SECOND LEVEL PARALLELISM USING SIMD ACCELERATORS ON HETEROGENEOUS MAPREDUCE CLUSTERS

Masoud Ebrahimi¹ and Farshad Khunjush¹,²,³
¹ School of Electrical & Computer Engineering
Shiraz University, Shiraz, Iran
² Department of Electrical and Electronics Engineering
Hormozgan University, Bandar Abbas, Iran
³ School of Computer Science,
Institute for Research in Fundamental Sciences (IPM), Tehran, Iran.
{3brahimi, khunjush@cse.shirazu.ac.ir}

ABSTRACT
The MapReduce programming model introduced by Google is one of the most successful efforts to cope with the growth of demand for processing large amount of data in large-scale clusters. Although MapReduce programming paradigm has never been easier or more scalable, distributed platforms have changed drastically in recent years. These days, most of the data centers and clusters are equipped with new processing elements such as Multi-Core CPUs, SIMD accelerators particularly, and FPGAs. Unfortunately, current MapReduce frameworks are incapable in harnessing the computational power of these available nodes. In this paper, we propose a new design philosophy to implement MapReduce frameworks in order to comply with above-mentioned multi-level parallelism that exists in modern data centers. We designed a novel architecture to leverage all types of SIMD architectures in distributed platforms. Experiments and evaluations show our novel implementation not only complies with the characteristics of MapReduce applications but also outperforms Hadoop in terms of speedup and throughput.

KEY WORDS
MapReduce; Heterogeneous Computing; Distributed Computing; Algorithms for Heterogeneous Systems; Cluster Computing

1. Introduction
Recently, there has been an increasing demand for processing huge amount of data that has been continuously generated by millions of users or computers around the world. The MapReduce programming model introduced by Google [1] is one of the most successful efforts to cope with growing demand for processing large amount of data in large-scale clusters. The key benefit of this model is that it hides the complex details of parallelism such as scheduling, load balancing, communication, and synchronization from programmers. The nature of the MapReduce model is based on a coarse-grained parallel model, and these types of applications rarely need to communicate. On the other hand, another important trend in the parallel programming community has been the use of fine-grained accelerators on each node of a cluster to boost its total computational power. Most of successful existing MapReduce frameworks were implemented based on these architectures (e.g. Phoenix [2] for multi-core CPUs and GPU-driven frameworks named Soren [3] and Mars [4]). Current implementations of MapReduce model such as Hadoop [5] enable programmers to harness the computational resources in clusters.

Nowadays data centers not only are used to handle massive amounts of data but also to perform large-scale processing; thus, most of them consist of various types of computational accelerators. There are even clusters that include one or more types of modern processors without any conventional CPUs involved in their computations [6]. Simultaneously with the emergence of new MapReduce frameworks, specialized parallel processor architectures such as programmable GPUs, the IBM Cell processors and custom-made accelerators were sometimes used alongside traditional processors, for accelerating specific tasks in data centers. In reality, the architectural differences between conventional CPUs and accelerators pose major obstacles in achieving a seamless hierarchical MapReduce execution. Current MapReduce implementations are supporting conventional CPUs and accelerators pose major obstacles in achieving a seamless hierarchical MapReduce execution. Current MapReduce implementations are supporting conventional CPUs and make the most out of their processing powers, but they are not designed to harness all available computational powers on cluster nodes.

