2nd IEEE International Conference on Cloud Computing Technology and Science

Hybrid Map Task Scheduling for GPU-based Heterogeneous Clusters

Koichi Shirahata*, Hitoshi Sato* and Satoshi Matsuoka*†‡

*Tokyo Institute of Technology

†CREST, Japan Science and Technology Agency

‡National Institute of informatics

***Abstract*—MapReduce is a programming model that enables efficient massive data processing in large-scale computing environments such as supercomputers and clouds. Such large-scale computers employ GPUs to enjoy its good peak performance and high memory bandwidth. Since the performace of each job is depending on running application characteristics and underlying computing environments, scheduling MapReduce tasks onto CPU cores and GPU devices for efficient execution is difficult. To address this problem, we have proposed a hybrid scheduling technique for GPU-based computer clusters, which minimizes the execution time of a submitted job using dynamic profiles of Map tasks running on CPU cores and GPU devices. We have implemented a prototype of our proposed scheduling technique by extending MapReduce framework, Hadoop. We have conducted some experiments for this prototype by using a K-means application as a benchmark on a supercomputer. The results show that the proposed technique achieves 1.93 times faster than the Hadoop original scheduling algorithm at 64 nodes (1024 CPU cores and 128 GPU devices). The results also indicate that the performance of map tasks, including both CPU and GPU tasks, is significantly affected by the overhead of map task invocation in the Hadoop framework.**

***Index Terms*—Large-scale data processing; MapReduce; GPGPU; Job Scheduling;**

I. Introduction

Nowadays, the data generated by human activities is rapidly increasing. For example, the IDC white paper has reported that 1800 exabytes of information will be created in 2011 [1]. HPC applications, such as biology, astronomy, and high-energy physics, etc., also require to process large amounts of data, which is typically generated by scientific experiments and observations, in supercomputers and cloud data centers for computational analyses. MapReduce [2] is a programming model for efficient scalable massive data processing in a large computer cluster. Recent computer clusters employ modern graphic processing units (GPUs) with compute nodes with general purpose CPUs [3], since GPUs provide high peak performance and memory bandwidth for applications with simple computation patterns, while CPUs offer flexibility and generality over wide-ranging classes of applications.

GPU-based heterogeneous computer clusters can be used for a MapReduce execution environment; however, scheduling map and reduce tasks onto CPU cores and GPU devices for efficient execution depends on running task characteristics and underlying computing environments. For example, tasks which contain data parallelism may suit for GPU execution, while tasks which contain many branches or synchronizations may not. The performance of task execution may also vary according to the resource configurations: the number of CPU cores and GPU devices, memory size and bandwidth, local I/O performance to secondary storage systems. Ad hoc scheduling strategies, such as allocating tasks to only GPU devices, or to idle CPU cores and GPU devices in a FIFO manner, may not achieve optimal job throughput and may cause inefficient resource utilization and energy consumption.

To address this problem, we propose a hybrid map task scheduling technique for GPU-based heterogeneous computer clusters. When a client submits a MapReduce job whose tasks can run on both CPU cores and GPU devices, a master job scheduler assigns the map tasks onto CPU cores and GPU devices in order to minimize the overall MapReduce job execution time by using profiles collected from dynamic monitoring of map task's behavior. (We currently employ the elapsed time of map tasks.) Worker nodes execute the scheduled map tasks on CPU cores or GPU devices. We implemented this scheduling technique to an existing widely used MapReduce system, Hadoop [4].

We evaluated the proposed technique on our GPU-based supercomputer, TSUBAME, by using a K-means cluster analysis application. The results show that the proposed technique achieves 1.93 times faster than the Hadoop original scheduling at 64 nodes with 1024 CPU cores and 128 GPUs. The results also indicate that the performance of map task execution, including both CPU tasks and GPU tasks, is significantly affected by the overhead of map task invocation from the Hadoop framework.

II. Background

This section introduces the brief overviews of MapReduce and GPGPU as the bases of our work.

A. MapReduce

MapReduce is a programming model for large-scale data processing in a computer cluster. By applying an uniform

978-0-7695-4302-4/10 $26.00 © 2010 IEEE
DOI 10.1109/CloudCom.2010.55

IEEE computer society

operation to distributed key-value pairs, MapReduce utilizes data access locality and achieves scalable data processing. The process consists of three phases: *map*, *shuffle*, and *reduce*. The map phase generates intermediate key-value pairs from initial key-value pairs. Then, the shuffle phase generates a list of values for the same key. Finally, the reduce phase aggregates values in a list for a key and generates the final output key-value pairs. Application users only have to write map and reduce functions, while the system parallelizes the application transparently; therefore, MapReduce conceals underlying computing resource complexities and provides high productivity for data-intensive applications such as data analysis and machine learning, etc.

