March 2016

Accelerated Iterative Algorithms with Asynchronous Accumulative Updates on a Heterogeneous Cluster

Sandesh Gubbi Virupaksha
University of Massachusetts Amherst

ACCELERATED ITERATIVE ALGORITHMS WITH ASYNCHRONOUS ACCUMULATIVE UPDATES ON A HETEROGENEOUS CLUSTER

A Thesis Presented

by

SANDESH GUBBI VIRUPAKSHA

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL AND COMPUTER ENGINEERING

February 2016

Electrical and Computer Engineering

ACCELERATED ITERATIVE ALGORITHMS WITH ASYNCHRONOUS ACCUMULATIVE UPDATES ON A HETEROGENEOUS CLUSTER

A Thesis Presented

by

SANDESH GUBBI VIRUPAKSHA

Approved as to style and content by:

Russell Tessier, Chair

Lixin Gao, Member

David Irwin, Member

Christopher V. Hollot, Department Chair
Electrical and Computer Engineering

ACKNOWLEDGMENTS

First and foremost, I would like to thank my parents who have stood by me at my best and worst times and always had faith in my efforts.

I would like to thank my advisor Professor Russell Tessier, who provided me constant encouragement and guided me at all times during the thesis. His availability almost all the time in person and his prompt responses to e-mails is truly remarkable. I would like to thank Deepak Unnikrishnan, who helped me understand the basics of this project and answered all my queries even with his busy work schedule. I sincerely thank Prof. Lixin Gao and Prof. David Irwin for being on my thesis committee.

I would like to thank all the past and present lab mates of Reconfigurable Computing Group, who made my stay comfortable and cherishing. I would like to thank all my friends, who I met in these 3 years and made my stay in Amherst a memorable one.

ABSTRACT

ACCELERATED ITERATIVE ALGORITHMS WITH ASYNCHRONOUS ACCUMULATIVE UPDATES ON A HETEROGENEOUS CLUSTER

FEBRUARY 2016

SANDESH GUBBI VIRUPAKSHA

B.E., VISVESVARAYA TECHNOLOGICAL UNIVERSITY

M.S.E.C.E., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Russell Tessier

In recent years with the exponential growth in web-based applications the amount of data generated has increased tremendously. Quick and accurate analysis of this 'big data' is indispensable to make better business decisions and reduce operational cost. The challenges faced by modern day data centers to process big data are multi fold: to keep up the pace of processing with increased data volume and increased data velocity, deal with system scalability and reduce energy costs. Today's data centers employ a variety of distributed computing frameworks running on a cluster of commodity hardware which include general purpose processors to process big data. Though better performance in terms of big data processing speed has been achieved with existing distributed computing frameworks, there is still an opportunity to increase processing speed further. FPGAs, which are designed for computationally intensive tasks, are promising processing elements that can increase processing speed. In this

thesis, we discuss how FPGAs can be integrated into a cluster of general purpose processors running iterative algorithms and obtain high performance.

In this thesis, we designed a heterogeneous cluster comprised of FPGAs and CPUs and ran various benchmarks such as PageRank, Katz and Connected Components to measure the performance of the cluster. Performance improvement in terms of execution time was evaluated against a homogeneous cluster of general purpose processors and a homogeneous cluster of FPGAs. We built multiple four-node heterogeneous clusters with different configurations by varying the number of CPUs and FPGAs.

We studied the effects of load balancing between CPUs and FPGAs. We obtained a speedup of 20X, 11.5X and 2X for PageRank, Katz and Connected Components benchmarks on a cluster cluster configuration of 2 CPU + 2 FPGA for an unbalancing ratio against a 4-node homogeneous CPU cluster. We studied the effect of input graph partitioning, and showed that when the input is a Multilevel-KL partitioned graph we obtain an improvement of 11%, 26% and 9% over randomly partitioned graph for Katz, PageRank and Connected Components benchmarks on a 2 CPU + 2 FPGA cluster.

TABLE OF CONTENTS

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

CHAPTER 1

INTRODUCTION

1.1 Motivation for Heterogeneous Computing

The rapid advancement in networking, storage and sensing technologies has resulted in a large volume of data being generated in a very short duration of time. With the advent of social networking platforms, e.g. Facebook and Twitter, a vast amount of data is being collected every second. The analysis of big data remains one of the biggest challenges in the computer area [22]. The volume of business data is expected to double every 1.2 years [3] and poor data management can cost up to 30% of the operating revenue of a business [4].

Data analytics are very critical to the operation of a business [22]. Better data analytics can lead to effective marketing strategies, better customer service and new revenue sources. Most data mining and data analytic algorithms [26][10][31] are based on iterative calculations. With large amounts of data requiring large numbers of iterations, timely analysis of big data has become challenging.

There have been a number of frameworks [40][28][11][37] proposed to accelerate the iterative computation of big data on a cluster of commodity processing nodes. MapReduce [15] is a popular parallel processing framework which is scalable, fault tolerant and can be easily implemented on a cluster of commodity computers. Even though MapReduce provides good processing speed, the synchronization barrier between iterations remains a bottleneck to higher speedup [38].

iMapReduce [40] tries to improve the speedup compared to MapReduce by reducing the overhead of creating the new tasks in every iteration and allowing asyn-

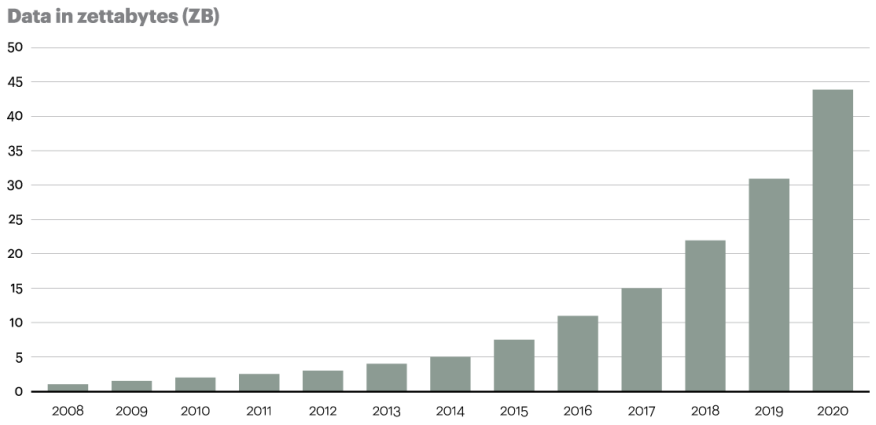

Figure 1.1: Growth of data from 2008 to 2020 (Source : Oracle 2012)

chronous execution of iterations. Piccolo [28] launches application kernel functions on multiple computing nodes. The computing nodes share a global state table which can be implemented both in memory and on disk. The computation is performed with a global barrier between kernel invocations. Spark [37] introduces the concept of resilient distributed datasets (RDD). An RDD is a read only collection of partitions across multiple machines. Spark supports caching of RDD in memory across multiple computing machines and reuse it in MapReduce like operations.

The concept of asynchronous accumulative updates (AAU), introduced in [42], overcomes limitations of synchronous frameworks. In AAU, intermediate results are accumulated asynchronously from both the current and previous iterations. AAU is known to accelerate the convergence of iterative computations and provide better speedup [42].

FPGAs (Field Programmable Gate Arrays) have long been used to perform computationally intensive tasks [12]. The flexibility of reprogramming the devices after their deployment provides an opportunity to change system functionality to suit instantaneous needs. The parallelism of FPGAs helps accelerate iterative computation. For example, Maestro [34] implements AAU on a cluster of four FPGAs leading to a 40× speedup with respect to performance on a four-node CPU cluster.

Figure 1.2: Inside view of a Google data center (Source : Google).

In data centers, it is not feasible to replace all processors with FPGAs for acceleration. Providing an option to integrate hardware accelerators with existing CPU infrastructure in data centers is an alternative. In this thesis, we propose a prototype of a heterogeneous data center, where FPGAs are included with general purpose processors and work in tandem to provide improved application performance. Our work aims to provide system level heterogeneity. The existing hardware infrastructure need not be replaced. The FPGA boards are used as plug and play devices which can be easily added and removed from a cluster.

In this thesis, we use a heterogeneous cluster with general purpose CPUs and FPGAs to solve iterative computation. The cluster is evaluated for performance using a number of iterative algorithms such as PageRank, Katz and Connected Components. We present a prototype of the heterogeneous cluster, integrating special purpose hardware (FPGAs) into a largely homogeneous CPU computing environment. We show that the heterogeneous cluster yields better performance in terms of execution time than its homogeneous counterpart.

1.2 Thesis Outline

In the background chapter, we review the concept of iterative algorithms and AAU. The execution of iterative algorithms with AAU on a CPU and FPGA cluster will be discussed in brief in Chapter 2. Chapter 3 focuses on the detailed architecture of the heterogeneous cluster which runs iterative algorithms with AAU. Various heterogeneous cluster configurations are discussed in chapter 3. Chapter 4 discusses the concept of load balancing and graph partitioning. Chapter 5 evaluates the effect of load balancing and input graph partitioning on various configurations of heterogeneous clusters. Chapter 6 concludes the thesis with a discussion of future work.

CHAPTER 2

BACKGROUND

In this chapter we discuss the concept of Asynchronous Accumulative Updates (AAU) in detail. Previous work [42] [34] has implemented AAU on homogeneous CPU and FPGA clusters, respectively. These implementations are briefly reviewed in this chapter.

2.1 Introduction to Iterative Algorithms

In iterative algorithms, the final result is obtained by executing the same set of operations over the input data set for a number of iterations. Iterations continue until the termination criterion is met. The results of the previous iteration are used in the current iteration. Mathematically, an iterative algorithm can be represented as

$$v^k = G(v^{k-1}) \tag{2.1}$$

where the update function $G()$ is applied on the $(k-1)^{th}$ iteration of a n-dimensional vector $v^k = \{v_1^k, v_2^k .....v_n^k\}$. Since each element of v^k can be computed separately, iterative algorithms are highly data parallel in nature. Programming models such as MapReduce exploit the parallelism of iterative models to accelerate the data convergence using a distributed cluster.

PageRank algorithm is used in Google's search engine to obtain the importance of web pages [10]. In computing the PageRank value, the web is viewed as a graph and individual web pages are treated as nodes in the graph. Consider a web graph with N web pages, an edge exists between two nodes in the graph if there is a hyper-link

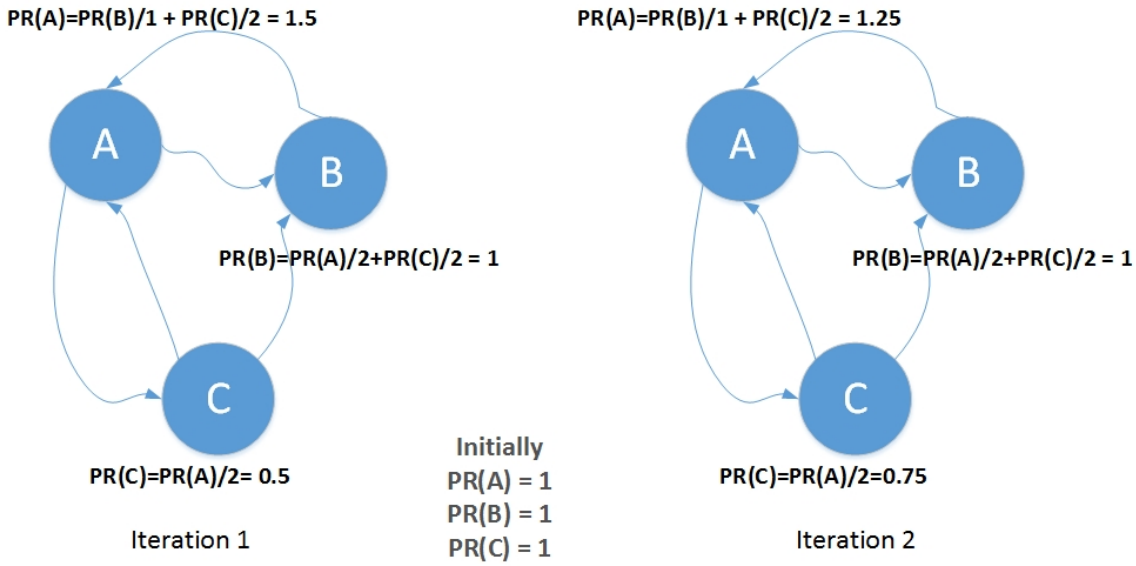

Figure 2.1: Computation of PageRank

between two corresponding web pages. The PageRank of a node at $(i+1)^{th}$ iteration is calculated using Eq. 2.2.

$$PR^{(i+1)} = \frac{1-d}{N} + \sum_{u \in B_u} \frac{d \times R^{(i)}(u)}{L(u)} \tag{2.2}$$

During the start of an iteration, each node in the graph is assigned an initial PageRank value $\frac{1-d}{N}$, where d is the damping factor, N is the number of nodes in the graph, $L(u)$ is the number of outgoing edges of a node k and u is the number of incoming links of a node k.

