC) as an approach to realize iterative algorithms on clusters. We observed that conventional implementations of iterative algorithms using MapReduce are quite inefficient as a result of several factors. Complementary to prior work, we focused on addressing the challenges of high network traffic due to frequent model updates, and lack of parallelism across iterations. PIC partitions the problem, and runs the sub-problems where the locality can be exploited better. The results can be numerically inaccurate (about 3% based on experimental results), but the forgiving nature of iterative-convergence algorithms allows us to utilize PIC and still get acceptable answers in much faster run times.

Finally we introduced two GPU-based projects that try to increase the performance of MapReduce style functions in GPUs. The first one is '*loop maximizing*, a code transformation for GPUs that can eliminate code flow divergence and result in better usage of GPU processing elements. Using this technique, we have achieved the highest reported speedups for gene alignment algorithms. The second project is *plasma*, a library for dynamic shared memory allocation and access in GPUs assuming independent execution of the GPU threads, which can be satisfied in *map()* functions.

REFERENCES

[1] J. Rehr, F. Vila, J. Gardner, L. Svec, and M. Prange, "Scientific computing in the cloud," *Computing in Science Engineering*, vol. PP, no. 99, p. 1, 2011.

[2] U.S. Department of Energy (DOE) project DE-AC02-05CH11232 final report, "Magellan final report," Tech. Rep., 2012.

[3] G. Fedak, G. Fox, G. Antoniu, and H. He, "Future of MapReduce for scientific computing," in *Proceedings of the second international workshop on MapReduce and its applications*, ser. MapReduce '11. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/1996092.1996108, pp. 75–76.

[4] J. Ekanayake, X. Qiu, T. Gunarathne, S. Beason, and G. C. Fox, *High Performance Parallel Computing with Clouds and Cloud Technologies*. CRC Press (Taylor and Francis), 2010.

[5] C. Evangelinos and C. N. Hill, "Cloud computing for parallel scientific HPC applications: Feasibility of running coupled atmosphere-ocean climate models on Amazon EC2," *Cloud Computing and Its Applications (CCA08)*, 2008.

[6] A. Iosup, S. Ostermann, M. Yigitbasi, R. Prodan, T. Fahringer, and D. Epema, "Performance analysis of cloud computing services for many-tasks scientific computing," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 22, no. 6, pp. 931 –945, June 2011.

[7] K. Jackson, L. Ramakrishnan, K. Muriki, S. Canon, S. Cholia, J. Shalf, H. Wasserman, and N. Wright, "Performance analysis of high performance computing applications on the amazon web services cloud," in *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, December 2010, pp. 159 –168.

[8] J. Napper and P. Bientinesi, "Can cloud computing reach the top500?" in *Proceedings of the combined workshops on UnConventional high performance computing workshop plus memory access workshop*, ser. UCHPC-MAW '09. New York, NY, USA: ACM, 2009. [Online]. Available: http://doi.acm.org/10.1145/1531666.1531671, pp. 17–20.

[9] A. Gupta and D. Milojicic, "Evaluation of HPC applications on cloud," *Open Cirrus Summit*, 2011.

[10] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox, "Twister: a runtime for iterative MapReduce," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, ser. HPDC '10. New York, NY, USA: ACM, 2010. [Online]. Available: http://doi.acm.org/10.1145/1851476.1851593, pp. 810–818.

[11] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "Haloop: efficient iterative data processing on large clusters," *Proc. VLDB Endow.*, vol. 3, no. 1-2, pp. 285–296, Sep. 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1920841.1920881,

[12] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," in *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, ser. HotCloud'10. Berkeley, CA, USA: USENIX Association, 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1863103.1863113, pp. 10–10.