Several systems, such as Google MapReduce [2], Hadoop [4], Phoenix [5], and Mars [6], implements the MapReduce model. Google MapReduce is the original implementation, which includes a distributed file system and a MapReduce framework itself. Hadoop is well-known for an open-source-based Java software framework that implements a clone of the Google MapReduce system. Phoenix provides programming APIs and runtime systems for shared memory systems. Mars is a generic framework for GPUs, which enables application users to implement data- and computation-intensive tasks correctly, efficiently, and easily on GPUs. Our study focuses on Hadoop, since Hadoop is widely used in academic and industrial domains.

Hadoop consists of several components, mainly Hadoop Distributed File System (HDFS) and Hadoop MapReduce. HDFS is a distributed file system that employs a master-worker model. A master node, called NameNode, manages information related to file system namespace, such as directory tree and meta data of stored files, etc., while worker nodes, called DataNodes, accommodate actual file data. A single file is divided into several chunks (typically 64 MB). Then, the divided chunks are stored across DataNodes and replicated to different DataNodes (typically 3 times). On the other hand, Hadoop MapReduce provides a MapReduce execution environment on top of HDFS, whose environment also employs master-worker model, JobTracker as a master node and TaskTrackers as worker nodes. JobTracker is responsible to manage submitted jobs, while TaskTrackers execute actual map and reduce tasks in the submitted jobs. Thanks to the localized data accesses provided by HDFS, Hadoop achieves scalable data processing for large computer clusters.

B. GPGPU

GPGPU (General-purpose computing on GPU) [3] is a technique to apply commodity GPUs, which is typically used for running specific graphic operations, to general purpose computing in applications traditionally handled by CPUs. Recent advancement of GPU, in architecture by adding programmable stages and higher precision arithmetic to the rendering pipeline and in programmability by providing integrated development environments embodied as CUDA [7] and OpenCL [8], enables application programmers to use stream processing on non-graphics data.

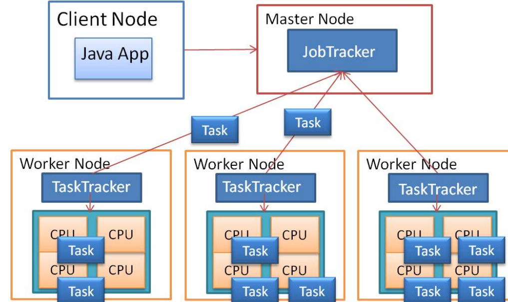

Fig. 1. Overview of MapReduce job scheduling in Hadoop

GPU suits for parallel computing, since the architecture employs SIMD-based processing; therefore, GPU achieves much higher perk performance and memory bandwidth than CPU by using tens of thousands of fine-grain threads. However, computation in GPUs requires to transfer data from main memory in a host compute node to global memory in a GPU device and introduces significant overheads to applications running on GPU devices. Moreover, applications with many branches and synchronizations may cause inefficient execution on GPU devices, whereas CPU suits general purpose computation and plays a main role in data transfer to GPU devices and in post- and pre-processing in a host compute node. Note that GPU cannot work as a stand-alone system.

Several programming environments, such as CUDA [7] and OpenCL [8], etc, focus on GPU computing. CUDA is a widely-used programming environment, which provides C- and C++-based programming environment for NVIDIA CPUs with high level abstraction in a SIMD-style. CUDA is applied to various applications such as chemistry, sparse matrix, sorting, searching, and physical modeling, etc. in order to accelerate their computing performance.

III. Hybrid Map Task Scheduling

We propose a technique to accelerate map tasks by using hybrid scheduling onto CPU cores and GPU devices in a GPU-based heterogeneous computer cluster. Figure 1 shows how MapReduce job scheduling works in the Hadoop framework. When a MapReduce job, typically written in Java, is submitted to the system, the JobTracker schedules map and reduce tasks in the MapReduce job to idle CPU slots on the TaskTrackers, then the tasks run on the assigned CPU slots. In order to introduce the hybrid map task scheduling to the Hadoop framework, we have to consider the following problems:

- How to invoke C- and C++-based CUDA codes from Hadoop in order to execute map tasks on GPU devices.
- How to schedule map tasks onto CPU cores and GPU devices in order to minimize job execution time

This section presents the invocation technique of CUDA code and the hybrid map task scheduling algorithm in the Hadoop framework.