Lack of the possibility to take the advantage of SIMD accelerators alongside modern multi-core CPUs in massive scale is a big loss in today's MapReduce implementations; However, there have been various efforts to exploit the computational power of programmable GPUs in Hadoop as a well-known MapReduce framework [7], [8], [9]. Studying the Hadoop framework and recent efforts on making this homogeneous framework leverage heterogeneous processors shows there are two major challenges that pose major obstacles in achieving a seamless hierarchical MapReduce execution. First, accelerators with different architectures result in having different programming languages, and most of them are derived from the C language and with a Semi-C syntax such as CUDA [10] and OpenCL [11], while MapReduce frameworks were formed around high-level languages such as Java. This gap among high-level languages and Semi-C has created difficulties in implementation and utilizing accelerators in frameworks such as Hadoop. Furthermore, processor scheduling is significantly more challenging in heterogeneous clusters consisting of CPUs and accelerators compared to that in homogeneous clusters. For example, due to architectural differences, CPUs and accelerators have different optimal
Prior to this, computational slot and the container were used for performing concurrent tasks. The number of computational slots is limited to the amount of available resources on distributed environments. In fact, each slot is a subset of available resources on a particular node. Lee et al. [12] provide a solution to the above-mentioned problem about the differences between accelerators and CPUs by defining heterogeneous slots. What is missing in previous approaches using this concept is the amount of resource usage for each task assigned to the slots. There are situations in which these resources are underutilized or captured for a long period of time that both may result in performance degradations.

This paper presents a concept called Executor to solve above-mentioned problems in the distributed MapReduce platforms. An executor is not an abstraction of resources but an execution engine aims to perform a set of tasks on the cluster nodes. We propose a new design philosophy for MapReduce platforms wherein Executors along with several techniques such as resource monitoring and multiple-issue are leveraged to sustain a high throughput in running MapReduce applications in cluster environments.

As discussed, one of the serious shortcomings in the Hadoop platform is the lack of multi-core support in its current implementation. Considering the current trend of using multi-core processors on almost every node of modern clusters and data centers, the ability of leveraging these resources is vital to reach high-performance in large-scale data processing centers. Therefore, supporting multi-core processors in MapReduce implementations for cluster environments is another objective in our proposed framework. We have successfully substituted CPUs with programmable GPUs and then evaluated the efficacy of our proposed framework. We claim this architecture can be driven by any kind of processor solely or in conjunction with other accelerators with almost zero change in its core implementation.

The remainder of this paper is organized as follows. We provide a concise background on the MapReduce programming model and available implementations in Section 2. Design philosophy and key design factors are provided in Section 3. Section 4 describes the experimental setup and the characteristics of MapReduce applications, which we have chosen. In Section 5, we evaluate the performance of our proposed framework. We present related work in Section 6; and finally we conclude in Section 7.

2. Background

2.1 MapReduce Programming Model

In the MapReduce model, programmers must specify two core functions: a map function that receives a chunk of input data and emits a set of intermediate \( \langle \text{key}, \text{value} \rangle \) pairs, and a reduce function that aggregates \( \langle \text{key}, \text{value} \rangle \) pairs with the same key and reduces them to a final form. In this model, the details of parallelism, including data decomposition, task creation, synchronization, and data communication, are handled by a MapReduce run-time system.

2.2 Hadoop's Architecture and Overall Workflow

In this sub-section, we describe the architecture of the Hadoop framework by investigating its major components; then, we will overview its overall workflow. Hadoop is an open source implementation of the MapReduce programming model for homogeneous clusters. The Hadoop framework includes a two-tier architecture. The upper layer, which is MapReduce, relies on a distributed file system named Hadoop Distributed File System (HDFS) as shown in Figure 1. Jobs in Hadoop are defined as a composition of Java map, reduce functions along with addresses to the input file, and desired output locations on HDFS. A Hadoop cluster consists of two types of nodes: a master node, which acts as the central regulation point in each cluster, and slave nodes, administrated by the master node. The Hadoop MapReduce run-time system includes two major components: JobTracker running on the master node, and TaskTrackers that run on slave nodes. The JobTracker is the main server in this architecture. It splits jobs into several maps and/or reduces tasks, and waits for TaskTrackers to request tasks from this task pool. The JobTracker's scheduler selects zero or more tasks waiting for being initiated by receiving a task request from any TaskTracker and assigns them to the requester. Afterwards, the TaskTracker forks out independent JVMs to process its assigned tasks. Then, the map tasks output intermediate results on local file system of slave nodes before reducers request to fetch them to start reducing the generated results.