[13] "Apache mahout project." [Online]. Available: mahout.apache.org/

[14] K. Kambatla, N. Rapolu, S. Jagannathan, and A. Grama, "Asynchronous algorithms in MapReduce," in *Proceedings of the 2010 IEEE International Conference on Cluster Computing*, ser. CLUSTER '10. Washington, DC, USA: IEEE Computer Society, 2010. [Online]. Available: http://dx.doi.org/10.1109/CLUSTER.2010.30, pp. 245–254.

[15] S. Herrero-Lopez, "Accelerating SVMs by integrating GPUs into MapReduce clusters," in *Systems, Man, and Cybernetics (SMC), 2011 IEEE International Conference on*, Oct. 2011, pp. 1298 –1305.

[16] R. Farivar, A. Verma, E. Chan, and R. Campbell, "MITHRA: Multiple data independent tasks on a heterogeneous resource architecture," in *Cluster Computing, 2009. CLUSTER '09. IEEE International Conference on*, September 2009, pp. 1 –10.

[17] R. Farivar, H. Kharbanda, A. Raghunathan, S. Chakradhar, and R. Campbell, "PIC: Partitioned iterative convergence for clusters," in *2012 IEEE International Conference on Cluster Computing*, ser. CLUSTER '12, 2012.

[18] R. Farivar, H. Kharbanda, S. Venkatraman, and R. H. Campbell, "An algorithm for fast edit distance computation on GPUs," in *InPar 2012 Innovative Parallel Computing: Foundations & Applications of GPU, Manycore, and Heterogeneous Systems*, IEEE. San Jose, California: IEEE, 2012.

[19] R. Farivar and R. Campbell, "Plasma: Shared memory dynamic allocation and bank-conflict-free access in GPUs," in *Proceedings of The 41st International Conference on Parallel Processing (ICPP), Poster*, 2012.

[20] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[21] "The open source apache hadoop project." [Online]. Available: http://hadoop.apache.org/

[22] S. Shehu, "Scaling-up GPUs for big data analytics, MapReduce and fat nodes," 2010. [Online]. Available: http://www.azintablog.com/2010/10/16/scaling-up-GPUs-big-data-analytics-mapreduce-fat-nodes/

[23] T. Kaldewey, "Large-scale GPU programming," 2012. [Online]. Available: http://cis565-spring-2012.github.com/lectures/02-13-MapReduce-Tim-Kaldewey.pdf

[24] E. Walker, "benchmarking amazon ec2 for," *Program*, vol. 33, no. 5, pp. 18–23, 2008.

[25] Q. He, S. Zhou, B. Kobler, D. Duffy, and T. McGlynn, "Case study for running hpc applications in public clouds," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, ser. HPDC '10. New York, NY, USA: ACM, 2010. [Online]. Available: http://doi.acm.org/10.1145/1851476.1851535 pp. 395–401.

[26] P. Colella, "Defining Software Requirements for Scientific Computing," *Slide of 2004 presentation included in David Pattersons 2005 talk*, 2004. [Online]. Available: http://www.lanl.gov/orgs/hpc/salishan/salishan2005/davidpatterson.pdf

[27] J. Wittwer, "Monte Carlo Simulation Basics," *A Practical Guide to Monte Carlo Simulation*, June 2004. [Online]. Available: http://vertex42.com/ExcelArticles/mc/MonteCarloSimulation.html

[28] D. Landau and K. Binder, *A Guide to Monte Carlo Simulations in Statistical Physics*. New York, NY, USA: Cambridge University Press, 2005.

[29] J. Feldman, S. Muthukrishnan, A. Sidiropoulos, C. Stein, and Z. Svitkina, "On the complexity of processing massive, unordered, distributed data," May 2007. [Online]. Available: http://arxiv.org/abs/cs/0611108

[30] Z. Hu, H. Iwasaki, and M. Takechi, "Formal derivation of efficient parallel programs by construction of list homomorphisms," *ACM Trans. Program. Lang. Syst.*, vol. 19, no. 3, pp. 444–461, 1997.

