論文 / 著書情報
Article / Book Information

題目(和文)	GPU搭載ヘテロジーニアススーパーコンピュータ上でのスケーラブルで階層的なMapReduce型大規模データ処理
Title(English)	Scalable and Hierarchical MapReduce-based Large-scale Data Processing on GPU-based Heterogeneous Supercomputers
著者(和文)	白幡晃一
Author(English)	Koichi Shirahata
出典(和文)	学位:博士（理学）, 学位授与機関:東京工業大学, 報告番号:甲第9746号, 授与年月日:2015年3月26日, 学位の種別:課程博士, 審査員:松岡 聡,遠藤 敏夫,増原 英彦,脇田 建,渡辺 治,藤澤 克樹
Citation(English)	Degree:, Conferring organization: Tokyo Institute of Technology, Report number:甲第9746号, Conferred date:2015/3/26, Degree Type:Course doctor, Examiner:,,,,,
学位種別(和文)	博士論文
Type(English)	Doctoral Thesis

TOKYO INSTITUTE OF TECHNOLOGY

Scalable and Hierarchical MapReduce-based Large-scale Data Processing on GPU-based Heterogeneous Supercomputers

by

Koichi Shirahata

A thesis submitted in partial fulfillment for the
degree of Doctor of Science
in
Mathematical and Computing Sciences

in the
Graduate School of Information Science and Engineering

Committee Chair in charge:
Professor Satoshi Matsuoka
February 2015

TOKYO INSTITUTE OF TECHNOLOGY

Abstract

Graduate School of Information Science and Engineering

Doctor of Science
in
Mathematical and Computing Sciences

by Koichi Shirahata

Fast processing for extremely large-scale data is becoming increasingly important in various domains such as health care, social networks, system biology, and electric power grids, which typically consists of millions to trillions of elements. Although supercomputers has accelerated a wide range of applications such as physical simulations, large-scale data processing is also considered an important application on supercomputers. Recent supercomputers employ many-core processors such as GPUs in addition to general purpose CPUs, since many-core processors can provide high peak performance and high memory bandwidth for applications with specific computation patterns.

Although many-core-based supercomputers are possible environments for large-scale data processing, how to accelerate extremely large-scale data processing with careful consideration of scalability of multiple many-cores and management of deep memory hierarchy on many-core-based heterogeneous supercomputers is an open problem. Firstly, the validity of acceleration, including optimization techniques, of large-scale data processing using many-cores is an open problem. Also, efficient techniques to handle many-core memory overflows, including overflow detection and performance analysis in large-scale systems, are not well investigated.

To address the problem, we propose scalable and hierarchical multi-GPU MapReduce-based large-scale data processing techniques for GPU-based heterogeneous supercomputers. Our implementation applies a number of optimization techniques for improving scalability such as load balance optimization, as well as thread assignment optimization and data compression. Our implementation also handles GPU memory overflow by applying a technique that automatically divides input data into multiple chunks and overlaps CPU-GPU data transfer and computation on GPUs as much as possible.

Our experimental results on TSUBAME2.5 using 1024 nodes (12288 CPU cores, 3072 GPUs) exhibit that our GPU-based implementation performs 2.10x faster than running on CPU when graph data size does not fit on GPUs. We also see that our implementation exhibits 186.6 times performance improvement compared with an existing widely used MapReduce-based graph processing implementation.

Acknowledgements

I would like to express my special appreciation and thanks to my advisor Prof. Satoshi Matsuoka, you have been a tremendous mentor for me. I would like to thank you encouraging my research and for allowing me to grow as a research scientist. Your advice on both research as well as on my career have been priceless. I owe sincere and earnest thankfulness to Prof. Hitoshi Sato, Prof. Toyotaro Suzumura, and the members at Tokyo Institute of Technology, for their help and support for research activities, for providing me with an excellent atmosphere for doing research. I would especially like to express my greatest gratitude to the people who have helped me with my research work. I would like to thank Japan Society for the Promotion of Science (JSPS) and Japan Science and Technology Agency (JST), which gave me the great opportunity to study at Tokyo Institute of Technology.

I would also like to show a special thanks to my family. Words cannot express how grateful I am to my mother-in-law, father-in-law, my mother, father, and my sisters for all of the sacrifices that you've made on my behalf. Your prayer for me was what sustained me thus far. At the end I would like to express appreciation to my beloved wife who spent sleepless nights with and was always my support in the moments when there was no one to answer my queries.