The Hadoop output collector is the entity responsible for gathering and handling the intermediate results of the map phase. It tries to enhance performance of the map phase, by employing a combination of three operations on the \( \langle \text{key}, \text{value} \rangle \) pairs in its buffer: sort, optional incremental combine, and spill. The first and the last operations are abbreviated as SortAndSpill in the Hadoop implementation. A sort operation based on the keys is applied on all \( \langle \text{key}, \text{value} \rangle \) pairs; and then an optional combine operation is employed by the output collector to aggregate the intermediate pairs for each machine. This combine phase helps to cut down the size of data transfers from the mappers to the reducers. Spill-files are generated using these intermediate results on every mapper. Afterwards, all the spill-files will be merged into a single intermediate file. Prior to the reduce phase, there is a compulsory middle phase named Shuffle. The outcome of the merge is scattered into several partitions in the shuffle phase and then these partitions are transferred from mappers to the reducers.

2.3 Java Alternatives in Hadoop

Besides Java language to write applications for the Hadoop platform, there are alternatives for this purpose such as Hadoop Streaming and Java Native Interface (JNI). By using these language alternatives researchers have tried to stitch accelerators' codes into the Hadoop MapReduce in heterogeneous environments [7], [8]. In this sub-section, we briefly describe these Java alternatives.
2.3.1 Hadoop Streaming

Hadoop Streaming allows us to create and run MapReduce jobs with any executable or script as the mapper and/or the reducer [13]. This utility comes with the Hadoop distribution. Hadoop Streaming is a text protocol, which uses the *Stdin* as its input and writes its outputs to the *Stdout*. Handling arbitrary data streams using this method is not straightforward but our experiments revealed if a programmer handles the input stream carefully and generates the intermediate results of the map phase appropriately, there is no shortage in this aspect.

2.3.2 Hadoop Pipes

This Java alternative allows developers to implement their MapReduce applications using C++. Primary approach is to split C++ code into a separate process that runs the application specific code. Although C++ interface is swig-able meaning all C++ standard functions and classes can be included in the Hadoop Pipes namespace, there may be unexpected behaviors while using customized libraries for implementing accelerator driven codes. It is worth mentioning that in Hadoop 0.20.1 the support for Pipes is deprecated; thus, further experiments in this area seem unnecessary to us.

It has been shown in [8] that extremity of Hadoop acceleration is reached using these alternatives or any architecture-specific interfaces such as JCUDA [14]. Previous approaches [7], [8], [9] ignored the importance of resource awareness in distributed computing. They accelerated Hadoop by targeting each machine’s hardware; thus, they can be referred as standalone optimizations of Hadoop framework.

3. Design and Implementation

Handing the available resources in a cluster with heterogeneous node architectures efficiently is the key problem in sustaining high performance in cluster-based MapReduce run-time systems. In current implementations, the computational slots of a certain node, which runs various tasks, are defined statically for each node. This policy might result in situations that all of the assigned tasks underutilize their resources due to their non-preemptive scheduling method, which in turn causes major penalties in power consumption and the overall throughput. On the other hand, increasing the resource utilization to solve this shortcoming is not the proper solution as it may result in capturing a non-preemptive resource such as CPU for a long time. Therefore, all other concurrent tasks might be postponed while they have the chance to be executed on remote nodes.

We propose a new design philosophy to sustain the performance of the MapReduce programming model on heterogeneous clusters. Unlike the architecture of Hadoop and recent studies [7], [8], [9] the proposed architecture is aware of the existence of different SIMD accelerators in the operational data center. In this design, the nodes should clarify their computing facilities once they join the platform and provide the status of their facilities in certain intervals. Using this heartbeat method our platform is always aware of all busy processing elements in the cluster. We employ this processor-aware scheduling in our platform due to its benefits where more than one type of processing element is available on each node.

Similar to the original MapReduce model, the proposed architecture is designed as a two-tier framework as shown in Figure 1. As depicted, the upper layer is the operating layer implemented on the Hadoop Distributed File System (HDFS) as the underlying layer. However, unlike the Hadoop framework, in our framework HDFS is used not only to provide the input data and to store results, but also to store the intermediate results. Storing intermediate results over the distributed file system is based on a motive; intermediate results are used for synchronization and ensure integrity of the framework.