[31] Y.-K. Chen, J. Chhugani, P. Dubey, C. Hughes, D. Kim, S. Kumar, V. Lee, A. Nguyen, and M. Smelyanskiy, "Convergence of recognition, mining, and synthesis workloads and its implications," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 790 –807, may 2008.

[32] A. Vahdat, M. Al-Fares, N. Farrington, R. N. Mysore, G. Porter, and S. Radhakrishnan, "Scale-out networking in the data center," *IEEE Micro*, vol. 30, no. 4, pp. 29–41, July 2010. [Online]. Available: http://dx.doi.org/10.1109/MM.2010.72

[33] J. MacQueen, *Some methods for classification and analysis of multivariate observations.* University of California Press, 1967, vol. 1, no. 233, pp. 281–297.

[34] S. Che, J. Li, J. Sheaffer, K. Skadron, and J. Lach, "Accelerating compute-intensive applications with GPUs and fpgas," in *Application Specific Processors, 2008. SASP 2008. Symposium on*, June 2008, pp. 101 –107.

[35] Y. Liu, D. Maskell, and B. Schmidt, "Cudasw++: optimizing smith-waterman sequence database searches for cuda-enabled graphics processing units," *BMC Research Notes*, vol. 2, no. 1, p. 73, 2009. [Online]. Available: http://www.biomedcentral.com/1756-0500/2/73

[36] T. Siriwardena and D. Ranasinghe, "Accelerating global sequence alignment using cuda compatible multi-core GPU," in *Information and Automation for Sustainability (ICIAFs), 2010 5th International Conference on*, dec. 2010, pp. 201 –206.

[37] J. Blom, T. Jakobi, D. Doppmeier, S. Jaenicke, J. Kalinowski, J. Stoye, and A. Goesmann, "Exact and complete short read alignment to microbial genomes using GPU programming," *Bioinformatics*, March 2011. [Online]. Available: 10.1093/bioinformatics/btr151

[38] H. Niederreiter, *Random number generation and quasi-Monte Carlo methods.* Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1992.

[39] J. E. Gentle, *Random number generation and monte carlo method.* Springer-Verlag, 1998.

[40] "Niederreiter quasirandom sequence generator," *Available in NVIDIA CUDA SDK*, 2012.

[41] U. Cherubini and G. D. Lunga, *Structured Finance: The Object Oriented Approach (The Wiley Finance Series).* John Wiley & Sons, 2007.

[42] M. P. Craig Kolb, *Chapter 45. Option pricing on the GPU, GPU Gems 2.* Addison-Wesley Professional, 2005.

[43] D. P. Bertsekas and J. N. Tsitsiklis, *Parallel and distributed computation: numerical methods.* Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1989.

[44] R. Rojas, *Neural Networks - A Systematic Introduction.* Springer-Verlag, 1996.

[45] "Apache nutch open source search engine." [Online]. Available: http://nutch.apache.org/

[46] S. Simsek, "An edit-distance algorithm to detect correlated attacks in distributed systems," *International Journal of Computer and Information Engineering*, vol. 2, 2008.

[47] J.-M. Koo and S.-B. Cho, "Effective intrusion type identification with edit distance for hmm-based anomaly detection system," in *Pattern Recognition and Machine Intelligence*, ser. Lecture Notes in Computer Science, S. Pal, S. Bandyopadhyay, and S. Biswas, Eds. Springer Berlin / Heidelberg, 2005, vol. 3776, pp. 222–228.

[48] J. Droppo and A. Acero, "Context dependent phonetic string edit distance for automatic speech recognition," in *Acoustics Speech and Signal Processing (ICASSP), 2010 IEEE International Conference on*, march 2010, pp. 4358 –4361.

[49] I. Yalniz and R. Manmatha, "A fast alignment scheme for automatic ocr evaluation of books," in *Document Analysis and Recognition (ICDAR), 2011 International Conference on*, Sept. 2011, pp. 754 –758.