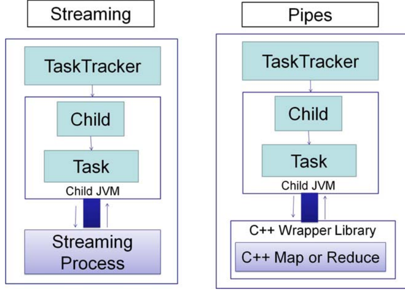

Fig. 2. Hadoop Streaming (left) and Hadoop Pipes (right)

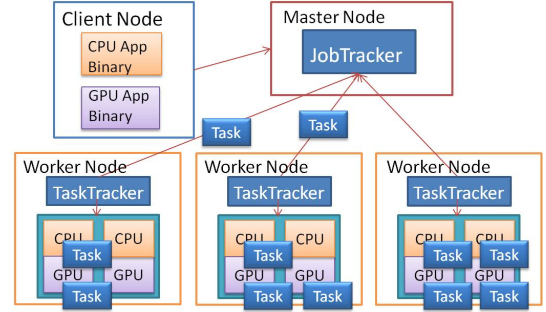

Fig. 3. Overview of our proposed hybrid scheduling technique

A. Comparison of CUDA code invocation techniques from Hadoop

Our proposed technique requires to execute a map task on both CPU cores and GPU devices. To achieve this hybrid execution feature, we investigate CUDA code invocation techniques from Hadoop. The Hadoop framework, including HDFS and Hadoop MapReduce, is currently implemented in Java. Thus, user applications are also typically implemented in Java by using the Hadoop libraries. On the other hand, CUDA provides a C- and C++-based programming environment. Therefore, we have to translate a Java program to a CUDA code. We have several solutions to invoke CUDA codes from the Hadoop framework: Hadoop Streaming, Hadoop Pipes, and Java Native Interface (JNI).

- Hadoop Streaming
  Hadoop Streaming (Figure 2 left) is an API that allows application users to write their map and reduce functions in languages other than Java. Using Unix standard streams as the interface between Hadoop and user's program, application users can use any languages with standard I/O operations to implement their MapReduce programs.
- Hadoop Pipes
  Hadoop Pipes (Figure 2 right) is a C++ interface to Hadoop MapReduce. Unlike Streaming, Pipes uses sockets as the channel over which the TaskTracker communicates with the process running the C++-based map and reduce functions without using JNI.
- JNI
  JNI is a native programming interface that allows Java code running in a Java Virtual Machine (JVM) to invoke or to be invoked by applications and libraries written in other programming languages such as C, C++, and assembly. Using JNI as a code translator between Java and other languages, application users can enjoy various benefits: invoking platform specific features and program libraries which the standard Java class library does not support. Several existing libraries [9], [10] wraps CUDA code invocation by using JNI.

Hadoop Streaming supports wide-ranging map and reduce programs written in any languages with the standard I/O; however, application users have to write parser codes of the standard I/O manually, which may introduce complex programmability. By contrast, Hadoop Pipes does not require to parse data via the standard I/O, since the runtime can communicate with key-value abstractions by using the Hadoop Pipes library. JNI may provide transparency to map and reduce programs by encapsulating the differences between Java and CUDA codes; however, such native method invocations usually introduce significant overheads, since accesses to Java data structures in JNI, such as methods and fields, have to invoke functions via JNI's interfaces indirectly. In addition, JNI-based applications require platform-specific libraries, which may ruin pure Java portability.

As discussed above, we have several techniques to invoke CUDA codes from the Hadoop framework. Considering that CUDA extends C and C++ programs and Hadoop Pipes provides C++-based interfaces for user MapReduce applications, we use Hadoop Pipes for our proposed technique.

B. Hybrid Map Task Execution on CPU cores and GPU devices

We assume that application binary programs can run on both CPU cores and GPU devices in Hadoop Pipes. Figure 3 shows the overview of our hybrid scheduling technique. When a client submits a MapReduce job to the JobTracker by specifying CPU and GPU binaries, the JobTracker firstly assigns the map tasks to idle slots and executes the assigned tasks on the available slots. while TaskTrackers monitor the behavior of each running map task: the elapsed time of a map task and the used CPU cores and GPU devices, etc. Using these available profiles, the JobTracker decides which map tasks run on which devices (CPU cores and GPU devices) and binds the decided map tasks to the corresponding application binary programs, i.e., CPU map tasks use a CPU binary program, and GPU map tasks use a GPU binary program. Then, TaskTrackers execute given map tasks by using the specified binary program. After the map phase, reduce tasks are invoked based on the map results. Since both CPU and GPU map tasks have the same output format, map tasks running on GPU devices never influence the reduce phase.

C. Scheduling Strategy

The basic idea of the proposed scheduling algorithm is to minimize the elapsed time of a submitted MapReduce job. To achieve this algorithm, we allocate map tasks based on the performance ratio between CPU and GPU map task executions. We collect profiles of map tasks and calculate the average elapsed time of the finished map tasks for CPU cores and GPU devices respectively. Applying the collected profiles of map tasks dynamically, we can adapt our hybrid scheduling on a heterogeneous environment. Since the performance of map tasks depends on running application characteristics and underlying computing environments, we execute both CPU and GPU map tasks simultaneously in the initial phase in order to collect profiles as many as possible and to allocate successive tasks to fast processors.

