A Study of Many-Core Hardware Accelerated Hadoop MapReduce

Min Luo † Toshimori Honjo ‡

Software Innovation Center, NTT Corporation
3-9-11 Midoricho, Musashino-shi, Tokyo, 180-8585 Japan
E-mail: † luo.min@lab.ntt.co.jp, ‡ honjo.toshimori@lab.ntt.co.jp

Abstract MapReduce is a widely used framework for massive data processing. It was originally designed to overcome the I/O bottleneck, and enabled us to process Bigdata with the commodity clusters systems. However, several existing work have recently shown that the emerging high speed storage and network devices are capable to remove the I/O bottleneck and made the CPU the next serious bottleneck in the MapReduce framework. In this paper, we propose hardware accelerated (HA) Hadoop MapReduce framework and implement it on a Tilera's many-core processor board to overcome the CPU bottleneck. Our proposed solution offloads the main parts of Map and Reduce procedures, including the data parsing, sorting and merging. Based on our experimental evaluations, we verify the feasibility of our proposal.

Keyword Hardware Acceleration, MapReduce, Many-core, Hadoop

1. Introduction

The explosion of information has created a new area of research for computer scientists, creatively named Big data. However, the limited HDD I/O performance is a typical bottleneck in processing the massive amount of big data. Therefore, cluster-based parallel and distributed approach is proposed and achieves cost-effective total I/O performance. Recently, there is popular cluster-based big data processing application named Apache Hadoop [1], which contains an open source implementation of Google MapReduce framework [2]. It further relieves the data moving overhead among servers by combining data storage and data computation into an integrated system, and enables massive data processing at high speed by the commodity clusters systems. However, the data moving overhead required in MapReduce framework still makes the limited HDD I/O speed in the commodity server and the limited network bandwidth among servers the typical performance bottleneck in Hadoop.

On the other hand, several kind of high speed storage devices and networks have emerged recently, such as Solid State Device (SSD) and Infiniband. The throughputs of these devices are faster than those of conventional HDD and connections by at least one order of magnitude. Our previous research [3] has shown that these devices are able to remove the previous bottlenecks in MapReduce cluster systems and verified the next bottleneck turns to be the limited CPU capability on the commodity cluster node.

In addition, to improve the calculation capability of a single node in commodity clusters, many solutions have been proposed including the compound use of GP-GPU and CPU [4], the hardware accelerators by many-core or FPGA [5-9]. However, these works focused on the use of accelerators to improve general system performance; there is no offloading efficiency study or optimization for the MapReduce framework on these accelerated proposals.

In light of the existing achievements from hardware technology and their widely use in commodity servers in near future, it is time to have a redesign of MapReduce framework and have a detailed performance study of MapReduce phases on the hardware accelerated platforms.

In this paper, we propose our hardware accelerated (H.A.) Hadoop MapReduce framework which is implemented on a Tilera's many-core processor board. In our prototype system, the main parts of Map and Reduce procedures, including data parsing, sorting and merging, are able to be offloaded. We verified the feasibility of our proposal in overcoming the CPU bottleneck through the benchmark experiments.

The remainder of the paper is organized as follows. Section 2 gives the background of MapReduce framework and its performance issue. Section 3 presents the design of our H.A. MapReduce framework, the offloading processes and implementation. Experimental benchmarks and the evaluation results are described in Section 4. Section 5 discusses the related work on H.A. MapReduce. We conclude the lessons we learned and our future work in Section 6.

2. An Introduction of MapReduce

2.1. MapReduce Framework

MapReduce is a parallel programming model proposed by Google to facilitate large scale data processing in a distributed computing environment [2].

The MapReduce's programming model contains two phases, map and reduce. As shown in Figure 1, the input data to a computing task is split into many <K, V> tuples and a map function processes there pairs to generate a set of intermediate <K', V'> pairs. The intermediate pairs with the same intermediate key are grouped together and passed to reduce function. The communication model within MapReduce is transparent to users to alleviate the development cost. MapReduce framework takes care of the parallel execution by issuing and managing the map and reduces tasks to computation nodes. It greatly relieves the complexity of designing parallel computing programs and provides efficiency for data-intensive applications [2].