Distributed platforms such as Hadoop are formed around central regulation points. Management and coordination among other components are achieved through these central points. For example, the *JobTracker* is the central regulation point of the Hadoop run-time system responsible for creating tasks, scheduling, managing resources, and dispatching jobs to nodes. On the contrary, our framework does not contain any mandatory central point. These servers usually become crowded spaces in the MapReduce implementations. Even without considering the complexities imposed on developers by implementing and debugging such components, these servers respond to received requests slower. We have considered replacing central regulation points by a combination of synchronized shared-memory access policy and timeout policy. This substitution has been proposed not only to eliminate development complexity resulted by different implementation paradigms but also with hope to speed up the execution time of running applications.

Our proposed MapReduce framework is designed as a fully distributed parallel platform which means there is no mandatory central regulation point in our design. However, in the case of a conservative resource management each cluster may consist of a single master node.

3.1 Overall Workflow

Workflow of the proposed architecture is discussed in this section. MapReduce jobs include an address to the input file,
mapping and reduction methods, and a location to store the results. The input files are stored in a distributed file system and results are to be saved under the location of the output directory. The following describes the scenario of a MapReduce job execution since a user initiates the job until the results come out.

### 3.1.1 Job Initiation

Each node in this design philosophy acts as an endpoint enabled to receive MapReduce jobs from clients. Users are able to initiate their jobs from any arbitrary node using a component named Job Initiator, see Figure 2. The Job Initiator performs a primary estimation of job and validates the existence of input file and output location.

### 3.1.2 Task Creation

Tasks are to be created by schedulers according to input size of the running MapReduce application.

### 3.1.3 Map Phase

Map tasks are deployed gradually by the scheduler over the data center in order to be executed on proper nodes. Each map task generates and stores its intermediate results on the HDFS. The stored results are meant to act as a completion flag for the relevant map task. Once all intermediate results are generated the map phase is completed.

### 3.1.4 Shuffle & Merge

The framework processes the intermediate results which are generated in the former map phase and groups all the values of similar keys altogether. Then, a single file on the HDFS is created. The generated file consists of no more than one tuple for each key, and it includes the input to the reduce phase.

### 3.1.5 Reduce Phase

Unlike the map phase, reduce tasks are deployed at once; each task has to process a group of shuffled keys generated in the previous phase.Reducers store their output as the final result in a single file over the HDFS. These write backs are in order because HDFS does not support simultaneous write back to a single file.

### 3.1.6 Incremental Combine Phase

Incremental combine/reduce is defined as the process of reducing the intermediate results during the map phase locally on each node [15]. Intermediate results of MapReduce applications with commutative and associative properties can be combined incrementally during the map phase in order to minimize the overheads of data transfer and to cut down the spent time in reduce phase.

### 3.2 Shared Memory

Inherent complexity of JobTracker leads to a complex control flow for the main server in the Hadoop MapReduce implementation. We intend to simplify our design by eliminating equivalent crowded spaces and delegate their responsibilities among distributed components.

By eliminating the central regulation points from our architecture, a shared memory to store the status of operational clusters is vital in order to allocate resources accurately and schedule initiating jobs according to available resources. The data structure that represents the status of clusters is stored on a distributed storage and retrieval system whereas it has to be shared among all nodes. In this model, this shared-memory is represented by a table. Each tuple in this table is dedicated to a node; therefore, its attributes should be related to computational capabilities and available resources of each node. This table is used for obtaining recent status of computational facilities. For this, an extra timestamp attribute is required to store the latest time that the equivalent node has sent a heartbeat (updated its status). In addition, querying current workload of the cluster (i.e., Job Running Status), active nodes, passive/dead nodes and the resource management are possible by investigating the tuples of this table.