D. Scheduling Algorithm

In order to minimize the elapsed time of a submitted MapReduce job in a GPU-based heterogeneous computing environment, we introduce a performance model of map task executions. Let N be the number of map tasks, n be the number of CPU cores, and m be the number of GPU devices. We denote the performance ratio of a GPU map task execution to a CPU map task execution as α, which we call the acceleration factor, and the mean elapsed time of the finished map tasks on GPU devices as t. We assume that each map task uses a single available slot exclusively, i.e., a single CPU core or a single GPU device. Based on these parameters, the acceleration factor, α, is described as follows:

$$\alpha = \frac{mean\ map\ task\ execution\ time\ on\ CPU\ cores}{mean\ map\ task\ execution\ time\ on\ GPU\ devices}$$

Note that, by using acceleration factor, α, we denote the mean elapsed time of the finished map task on CPU cores as $\alpha \cdot t$. Here, let x be the number of map tasks running on CPU cores, and y be the number of map tasks running on GPU devices. We can model the estimated total map task execution time (including both CPU and GPU map tasks) as follows:

$$f(x, y) = \max\{\frac{x}{n} \cdot \alpha \cdot t, \frac{y}{m} \cdot t\}$$

Using the above parameters, we determine the number of CPU map tasks, x, and the number of GPU map tasks, y, to minimize the overall elapsed time of a MapReduce job as follows:

Minimize

$$f(x, y) \tag{1}$$

Subject to

$$f(x, y) = \max\{\frac{x}{n} \cdot \alpha \cdot t, \frac{y}{m} \cdot t\} \tag{2}$$

$$x + y = N \tag{3}$$

$$x, y \geq 0 \tag{4}$$

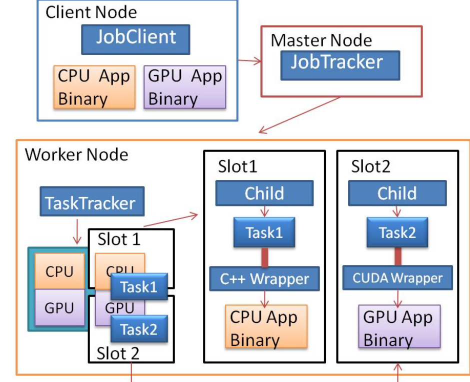

Fig. 4. Overview of our prototype implementation

The objective function (1) minimizes the elapsed time of a submitted MapReduce task, i.e., the elapsed time of x CPU map tasks and y GPU map tasks as described in (2). We introduce additional constraints: (3) and (4). Constraint (3) states that the sum of CPU and GPU map tasks is constant and is equal to the total map tasks. Constraint (4) avoids the number of CPU and GPU map tasks to be negative. Note that our current hybrid scheduling algorithm does not consider reduce tasks, since reduce tasks typically run on all available slots at the same time; therefore, the hybrid scheduling cannot optimize the resources.

Based on the determined number of map tasks, x and y, the scheduler allocates CPU and GPU map tasks to the corresponding slots. The scheduler applies this algorithm by using the collected profiles in the heartbeat messages sent from worker nodes and holds the updated results.

IV. Implementation

We implemented our proposed mechanism in Hadoop version 0.20.1 as a prototype. Figure 4 shows the overview of our prototype implementation. The rest of this section describes the detailed implementation of Hadoop and our extensions for the proposed hybrid scheduling.

A. CUDA code invocation from Hadoop Pipes

As discussed in previous section, We use Hadoop Pipes for implementing our prototype. In the Hadoop framework, the MapReduce job invocation process consists of several steps. First, a user program launches a job as a Job instance and invokes a JobClient instance. Then, the JobClient instance submits the job to the JobTracker. After submitting the job, the JobTracker allocates map and reduce tasks to idle slots on TaskTrackers. Each TaskTracker launches child JVM processes and runs the allocated tasks on the assigned slots.

In the case of Hadoop Pipes, each C++ wrapper process, running map and reduce tasks, establishes a permanent socket connection to the child JVM process by passing a port number of the given execution environment. During the task execution, the JVM process sends input key-value pairs to the C++

wrapper process. Then, the wrapper process starts user defined map and reduce functions and passes the output result key-value pairs to the JVM process after the computation. From the TaskTracker's point of view, this behavior can be seen as if the TaskTracker child process executes the map or reduce codes by itself.

Our prototype requires to invoke a map task on both CPU cores and GPU devices; therefore, we prepare a C++-based CPU map task code and a CUDA-based GPU map task code, both of which the JVM process can communicate with by passing key-value pairs via the standard I/O. By specifying both CPU and GPU binary programs at the program invocation from command line, users can run map and reduce tasks on both CPU cores and GPU devices, while the scheduler controls the task execution by binding specific tasks to corresponding binary programs: CPU map tasks to a CPU binary program and GPU map tasks to a GPU binary program.