Koichi Shirahata

February 2015

Publications (International)

[1] Koichi Shirahata, Hitoshi Sato, and Satoshi Matsuoka. Out-of-core GPU memory management for MapReduce-based large-scale graph processing. In *2014 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 221–229. IEEE, 2014.

[2] Koichi Shirahata, Hitoshi Sato, Toyotaro Suzumura, and Satoshi Matsuoka. A scalable implementation of a MapReduce-based graph processing algorithm for large-scale heterogeneous supercomputers. In *2013 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 277–284. IEEE, 2013.

[3] Koichi Shirahata, Hitoshi Sato, Toyotaro Suzumura, and Satoshi Matsuoka. A GPU implementation of generalized graph processing algorithm GIM-V. In *the 3rd International Workshop on Parallel Algorithm and Parallel Software (IWPAPS 2012)*, pages 207–212. IEEE, 2012.

[4] Koichi Shirahata, Hitoshi Sato, and Satoshi Matsuoka. Hybrid map task scheduling for GPU-based heterogeneous clusters. In *the 1st International Workshop on Theory and Practice of MapReduce (MAPRED '2010)*, pages 733–740. IEEE, 2010.

Publications (Domestic)

[1] Zhang Chaojie, Koichi Shirahata, Shuji Suzuki, Yutaka Akiyama, and Satoshi Matsuoka. Performance analysis of MapReduce implementations for high performance homology search. *IPSJ SIG Technical Reports 2014-HPC-147*, 2014.

[2] Koichi Shirahata, Hitoshi Sato, and Satoshi Matsuoka. Preliminary I/O performance evaluation on GPU accelerator and external memory. *IPSJ SIG Technical Reports 2013-HPC-141*, 2013.

[3] Koichi Shirahata, Hitoshi Sato, Toyotaro Suzumura, and Satoshi Matsuoka. A multi GPU implementation of generalized graph processing model GIM-V with data transfer optimization. *IPSJ SIG Technical Reports 2012-HPC-133*, 2012.

[4] Koichi Shirahata, Hitoshi Sato, Toyotaro Suzumura, and Satoshi Matsuoka. Towards GPGPU-based large-scale fast graph processing. *IPSJ SIG Technical Reports 2011-HPC-130*, 2011.

[5] Koichi Shirahata, Hitoshi Sato, and Satoshi Matsuoka. Improving MapReduce task scheduling for CPU-GPU heterogeneous environments. *IPSJ SIG Technical Reports 2010-HPC-126*, 2010.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Recent emergence of extremely large-scale data in various application fields, such as health care, social networks, intelligence, system biology, and electric power grids, which typically consists of millions to trillions of elements, requires fast and scalable analysis by using HPC technologies. For example, a friend network in an existing social network service [4] is expressed as a graph with over 1.31 billion vertices and over 170 billion edges, and is required to analyze mutual relationships of the graph. Furthermore, these large-scale graph applications attract recent attention to the Graph500 benchmark [5] in the HPC community, which ranks supercomputers by executing a large-scale graph search problem as an instance of data-intensive supercomputing applications. Also, homology search to be used in emerging bioinformatics problems such as metagenomics is of increasing importance and challenge as its application area grows more broadly while the computational complexity is increasing. Required dataset for metagenomic search consists of queries and database, each of whose size will reach Gigabytes to Terabytes, and total data size to compute will grow to product of these two datasets (i.e. Exabytes to Zettabytes).

MapReduce [3] is a successful programing model for efficient, scalable, and massive data processing with large-scale commodity compute clusters, which conceals elaborate efforts in distributed systems such as communication between thousands of nodes, data management for petabyte-scale large data volumes, and fault tolerance. MapReduce is also applied to graph processing with petabyte-scale data; PEGASUS [6], which is an Hadoop [7]-based peta-scale graph mining system that employs the GIM-V (Generalized

Iterative Matrix-Vector multiplication) algorithm, has been proposed. GIM-V enables users to describe important graph algorithms, such as PageRank, Random Walk, and Connected Component, without any difficulties in distributed systems. Kang et al. [6] have reported that GIM-V exhibits good scalability in a compute cluster; however, such CPU-based implementation introduces significant performance overheads when we increase the size of a graph.