### 3.3 Node Trackers

Each node consists of a Node Tracker that (1) updates its status in certain time intervals and (2) allocates resources for each job, which is passed to it by the local Job Initiator. Node Trackers inquiry the existence of a master node; if there exists one, they send resource request to the master node. Otherwise, Node Trackers inquiry the Shared-Memory to find and reserve required resources by themselves. Once the resource allocation is made for each job, a data structure named capsule, which consists of the job, its estimation, and the relevant resources will be passed to the local Scheduler.

### 3.4 Resource Tracker

Albeit having a master node is not compulsory in this framework, yet the data center's policy may decide to have a conservative resource allocation. In this case, Resource Tracker is running on the master node as a server daemon. This server is responsible for handling the resource requests that are passed to the master node by remote Node Trackers.

Figure 2. Simplified workflow of proposed framework
Resource Tracker is the high-level resource manager in the cluster. It may search, update and manipulate the status whenever needed.

3.5 Executors

In general, executors are binaries that run MapReduce phases (i.e., map phase or reduce phase) on remote nodes. Other implementations of MapReduce use built-in virtual functions that need to be overridden by programmers and calling them remotely (i.e., Remote Procedure Calls). In our approach, this part of the system is implemented as a plugin component; therefore, any binary file or even source-code can be passed to our framework as an executor. The framework is capable of compiling the source-code on the destination node.

Executors are dispatched over clusters, and they can be invoked by Schedulers to perform tasks of a MapReduce phase. Users of this framework do not need to know how the data is provided to their executors or how their results are to be written on the HDFS. The executor is the key feature in our design philosophy, considering these two facts: (1) the framework is aware of all available SIMD accelerators and (2) invocation of the corresponding executor for any individual SIMD architecture is possible in our approach; this is the point where we can claim that we are proposing a high-performance heterogeneous MapReduce framework.

There is no a multi-platform programming technology yet except for OpenCL; therefore, a redundant implementation of the executors for different hardware architectures is a vital prerequisite for a heterogeneous executing platform. Meanwhile the proposed design philosophy leads us to establish a new high performance heterogeneous framework, which deviates from crowded spaces previously demonstrated as a similar concept. Potential advantages of this design policy are beyond the potentials of its predecessors. Here are two examples, (1) having characteristics of all processors in the entire operational distributed platform provides the opportunity to devise any cost-benefit aware scheduler regarding the relevant information; (2) there are many successful standalone implementations of MapReduce model, such as Phoenix for multi-core CPUs, Mars and Soren for programmable GPUs. These frameworks are based on different system architectures. Using our approach one can use all the potentials of such frameworks such as, simplified second-level parallelism and scalable performance on shared-memory systems in cluster computing without any major change in their implementations.

Phoenix++ [16] is the execution engine for our framework. This C++ edition of the MapReduce programming model has changed to suit our needs. However, the I/O interface for Phoenix MapReduce applications is substituted with our alternatives to redirect the path of input data from local files to HDFS.

3.6 Schedulers

This framework consists of multiple task schedulers each located on a separated node. Dynamic scheduling in this framework is not possible due to lack of central regulation point in our architecture. Each scheduler is responsible for handling local jobs.

The implemented scheduler is a static task scheduling system that accepts capsules from the local Node Tracker as its input. Each capsule consists of a job, estimation about the number of map tasks with respect to its input size, and a set of correspondent resources. Then, the Scheduler divides these jobs into tasks and employs a FIFO algorithm for task scheduling regarding reserved resources. The following discusses various components in the scheduler.

3.6.1 Dispatcher

Once task scheduling is done, dispatcher initiates appropriate Executors over the data center in order to perform tasks execution. Tasks are transmitted to executors gradually according to the FIFO scheduling policy.

3.6.2 Integrity & Timeout Policy

Static schedulers schedule a set of related tasks at once with the start of MapReduce applications, therefore the probability of task failure will risk the integrity of the framework. In order to resolve such shortage in the proposed framework we are considering a due time for each task. By dispatching tasks if a timeout event occurs for a certain task, scheduler investigates the HDFS and checks whether results for the task is ready or not; if not the scheduler will assign that task to another node while it may refuse to send kill signal to the primary Executor. This policy gives a second chance to the first Executor to finish its task and to write its results back to HDFS while the secondary Executor is trying to overcome its predecessor. Whenever an Executor finishes its task, it will try to write back the result. If the result exists, Executor discards its own.