In order to implement the above mechanisms, we add a task management feature in Hadoop, whose feature consists of two steps: specifying CPU and GPU map task binary programs and allocating CPU and GPU map tasks. First, users on a client submit a MapReduce job by specifying both CPU and GPU binary programs. Then, the client sends the submitted job, including CPU and GPU task information, to the JobTracker. Each TaskTracker manages their idle slots, i.e., idle CPU cores and GPU devices, and periodically sends a heartbeat message to the JobTracker in order to ask tasks to run on the TaskTracker. Meanwhile, the TaskTracker notices the idle slots, i.e., whether CPU cores or GPU devices are idle or not, to the JobTracker. Each TaskTracker also collects application and resource profiles: the elapsed time of running map tasks, the number of CPU cores and GPU devices, and the device number of GPUs that map tasks run on, etc. We include these profiles in a heartbeat message. Then, when the JobTracker receives the heartbeat message, the scheduler decides the map task allocation by using our proposed algorithm based on the information collected from the heartbeat messages sent from TaskTrackers, including the information such as which slots the previous map tasks used, how long the previous tasks took for computation, etc. The decided task allocation is sent to the TaskTracker with task information. Then, the TaskTracker invokes the corresponding map task binary program based on the information.

If the TaskTracker node has multiple GPU devices, we have to manage which map tasks run on which GPU devices. since many current operating systems do not support resource management features for GPU devices; therefore, resource contentions to specific GPU devices may occur. This situation can be avoided by introducing the GPU device management to the TaskTrackers, i.e., which map tasks run on which GPU devices on the TaskTracker node. Hence, we set the GPU device number at the invocation of a GPU binary program. This GPU device usage information is also sent to the JobTracker in the heartbeat message. Thus, the JobTracker can recognize the GPU device usage of all TaskTracker nodes. Based on these profiles, the JobTracker allocates map tasks to idle GPU devices in a round-robin manner.

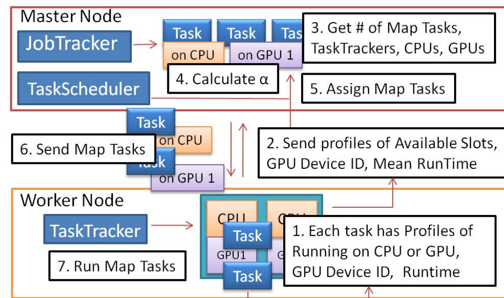

Fig. 5. Map task scheduling workflow of our prototype implementation

B. Map Task Scheduling

We implemented the proposed hybrid scheduling technique to the JobTracker and the TaskTrackers. Figure 5 shows the map task scheduling workflow of our prototype implementation. As described in the previous section, each TaskTracker fetches a new map task to the JobTracker every time the slots on the TaskTracker are idle. Then, the JobTracker allocates CPU or GPU map tasks to the TaskTrackers according to the results of the proposed scheduling algorithm. In order to allocate these map tasks efficiently, the scheduler calculates the mean elapsed time of the finished CPU and GPU map tasks and the acceleration factor of map tasks. Since the acceleration factor is unknown at the start time, the scheduler allocates map tasks to the available CPU cores and GPU devices. When the JobTracker receives a heartbeat message, which includes a query of map task allocations to available slots on the TaskTracker, the scheduler calculates the mean elapsed time of CPU and GPU map tasks and the acceleration factor. Based on the results, the scheduler allocates the remaining map tasks to idle slots by using the proposed algorithm. Here, the scheduler checks the progress of map tasks and the availability of CPU cores and GPU devices on the TaskTrackers. If the TaskTrackers do not have available CPU or GPU slots, the scheduler waits to allocate map tasks to the TaskTrackers until the slots become available. We minimize the idle time by reducing the situation that map tasks running on fast processors wait for tasks running on slow processors to finish.

V. Experiments

In order to evaluate the validity of our proposed hybrid map task scheduling algorithm, we conduct performance studies by using a cluster analysis application. This section presents the results of our experimental studies based on our prototype.

A. Target Application

We use a cluster analysis application, K-means, in this experiment, since K-means is a common technique for data analysis and a typical MapReduce application. The K-means application works as follows:

TABLE I
SPECIFICATION OF A SINGLE COMPUTE NODE

CPU	Dual Core AMD Opteron 880 (2.4 GHz)
# of cores	16
Main MEM	32GB
GPU	NVIDIA Tesla S1070 (T10-based card × 4) shared by 2 compute nodes
# of cards	2
# of cores per card	240 cores (1.29 - 1.44 GHz)
Global MEM per card	4GB
Interconnect	SDR Infiniband × 2
PCI-Express Bandwidth	2GB/s
OS	Linux 2.6.16

1) Select a value K. Here, we divide data instances $x_i (i = 1, \ldots, n)$ to K clusters.
2) For each data instance x_i, assign a cluster in K clusters randomly.
3) Based on the clusters assigned to data instances x_i, calculate the centroid $V_j (j = 1, \ldots, K)$ of each of K clusters.
4) Calculate distances between the data instance x_i and each centroid V_j. Then reassign the data instance x_i to the closest cluster.
5) Recalculate each centroid $V_j (j = 1, \ldots, K)$ of each of K clusters and iterate the above process until the cluster assignment converges for all data instances.