Consider the graph shown in Figure 2.1. The graph has three nodes assigned an initial PageRank value of 1. During the start of the first iteration, the nodes transmit PageRank values to the nodes which are attached to their outgoing links. After receiving the values from all incoming links, a node calculates its new PageRank value according to Eq. 2.2. In Figure 2.1, after the first iteration, PageRank values of nodes A, B and C are 1.5, 1 and 0.5. This process continues until the PageRank values

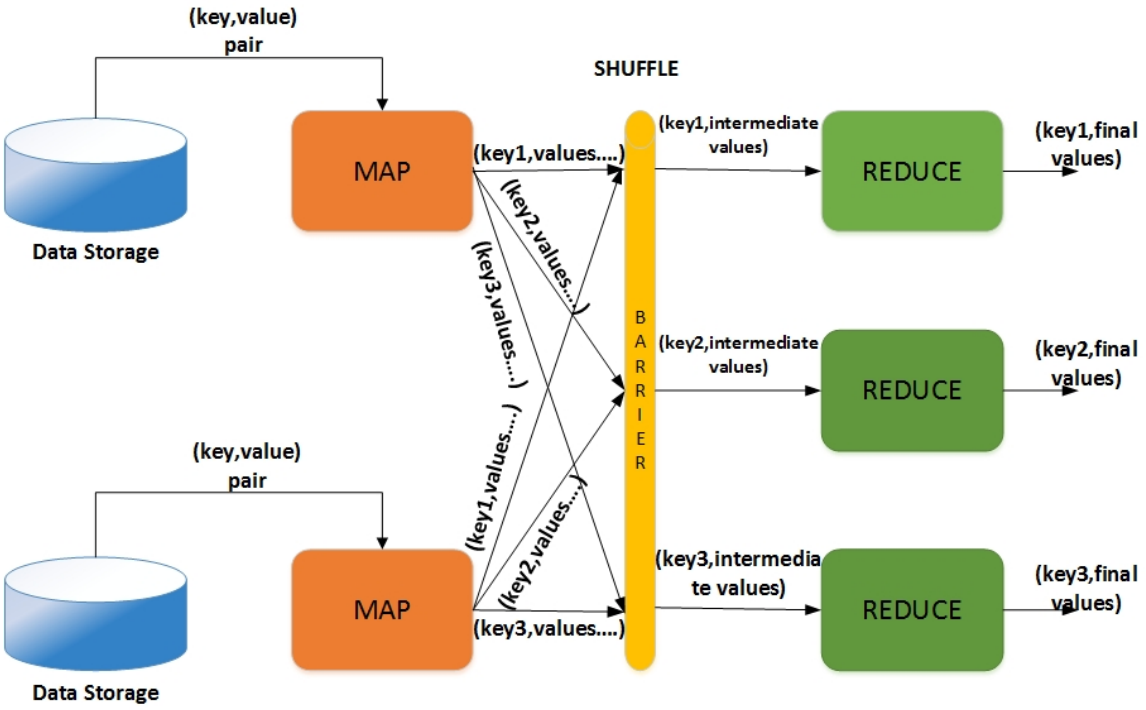

Figure 2.2: Graphical overview of MapReduce

of all nodes remains constant between successive iterations or for a predetermined number of iterations.

2.2 Synchronous programming models

2.2.1 MapReduce

The flow of iterative computation using MapReduce is described in Figure 2.2. MapReduce primarily executes a series of map and reduce tasks. Often there are supplemental phases such as sorting, partitioning and combining which occur between map and reduce tasks. The input and output of map and reduce functions are a set of Key-Value (KV) pairs. KV pairs contain information specific to the application. Map functions provide processed input values to the reduce function. The reduce function acts on the intermediate KV pairs obtained from the map function and generates

the final output. All mappers and reducers run in parallel and perform operations independently.

Some of the limitations of MapReduce (synchronous computing frameworks) are:

- MapReduce imposes a strict synchronization barrier between two iterations. The map task in the present iteration cannot start unless all the reduce tasks in the previous iterations have completed. There typically is no pipelining mechanism between map and reduce tasks [38].

- MapReduce handles the issue of slow computing nodes (stragglers) by speculatively executing the task of a slow node on another node to accelerate the computation. When MapReduce is implemented on a heterogeneous cluster, where some nodes are slow compared to others, it can result in performance degradation. Stragglers and speculative tasks often compete for system resources, such as network resources, with other active tasks.

2.2.2 Spark, Picollo and iMapReduce

Spark [37] is a framework for running large scale data intensive applications on commodity clusters. Spark is specially designed for machine learning applications. Spark uses an abstraction called resilient distributed datasets (RDD). An RDD is a read only collection of objects partitioned across commodity cluster of machines that can be rebuilt if the partition is lost. Spark has a built in fault tolerance feature, where the lost partition can be derived from the existing partitions. Spark preserves the static data in memory between iterations.

Picollo [28] uses a data centric programming model, where computations running on different machines can share distributed mutable state via a key-value table interface. The iterative algorithm implemented on Picollo updates the distributed tables iteratively. The input data is loaded into a shared memory from the distributed

file system and iterations are executed synchronously to arrive at the termination condition. Picollo achieves fault tolerance by checkpoint and restore mechanisms.

iMapReduce [40] is a distributed computing framework, where the users can perform iterative processing by specifying Map and Reduce functions. Some of the benefits of iMapReduce over traditional MapReduce framework are: no overhead of creating jobs at every iteration and asynchronous execution of iterations. In iMapReduce, data loading from the distributed file system (DFS) happens only once during initialization and data is written back to the DFS after termination. During iterations, the data from reduce functions are directly sent to the map function of the next iteration.

In the above techniques, synchronization is essential either between iterations or in an iteration. There are few asynchronous frameworks proposed such as GraphLab [27]. This platform achieves a high degree of parallel performance with asynchronous iterative computation with sparse computational dependencies. AAU proposes a unique concept for asynchronous updates and is applicable for a large collection of iterative computations.

2.2.3 Previous works on Implementation of Synchronous and Asynchronous Frameworks on Hardware Platforms

There have been numerous studies performed to implement synchronous iterative methods on hardware accelerators such as GPUs and FPGAs. HeteroSpark [6] integrates a graphics processing unit (GPU) in a Spark framework to achieve better speedup. HeteroSpark is primarily designed for machine learning algorithms. It provides "plug and play" capability for GPUs which can be enabled/disabled in the cluster.

Shan et al. [30] implements MapReduce on an FPGA with mappers and reducers implemented using an on-chip memory. During an iteration, data is fetched from the

DDR global memory and stored in an on-chip local memory. The intermediate values in the local memory have to be written back to the global memory after completion of every iteration. Yeung et al. [36] describes the implementation of MapReduce libraries supporting FPGAs and GPUs. The source code specified in ANSI C is compiled along with MapReduce libraries into a binary configuration file for processing units. Tsoi et al. [33] demonstrates the implementation of a MapReduce framework on a cluster of CPUs, FPGAs and GPUs. Choi et al. [13] demonstrates the implementation of an FPGA-based cluster to run a k-means algorithm based on MapReduce. Bingsheng et al. [19] discusses the implementation of MapReduce on GPUs.

2.3 Asynchronous Accumulative Updates

In a synchronous update model such as MapReduce, the update in the k^{th} iteration is performed after obtaining all the results from the $(k-1)^{th}$ iteration. In the synchronous model, the partial results from the iteration cannot be utilized by the next iteration. Due to the strict synchronization mechanism required between the two iterations, the synchronous update model slows down convergence.

The AAU model loosens this restriction. It accumulates results using information from both the previous and present iterations. Results are accumulated asynchronously in AAU, eliminating the need for synchronization barriers. In [17] the concept of asynchrony has been illustrated to accelerate the convergence of iterative computation. In the AAU model described in [42], a node disseminates the *change* in the node value instead of the entire value of the node. Changes received from all nodes are accumulated and the update then is asynchronously propagated to other nodes.

Consider a graph node, S with a value v. When the new increment is received from a neighboring node, the value will not be added to v immediately but will be

accumulated in Δv and asynchronously updated to v later. The concept of AAU is described in Figure 2.3.

The process of AAU can be expressed in two steps : Accumulate and Update.

- In accumulation phase, a compute node receives a message, m from its neighboring nodes. The message received is accumulated in Δv associated with the compute node. This operation is described in Eq. 2.3, where $\oplus$ is an abstract operator.

$$\Delta v \leftarrow \Delta v \oplus m \tag{2.3}$$

- The update operation is divided into three steps. In the first step the accumulated values from all nodes, Δv are added to v. In the second step, an update function $g()$ is applied to Δv, the change in the current value of the node. In the third step, the node will propagate, $g(\Delta v)$ to all its neighboring nodes and Δv is reset to 0. Eq. 2.4 describes the update operation.

$$\begin{aligned} & & v \leftarrow v \oplus \Delta v \\ send \quad g(\Delta v) \quad if \quad & g(\Delta v) \neq 0 & \\ & & \Delta v \leftarrow 0 \end{aligned} \tag{2.4}$$

2.3.1 Computation of PageRank using Asynchronous Accumulative Updates

Consider a graph with nodes A and B having incoming edges to node C as shown in Figure 2.3. Node D has an incoming edge from node C. Let $\Delta v_A = d(\frac{\Delta PR(A)}{L(A)})$ and $\Delta v_B = d(\frac{\Delta PR(B)}{L(B)})$ be the change in PageRank values of Node A and Node B respectively. Node C accumulates the received delta values from other nodes in $\Delta PR(C)$. The PageRank of node C is calculated by adding $\Delta PR(C)$ and v. After accumulating the values, node C applies the update function on the change in its delta value and propagates the result to node D.

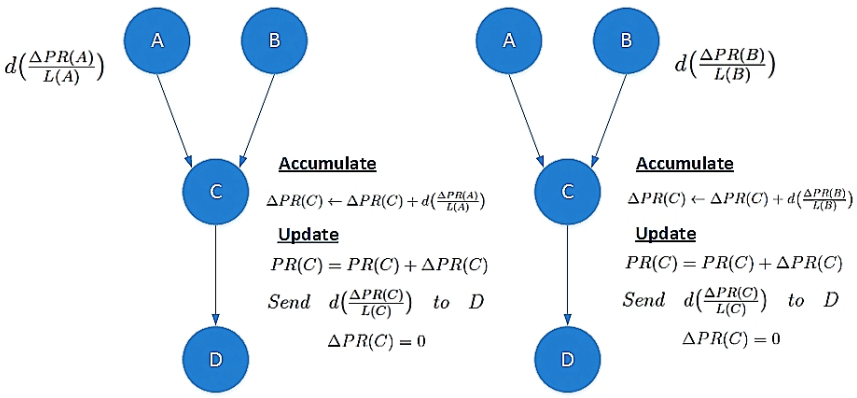

Figure 2.3: Computation of PageRank (PR) using AAU

When AAU is implemented on a cluster of commodity CPUs, processors perform update and accumulate operations in a completely asynchronous fashion and processors will not wait for other processors in the cluster to complete their operations. Thus, we can conclude that the concept of AAU is well suited to a heterogeneous cluster containing slow and fast processors. Chapter 3 discusses how AAU can be realized on a heterogeneous cluster of CPUs and FPGAs.

2.4 A Message-Passing Distributed Framework for Accumulative Iterative Computation on a CPU cluster

AAU is demonstrated on a cluster of homogeneous CPUs in [42]. The architecture of Maiter is shown in Figure 2.4. Maiter consists of a single master and multiple workers. The function of a master node is to coordinate and monitor the status of workers. A master node also takes part in the computation apart from monitoring computation of workers. Workers communicate with each other via MPI [9] and execute the iterative task in parallel. During the start of computation, a master node assigns an even amount of data (KV pairs) to workers and monitors workers for iteration termination.