Recent supercomputers employ many-core processors such as commodity graphics processing units (GPUs) and Intel Many Integrated Core Architecture (MIC) [8] in addition to general CPUs, since many-core processors can provide high peak performance and high memory bandwidth for applications with specific computation patterns, while CPUs offer flexibility and generality over wide-ranging classes of applications. These supercomputers are called heterogeneous supercomputers since these supercomputers employ two different types of processors. A large number of heterogeneous supercomputers have been ranked high order in terms of the TOP500 list [9]. For instance, Tianhe-2 at National Super Computer Center in Guangzhou, China, which employs Intel Xeon Phi many-core processors ranked 1st in June 2014. As for GPU-based heterogeneous supercomputers, Titan [10] at Oak Ridge National Laboratory, United States ranked 2nd and TSUBAME2.5 [11] at Tokyo Institute of Technology, Japan ranked 13th. This tendency is applied not only to HPC supercomputers, but cloud data centers as well. For example, Amazon EC2 provides Cluster GPU Instances as a GPU-based compute cluster [12]. In such environments, large-scale graph processing is also considered as an important kernel application. In practice, several existing GPU-based graph processing techniques have shown that the GPUs accelerate the performance on several graph applications, such as Breadth-First Search (BFS) [13], PageRank [14], etc.

1.2 Problem Statement

Although utilizing the big data software substrates using MapReduce enables to handle large-scale data on many-core-based heterogeneous supercomputers, how to accelerate extremely large-scale data processing with careful consideration of scalability of multiple many-cores and management of deep memory hierarchy on many-core-based heterogeneous supercomputers is an open problem. Firstly, although many-core-based heterogeneous compute clusters are also possible environments for large-scale data processing applications, how much the applications can be accelerated is unclear. Moreover, we have to consider overflow of graph data from a single many-core memory when applying the GIM-V algorithm to large-scale data. Using multiple GPU devices may relax the

overflow situation; however, even in such cases, load balance optimization techniques between GPU devices are required for efficient execution of the application. How much utilizing many-cores accelerates for large-scale data processing is a problem worth investigating. Also, the capacity of device memory on many-cores limits scalable large-scale data processing, since many-cores typically have smaller memory capacity than the CPU hosts. For example, the TSUBAME2.5 supercomputer [11] employs 1408 compute nodes, each of which equips 3 GPU devices and 2 CPU sockets, where the capacity of device memory on each GPU is 6GB, while that of CPU host memory is 54GB. Thus, in order to process larger-scale graphs whose size exceeds the capacity of many-core memory, data management techniques for handling many-core memory overflows are required. However, such out-of-core many-core data management techniques with detailed performance studies for large-scale graph processing are not well investigated. Furthermore, even if we apply the out-of-core many-core data management techniques, which execution approaches to use, only the device memory on many-cores (scale-out) or offload partial graph data to the secondary CPU memory (scale-up) on a multi-node environment, in terms of graph application's performance and its power efficiency, is considered another important issue.

1.3 Proposal

To address the problem, we propose scalable and hierarchical multi-GPU MapReduce-based large-scale data processing techniques for GPU-based heterogeneous supercomputers. First, we implement a multi-GPU-based GIM-V application with load balance optimization among GPU devices. Our implementation extends the existing MapReduce library for supporting multi-GPU-environments using the MPI library and optimizes load balance among GPU devices by employing task scheduling-based graph partitioning. We also propose an out-of-core GPU memory management technique for GPU-MapReduce-based graph applications. Our proposed technique automatically handles GPU memory overflows by dividing graph data into multiple chunks and hides CPU-GPU data transfer overheads by overlapping computations on GPUs and CPU-GPU data transfers. We also investigate the balance of the scale-up and scale-out approaches, in terms of the number of GPUs for processing graph data size per node, by comparing application's performance and power efficiency. We also implement large-scale data processing applications including a large-scale graph processing algorithm and a metagenomic homology search algorithm on top of MapReduce.

1.4 Contributions

This thesis presents several contributions towards fast and scalable MapReduce-based large-scale data processing on GPU-based heterogeneous supercomputers. The primary contributions are as follows:

1.4.1 A Scalable Implementation of a MapReduce-based Graph Processing Algorithm on GPUs

We implement a multi-GPU-based GIM-V application with load balance optimization between GPU devices. We conducted our implementation on the TSUBAME2.0 supercomputer using 256 nodes (6144 hyper-threaded CPU cores, 768 GPU devices). The results exhibit that our GPU-based implementation performed 87.04 ME/s on 2^{30} (1.07 billion) vertices and 2^{34} (17.2 billion) edges, and 1.52 times faster than the CPU-based naive implementation with 2^{29} (536.9 million vertices) and 2^{33} (8.6 billion) edges. We also exhibit the performance characteristics of our implementation and load balance optimization technique.