We implemented both CPU and GPU version [11] of K-means applications; we use C++ for the CPU version and CUDA for the GPU version. By using these applications, our MapReduce program works as follows:

1) In the map phase, the map tasks apply the K-means application to each file.
2) Then, in the reduce phase, the reduce tasks accumulate the results of the map tasks.

In this experiment, we set the number of clusters, K, to 128. We use 20GB of files that contain 4000 sets of 262144 2-dimensional points.

B. Experimental Settings

We conduct the K-means application on the TSUBAME supercomputer, which is a supercomputer located in the Tokyo Institute of Technology, Japan. The specification of each compute node is listed in Table I. We use up to 64 nodes (1024 CPU cores and 128 GPU cards). We use the Lustre file system as a distributed file system for Hadoop instead of HDFS, since TSUBAME has a few local storage volume in a compute node. The number of stripes of files in the file system is set to 4. The I/O performance to the file system is 180 MB/s for write and 610 MB/s for read for a single file with 32MB. In this experiment, we use Sun Java version 1.6.0 and the CUDA toolkit version 2.3.

C. Experimental Results

Figure 6 shows the result of the total MapReduce job execution time. We apply three policies for the map task execution as follows:

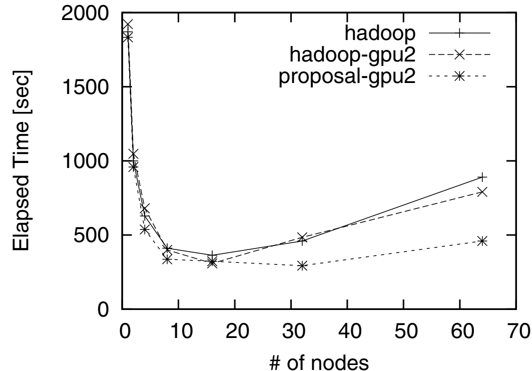

Fig. 6. Total MapReduce job execution time

- 16 CPU cores per node are used for the Map phase by using the original Hadoop scheduling algorithm, which always schedules map tasks to idle CPU slots (*hadoop*).
- 14 CPU cores and 2 GPU devices per node are used for the Map phase by using Hadoop-based scheduling algorithm, which always schedules map tasks to idle CPU and GPU slots (*hadoop-gpu2*).
- 14 CPU cores and 2 GPU devices per node are used for the Map phase by using our proposed hybrid scheduling technique (*proposal-gpu2*).

The x-axis corresponds to the number of nodes, and the y-axis corresponds to the elapsed time (sec) of the total MapReduce job. Here, we see that using GPU can reduce MapReduce job execution time; the *hadoop-gpu2* policy is 1.13 times faster than the *hadoop* policy at 64 nodes (1024 CPU cores and 128 GPU devices). However, there are negligible differences between *hadoop* and *hadoop-gpu2* policies. This is because the acceleration factor of our target application is low and the experimental environment has many CPU slots rather than GPU slots. On the other hands, the *proposal-gpu2* policy exhibits higher performance than the *hadoop* and *hadoop-gpu2* policies, since our proposed technique automatically determines the number of map tasks to execute on CPU cores and GPU devices; we see that the *proposal-gpu2* policy is 1.93 times faster at 64 nodes (1024 CPU cores and 128 GPU devices) and 1.02 times faster at 1 node (16 CPU cores and 2 GPUs).

We observe performance overheads when the number of nodes are increasing in Figure 6. To clarify this cause, we investigate the mean elapsed time of map tasks executed on CPU and GPU slots respectively. Figure 7 shows the results of the mean CPU and GPU elapsed time. The x-axis corresponds to the number of nodes, and the y-axis corresponds to the map task mean time (sec). We see that both of the map task mean times increase linearly in proportion to the number of nodes, which introduces the performance overheads of the overall MapReduce job execution. We consider that this is caused by I/O overheads, since the file system, we use Lustre instead of HDFS in this experiment, is configured

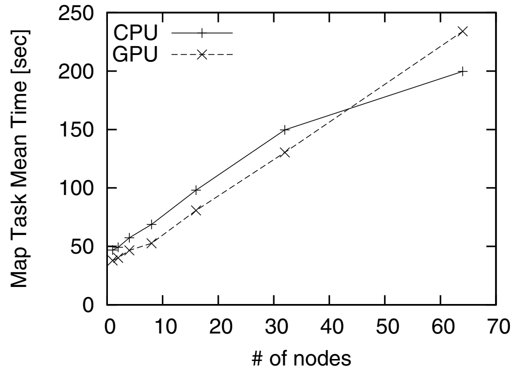

Fig. 7. Mean elapsed time of CPU and GPU map tasks

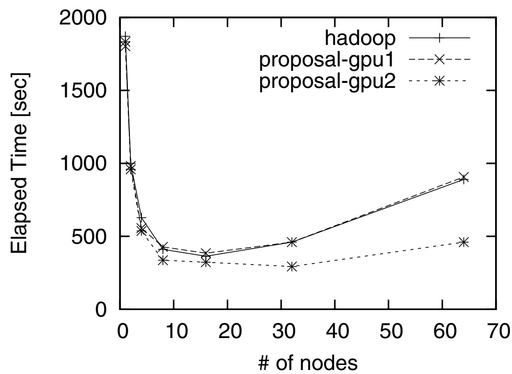

Fig. 8. Total MapReduce job execution time under the different number of GPU devices per node