The iterative computation on the Maiter cluster is divided into three phases:

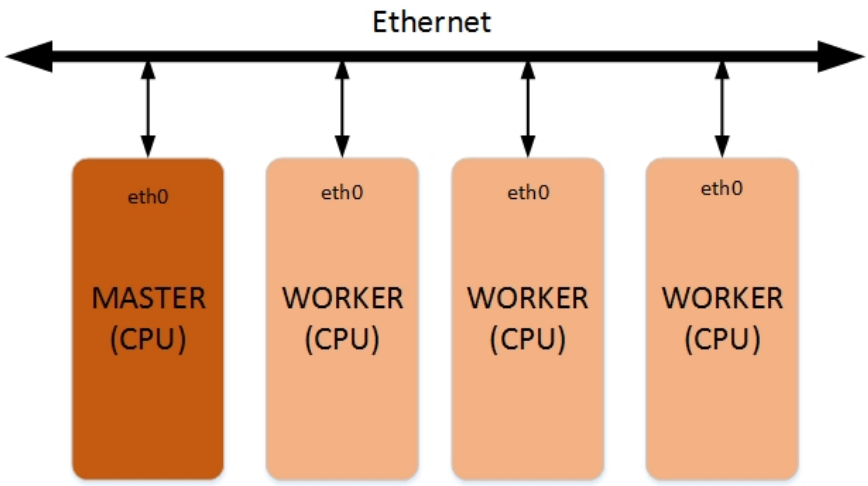

Figure 2.4: Architecture of Maiter

1. **Loading the data**: Workers are assigned a subset of data by the master node. Before the start of iterative computation, the data on the DFS will be partitioned into multiple shards and assigned to workers. Workers parse the data in parallel and populate their respective local state tables. A local state table on a worker contains KV pairs assigned to it. Every data element (KV pair) is associated with a unique global key. The data element corresponding to a unique key k is assigned to a worker based on the result of using a hash function, $h(k)$. MOD is used as the hash function in Maiter.

2. **Iterative computation**: The iterative computation involves update and accumulate operations. These operations are performed on each worker by two mutually exclusive receive and accumulate threads. The receive thread, obtains messages from all the workers in the cluster via MPI and accumulates messages to the Δv field of a data element. The update thread updates the value field with the Δv field and sends the change in the value to the other workers. Since the update thread performs both read and write operations on the Δv field, it is implemented in a critical section.

KV pairs are scheduled for an update operation with a priority scheduling policy. Priority scheduling accelerates the convergence of computation resulting in improved performance [41]. The update thread extracts a KV pair from the priority queue. The priority of a KV pair is calculated initially considering value(v) and Δv. The priority value is changed during an iterative computation when there is a change in the Δv field.

The communication between workers during an iterative computation occurs via message passing [9]. A message contains a key (destination element) and its corresponding value. The worker possessing the destination key receives the message to perform an accumulate operation. Output messages are first buffered and then flushed after a brief time out from each worker to reduce the communication cost.

3. **Iteration Termination**: The master node relays the termination check signal to all worker nodes periodically. After receiving the termination check signal, worker nodes calculate their local termination value and transmit it back to the master node. After receiving the iteration progress from all worker nodes, the master node makes a decision based on the global iteration progress, obtained from information received from workers. If the master node decides to terminate the iteration, it sends a termination signal to all workers. After receiving the termination signal from the master, workers stop update operations and dump the results from their local state tables onto the DFS.

Maiter is implemented [42] on an Amazon EC2 cloud [2]. Maiter has achieved a speedup of 60X over Hadoop MapReduce [5], a synchronous iterative algorithm.

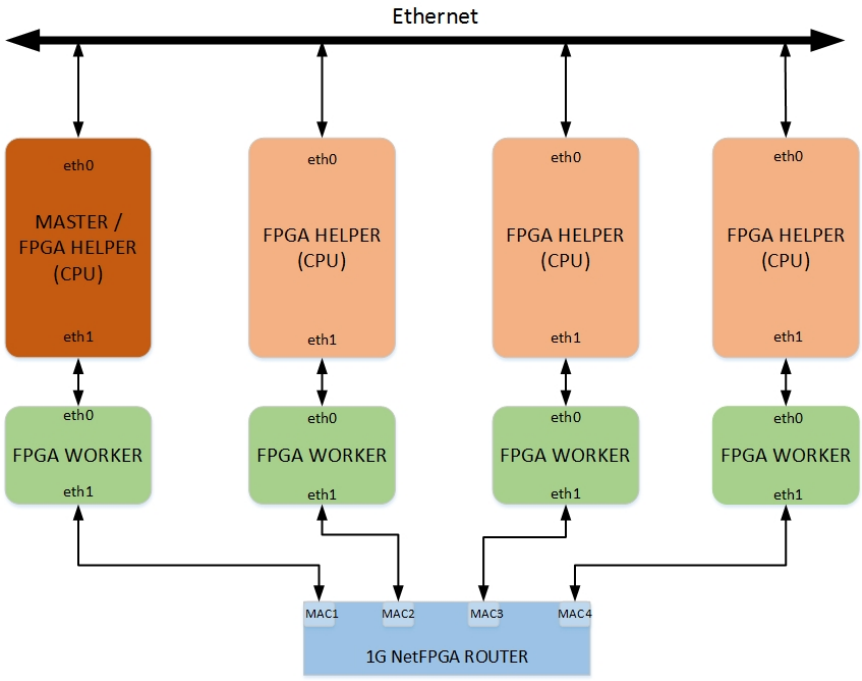

Figure 2.5: Architecture of Maestro

2.5 Accelerating Iterative Algorithms with Asynchronous Accumulative Updates on FPGAs

Maestro [34] demonstrates the implementation of AAU on a scalable homogeneous cluster of FPGAs. The scalability of the cluster can be exhibited by an increase in individual FPGA capacity and in the number of FPGAs.

The AAU model implemented in Maestro [34] on a cluster of FPGAs is shown in Figure 2.5. In Maestro, CPU[0] acts as a master which coordinates and monitors the computation on FPGA worker nodes. All FPGAs are connected to helper CPUs which serve as an interface between FPGA workers and the master. FPGA workers are implemented on Altera DE4 developmental boards [1]. During the computation, FPGAs operate in parallel on the subset of data assigned to them and communicate with each other via a NetFPGA router.

The execution of iterative algorithms on Maestro largely follows Maiter operations. The primary steps involved are:

1. **Loading the KV pairs on FPGAs**: The input data is stored on a DFS, an assemblage of hard disk drives accessible by CPUs. A MOD hash function is applied on all global keys The helper CPU transmits the initial data via Ethernet to the associated FPGA nodes to store in 1GB DDR2 DRAM next to each FPGAs.

   A state table is constructed to store KV pairs in the DRAM of each FPGA. The state table contains five fields: the key, corresponding value of the key, the delta value, the priority value of the key and the linkage information. After completion of loading the data into DRAM on all FPGAs, the master CPU broadcasts a start iteration message to all helper CPUs via MPI. Helper CPUs send the start iteration signal to their respective FPGAs, which in-turn start iterative processing.

2. **Iterative computation on FPGAs**: The AAU architecture implemented on FPGAs is shown in Figure 2.6. The packet parser module receives the incoming Ethernet packet and initiates suitable operations (eg. start iterative computation, check termination, etc.). The packet composer module constructs packets to be sent out of an FPGA worker to other workers. The computation unit on an FPGA is comprised of multiple processors (8 processors in the Maestro design). Each processor can be configured either as a transmit processor or as a receive processor. The processor in Rx mode performs only accumulate operations. The processor in Tx mode performs both accumulate and update operations.

   A priority scheduler is implemented on an FPGA worker to accelerate iteration convergence. A random sample of KV pairs are selected and sorted using a chain of shift registers. The threshold is selected as the priority value of the k^{th} highest KV pair in the sorted sample. The Tx mode processor choses a KV

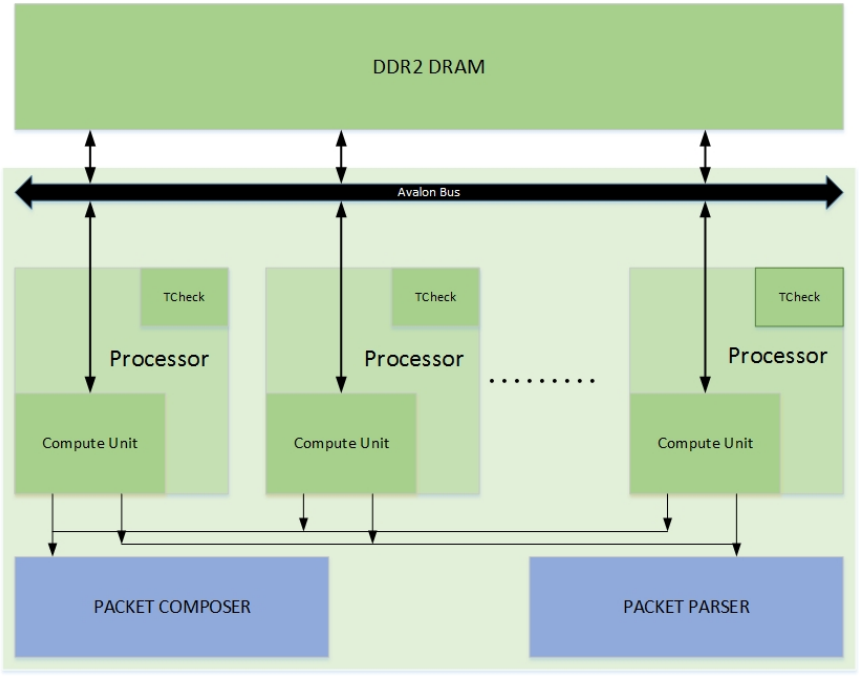

Figure 2.6: Implementation of AAU on an FPGA

pair for update only if the priority value is more than the threshold. Possible memory inconsistencies due to shared memory among multiple processors on an FPGA are averted by implementing a snoopy coherence protocol. The protocol implementation enforces strict memory consistency and serializes data accesses. During the remote update operation, an FPGA node sends KV pairs to other FPGA nodes. In order to minimize the communication cost, a sufficient number of key value pairs (150 KV pairs) are accumulated to fill the maximum size of the Ethernet frame before transmission.

3. **Iteration termination**: On an FPGA worker, each processor calculates the local iteration progress. Progress is computed by summing the values of all keys in the state table. Information from Tcheck modules is aggregated and transmitted to the helper CPUs. Helper CPUs communicate the progress of their respective FPGA nodes to the master node. The master node accumulates

local termination progress from all FPGA workers and decides if the iteration needs to be terminated.

The speedups of Maestro versus Maiter and Hadoop MapReduce were determined by varying the number of Tx and Rx processors on FPGA workers. A balanced Tx/Rx processor ratio of 4:4 provided the highest speedup. The speedup was evaluated for a fixed number of Tx/Rx processors on 1, 2 and 4 node clusters by using a Rx/Tx ratio of 4:4. Maestro reported a speedup of 40X, 18.7X and 7.5X speedups for PageRank, Katz and Connected Components benchmarks compared to similar node counts for Maiter and Hadoop MapReduce.

In the above implementations AAU has been implemented either on a homogeneous cluster of CPUs or homogeneous cluster of FPGAs. In our work we study the performance of a cluster consisting of both CPUs and FPGAs.

CHAPTER 3

HETERO: A HETEROGENEOUS COMPUTING CLUSTER

The asynchronous accumulative update model (AAU) is well suited to heterogeneous clusters. In Maiter, AAU is implemented on a cluster of homogeneous general purpose processors. Maestro implements AAU on a homogeneous cluster of FPGAs. In both implementations, the true potential of AAU was not realized. In this thesis we develop a heterogeneous architecture - **Hetero**, integrating general purpose CPUs and FPGAs to accelerate iterative computation with AAU.

3.1 Design of a heterogeneous cluster

The architecture of a 4-node heterogeneous cluster (Hetero) is shown in Figure 3.1. The 4-node Hetero consists of two CPUs and two FPGAs. CPU[0] acts as a master CPU node and also performs worker functions. FPGA workers are attached to FPGA helper CPUs. Worker CPUs and FPGA helper CPUs are connected via Ethernet. The master/worker CPU communicates with other workers via a 1G NetFPGA reference router.

3.1.1 Design of Master node

The master runs APIs implemented in C++. The APIs are borrowed from [7]. The master performs the functions listed below.

1. The master (CPU node) initiates the loading of data (KV pairs) into state tables of workers. The KV pairs are stored in the CPU worker in memory state tables and in DDR2 DRAM adjacent to the FPGA.