There are many implementations of MapReduce model on parallel computing platforms in the last decade [1, 2, 4-9].

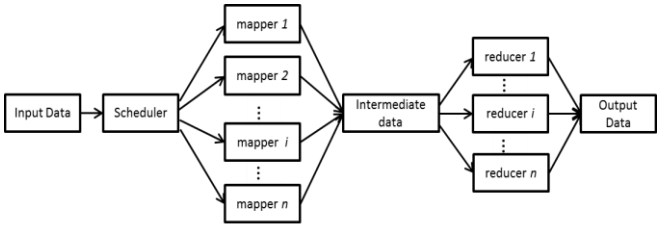

Figure 1. MapReduce Data Flow

2.2. MapReduce Execution Overview

Because Apache Hadoop [1] is the most popular and widely used open-source MapReduce implementation, we focus our discussion on Hadoop's MapReduce hereafter.

Hadoop contains a network file system, named Hadoop Distributed File System (HDFS), which is implemented as the storage layer for its MapReduce computation model. The HDFS consists of a single centralized management node, *NameNode*, which maintains all the metadata for the cluster along with multiple storage nodes; *DataNodes*, which contain all the data in large blocks size (default 64MB). The MapReduce computation model includes a single central management node, *JobTracker*, and multiple computation nodes, *TaskTrackers*. The JobTracker is responsible for initializing the specific map and reduce task for a submitted MapReduce job to a subset of the TaskTrackers in the cluster. The JobTracker also monitors the TaskTrackers' state to ensure redundancy and timely task completion. A single TaskTracker can be assigned both map and reduce tasks with the same job. Therefore, the computation and storage is able to be integrated because the DataNodes and TaskTrackers may exist on the same physical node, which are so critical to the performance of MapReduce.

There are five execution phases in Hadoop MapReduce. When a job is submitted to the JobTracker, it divides the input data into many data split and selects a number of TaskTracker and schedules them to run several MapTasks, one for each split.

In the first phase, the MapTask loads a chunk of its input split data from HDFS to memory, de-serializes and parses it to extract key-value pairs. The mapping function in each MapTask converts the original <K, V> pairs into a set of intermediated results, which are records in the form of <K', V'> pairs. The intermediate pairs are partitioned and sorted by the <K'> values, and buffered in memory. MapTask spills buffer content to disk based on a threshold mechanism in memory. It repeats the above sequence until all its input split data is processed.

In the second phase, MapTask merges all the spill files on disk and generates a single MOF (Map Output File) to disk for its split data. Note that, this *merge* process may contain a 'combine' function in its final MOF generation

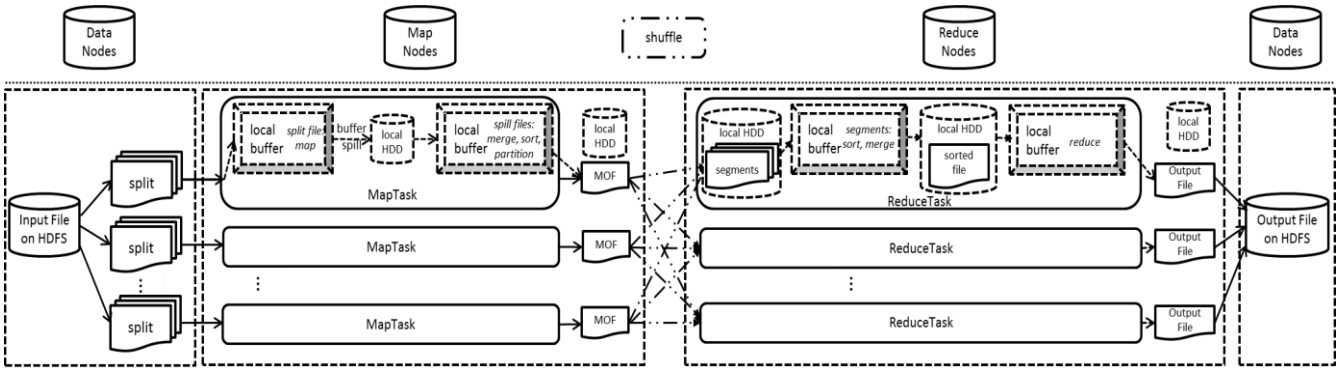

Figure 2. Hadoop MapReduce Execution Flow

to reduce the data transfer overhead during the next phase.