3.6.3 Multiple-Issue

Considering the situations in which resources are underutilized, we proposed the concept of Executors in conjunction with resource monitoring. As stated before Executors are programs written by final programmers; therefore our approach is prone to underutilize available resources the same as prior approaches. To overcome this issue schedulers are designed to saturate a node with all possible Executors and tasks proportional to node’s resource utilization. The act of transmitting more than one task at a time to be executed by a particular Executor or initiating multiple Executors on a certain node is called multiple-issue. Different Executors are targeting different processing elements on a certain node. Preemption, releasing or underutilizing any resource is recorded immediately through Node Tracker’s status update. Therefore, the Scheduler may issue another task to its relevant worker node whenever possible.

4. Experimental Setup

This section is devoted to describe the testing platform and some specifications about the applications we have selected to evaluate our work.
4.1 Platform Setup

A heterogeneous cluster with 3 nodes, whose nodes consist of a Quad-Core, Core-i7 and a Xeon processor, is our testing platform. The first two nodes have four gigabytes of RAM while Xeon benefits from 8 gigabytes of RAM. The interconnection is a gigabyte-class Ethernet.

To evaluate our framework, we have used Executors, which are implemented as threaded programs in the Phoenix++ framework. In other words, each executor is a multi-threaded application, which is run in a distributed way by our framework. The same implementation paradigm is used for both Hadoop and Phoenix++ applications. We evaluate our design philosophy for novel multi-core CPU architectures with the Hadoop baseline. This evaluation can be easily extended to other architectures.

4.2 Applications

In this sub-section, we introduce applications implemented to test the proposed framework and to compare it against Hadoop baseline. We have selected four different applications including Word Count, Histogram, Quasi-Monte Carlo and String Matching with similar implementation and computational complexity for both Hadoop and our framework.

4.2.1 Word Count

This application calculates the exact number of each word occurrences in a given document. Each mapper processes a block of data and generates a \( \langle \text{Word},1 \rangle \) pair per word. Reducers take a set of keys and sum their corresponding values up for each individual word represented by those keys.

4.2.2 Histogram

This application provides a graphical representation of the tonal distribution in a digital image. Each mapper receives a piece of a picture then emits intermediate results in the \( \langle \text{RGBValue},1 \rangle \) format. Reducers are meant to sum all the values for each RGBValue separately.

4.2.3 Quasi-Monte Carlo

This is a method for numerical integration and solving other mathematical problems using low-discrepancy sequences. This method is used to calculate the \( \pi \) number via Halton-Sequences. Each mapper iterates over a set of instructions in order to enumerate the number of occurrences of a random event. It provides the \( \langle \text{Boolean},1 \rangle \) pairs as its intermediate results in each step of the iteration.

4.2.4 String Matching

This application is intended to search datasets for lines matching a regular expression. Mappers traverse among input data in search of a match for the regular expression and whenever they find one, they print it out to the screen or write the line down to a file; so there will not be any intermediate results or any reducer involved in this application. This application is also called Distributed Grep.

5. Results

In this section, we compare our design policy versus Hadoop framework in terms of performance. In the first step, we provide a classification for the MapReduce applications, which can provide insights to the achieved results. Then, we continue with several design issues and study their overall impact on our framework. Finally, we present the total performance of our framework.

5.1 Application Taxonomy

Rizvandi et al. [17] have shown that the behaviors of MapReduce applications are more or less related, but they would not be necessarily similar. In [17], it is demonstrated that increasing either the input size or number of nodes will result in deviation of MapReduce applications’ behavior. In this section, we provide our classification for MapReduce applications based on our observations. Our experiments show MapReduce applications fall into three main categories.