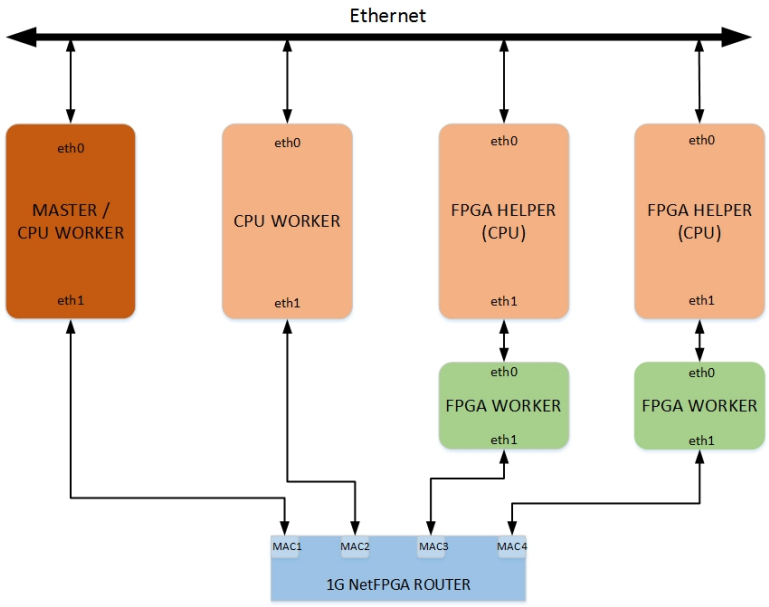

Figure 3.1: Architecture of a Heterogeneous cluster

2. The master initiates iterative computation by sending a message to CPU and FPGA workers.

3. The master periodically sends termination check messages to workers. Workers compute the progress and report to the master. In the case of FPGA nodes, FPGA helpers send the progress received from FPGAs to the master. The master computes the global progress value and if the global progress value is constant between two successive iterations, it sends a termination signal to stop the computation.

3.1.2 Design of Worker nodes

Hetero consists of two types of workers : a CPU worker and an FPGA worker. FPGA helper CPUs attached to the FPGAs transmit initial data to the FPGAs and serve as an interface between the master and the FPGAs.

The C++ APIs of the iterative kernel were obtained from the Maiter open source code base [7]. The kernel consists of receive and update threads. The receive thread

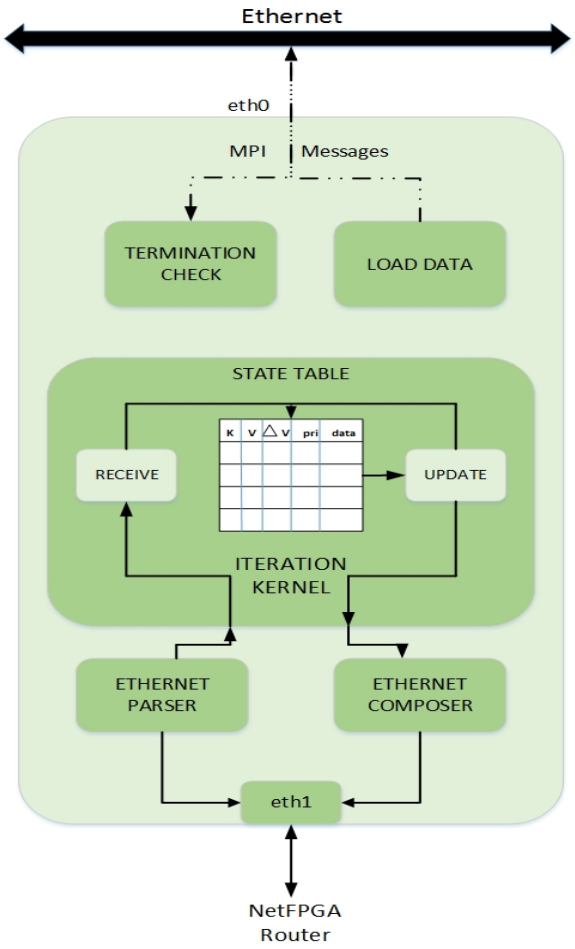

Figure 3.2: Architecture of Hetero CPU node

performs accumulate operations on a local key element by accumulating messages obtained from all key elements. The update thread updates values and Δv fields and sends messages to other key elements. A priority scheduling policy is implemented as in Maiter to accelerate iteration termination.

The workers exchange KV pairs via Ethernet during iterative computation. When an iterative kernel on a CPU worker performs a remote update, it needs to send a KV pair to other workers. An Ethernet composer module composes the Ethernet frame to be sent out to other workers. The raw Ethernet frame format is shown in Figure 3.3. We accumulate 150 KV pairs before sending the Ethernet packet. When a receiver receives an Ethernet frame, the Ethernet parser module extracts the KV

CTRL	NetFPGA 64bit Data Path							
-	Bits 0-7	Bits 8-15	Bits 16-23	Bits 24-31	Bits 32-39	Bits 40-47	Bits 48-55	Bits 56-63
0xFF	port_dst 16		word_length 16		port_src 16		byte_length 16	
0x00	mac_dst 48						mac_src_hi 16	
0x00	mac_src_lo 32				mac_ethertype 16		ip version 4 + ip_header_length 4	ip_ToS 8
0x00	ip_total_length 16		ip_id 16		ip_flags 3 + ip_flag_offset 13		ip_TTL 8	ip_prot 8
0x00	ip header checksum 16		ip src 32				ip dst hi 16	
0x00	ip_dst_lo 16		udp_src 16		udp_dst 16		udp_length 16	
0x00	udp_checksum 16		airfpga_reserved 16		airfpga_seq_num 32			
0x00	IQ 32				IQ 32			
0x00	IQ 32				IQ 32			
0x00	...				...			
0x01	IQ 32				IQ 32			

Figure 3.3: NetFPGA frame format.

Figure 3.4: Altera DE4 FPGA.

pairs, which is used by the iteration kernel to perform an accumulation operation. Workers communicate with the master via MPI.

3.2 Hardware Components of Heterogeneous Cluster

The heterogeneous cluster is built using various hardware components:

1. An **Altera DE4 FPGA board** (Figure 3.4) acts as an accelerator in the cluster. The DE4 board includes a Stratix IV GX (EP4SGX530C2) device. The DE4 board has a built-in USB blaster circuit for programming. Four Gigabit

Figure 3.5: NetFPGA.

Ethernet ports (GigE) with RJ-45 connectors provide Ethernet interfaces to the board. High performance external DDR2 DRAM offers off-chip storage.

2. A **NetFPGA reference router** [8] (Figure 3.5) is used for cluster network routing. The NetFPGA is a reconfigurable hardware platform used for high speed routing operations. In the NetFPGA, a complete data path is implemented in hardware. The design supports back-to-back packet transport at full Gigabit line rates. The NetFPGA board contains four 1 Gigabit/second Ethernet (GigE) interfaces and an FPGA. We download a 1G reference router design to the NetFPGA board and configure the NetFPGA routing table to forward Ethernet packets to appropriate destinations.

3. A **quad core processor** is also used. CPU nodes in the cluster run on an Intel Core2 quad processor with a clock frequency is 2.33GHz. The machines have 4GB of DRAM. The machines have two 1Gigabit/second network interface cards which connect to a LAN setup.

3.3 Heterogeneous Cluster Operation

The user specifies three parameters to execute an iterative algorithm on Hetero: a partitioning algorithm, an iterative kernel and a termination condition. The par-

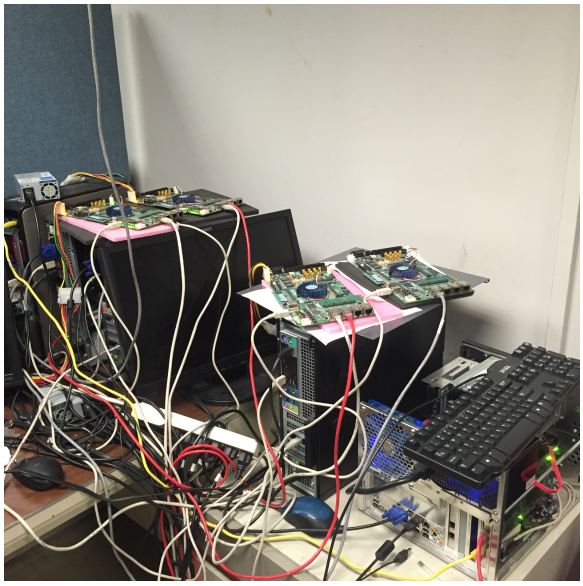

Figure 3.6: Laboratory prototype of a 4 node Hetero

titioner uses a hash function to assign data values to worker nodes. In this design we use the MOD function to assign KV pairs to compute nodes. The partitioner is implemented as a C++ API and resides on the master (CPU node).

The iterative kernel interface specifies update and accumulate functions. In our design, the kernel resides on both CPUs and FPGAs. For CPU nodes, the kernel is specified as a C++ API and on FPGA nodes it is implemented as hardware synthesized from a Verilog module. The termination checker interface specifies the condition to be met to terminate iterations. The termination checker is implemented as a C++ API and is located on the master.

The user provides two configuration files as input. The first configuration file contains the host-names/IP-address of the CPUs and the FPGA helpers. The second configuration file consists of the information about the type of each node (CPU/FPGA helper). A bitstream is downloaded onto the FPGA using a USB JTAG interface.

At the start of computation, the partitioner in the master node assigns KV pairs to workers based on the specified hash function. Each FPGA is assigned the same amount of KV pairs. The keys which are assigned to the FPGA nodes are loaded

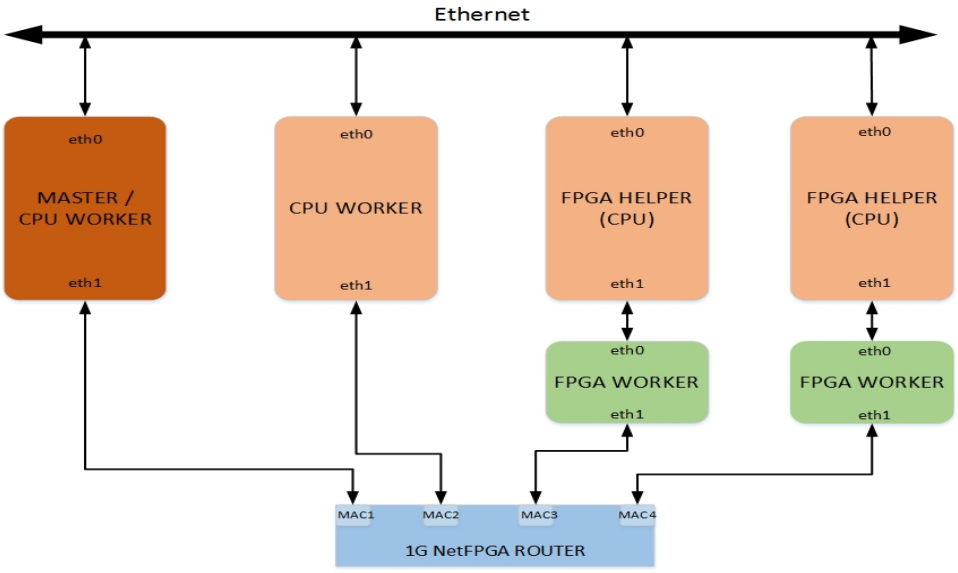

Figure 3.7: Laboratory prototype of a 2 CPU, 2 FPGA Hetero

from the local file system of the FPGA helpers into FPGA DRAM. When the loading of data is complete, the master initiates iterative computation. The iterative kernels on CPUs and FPGAs are executed in parallel by exchanging KV pairs via the 1G NetFPGA reference router. The termination condition is checked periodically (every 4 seconds) by the master node after collection of the progress of CPU and FPGA workers.

3.4 Heterogeneous Cluster Configurations

Different configurations of heterogeneous cluster are built by varying the number of CPUs and FPGAs. In this section, we discuss the architecture of three different configurations of a heterogeneous cluster and in Chapter 5, we evaluate the performance of each configuration for various benchmarks.