In the third phase, the JobTracker selects a set of TaskTrackers to run ReduceTasks when MOFs are available. Each ReduceTask fetches its intended data partition (also called segment) from the MOF across the network once it has been generated by the MapTasks. This all-map-to-all-reduce phase is also named *shuffle*.

The fourth phase starts after all the MOFs generation and shuffling is completed. The ReduceTask on each TaskTracker merges the shuffled segments into a single file, in which the data are partitioned and sorted by its key <K'>. Specifically, because the segments are already sorted and the memory size is generally insufficient for all the segments, segments only have a piece of their front-most data loaded in to memory for the *comparison*, *sorting* and *merging* processing. After writing the processed data back to HDD, the following data pieces in each segment will be loaded into memory and repeat the above sequence until all the segments data are processed.

In the last phase, the merged results of every key <K'> in the fourth phase, as <K', {$V'_{segment\_1}$, $V'_{segment\_2,\cdots}$, $V'_{segment\_n}$}>, is reduced to the final <K', V''> value pair by the ReduceTask. This process is named *reduce* phase.

2.3. Bottlenecks for MapReduce Performance

In MapReduce execution, as described in Sec.2.2, both the *map* (map computing, map intermediate data splitting, and map merging) and *reduce* (reduce merging) processes require massive disk I/O operations.

In addition, a reduce task will not start until all the MOFs are generated and all its corresponding segments in these MOFs are shuffled to the reduce node. In other word, the shuffling phase may incur considerable performance overhead in MapReduce framework, especially for the shuffle-weight tasks (e.g., sorting, inverted-indexing). Note that the shuffle moves data from the map nodes' disk rather than their memories, and through the cluster network. The affordable disk and network bandwidths of commodity clusters incur great latency and become the bottleneck to the MapReduce performance [11].

However, several kinds of high speed storage and network devices, such as Solid State Device (SSD) and Infiniband, have emerged. While conventional HDD has a transfer-rate of about 100MB/s, SATA-based SSD offers 500MB/s, and PCI Express-based SSD offers several GB/s. In addition, it is possible to install a number of these devices, so that each node can realistically offer the storage bandwidth of ~10GB/s. As for the network, 10 gigabit Ethernet is now widely used, and 40 gigabit Ethernet is available in the market. 100 Gigabit Ethernet will become common in the near future. Infiniband is another fascinating high speed network. Infiniband was widely applied to super computers. Vendors are now offering Infiniband interface cards for commodity servers. Current Infiniband network bandwidths are of the order of 56Gbit/s. In near future, these high speed devices will be found in commodity servers.

Our previous research [3] has shown that these new devices eliminate the I/O bottleneck in MapReduce and the next bottleneck in MapReduce turns to be the limited CPU capability on the commodity cluster node.

3. Proposed H.A. MapReduce framework

3.1. Hardware Acceleration in MapReduce

There are many reasons why the clock frequency increase had been becoming saturated. The unsustainable level of power consumption implied by higher clock rates is one of the obvious and stringent reasons. Advances now consist of increasing the number of cores. However, it seems unlikely that number of host CPU cores will reach levels sufficient to fully utilize the state-of-the-art storage and network devices. Some improvement in MapReduce performance is possible by increasing the number of servers, but this is inefficient in terms of power consumption neither.

In our solution, we extend the MapReduce framework with a hardware accelerator board equipped with a special processor such as FPGA or many core processors. This allows us to seamlessly increase the number of boards to keep up with performance demands and fully utilize state-of-the-art storage and network devices. In addition, as discussed in Sec. 2.2, the five main data processing phases consisted in MapReduce are isolated from each other, which logically implies the feasibility in accelerating these phases individually.