with separated compute and storage nodes connected with shared networks, which may cause I/O contentions affected by other processes. On the other hand, HDFS is configured with combined compute and storage nodes to localize file accesses, which avoids I/O contention situations. As future work, we are planning to conduct experiments to verify the cause of I/O contentions in a shared HPC filesystem such as Lustre by comparing with other large-scale computing environments with a HDFS-based distributed filesystem.

Figure 8 compares the total MapReduce job execution time when the number of GPU devices per node varies. Here, we introduce three policies as follows:

- 16 CPU cores per node are used for the Map phase by using the original Hadoop scheduling algorithm, which always schedules map tasks to idle CPU slots (*hadoop*).
- 15 CPU cores and 1 GPU device per node are used for the Map phase by using our proposed hybrid scheduling technique (*proposal-gpu1*).
- 14 CPU cores and 2 GPU devices per node are used for the Map phase by using our proposed hybrid scheduling technique (*proposal-gpu2*)

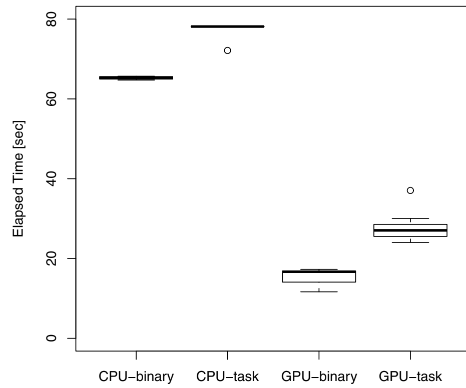

Fig. 9. The overhead of both CPU and GPU binary code invocations from Hadoop

The x-axis corresponds to the number of nodes, and the y-axis corresponds to the elapsed time (sec). We observe that there are negligible differences between the *hadoop* and *proposal-gpu1* policies; however, the *proposal-gpu2* policy exhibits good performance than other *hadoop* and *proposal-gpu1* policies. Thus, using many GPUs improves the overall MapReduce execution time.

Our prototype invokes both CPU and GPU binary codes form the Hadoop framework by using the Hadoop Pipes-based mechanism, which may introduce extra performance overheads to running applications. We evaluate the overhead of binary code invocation from the Hadoop layer by using a single local compute node, which consists of AMD Phenom 9850 Quad core (2.5GHz) and NVIDIA GeForce GTX 275 (1.4GHz) with 8.2GB of main memory, running Linux 2.6.27. Here, we compare four metrics as follows:

- CPU binary execution time (*CPU-binary*), which denotes the actual execution time of a C++-based application binary program.
- CPU map task execution time (*CPU-task*), which denotes the interval, including the CPU binary execution time, between the time when a CPU map task is allocated by the scheduler and the time that the map task is finished.
- GPU binary execution time (*GPU-binary*), which denotes the actual execution time of a CUDA-based application binary program.
- GPU map task execution time (*GPU-task*), which denotes the interval, including the GPU binary execution time, between the time when a GPU map task is allocated by the scheduler and the time that the map task is finished.

Figure 9 shows the result of the overhead of both CPU and GPU binary code invocations from the Hadoop framework. Here, we see that the mean CPU binary execution time is 65.2 sec, the mean CPU map task execution time is 77.4 sec, the mean GPU binary execution time is 15.5 sec, and the mean

GPU map task execution time is 27.9 sec; there are significant overheads between binary execution and map task execution: 6 % for CPU and 44 % for GPU on average. This arises from the implementation of the Hadoop framework. We also observe that GPU execution causes perturbation; the maximum GPU binary execution time is 17.3 sec and the minimum GPU binary execution time is 11.7 sec.

VI. Related Work

There are several MapReduce studies that considers GPU-based computing environments. He et al. have proposed a MapReduce framework, called Mars [6], for commodity GPUs, which focuses on providing a generic framework for developers to implement GPU-based applications. Their work mainly targets on a single machine with commodity GPUs; however, our target environments include a large-scale computer cluster with multiple commodity GPU devices. Stuart et al. have proposed a MapReduce-based volume rendering application for a multi-GPU computer cluster [12], whereas our study focuses on hybrid map task scheduling that considers running application characteristics and underlying computing environments.