3.4.1 Heterogeneous cluster with 2 CPUs and 2 FPGAs

A heterogeneous cluster configuration with 2 CPUs and 2 FPGAs is shown in Figure 3.7. Ethernet 1 ports of CPU worker-1 and CPU worker-2 are connected

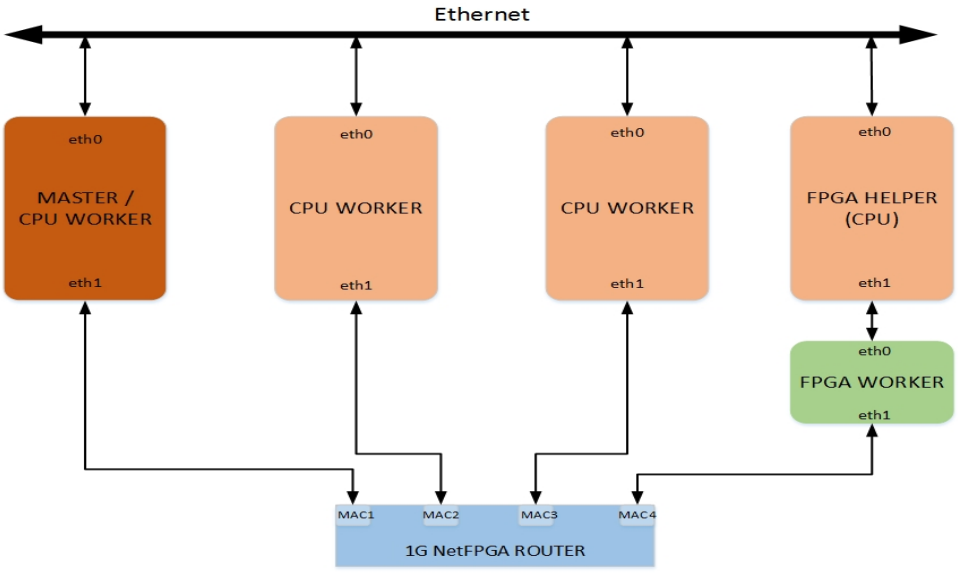

Figure 3.8: Laboratory prototype of a 3 CPU, 1 FPGA Hetero

to the MAC1 and MAC2 ports of the NetFPGA router. MAC3 and MAC4 ports of the NetFPGA router are connected to the eth1 ports of FPGA worker-1 and FPGA worker-2 respectively. CPU workers and the FPGA helper CPUs are connected via MPI to enable all the workers to send the termination condition to the master periodically.

CPU worker-1 acts as a master and coordinates parallel computations on all workers. Both CPU and FPGA workers in this configuration work in tandem by exchanging KV pairs to complete the task.

3.4.2 Heterogeneous cluster with 3 CPUs and 1 FPGA

In this configuration the majority of the workers are CPU workers (Figure 3.8). The eth1 port of the CPU worker 1, 2 and 3 are connected to MAC1, MAC2 and MAC3 ports of the NetFPGA respectively. MAC4 port of the NetFPGA is connected to the eth1 port of the FPGA worker. CPU workers and the FPGA helper CPU are connected to the master via MPI.

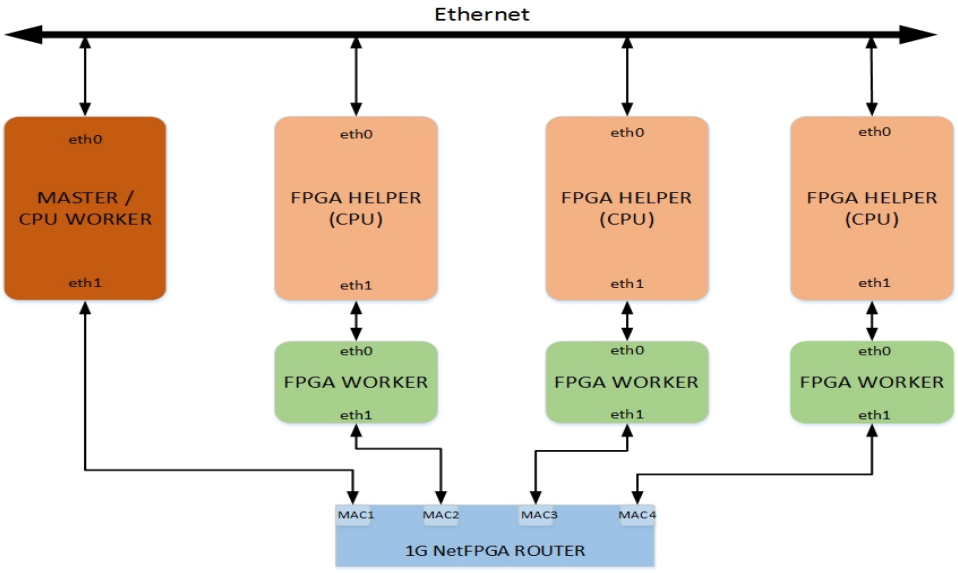

Figure 3.9: Laboratory prototype of a 1 CPU, 3 FPGA Hetero

3.4.3 Heterogeneous cluster with 1 CPU and 3 FPGAs

A heterogeneous configuration with 3 FPGAs and 1 CPU is shown in Figure 3.9. The eth1 port of the CPU worker is connected to MAC1 port of the NetFPGA. The eth1 ports of FPGA worker 1, 2 and 3 are connected to MAC2, MAC3 and MAC4 ports of the NetFPGA. The CPU worker in the cluster performs both master and worker functionality. The FPGA workers perform update and accumulate operations in parallel with the CPU to arrive at the termination condition.

3.5 Generation of a Synthetic Graph

Synthetic graphs are generated to evaluate the performance of the cluster. Graphs for Connected Components are weighted and graphs for Katz metric [24] and PageRank [10] are unweighted. The node IDs are whole numbers ranging from 0 to the size of the graph. The graphs are generated such that the in-degrees (number of edges with a graph node as terminal vertex) follow a log-normal distribution with parameters $\sigma = 0.5$ and $\mu = 2.3$.

Table 3.1: Iterative algorithms

Algorithm	$Init_j$	$g_j(x)$	$\oplus$
PageRank	$1-d$	$x.\Delta_j.a_{ji}$	$+$
Katz Metric	$1(j=s) \quad or \quad 0(j \neq s)$	$\beta.x.\Delta_j.a_{ji}$	$+$
Connected Components	j	$d.\frac{x}{L(j)}.\Delta_j.a_{ji}$	max

3.6 Iterative algorithms

To evaluate the performance of the heterogeneous cluster three iterative algorithms are considered. Table 3.1 specifies the initial values for the j^{th} key, update functions for the j^{th} key ($g_j(x)$) and accumulate operators ($\oplus$).

1. **Connected components**: The Connected Components algorithm is used to determine if all nodes in a graph are connected. In the initial phase, the value associated with each graph node is initialized to its node ID. All nodes propagate their values to their neighbors. When a node receives values from its neighboring nodes, it replaces its value with the largest received value from other nodes. The algorithm terminates when all the nodes which are connected have the same value.

2. **Katz metric**: Katz metric [24] provides a proximity measure between two nodes in a graph. The Katz metric is calculated by sum over all paths between two nodes exponentially dampened by the path length. The source node is initialized to 1 and all other nodes are initialized to 0. During an iteration, every node multiplies its current value with a constant dampening factor, β and propagates the value to all its neighbors. When a value message is received, the node accumulates the received value to its current value.

3. **PageRank**: The PageRank algorithm is used to rank websites for a search engine [10]. The detailed description of PageRank is found in Section 2.3.1.

This chapter explained the design of heterogeneous clusters with CPUs and FPGAs and their operation. We discussed the various iterative benchmarks which we use to evaluate the performance. In the succeeding chapters, we introduce the various configurations of heterogeneous cluster and evaluate their performance.

CHAPTER 4

LOAD BALANCING AND GRAPH PARTITIONING

In a cluster containing homogeneous processing elements, workload should be distributed evenly. In a heterogeneous cluster, with one or more processing units that are more powerful than the others, work partitioning is more complicated. For Hetero, we consider this issue by noting the relative performance of FPGAs and CPUs and performing unbalanced partitioning.

Graph partitioning aims to reduce the number of edge crossings between the partitioned graphs, thus reducing the communication costs between two graphs. In this chapter, we discuss how the technique of load balancing and graph partitioning are helpful in achieving better performance.

4.1 Asymmetric Load Balancing

Load balancing allocates workloads across multiple computing resources. Load balancing aims to make effective use of the available computing resources, prevent the overload of any particular resource and to reduce the idle waiting time of resources [39]. In a heterogeneous cluster with some computing elements having more processing power than others (e.g. FPGAs), we can improve performance by adopting an asymmetric load balancing strategy [35]. Asymmetric load balancing involves allocating more work to these computing resources with higher processing power.

Partitioning ratio	% of total nodes on CPUs	% of total nodes on FPGAs
50 : 50	50%	50%
40 : 60	40%	60%
30 : 70	30%	70%
20 : 80	20%	80%

Table 4.1: Example of load balancing on a heterogeneous cluster

4.2 Graph Partitioning

Graph partitioning often addresses the issue of dividing graph vertices into smaller sets such that there are few edge crossings between the sets [29]. One major application of graph partitioning is parallel computing. In a cluster of distributed processing elements which exchange data during computation, it is extremely beneficial to balance the workload among processing elements to reduce interprocess communication [20].

In most applications where input can be defined in terms of a graph, a vertex denotes computation and an edge between the two vertices denotes data dependency. For efficient execution time performance in a balanced computing environment, a graph must be partitioned into smaller graphs with approximately same number of vertices but fewer edge crossings [23].

Graph partitioning is known to be a NP complete problem [14] [18], but many heuristic algorithms have tried to solve it in an acceptable amount of time. One of the most popular graph partitioning algorithms is the K-L algorithm [25]. The algorithm follows greedy optimization technique and recursively moves vertices between partitions to reduce edge cuts between them. The K-L algorithm was originally developed for bisection and was later extended to quadrisection [32]. The quality of the partitions generated using K-L depends on the initial partition.

Partitioning algorithm	Total number of edges
Multilevel-KL	9828780
Spectral	10060000
Inertial	10270000
Linear KL	10295500
Random	10296200

Table 4.2: Total number of edge cuts for different partitioning methods

4.2.1 Chaco: Graph partitioning software

Chaco 2.0 [21] is an open source graph partitioning software. Chaco is available under license from Sandia National Laboratories. We obtained the source code along with technical documentation and sample input files via the Internet. Chaco is written entirely in ANSI C. Chaco offers functions such as:

1. Partition the graph using different partitioning methods with distinct properties.

2. Embed the partitions generated into several different topologies such as mesh and hypercube.

Chaco is designed to run on UNIX/LINUX systems. The five partitioning algorithms available in Chaco are: 1) Multilevel-KL, 2) Spectral, 3) Inertial, 4) Linear KL and 5) Random partitioning. Each of these partitioning algorithms partition the input graph into 2, 4 or 8 partitions. To evaluate the quality of partitions generated by each method, we chose an input graph of 2.1 million nodes and partitioned the graph into 4 sub parts using the above mentioned partitioning methods. The total number of edge crossings for each partitioning methods is listed in Table 4.2.

In random partitioning, vertices are assigned randomly to the sets to preserve balance. Multilevel-KL partitioning algorithm yields partitions with a smaller number of edge crossings between partitions. Multilevel K-L algorithm is divided into 3 phases. In phase one, an increasingly coarse approximation to the input graph is

Partition ratio	KL_IMBALANCE
50 : 50	0
40 : 60	0.4
30 : 70	0.85
20 : 80	1

Table 4.3: KL_IMBALANCE values for generating input graphs for 2 CPU + 2 FPGA heterogeneous cluster

constructed. In phase two, the smallest graph in the sequence is partitioned. In phase three the coarse partition is projected back through a sequence of graphs improving the quality with a local refinement algorithm (e.g. Kernighan and Lin).

Chaco keeps the number of vertices between partitions nearly as equal as possible. If we require unbalanced partitions for our applications we can introduce imbalance in the generated partitions though an user defined paramater KL_IMBALANCE [21]. When KL_IMBALANCE is set to q, between 0 and 1, the partitioning algorithms generate partitions having unequal number of vertices in each partition.

4.2.2 Generation of input partitioned graphs for various cluster configurations

To evaluate the effect of uneven load balancing on the cluster configurations discussed in section 3.4, we generate input graphs using Chaco as discussed below.

1. **Heterogeneous cluster with 2 CPUs and 2 FPGAs:** To evaluate the performance of this cluster for uneven load balancing, the input graph should be partitioned for different partitioning ratios as shown in Table 4.1. To obtain unbalanced partitions for different partitioning ratios, we set the KL_IMBALANCE value in Chaco as shown in Table 4.3.