[50] A. Pollack, "DNA sequencing caught in deluge of data," *The New York Times*, Nov. 30th 2011.

[51] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of Molecular Biology*, vol. 48, no. 3, pp. 443–453, March 1970.

[52] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE Transactions on Computers*, vol. 21, no. 9, pp. 948–960, September 1972.

[53] "CUDA programming guide." [Online]. Available: http://www.nvidia.com/cuda

[54] P. Jaeckel, *Monte Carlo Methods in Finance.* Wiley, 2002.

[55] B. Moro, "The full monte," *Union Bank of Switzerland, Published in RISK magazine*, 1995.

[56] A. Toselli and O. Widlund, *Domain Decomposition Methods - Algorithms and Theory*, ser. Springer Series in Computational Mathematics. Springer, 2004, vol. 34.

[57] B. F. Smith, P. E. Bjørstad, and W. Gropp, *Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations*. Cambridge University Press, 1996.

[58] Q. M. Al-Hassan, *On Powers of Tridiagonal Matrices with Nonnegative Entries*, ser. Applied Mathematical Sciences. New York: Springer Verlag, 2012, vol. 6, no. 48.

[59] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating mapreduce for multi-core and multiprocessor systems," in *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 13–24.

[60] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: a mapreduce framework on graphics processors," in *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. New York, NY, USA: ACM, 2008, pp. 260–269.

[61] "Hadoop streaming." [Online]. Available: hadoop.apache.org

[62] A. Jagota, "Novelty detection on a very large number of memories stored in a hopfield-style network," in *Neural Networks, 1991., IJCNN-91-Seattle International Joint Conference on*, vol. ii, July 1991, p. 905 vol. 2.

[63] Y. Zhang and J. D. Owens, "A quantitative performance analysis model for GPU architectures," in *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, ser. HPCA '11. Washington, DC, USA: IEEE Computer Society, 2011. [Online]. Available: http://dl.acm.org/citation.cfm?id=2014698.2014875 pp. 382–393.

[64] R. S. Bird, "An introduction to the theory of lists," in *Proceedings of the NATO Advanced Study Institute on Logic of programming and calculi of discrete design*. New York, NY, USA: Springer-Verlag New York, Inc., 1987, pp. 5–42.

[65] M. Cole, "Parallel programming with list homomorphisms," *Parallel Processing Letters*, vol. 5, pp. 191–203, 1995.

[66] S. Gorlatch, "Constructing list homomorphisms," Universitt Passau, Germany, Tech. Rep., 1995.

[67] K. Kakehi, Z. Hu, and M. Takeichi, "List homomorphism with accumulation," in *Proceedings of Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD*, 2003, pp. 250–259.

[68] A. Morihata, K. Matsuzaki, Z. Hu, and M. Takeichi, "The third homomorphism theorem on trees: downward & upward lead to divide-and-conquer," in *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA: ACM, 2009, pp. 177–185.

[69] R. Lämmel, "Google's MapReduce programming model — revisited," *Sci. Comput. Program.*, vol. 68, no. 3, pp. 208–237, 2007.

[70] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng, "Merge: a programming model for heterogeneous multi-core systems," *SIGPLAN Not.*, vol. 43, no. 3, pp. 287–296, 2008.

[71] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover, "GPU cluster for high performance computing," in *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*. Washington, DC, USA: IEEE Computer Society, 2004, p. 47.

[72] Z. Fan, F. Qiu, and A. E. Kaufman, "Zippy: A framework for computation and visualization on a GPU cluster," *Computer Graphics Forum*, vol. 27, no. 2, pp. 341–350, Apr. 2008.

[73] T. Tu, C. A. Rendleman, D. W. Borhani, R. O. Dror, J. Gullingsrud, M. O. Jensen, J. L. Klepeis, P. Maragakis, P. Miller, K. A. Stafford, and D. E. Shaw, "A scalable parallel framework for analyzing terascale molecular dynamics simulation trajectories," in *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. Piscataway, NJ, USA: IEEE Press, 2008, pp. 1–12.