In this paper, we propose our H.A. MapReduce in which data mapping, data merging (including sorting) and data reducing processing are able to be offloaded onto the accelerators.

3.2. Prototype Implementation

We implement our prototype on a many core processor board TILEncore-Gx36, which is developed by Tilera [12]. It contains a cache-coherent mesh network of 36 tiles, where each tile houses a general purpose processor, cache

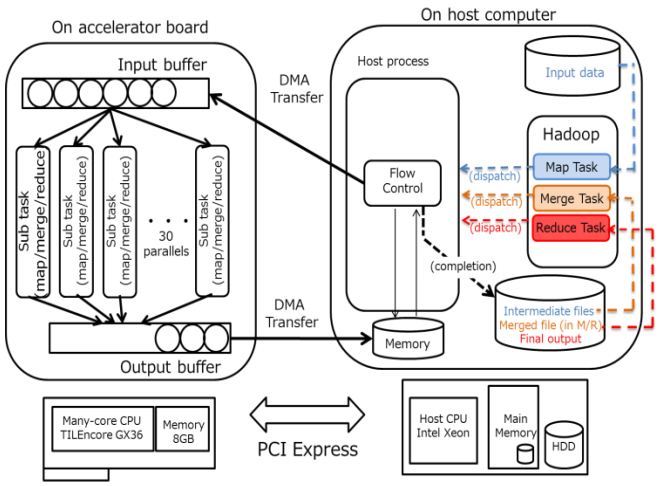

Figure 3. Overview of the Proposed H.A. MapReduce Framework

and a non-blocking router. Low power consumption is exciting feature of this processor. The power consumption of TILEncore-Gx36 is around 35W. One 1-GHz clocked CPU (36cores) and 8GB RAM was implemented on the acceleration board used in this experiment. The board has a PCI Express interface, and communicates with its host via this interface. The board runs SMP Linux so that we can easily develop several applications in user mode

Now, we give the overview of our map-offloading, reduce-offloading and merge-offloading, respectively.

At first, we give the overview of our proposed H.A. MapReduce framework. As shown in Figure 3, there are three main parts consisted in our framework: Hadoop task, host process, and on-board process.

For map acceleration, when MapReduce executed, these offloaded tasks are launched on the accelerator board node. We implemented a hook in Map of Hadoop MapReduce such that calling it dispatches the tasks to the outside host process. This host process receives the data assigned to the tasks, splits them into chunks, and transfers them to the onboard process by using DMA transfer. The on-board process uses multi-thread processing, and performs key-value pair generation and sort by key in 30 parallel streams. The intermediate data is transferred back to the host disk, and wait to be sorted into one merged file (MOF) during the merge offload phase. When all data assigned in this MapTask is processed, control is returned to the Map task, and the MapReduce job is resumed.