   For each partitioning ratio, Chaco generates 4 partitions (part0, part1, part2 and part3) as output. The pairs, part0 and part1, and part2 and part3 contain

Partitioning ratio	KL_IMBALANCE		
	1st Partitioning	2nd Partitioning	3rd Partitioning
50 : 50	0	0.73	0
40 : 60	0.4	0.73	0
30 : 70	0.85	0.73	0
20 : 80	1	0.73	0

Table 4.4: KL_IMBALANCE values for generating input graphs for 1 CPU + 3 FPGA heterogeneous cluster

equal numbers of nodes. One partition pair (part0 and part1) is assigned to two CPUs and the other partition pair (part1 and part2) is assigned to two FPGAs.

2. **Heterogeneous cluster with 1 CPU and 3 FPGAs:**

   For this configuration, to generate input graphs we perform graph partitioning multiple times. For example, consider a load balancing ratio of 40:60, CPU-1 should be assigned a graph containing 40% of the input load and each FPGA should be assigned 20% of the load. To generate the necessary graphs using Chaco, we first partition the input graph into 2 partitions of 40% (part0) and 60% (temp1) of the nodes with KL_IMBALANCE = 0.4. We then partition the graph, temp1 into two partitions, with KL_IMBALANCE = 0.73 to obtain graphs with 20% (part1) and 40% (temp2) of the total nodes. In the third partitioning, we bipartition the graph, temp2 with KL_IMBALANCE = 0 to obtain partitions with 20% of total input nodes (part2 and part3).

   The above procedure is followed to generate the required graphs for the other partitioning ratios. Table 4.4 shows the values of KL_IMBALANCE to be used while generating input graphs for all partitioning ratios.

3. **Heterogeneous cluster with 3 CPUs and 1 FPGA:**

In the heterogeneous cluster of 3 CPUs and 1 FPGA, we follow a similar methodology to the one followed for a heterogeneous cluster of 1 CPU and 3 FPGAs to generate the input graphs.

For example, to generate graphs with a partition ratio of 40:60, we partition the input graph with KL_IMBALANCE = 0.4, to obtain the partitions containing 40% (temp1) and 60% (part0) of the nodes. We choose the graph temp1 to do the partitioning the second time with KL_IMBALANCE = 0.73 and obtain the graphs with 26.66% (temp2) and 13.34% (part1) of nodes. In the third partitioning, we partition the graph temp2 with KL_IMBALANCE = 0 to obtain the partitions with 13.33% (part2 and part3) of input nodes.

To generate the input graphs with other partitioning ratios, we use KL_IMBALANCE values given in Table 4.4. Here, we select the same KL_IMBALANCE values as in 1 CPU + 3 FPGA cluster at each stage of partitioning, but select a different graph from the previous partition output as the input graph in the current partitioning.

In this chapter, we discussed the concept of static load balancing and graph partitioning. We illustrated how we can generate the input graphs for different cluster configurations using Chaco. In Chapter 5 we present the performance of different cluster configurations for uneven load balancing and graph partitioning techniques.

CHAPTER 5

EXPERIMENTAL RESULTS

In this chapter, we discuss the performance measurements of various cluster configurations described in section 3.4. First, we obtain performance measurements of all clusters by executing PageRank, Katz and Connected Components benchmarks for a randomly partitioned input graph. Second, the execution time is compared with the input graphs created with multi-level K-L partitioning. All the experiments are conducted for Tx/Rx ratio of 4:4 on FPGAs, as a ratio of 4:4 is shown to provide maximum speedup compared to other Tx/Rx ratios.

Input graphs of 2.4 million nodes on a heterogeneous cluster of 3 CPUs and 1 FPGA, 4.8 million nodes on a heterogeneous cluster of 1 CPU and 3 FPGAs and 3.6 million nodes on a heterogeneous cluster of 2 CPUs and 2 FPGAs are used.

5.1 Performance Variation across Different Cluster Configurations

The speedup of heterogeneous clusters with respect to a homogeneous cluster of CPUs for randomly partitioned graphs for the Katz benchmark is shown in Figure 5.1. For all cluster configurations, the execution time decreases as more graph load is assigned to FPGAs. We observe that for all the heterogeneous cluster configurations, the execution time is smaller than for a homogeneous CPU cluster. A homogeneous FPGA cluster offers highest speedup because the accumulate and update operations are performed entirely on FPGAs. In a heterogeneous cluster, the computational capability of CPUs is the main impediment to achieve better speedup.

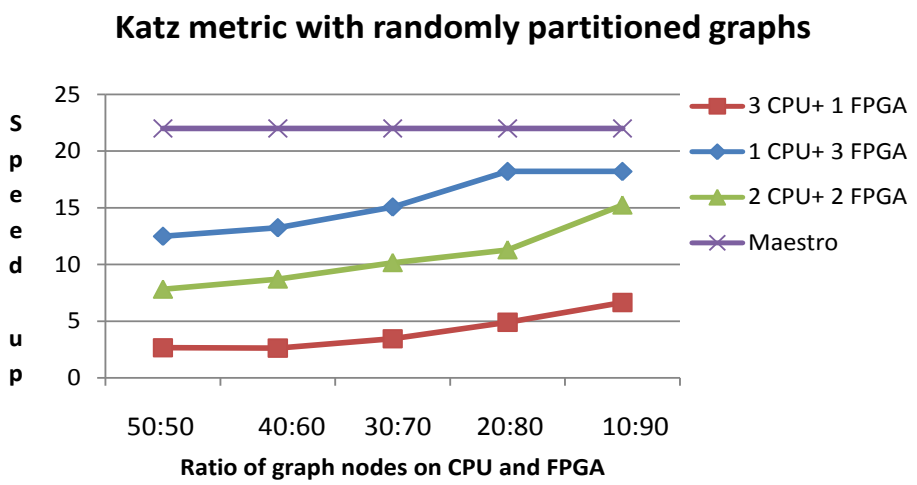

Figure 5.1: Performance of various cluster configurations for different partitioning ratio versus a four processor node configuration

Load Ratio	1 CPU + 3 FPGA	2 CPU + 2 FPGA	3 CPU + 1 FPGA
50 : 50	12	7	2.5
10 : 90	18	15	6

Table 5.1: Speedup of different cluster configurations for Katz benchmark versus a four-processor configuration

As shown in Figure 5.1, a heterogeneous 1 CPU + 3 FPGA cluster achieves higher speedup compared to other configurations. More number of FPGAs (accelerators) are involved in the computation compared to other configurations. The 3 CPU + 1 FPGA cluster includes a single FPGA and provides less speedup due to fewer FPGAs.

As illustrated in Figure 5.1, as we assign more load on the FPGAs, the speedup increases irrespective of number of CPUs and FPGAs in the cluster. FPGAs are computationally faster than CPUs (Figure 5.7), and can process more load than a CPU in a given amount of time. As more load is assigned to FPGAs, most of the computation happens on FPGAs and results in less execution time.

In Table 5.1, the speedup values for 50% and 90% load on FPGAs have been tabulated. We can observe a decrease in speed up as less work is assigned on FPGAs and increase in speedup as number of FPGAs in a cluster increases.

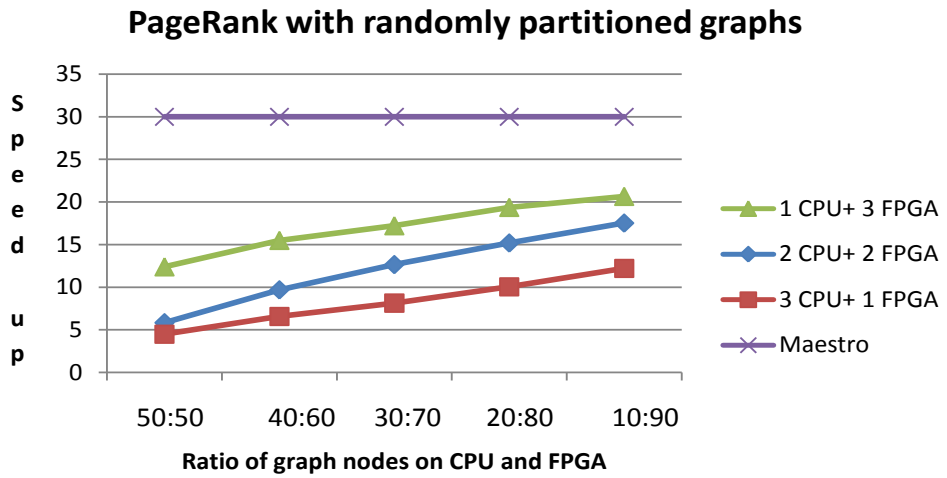

Figure 5.2: Performance of various cluster configurations for different partitioning ratios versus a four-processor cluster

Load Ratio	1 CPU + 3 FPGA	2 CPU + 2 FPGA	3 CPU + 1 FPGA
50 : 50	12	6	4
10 : 90	20.5	17	12

Table 5.2: Speedup of different cluster configurations for PageRank benchmark versus a four-node processor cluster

Figure 5.2 shows the speedup of different cluster configurations for different partitioning ratios for the PageRank banchmark. Similar to the Katz benchmark, we observe an increase in speedup when more load is assigned to FPGAs in a cluster and the overall speedup of a cluster decreases when CPUs outnumber FPGAs. Similar explanations as for the Katz benchmark can be applied here. Table 5.2 quantifies the speedup for loads of 50% and 90% on the FPGAs for the PageRank benchmark.

Speedup measurements of the Connected Components benchmark for different cluster configurations are shown in Figure 5.3. Analogous to other benchmarks (Katz and PageRank), performance measurements of the Connected Components exhibit similar characteristics. The higher speedup of the 1 CPU + 3 FPGA cluster is attributed to the presence of 3 accelerating elements (FPGAs). In Table 5.3, we notice the speedup of the Connected Components benchmark is lower than Katz

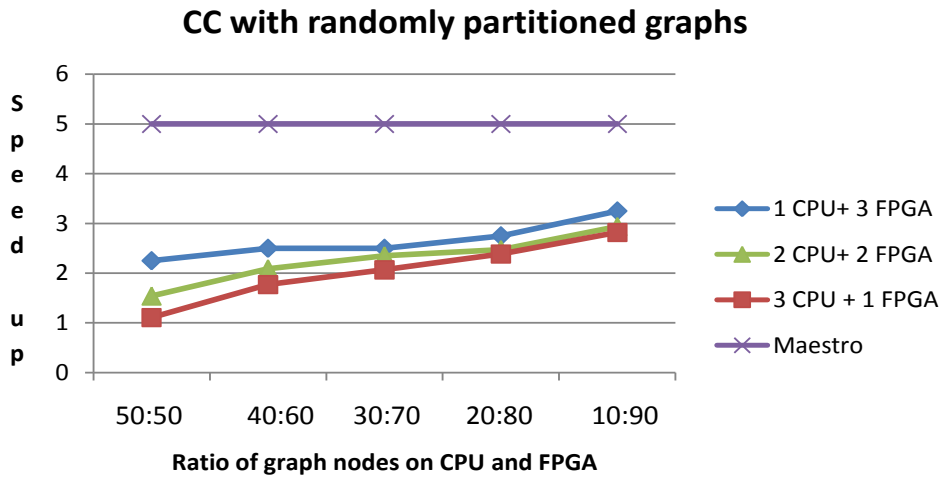

Figure 5.3: Performance of various cluster configurations for different partitioning ratio

Load Ratio	1 CPU + 3 FPGA	2 CPU + 2 FPGA	3 CPU + 1 FPGA
50 : 50	1.1	1.5	2.25
10 : 90	3.25	2.9	2.8

Table 5.3: Speedup of different cluster configurations for Connected Components benchmark

and PageRank. The Connected Components algorithm does not involve arithmetic calculations like PageRank and Katz. FPGA parallelism helps accelerate complex floating point calculations. In connected components, the update function is "Max", but in Katz and PageRank the update involves floating point division (Table 3.1). The performance of CPUs and FPGAs are comparable for the "Max"" update function, hence we don't observe significant speedup in heterogeneous clusters with FPGAs.

5.2 Performance Variation Using Different Graph Partitioning Methods

Figure 5.4 shows a speedup comparison between a randomly partitioned input graph and an input graph partitioned using multilevel-KL for the Katz benchmark.