[74] J. C. Phillips, J. E. Stone, and K. Schulten, "Adapting a message-driven parallel application to GPU-accelerated clusters," in *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. Piscataway, NJ, USA: IEEE Press, 2008, pp. 1–9.

[75] K. Shirahata, H. Sato, and S. Matsuoka, "Hybrid map task scheduling for GPU-based heterogeneous clusters," in *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, Dec. 2010, pp. 733 –740.

[76] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," in *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, ser. EuroSys '07. New York, NY, USA: ACM, 2007. [Online]. Available: http://doi.acm.org/10.1145/1272996.1273005 pp. 59–72.

[77] "Hadoop pig." [Online]. Available: http://hadoop.apache.org/pig/

[78] "Apache hive." [Online]. Available: http://hadoop.apache.org/hive/

[79] Y. Zhang, Q. Gao, L. Gao, and C. Wang, "iMapReduce: A distributed computing framework for iterative computation," in *Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, ser. IPDPSW '11. Washington, DC, USA: IEEE Computer Society, 2011. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2011.260 pp. 1112–1121.

[80] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 international conference on Management of data*, ser. SIGMOD '10. New York, NY, USA: ACM, 2010. [Online]. Available: http://doi.acm.org/10.1145/1807167.1807184 pp. 135–146.

[81] G. E. Blelloch, S. Chatterjee, J. C. Hardwick, J. Sipelstein, and M. Zagha, "Implementation of a portable nested data-parallel language," Pittsburgh, PA, USA, Tech. Rep., 1993.

[82] J. Meng, S. Chakradhar, and A. Raghunathan, "Best-effort parallel execution framework for recognition and mining applications," in *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, ser. IPDPS '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1–12.

[83] J. Meng, S. Chakradhar, and A. Raghunathan, "Exploiting the forgiving nature of applications for scalable parallel execution," in *Proceedings of the 2010 IEEE International Symposium on Parallel&Distributed Processing*, ser. IPDPS '10, 2010.

[84] S. Byna, J. Meng, A. Raghunathan, S. Chakradhar, and S. Cadambi, "Best-effort semantic document search on GPUs," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, ser. GPGPU '10. New York, NY, USA: ACM, 2010. [Online]. Available: http://doi.acm.org/10.1145/1735688.1735705 pp. 86–93.

[85] S. Venkatasubramanian and R. W. Vuduc, "Tuned and wildly asynchronous stencil kernels for hybrid CPU/GPU systems," in *Proceedings of the 23rd international conference on Supercomputing*, ser. ICS '09. New York, NY, USA: ACM, 2009. [Online]. Available: http://doi.acm.org/10.1145/1542275.1542312 pp. 244–255.

[86] J. Liu, N. Ravi, S. Chakradhar, and M. Kandemir, "Panacea: towards holistic optimization of MapReduce applications," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, ser. CGO '12. New York, NY, USA: ACM, 2012. [Online]. Available: http://doi.acm.org/10.1145/2259016.2259022 pp. 33–43.

[87] D. Chazan and W. Miranker, "Chaotic relaxation," *Linear Algebra and its Applications*, vol. 2, no. 2, pp. 199 – 222, 1969. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0024379569900287

[88] Takahashi W., "Iterative methods for approximation of fixed points and their applications," *Journal of the Operations Research Society of Japan*, vol. 43, no. 1, pp. 87–108, 2000.

[89] W.-L. Cao and W. J. Stewart, "Iterative aggregation/disaggregation techniques for nearly uncoupled markov chains," *J. ACM*, vol. 32, no. 3, pp. 702–719, July 1985. [Online]. Available: http://doi.acm.org/10.1145/3828.214137

[90] "APARAPI; API for data parallel java, allowing suitable code to be executed on GPU via OpenCL." [Online]. Available: http://code.google.com/p/aparapi/