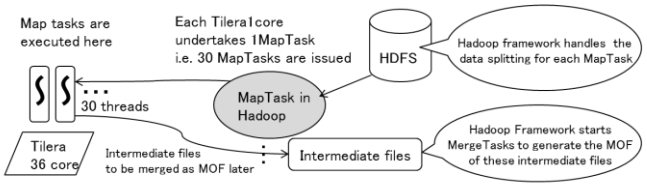

Figure 4. Map Acceleration

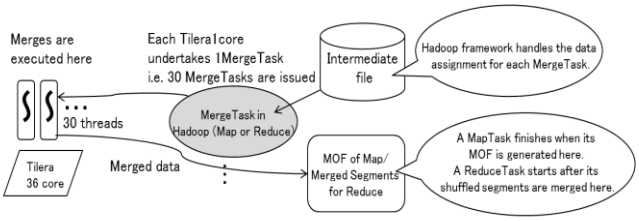

Figure 5. Merge Acceleration

In our implementation, the accelerated merge phase contains data sorting processing implicitly. It is suitable for accelerating the merge phase in both MapTask and ReduceTask. To offload the intermediate file merging phase which happens at the end of a MapTask, Map of Hadoop MapReduce dispatches the MergeTasks to the outside host process. This host process transforms the data in the intermediate file assigned to each MergeTask into (key, value-pointer) format so as to reduce transfer amount and memory usage on Tilera board. Multiple on-board processes perform key sorting and value combining (if the user designed) in parallel for one MergeTask, and output the merged file into host disk by using DMA transfer. Note that the returned merged file only contains (keys & value-pointers), the values are filled into the merge file on the host node. When all the intermediate files of a MapTask are processed, the next MapReduce job is resumed. To offload the merge phase which happens at the beginning of a ReduceTask, Reduce of Hadoop MapReduce dispatches the MergeTasks to the acceleration board in the similar way above. Note that the data to be transferred is the segments that are shuffled to current ReduceTask's host node. Because the records in these segments are already sorted, we propose and implement a multi-phase merging strategy to make the merging acceleration of Reduce-Merge more efficient.

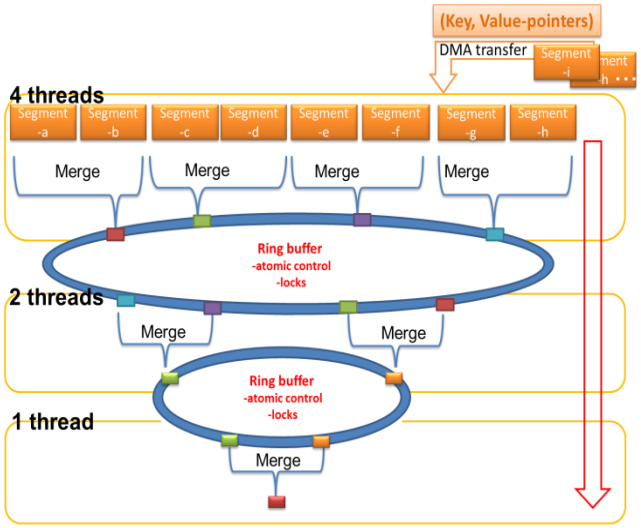

Figure 6. Multi-phase Merge acceleration

By using this multi-phase merge strategy, as shown in Figure 6, segments of a ReduceTask are sequentially loaded onto Tilera board. Every two loaded segments will form a pair and be merged by a thread (core) in the highest phase. The merged output data of these threads are stored in a ring buffer. Threads of the lower phase will pull and merge the upper phase output, and store their own merged output in another ring buffer while releasing the memory space of the upper ring buffer. Thus these multiple phases form a pipeline so that all the Tilera cores are running in parallel to merge the segments. Note that a maximum of [n/2] threads can be issued in the first phase of our implementation (n is the core number on Tilera board).

In addition, we propose and implement an index structure for the final output file in the multi-phase merge process. This index is useful in the *reduce* phase offloading, because the amount of data to be reduced in each core could be assigned as equal as possible for better data skew balancing among the offloaded reduce tasks.

Because of the paper length limitation, we only provide the indexed output file image below in Figure 7

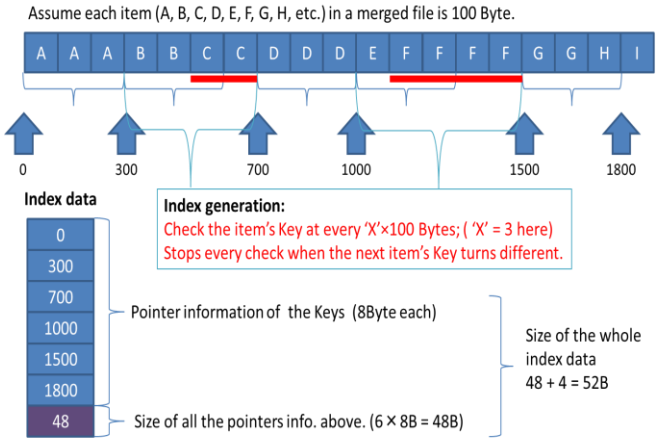

Figure 7. An Image of the Merged File Index

For reduce acceleration, the offloaded reduce tasks are launched on the accelerator board node. The host process of ReduceTasks reads the assigned merged file, splits them into chunks, and transfers them to the onboard process by using DMA transfer. Note that, index information of the merged file is used, such that items of the same key are not split into different chunks and chunk sizes are as equal as possible. The on-board process uses multi-thread processing, and performs key-based value reduction in 30 parallel streams for these chunks. The intermediate reduced data is transferred back to the host disk, and forms one final output file at the host node. Figure 8 shows the image of our reduce acceleration flow.