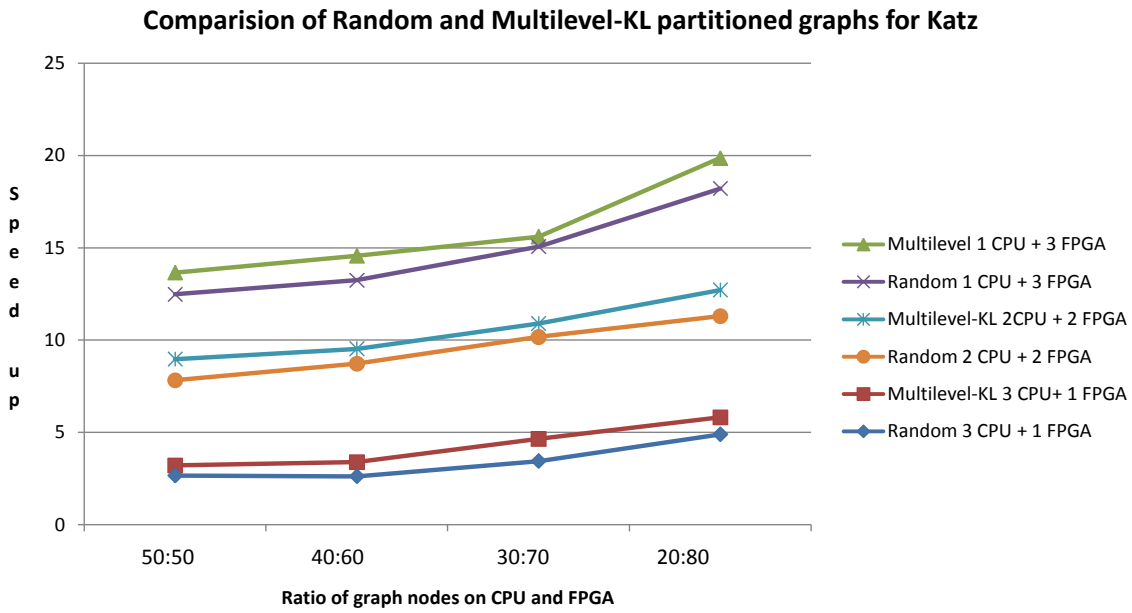

Figure 5.4: Comparison of performance of heterogeneous clusters for input graphs partitioned using different partitioning methods for the Katz benchmark versus a four-processor cluster

Load Ratio	1 CPU + 3 FPGA	2 CPU + 2 FPGA	3 CPU + 1 FPGA
10 : 90	15%	11%	20%

Table 5.4: Improvement in speedup for multilevel K-L partitioned graph over randomly partitioned graph for the Katz benchmark versus a four-processor cluster

For all cluster configurations in Figure 5.4, the multilevel K-L input graph shows improvement in speedup compared to the randomly partitioned graph input.

Table 5.5 compares the total number of edges for different partitioning methods. We notice that the total number of cut edges for multilevel K-L partitioning is less than random partitioning for different partition ratios. The reduced number of edge cuts between graph partitions generated using multilevel K-L partitioning results in a reduction in the number of messages exchanged between computing nodes.

Figures 5.5 and 5.6 show a speedup comparison for a randomly partitioned input graph versus an input graph partitioned using multilevel-KL for PageRank and Connected Components benchmarks. Similar to the Katz benchmark, we observe an

Partitioning ratio	Total number of Edge Cuts	
	Random	Multilevel K-L
50 : 50	28356180	26176001
40 : 60	27564199	25391169
30 : 70	23081222	21781210
20 : 80	22491320	19001125

Table 5.5: Total number of edges cuts for Random and Multilevel K-L partitioning methods

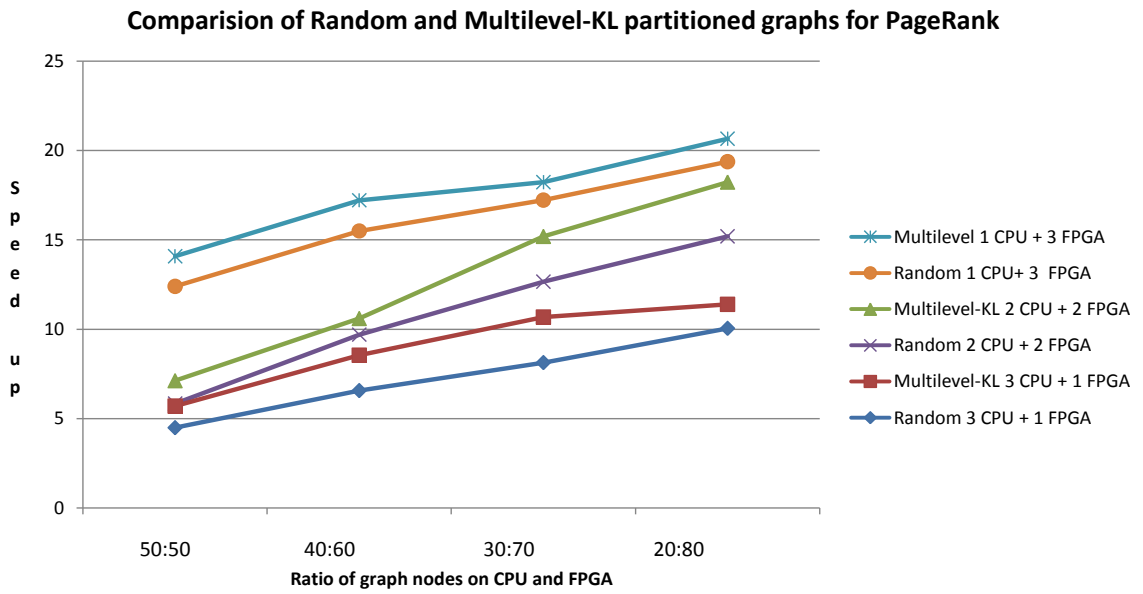

Figure 5.5: Comparison of performance of heterogeneous clusters for input graphs partitioned using different partitioning methods for the PageRank benchmark

improvement in speedup for input graphs partitioned using the multilevel K-L partitioning method. The increase in speedup is associated with decrease in the edge cuts between partitions. Table 5.6 and Table 5.7 show the percentage improvement in speedup for multilevel K-L partitioned graphs for PageRank and Connected Component benchmark respectively.

Load Ratio	1 CPU + 3 FPGA	2 CPU + 2 FPGA	3 CPU + 1 FPGA
10 : 90	11%	26%	15%

Table 5.6: Improvement in speedup for a multilevel K-L partitioned graph over a randomly partitioned graph for the PageRank benchmark

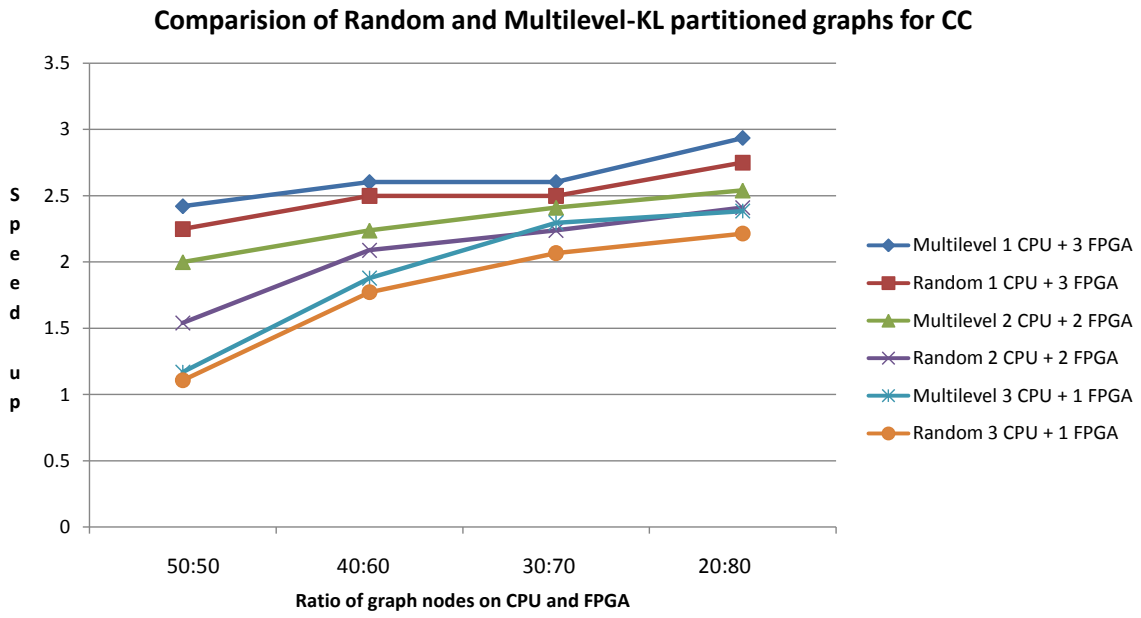

Figure 5.6: Comparison of performance of heterogeneous clusters for input graphs partitioned using different partitioning methods for the Connected Components benchmark

5.3 Modeling partitioning ratio

In Figure 5.7, we can observe that for the PageRank benchmark, 1 FPGA node gives a 4.5X speedup versus 1 CPU node. For Katz and Connected Components benchmarks, a 1 node FPGA configuration provides 3.8X and 1.5X speedup, respectively versus a 1 node CPU configuration.

For the PageRank benchmark, since a single FPGA performs computation 4.5X faster than a single CPU, hypothetically we can assign 4.5X more load to an FPGA than a CPU to complete the computation in an equal amount of time. In a heterogeneous cluster of CPUs and FPGAs, we need to balance the load such that neither the CPUs nor the FPGAs are starved or overloaded.

Load Ratio	1 CPU + 3 FPGA	2 CPU + 2 FPGA	3 CPU + 1 FPGA
10 : 90	8%	9%	8%

Table 5.7: Improvement in speedup for a multilevel K-L partitioned graph over a randomly partitioned graph for the Connected Components benchmark

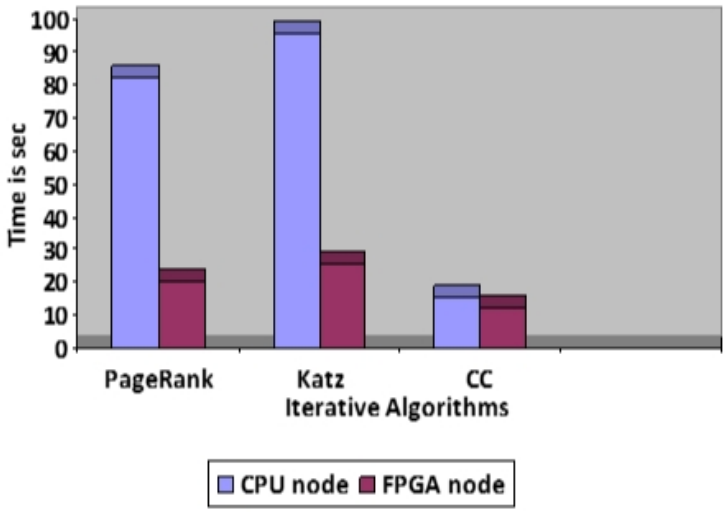

Figure 5.7: Execution time of a single node FPGA vs. a single node CPU for different benchmarks

Consider a cluster of 'm' CPUs and 'n' FPGAs. Assume, we have a load of size 'K'. Let 's' be the speedup of 1 FPGA node over 1 CPU node for a benchmark 'B'.

By assigning 's' times more load to individual FPGA than CPU, we have.

$$m \cdot x + s \cdot n \cdot x = K \tag{5.1}$$

$$x = \frac{K}{m + (s \cdot n)} \tag{5.2}$$

Hence, the load assigned to m CPUs = $\frac{mK}{m+sn}$ and the load assigned to n FPGAs = $\frac{snK}{m+sn}$. Therefore, the appropriate load balancing ratio = $m : s \cdot n$ Table 5.8 shows the partitioning ratio for different heterogeneous cluster configurations and

Config.	Benchmark					
	PageRank		Katz		CC	
	Ratio (CPU : FPGA)	Speedup	Ratio (CPU : FPGA)	Speedup	Ratio (CPU : FPGA)	Speedup
1 CPU + 3 FPGA	7 : 93	22	10 : 90	18.2	20 : 80	2.75
2 CPU + 2 FPGA	15 : 85	20	20 : 80	11.5	40 : 60	2
3 CPU + 1 FPGA	40 : 60	10	45 : 55	2.6	65 : 35	1.75

Table 5.8: Speedup of various configurations for appropriate load balancing ratio

Configuration	Speedup for Appropriate Partitioning			Estimated Cost
	PageRank	Katz	CC	
Maestro	30	22	5	$12,000
1 CPU + 3 FPGA	22	18.2	2.75	$9,500
2 CPU + 2 FPGA	20	11.5	2	$7,000
3 CPU + 1 FPGA	10	2.6	1.75	$4,500

Table 5.9: Estimated cost vs. speedup of different cluster configurations

the corresponding speedup for various clusters. A 1 CPU + 3 FPGA cluster with appropriate load balancing records higher speedup. In contrast, a 3 CPU + 1 FPGA records less speedup for an appropriate load balancing ratio which can be attributed to the presence of more accelerators (FPGAs) in the former configuration than in the latter configuration.