First category consists of those applications that emit many intermediate \( \langle \text{key}, \text{value} \rangle \) pairs with a high number of unique keys generated per map task (e.g. Word Count). The second category is defined for those that are generating lots of intermediate pairs but with a moderate number of unique keys (e.g. Histogram, Quasi-Monte Carlo). Finally, the third category includes applications emitting a moderate number of pairs (e.g. String Matching). We call these three categories the ungovernable, modest and sedate, respectively.

Ungovernable class may take the advantage of incremental combiners but still produces many intermediate pairs; therefore, the required time in writing and reading intermediate results, then reducing them take a considerable percentage of the whole application. On the other hand, the modest category shows a desired behavior while using the incremental combiners because there is an upper bound for generated individual keys for these kinds of applications. Incremental combiners are limiting the number of \( \langle \text{key}, \text{value} \rangle \) pairs to the upper bound of total number of unique keys for the second category, which saves a considerable amount of time in the reduction phase.

<table>
<thead>
<tr>
<th>Class</th>
<th>Key Distribution</th>
<th>Total Ratio</th>
<th>Uniqueness Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ungovernable</td>
<td>High</td>
<td>High</td>
<td></td>
</tr>
<tr>
<td>Modest</td>
<td>High</td>
<td>Limited / Relatively Low</td>
<td></td>
</tr>
<tr>
<td>Sedate</td>
<td>Medium / Low</td>
<td>---</td>
<td></td>
</tr>
</tbody>
</table>

Not all MapReduce applications need to include the reducers, nor does their reduce phase fall into associative class; these applications belong to sedate applications that produce their results in the map phase and perhaps a single more phase is required to organize the desired results. This final phase can be treated as another MapReduce job, which accepts the sedate application’s output as its input. I/O bound applications from the third category have the lowest compliance with the resultant acceleration achieved for our
proposed architecture, so the I/O bound *sedate* applications have the least scalable performance among the MapReduce applications on this framework.

5.2 Data-Awareness

Data-aware schedulers attempt to assign tasks to nodes that are storing the relevant input data. If finding such node is impossible due to the lack of available resources, schedulers will attempt to assign tasks to other nodes of the same rack. In terms of data-awareness our implementation is similar to Hadoop; meanwhile, schedulers are aware of both the rack in which the data is located and the exact node where each block of data is stored on. The impact of this policy is shown in Figure 3 compared to rack-level data-awareness of Hadoop. The contribution of CPU-time and I/O time is shown as percentiles in this figure. CPU-time is equal in the scale of time unit for both node-level and rack-level columns of each application in this figure. As shown, the key factor in the performance of applications is the I/O time, and our implementation by leveraging a data-aware policy reduces the I/O time of the Map-Reduce applications.

5.3 Concurrent Task Initiation

Considering CPUs as one of the most valuable resources in data centers, we implemented a multiple-issue approach according to reported CPU usage by each node. Figure 4 depicts the effect of multiple-issue on the scalability of Quasi-MonteCarlo application for two different executors. As shown in Figure 4, increasing the multiple-issue factor decreases the execution time. It should be mentioned that CPU utilization is a key factor in benefiting multiple issue execution. In other words, if the CPU utilization is already high, the multiple-issue will not help increase the application's performance.

5.4 Performance Analysis

This section is dedicated to describe the factors affecting the performance of our proposed framework. For this, we provide the results in two different figures because of large differences for the achieved speedups in different benchmarks.

The most influential factor in the performance of our framework is the HDFS. Word Count, String Matching and Histogram are excessive I/O-bound applications due to low performance random access in HDFS. Figure 3 shows almost 80% of String Matching application is dedicated to the I/O operations. By using HDFS as the same underlying layer, this I/O time is analogous to the Hadoop framework; therefore, we claim 3-4x speedup for String Matching and 7-12x speedup for WordCount is the final range for accelerat-
ing Hadoop framework in our experimental environment as shown in Figure 5. In this figure, some columns are labeled as remote call. These are the situations, in which our framework performs worse than normal. It happens when all input data are stored on nodes, which are remote from the Job Initiator. In these situations, all Executors are invoked remotely. As can be seen in Figure 5 and Figure 6, this phenomenon does not have a major impact on our framework’s performance.