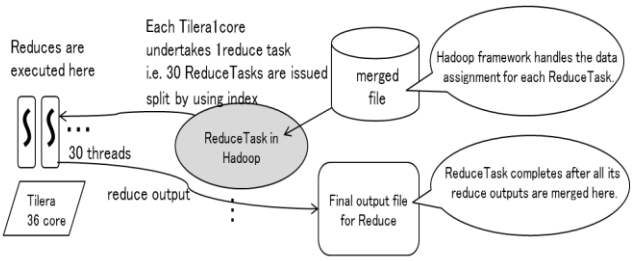

Figure 8. Reduce Acceleration

4. Experimental Evaluation

In this section, we will examine the effectiveness of our H.A. MapReduce. We first demonstrate the multi-phase merge efficiency, with "Terasort" benchmark. Then, we provide a performance comparison between H.A. MapReduce and the original Hadoop MapReduce, whose tasks are not accelerated but only run on the hosting server, with "Grep" and "Wordcount" applications.

Our experimental environment information is list below.

Table 1. Experimental Environment Specification

Host Server	CPU	Intel E5-1660 (3.3GHz, 6-cores, HT)
	RAM	128GB (DDR3-1600, 16GBx8)
	Storage	Intel SSD910 (PCIe 800GB)
Tilera Board	CPU	TILEncore-Gx36(1.2GHz,30cores)
	RAM	16GB (DDR3-1333)

4.1. Evaluation of Multi-phase Merge

In this evaluation, we use the "Terasort" benchmark scripts and dataset. By executing the TeraGen script with Hadoop MapReduce, we generate 30GB text data first. Then, we execute the "Terasort" benchmark script on this dataset with two types of MapReduce strategies. The first type is called "single-merge" MapReduce, it offloads *reduce-merge* tasks to the Tilera board with the original Hadoop's "singe-phase merge" solution. The second is called "multi-phase merge", which offloads *reduce-merge* task with our proposed "multi-phase merge" solution in Section 3.3. The offloaded *reduce-merge* is defined as the fourth phase that described in Section 2.2. Note that both the offloaded *reduce-merge* tasks and the offloading hardware environment are totally same in both executions.

Result in Figure 9 demonstrates the significance of our proposed "multi-phase merge" solution. As it shows, the proposed solution completes the reduce-merge tasks with nearly 2.5 times less time than original hadoop's "single merge" solution. We will further demonstrate the overall MapReduce performance achievement in the next section.

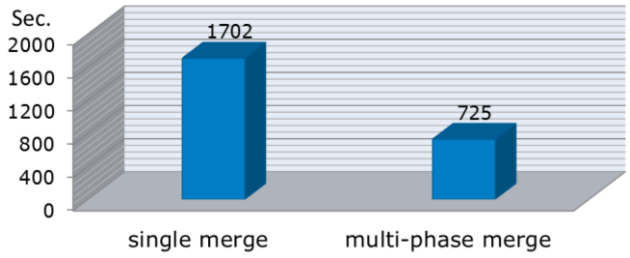

Figure 9. multi-phase merge efficiency

4.2. Overall Evaluation of Proposed MapReduce

In this evaluation, we provide an overall examination of the proposed H.A. MapReduce performance and compare it with that of original Hadoop's MapReduce.