5.4 Cost Analysis

Table 5.9 shows the estimated cost and speedup trade-off of all heterogeneous cluster configurations for three different benchmarks and appropriate load balancing ratio. For these comparisons, we assume that a CPU workstation costs $500 and each Altera DE4 board costs $3,000. We do not consider the cost of FPGA helper CPUs as they are used only for experimental prototyping and experimentation and can be

replaced by a soft processor on FPGAs. A 1 CPU + 3 FPGA cluster yields higher speedup for all benchmarks but is more expensive among all other cluster configurations. A 3 CPU + 1 FPGA cluster is least expensive among all configurations, but provides less speedup. We can conclude that a 2 CPU + 2 FPGA is a better trade-off between cost and performance among all clusters.

The Connected Components benchmark provides less speedup compared to other benchmarks for an appropriate load balancing ratio. We can speculate that if we are running a non-compute intensive task we should choose a heterogeneous cluster which has more CPU computing nodes and for a compute intensive task a heterogeneous cluster with more FPGA computing nodes for a better speed versus cost trade-off.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

Asynchronous Accumulative Updates (AAU) provide improved performance compared to conventional iterative algorithms. In previous research, AAU was implemented on a homogeneous cluster of CPUs and FPGAs. The absence of synchronization barriers between two iterations in AAU is employed to build different configurations of heterogeneous clusters containing CPUs and FPGAs.

In this thesis, we built 3 different configurations of heterogeneous clusters: 3 CPU and 1 FPGA, 2 CPU and 2 FPGA and 1 CPU and 3 FPGA. We executed three different benchmarks (PageRank, Katz and Connected Components) on each configuration.

We varied the input load on the CPUs and FPGAs by gradually increasing the input load on the FPGAs from 50% to 90% of total graph nodes. We showed that the performance of a heterogeneous cluster can be increased by incorporating more FPGAs in the cluster. We also observed that increased data load on the FPGAs reduces execution time.

We obtained a speedup of 10X, 20X and 22X for a 3 CPU + 1 FPGA, 2 CPU + 2 FPGA and 1 CPU + 3 FPGA cluster for the PageRank benchmark, 2.6X, 11.5X and 18.2X for a 3 CPU + 1 FPGA, 2 CPU + 2 FPGA and 1 CPU + 3 FPGA cluster for the Katz benchmark and 1.75X, 2X and 2.75X for 3 CPU + 1 FPGA, 2 CPU + 2 FPGA and 1 CPU + 3 FPGA clusters for the Connected Components benchmark. All experiments were conducted for an appropriate load balancing ratio

for each configuration. This ratio was determined by examining the runtime speedup for an FPGA versus a processor.

Chaco graph partitioning software was used to partition input graphs using multilevel KL. We ran the partitioned graphs on the cluster configurations to obtain execution time for different input load ratios for CPUs and FPGAs. We observed that when input graphs partitioned using multilevel-KL method were used, execution time was less compared to when input graphs generated with random partitioning were used.

In future work, we plan to implement a dynamic load balancing mechanism. We plan to design a dynamic load balancer which runs on the master CPU. The balancer examines the load on CPUs and FPGAs and balances the load by transferring some work from a heavily loaded node to a lightly loaded node. The dynamic load balancer will carefully track the load on all computing nodes in the cluster [16].

We plan to implement the fault tolerance in the heterogeneous cluster by periodically saving the state table of CPUs to disk and the state table in DRAM adjacent to FPGAs to the disk of the FPGA helper CPUs. We plan to scale the 4 node heterogeneous cluster to include more CPUs and FPGAs.

BIBLIOGRAPHY

[1] Altera de4 development and education board. http://www.altera.com/education/univ/materials/boards/de4/unv-de4-board.html. Altera Corporation.

[2] Amazon ec2. http://aws.amazon.com/ec2/.

[3] Big data facts and statistics that will shock you. http://www.fathomdelivers.com/blog/analytics-and-big-data/big-data-facts-and-statistics-that-will-shock-you/.

[4] Ebay study: How to build trust and improve the shopping experience. http://research.wpcarey.asu.edu/managing-it/ebay-study-how-to-build-trust-and-improve-the-shopping-experience/.

[5] Hadoop. http://hadoop.apache.org/.

[6] Heterospark. https://spark-summit.org/2015-east/wp-content/uploads/2015/03/SSE15-28-Peilong-Li-Yan-Luo.pdf.

[7] Maiter project. http://code.google.com/p/maiter/.

[8] Netfpga wiki. https://github.com/NetFPGA/netfpga/wiki/Guide.

[9] Open mpi. http://www.open-mpi.org/.

[10] Brin, Sergey, and Page, Lawrence. The anatomy of a large-scale hypertextual web search engine. *Comput. Netw. ISDN Syst. 30*, 1-7 (Apr. 1998), 107–117.

[11] Bu, Yingyi, Howe, Bill, Balazinska, Magdalena, and Ernst, Michael D. Haloop: Efficient iterative data processing on large clusters. *Proc. VLDB Endow. 3*, 1-2 (Sept. 2010), 285–296.

[12] Che, Shuai, Li, Jie, Sheaffer, Jeremy W., Skadron, Kevin, and Lach, John. Accelerating compute-intensive applications with gpus and fpgas. In *Proceedings of the 2008 Symposium on Application Specific Processors* (Washington, DC, USA, 2008), SASP '08, IEEE Computer Society, pp. 101–107.

[13] Choi, Yuk-Ming, and So, H.K.-H. Map-reduce processing of k-means algorithm with fpga-accelerated computer cluster. In *Application-specific Systems, Architectures and Processors (ASAP), 2014 IEEE 25th International Conference on* (June 2014), pp. 9–16.

[14] Cioaba, Sebastian M. The np-completeness of some edge-partitioning problems. Master's thesis, Queens University Kingston, Ontario, Canada.

[15] Dean, Jeffrey, and Ghemawat, Sanjay. Mapreduce: Simplified data processing on large clusters. *Commun. ACM 51*, 1 (Jan. 2008), 107–113.

[16] Devine, Karen D, Boman, Erik G, Heaphy, Robert T, Hendrickson, Bruce A, Teresco, James D, Faik, Jamal, Flaherty, Joseph E, and Gervasio, Luis G. New challenges in dynamic load balancing. *Applied Numerical Mathematics 52*, 2 (2005), 133–152.

[17] Frommer, Andreas, and Szyld, Daniel B. On asynchronous iterations. *J. Comput. Appl. Math. 123*, 1-2 (Nov. 2000), 201–216.

[18] Garey, M. R., Johnson, D. S., and Stockmeyer, L. Some simplified np-complete problems. In *Proceedings of the Sixth Annual ACM Symposium on Theory of Computing* (New York, NY, USA, 1974), STOC '74, ACM, pp. 47–63.

[19] He, Bingsheng, Fang, Wenbin, Luo, Qiong, Govindaraju, Naga K., and Wang, Tuyong. Mars: A mapreduce framework on graphics processors. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques* (New York, NY, USA, 2008), PACT '08, ACM, pp. 260–269.

[20] Hendrickson, Bruce, and Kolda, Tamara G. Graph partitioning models for parallel computing. *Parallel Comput. 26*, 12 (Nov. 2000), 1519–1534.

[21] Hendrickson, Bruce, and Leland, Robert. The chaco users guide version 2.0.

[22] Hu, Han, Wen, Yonggang, Chua, Tat-Sang, and Li, Xuelong. Toward scalable systems for big data analytics: A technology tutorial. *IEEE Access 2* (2014).

[23] Jones, C.A. *Vertex and Edge Partitions of Graphs.* PhD thesis, Penn State, Dept of Computer Science State College, PA,

[24] Katz, Leo. A new status index derived from sociometric analysis. In *Psychometrika.*

[25] Kernighan, B. W., and Lin, S. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal 49*, 2 (1970), 291–307.

[26] Liben-Nowell, D., and Kleinberg, J. The link-prediction problem for social networks. *Journal of the Association for Information Science and Technology* (2007).

[27] Low, Yucheng, Bickson, Danny, Gonzalez, Joseph, Guestrin, Carlos, Kyrola, Aapo, and Hellerstein, Joseph M. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow. 5*, 8 (Apr. 2012), 716–727.

[28] Power, Russell, and Li, Jinyang. Piccolo: Building fast, distributed programs with partitioned tables. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2010), OSDI'10, USENIX Association, pp. 1–14.

[29] Sanders, Peter, and Schulz, Christian. High quality graph partitioning.

[30] Shan, Yi, Wang, Bo, Yan, Jing, Wang, Yu, Xu, Ningyi, and Yang, Huazhong. Fpmr: Mapreduce framework on fpga. In *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (New York, NY, USA, 2010), FPGA '10, ACM, pp. 93–102.

[31] Song, Han Hee, Cho, Tae Won, Dave, Vacha, Zhang, Yin, and Qiu, Lili. Scalable proximity estimation and link prediction in online social networks. In *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement Conference* (New York, NY, USA, 2009), IMC '09, ACM, pp. 322–335.

[32] Suaris, P.R., and Kedem, G. An algorithm for quadrisection and its application to standard cell placement. *Circuits and Systems, IEEE Transactions on 35*, 3 (March 1988), 294–303.

[33] Tsoi, Kuen Hung, and Luk, Wayne. Axel: A heterogeneous cluster with fpgas and gpus. In *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (New York, NY, USA, 2010), FPGA '10, ACM, pp. 115–124.

[34] Unnikrishnan, D., Virupaksha, S.G., Krishnan, L., Gao, L., and Tessier, R. Accelerating iterative algorithms with asynchronous accumulative updates on fpgas. In *Field-Programmable Technology (FPT), 2013 International Conference on* (Dec 2013), pp. 66–73.

[35] Vöcking, Berthold. How asymmetry helps load balancing. *J. ACM 50*, 4 (July 2003), 568–589.

[36] Yeung, Jackson H. C., Tsang, C. C., Tsoi, K. H., Kwan, Bill S. H., Cheung, Chris C. C., Chan, Anthony P. C., and Leong, Philip H. W. Map-reduce as a programming model for custom computing machines. In *Proceedings of the 2008 16th International Symposium on Field-Programmable Custom Computing Machines* (Washington, DC, USA, 2008), FCCM '08, IEEE Computer Society, pp. 149–159.

[37] Zaharia, Matei, Chowdhury, Mosharaf, Franklin, Michael J., Shenker, Scott, and Stoica, Ion. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing* (Berkeley, CA, USA, 2010), HotCloud'10, USENIX Association, pp. 10–10.

[38] Zaharia, Matei, Konwinski, Andy, Joseph, Anthony D., Katz, Randy, and Stoica, Ion. Improving mapreduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2008), OSDI'08, USENIX Association, pp. 29–42.

[39] Zhang, Y, Kameda, H, and Hung, S-L. Comparison of dynamic and static load-balancing strategies in heterogeneous distributed systems. *IEE Proceedings-Computers and Digital Techniques 144*, 2 (1997), 100–106.

[40] Zhang, Yanfeng, Gao, Qinxin, Gao, Lixin, and Wang, Cuirong. 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* (Washington, DC, USA, 2011), IPDPSW '11, IEEE Computer Society, pp. 1112–1121.

[41] Zhang, Yanfeng, Gao, Qixin, Gao, Lixin, and Wang, Cuirong. Priter: A distributed framework for prioritized iterative computations. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing* (New York, NY, USA, 2011), SOCC '11, ACM, pp. 13:1–13:14.

[42] Zhang, Yanfeng, Gao, Qixin, Gao, Lixin, and Wang, Cuirong. Accelerate large-scale iterative computation through asynchronous accumulative updates. In *Proceedings of the 3rd Workshop on Scientific Cloud Computing Date* (New York, NY, USA, 2012), ScienceCloud '12, ACM, pp. 13–22.