There are also several studies related to task scheduling for GPU-based computing environments [13], [14]. Ravi et al. have proposed a parallel reduction system for CPU-GPU heterogeneous environments [13]. They have reported fluctuation of task execution time by varying chunk sizes to allocate CPU cores and GPU devices; however, their work does not focus on dynamic scheduling and multi-GPU computer clusters. Lu et al. have proposed a programming system, called Qilin [14], which adaptively maps computations to processing elements on a CPU-GPU hybrid machine by using training mechanisms. Their work is similar to ours in terms of distributing tasks onto CPUs and GPUs automatically; however, our proposal employs dynamic scheduling and does not require any training processes for optimization.

There are vast amounts of studies related to task scheduling on heterogeneous computing environments. Zaharia et al. have presented a task scheduling technique by using speculative execution in order to avoid straggler tasks and to improve efficiency for CPU-based heterogeneous environments [15]. However, our work considers GPU-based heterogeneous computing environments, which arise different problems to CPU-based ones, such as data transfer overheads between CPU cores and GPU devices, latency hidings between compute nodes, etc. Current our prototype only considers the execution time of map tasks running on both CPU cores and GPU devices. We are planning to improve our scheduling technique by monitoring detailed profiles of map tasks, such as memory bandwidth performance and disk access time, etc.

VII. Conclusion

We have presented a hybrid map task execution technique for GPU-based heterogeneous computer clusters in the Hadoop framework. We have also proposed a hybrid task scheduling algorithm, which minimizes the total MapReduce job time by using dynamic monitoring of map task's behavior, such as the elapsed time of map tasks. Our experimental results based on using our Hadoop-based prototype have showed that our proposed technique achieves 1.93 times faster application performance than the Hadoop original scheduling algorithm at 64 nodes (1024 CPU cores and 128 GPU devices). The results also indicates that the performance of map tasks, including both CPU tasks and GPU tasks, is significantly affected by the overhead of map task invocation from Hadoop.

As future work, we are planning to monitor GPU task behavior, such as memory usage and data transfer performance between CPU cores and GPU devices, and I/O performance to storage system, and to improve the scheduling model and overheads by using the detailed profiles.

Acknowledgment

This work is partially supported by the Ministry of Education, Culture, Sports Science, and Technology, the Grants-in-Aid for Scientific Research on Priority Areas (18049028) and the JST-CREST Ultra-Low Power HPC Project.

References

[1] J. F. Gantz, C. Chute, A. Manfrediz, S. Minton, D. Reinsel, W. Schlichting, and A. Toncheva, "The diverse and exploding digital universe," IDC white paper, 2008.

[2] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *OSDI '04, Sixth Symposium on Operating System Design and Implementation*, pp. 137–150, 2004.

[3] J. D.Owens, M. Houston, D. Luebke, S. Green, J. E.Stone, and J. C.Phillips, "GPU computing," *Proc IEEE*, vol. 96, no. 5, pp. 879–899, 2008.

[4] "Hadoop homepage," http://hadoop.apache.org.

[5] R. Colby, R. Ramanan, P. Arun, B. Gary, and K. Christos, "Evaluating mapreduce for multi-core and multiprocessor systems," in *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 13–24.

[6] B. He, W. Fang, Q. Luo, N. K.Govindaraju, and T. Wang, "Mars: A mapreduce framework on graphics processors," *Parallel Architectures and Compilation Techniques*, pp. 260–269, 2008.

[7] "Nvidia cuda," http://developer.nvidia.com/cuda.

[8] "Khronos Group Open Computing Language," http://www.khronos.org/opencl/.

[9] Y. Yan, M. Grossman, and V. Sarkar, "Jcuda: A programmer-friendly interface for accelerating java programs with cuda," *Lecture Notes in Computer Sciences*, vol. 5704 (2009), pp. 887–899, 2009.

[10] "jcuda - java for cuda," http://hoopoe-cloud.com/Solutions/jCUDA/".

[11] B. Hong-tao, H. Li-li, O. Dan-tong, L. Zhan-shan, and L. He, "K-means on commodity gpus with cuda," *Computer Science and Information Engineering, 2009 WRI World Congress*, pp. 651–655, 2009.

[12] J. A. Stuart, C.-K. Chen, K.-L. Ma, and J. D. Owens, "Multi-gpu volume rendering using mapreduce," *1st International Workshop on MapReduce and its Applications*, June 2010.

[13] T. R. Vignesh, M. Wenjing, C. David, and A. Gagan, "Compiler and runtime support for enabling generalized reduction computations on heterogeneous parallel configurations," in *ICS '10: Proceedings of the 24th ACM International Conference on Supercomputing*. New York, NY, USA: ACM, 2010, pp. 137–146.

[14] C.-K. Lu, S. Hong, and H. Kim, "Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping," *MICRO '09*, pp. 45–55, 2009.

[15] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, "Improving mapreduce performance in heterogeneous environments," *The 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI2008)*, pp. 29–42, 2008.