Now we describe the detail of our experimental settings. At first, the original Hadoop on host PC initializes 30 map tasks. For the H.A. MapReduce evaluation, all these tasks are offloaded onto Tilera board and being processed by 30 cores (1.2*30GHz) there at the same time in parallel. While for the original Hadoop's MapReduce evaluation, we use the hosting PC's resources, 12 hyper-threads (12*3.3GHz), to complete the whole *map* workload. Note that, only 12 tasks are online at the same time and each task is designated to one thread. Our reason for this setting is to reduce the catch switching and resource competition overhead within a single thread if multiple tasks are being processed there at the same time. The *shuffle* and *merge* phases are not offloaded here, they are completed by the hosting PC in both evaluations. At last, we offload the *reduce* phase in offloaded MapReduce.

We carry out above processes for three types of evaluations. The first two are "Grep" applications; we examine both *Fuzzy* and *Exact* match performance here. To fairly compare the two methods, we did not use Grep sample implementation included in the original Hadoop distribution. We implemented a very simple Grep program using the java standard tokenizer to split words, and regex to check the regular expression. We used a 50GB randomly generated dictionary, and searched for words that matched the regular expression "a*b*c*1" or the exact string of "aabbcc1" in *Fuzzy* and *Exact* match, respectively. The third evaluation is a "Wordcount" application. We use the same method above to generate 10GB dictionary dataset, and we count the number of occurrences of each word. Note that the combiner in Map Task is enabled.

Figure 10 shows our evaluation result. For both "Grep" applications, our proposed solution achieves almost the same execution time (81 & 75 seconds, respectively); because it has fully utilized the I/O bandwidth of our SSD 775MB/s when loading the dataset from hotsing PC to Tilera board. For the "Wordcount" applications, the offloaded version completes in 135 seconds, which is 3.4 times faster than original MapReduce.

4.3. Efficiency Study

The results of above experiment show the feasibility of our proposal. In this experiment, we used only one acceleration board which implemented one CPU with has 36 cores. We can seamlessly increase the performance by increasing the number of boards depending on the performance needed.

Note that the computation capability in hosting PC and the many-core Tilera board are almost same, so that our competition results should be also comparable. However, hardware accelerated version overwhelmed the original version. For example, although the clock speed of Tilera core is lower, its single *map* task completion time (on average) is 3-5 times faster than that of hosting PC. This must be due to the overhead of Java. If a C version of MapReduce is implemented, we can achieve performance comparable to the hardware accelerated results given above.

We also note that the *merge* phase of offloaded version in "Wordcount" application takes much longer time than original version. This is because our current reduce offloading requires the MOF (that generated in reduce-merge phase) to be reduced should be located on disk. This generates disk I/O overhead at the end of reduce-merge phase. We will improve the merge part implementation as one of our future work.

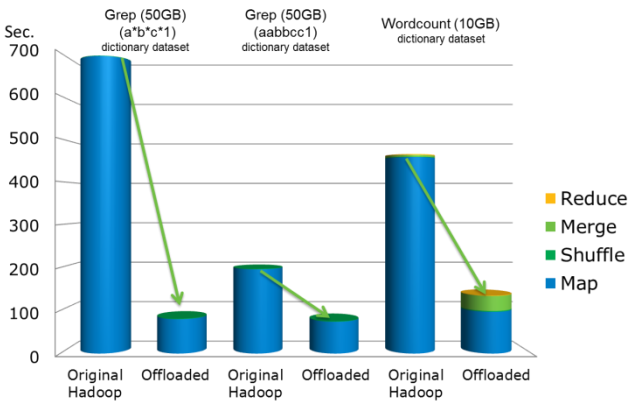

Figure 10. Offloaded MapReduce Efficiency

5. Related work on Hardware Acceleration

There are some solutions proposed to improve the data processing capability of single node in commodity clusters by using some specialized hardware.