The speedups achieved for the Histogram and Quasi-Monte Carlo applications are presented in Figure 6. As shown, the maximum speed up in Histogram is about 185 times. The Histogram application generates an intermediate pair for each byte of a digital image. An implicit time-complexity that is lost in run-time, is the time required for shuffling and combining intermediate pairs. This overhead is related to the total number of emitted intermediate pairs, which is a large number in terms of Histogram application. Incremental combiners are reducing the amount of explicit emit intermediate operations and aggregate values of certain keys locally. Therefore, less time is spent to store, group and merge intermediate pairs. Hence, we show that the computational-bound aspect of Histogram outweighs its I/O-bound nature especially in the Hadoop framework. The other application, Quasi-Monte Carlo, is a pure computational-bound application, which falls into the moderate category. Acceleration rate gained for this application is about 44 times to 60 times, compared to Hadoop baseline; see Figure 6.

The huge acceleration gap for Quasi-Monte Carlo and Histogram as two moderate applications is because there is more than a solution for implementing MapReduce applications. For example, one can implement a Quasi-Monte Carlo method in such a way that in every step of its iterations the mapper would emit two intermediate pairs. Meanwhile the mapper of another implementation may explicitly reduce these intermediate results, and eventually emits two intermediate pairs. Farivar et al. [9] used the first method to challenge the performance issues of Hadoop framework performing BlackScholes. Considering this unfair comparison, we have decided to use explicit reduction approach to be the comparative baseline. Using the first implementation, we have gained 360 times acceleration issuing a single task per node and an approximation of 720 times acceleration while using double issue policy over Hadoop baseline.

6. Related Work

Abbasi et al. [8] introduced Surena; a framework that uses Hadoop architecture as its backbone architecture. Surena accelerates Hadoop architecture by stitching a GPU-driven MapReduce framework to TaskTracker and uses it as the main task executor instead of the CPU. It achieves 4-21x speed up over Hadoop baseline. Surena used a 4-node cluster; each node was equipped with a Core-i7 860 and Nvidia GTX-570.

Farivar et al. [9] introduced MITHRA a framework which leverages GPU processing power using Nvidia CUDA language in conjunction with Hadoop Streaming to perform distributed Monte Carlo computations. This research results in 254x speedup through employing GPUs on a small 4-node cluster versus a 62-node Hadoop cluster for BlackScholes application. The huge acceleration is achieved due to several factors. First, there is no I/O involved with the BlackScholes application; second factor is in the nature of the BlackScholes application because it only generates two keys which make this application a perfect candidate for performing incremental combine. Finally the implementation is the last factor. In other words, emitting two pairs of intermediate results in each iteration is not ideal, it is better to reduce these pairs and emit only two pairs for each map task.

All related work in this era can be labeled as standalone optimizations of a well-known MapReduce framework. These kinds of researches, targeting existing MapReduce frameworks, have a high cost-benefit ratio, showed in [7] even if we gain 20x speedup over the base implementation.

7. Conclusion and Future Work

In this paper, we have presented a novel MapReduce architecture aimed to fully utilize heterogeneous processors in a MapReduce cluster in the hope of accelerating MapReduce applications. We have shown characteristics and behavior of some MapReduce applications using this architecture. In terms of throughput, we have shown there is a correlation between amount of I/O involved in MapReduce applications and total number of intermediate results with the acceleration rate that will be achieved. Preserving our experimental results, we have defined a new taxonomy for MapReduce applications in this paper.

Use of new SIMD processors in MapReduce frameworks to parallelize MapReduce applications in the second level is inevitable in data centers. Future studies in the aspects of energy consumption of novel design philosophies will reveal the cost-benefit ratio of using different types of SIMD accelerators such as, GPGPUs, IBM Cell Processors and FPGAs solely or in collaboration in large-scale clusters.

Acknowledgment

This work was supported in part by the IPM under grant number 90/80/5813.

References