Phoenix [5] implements a MapReduce model on a share-memory multi-core system to explore its parallelism. However, its memory and I/O usage of one task may severely affect others and this problem becomes critical when thread number increases for processing big data. Mars [4] accelerates MapReduce with a general GPU platform and achieved 1.5-16X higher speed than Phoenix. Although GPUs have an order of magnitude higher computation power and memory bandwidth compared with CPUs, it is not suitable for the data-intensive tasks such as *word-count*, and it is hard to program due to their special-purpose architecture design. In FPMR framework [6], dedicated processors are designed for different applications in which map and reduce operations are done by mappers and reducers on FPGA. The dynamic on-chip scheduling and efficient data control hide the task control, communication, and data synchronization away from designers. However, because the task scheduling and data dispatching are hard coded on chip, there is no reconfiguration ability in FPMR framework. [7] also proposes a FPGA based acceleration framework. It studies the scalability of this framework by increasing both FPGA boards and servers. However, it does not support or study the offloading of merge and sort processes. The H.A. frameworks proposed in [8, 9] enable MR framework to run on hybrid clusters so as to exploit the capabilities of heterogeneous hardware. Their authors provide a scheduling policy for resource locating based on job progress. They also show the communication and synchronization overhead in data intensive applications may still hide the benefit of hardware accelerators, however, they used commodity network in their experiments.

These previous works focus on using accelerators to improve commodity system performance and the MapReduce framework is deployed on their systems straightforwardly only to verify the feasibility of their proposal. No effort is paid in these works in studying or optimization MapReduce on the H.A. infrastructure.

6. Future Work

In our future work, we will support multiple Tilera boards in our framework for scalable and elastic offloading solution. And we will also study the FPGA board efficiency in improving the offloading performance, especially for the *merge* tasks, in our framework. We also feel interested in accelerating Hive applications by using our H.A. MapReduce proposal.

Reference

[1] Apache Hadoop (2014.2). http://hadoop.apache.org

[2] J. Dean and S. Ghemawat. “MapReduce: Simplified data processing on large clusters”, In 6th Symposium on Operating System Design and Implementation (OSDI '04), 2004.

[3] T. Honjo and K. Oikawa, “Hardware acceleration of Hadoop MapReduce”, IEEE Intl. Conf. on Big Data (IEEE BigData '13), 2013.

[4] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. “Mars: a MapReduce framework on graphics processors”, In Proc. of the 17th intl. Conf. on Parallel architectures and compilation techniques (PACT '08), 2008.

[5] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, “Evaluating MapReduce for Multi-core and Multiprocessor Systems”, In Proc. of the 13th Intel. Symposium on High Performance Computer Architecture (HPCA '07), 2007

[6] S. Yi, B.Wang, J. Yan, Y. Wang, N. Xu, H. Yang: “FPMR: MapReduce framework on FPGA”, In Proc. of the 18th Annual ACM/SIGDA Intl. Symposium on Field Programmable Gate Arrays, pp. 93–102, 2010.

[7] D. Yin, G. Li, K. Huang, “Scalable MapReduce Framework on FPGA Accelerated Commodity Hardware”, NEW2AN/ruSMART’12, LNCS 7469, 280-294, 2012

[8] Y. Becerra, V. Beltran, D. Carrera, M. Gonzalez, J. Torres, and E. Ayguade, “Speeding Up Distributed MapReduce Applications Using Hardware Accelerators”, in Proc. of the Intel. Conf. on Parallel Processing (ICPP '09). 2009.

[9] J. Polo, D. Carrera, Y. Becerra, V. Beltran, J. Torres, and E. Ayguade, “Performance Management of Accelerated MapReduce Workloads in Heterogeneous Clusters”, in Proc. of the Intel. Conf. on Parallel Processing (ICPP '10), 2010.

[10] F. Ahmad, S. Lee, M. Thottethodi, T.N. Vijaykumar, “PUMA: Purdue MapReduce Benchmarks Suite”, Purdue Technical Report TR-ECE-12-11, 2012.

[11] F. Ahmad, S. Lee, M. Thottethodi, T. N. Vijaykumar “MapReduce with communication overlap (MaRCO)”, J. Parallel Distrib. Comput. 73, 5, 608-620, 2013.

[12] Tilera Corporation: TILE-Gx8036 Processor Specification Brief (online), available from <http://www.tilera.com/sites/default/files/productbriefs/TILEGx8036_PB033-02_web.pdf>.