Here is a quick summary of contributions of this work:

- We implemented a multi-GPU-based GIM-V application by extending an existing MapReduce library that supports a single GPU environment.
- We applied load balance optimization between GPU devices for large-scale graphs.
- We studied the performance characteristics of our multi-GPU-based GIM-V implementation and load balance optimization.

1.4.2 Out-of-core GPU Memory Management for MapReduce-based Graph Processing

We propose an out-of-core GPU memory management technique for GPU-MapReduce-based graph applications. We conduct experiments on TSUBAME2.5 using up to 1024 nodes (12288 CPU cores, 3072 GPU devices). The results exhibit that our GPU-based implementation performs 2.81 GE/s (billion edges per second) on a large-scale graph with 2^{34} (17.18 billion) vertices and 2^{38} (274.9 billion) edges. These results indicate that our GPU-based implementation performs 2.10x faster than the multi-core CPU-based implementation even when the graph data size exceeds the device memory capacity on

the multiple GPUs. We also show that the scale-up approach outperforms the scale-out approach by 1.71x in power efficiency on the TSUBAME-KFC supercomputer.

Here we describe a summary of contributions of this work:

- We propose an out-of-core GPU data management technique for GPU-based-MapReduce-based large-scale graph processing.
- We demonstrate the scalability of our proposed technique on heterogeneous large-scale GPU-based supercomputers by utilizing several optimization techniques.
- We investigate the balance of scale-up and scale-out approaches, i.e., the number of GPUs for processing graph data per node, whose results suggest that the scale-up approach helps power-efficient graph processing rather than the simple scale-out approach.

1.5 Thesis Outline

Before describing the main contributions of this thesis, some background information about large-scale data processing on supercomputers are provided, particularly on large-scale data processing applications, MapReduce-based large-scale data processing including graph processing and bioinformatic processing, GPU architecture, GPU-based heterogeneous supercomputers, programming models on GPUs.

Chapter 3 describes related work to our work. Existing work on graph processing on GPUs, large-scale graph processing on CPUs and on GPUs including our-of-core processing, as well as MapReduce implementations utilizing GPUs are introduced. Efforts on MapReduce-based large-scale data processing including graph processing and bioinformatic processing are also provided.

Chapter 4 addresses the problem of the scalability of large-scale graph processing on GPUs. We start by providing a general introduction to the problem domain of large-scale graph processing and a formal definition of the problem. We describe in detail our multi-GPU-based MapReduce implementation and analyze the implementation in terms of edge scan performance on GPU architecture.

Chapter 5 addresses the problem of out-of-core GPU memory management for graph processing. Particular attention is given to the problem of overlapping of moving data between host and GPU memories. This chapter also discusses the balance of the scale-up

and scale-out approaches, in terms of the number of GPUs for processing graph data size per node, by comparing application's performance and power efficiency.

Chapter 6 presents additional discussion on applicability of the proposed techniques on MapReduce-based large-scale data processing on GPU-based heterogeneous supercomputers. We also introduce another big data application case study: MapReduce-based designs and implementations of a metagenomic homology search algorithm. We also compare the MapReduce-based implementations with an existing MPI-based master-worker framework for homology search.

Chapter 7 concludes this work. It outlines the main findings from implementation experience and performance evaluation of our solutions. We also suggest some directions for future work in processing further larger data by utilizing deeper memory hierarchy utilizing Non-Volatile Memories.

Chapter 2

Background

This chapter provides basic background information for understanding GPU computing on heterogeneous supercomputers and large-scale data processing. We start by reviewing overview of GPU computing in Section 2.1. We then describe the MapReduce programming model and existing MapReduce implementations in Section 2.2, followed by summarizing large-scale data processing including large-scale data in real world, existing large-scale data processing applications, as well as MapReduce-based large-scale data processing in Section 2.3.

2.1 GPU Computing

In this section, firstly we give overview of GPU architecture, followed by introduction of GPU-based heterogeneous supercomputers, then we summarize existing programming models for GPU computing.

2.1.1 GPU Architecture

A graphics processing unit (GPU) is a specialized electronic circuit designed to rapidly manipulate and alter memory to accelerate the creation of images. Although GPU is originally designed for accelerating image processing, GPGPU (General-purpose computing on GPU) [15] became practical and popular, with the advent of both programmable shaders and floating point support on graphic processors. GPGPU 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.

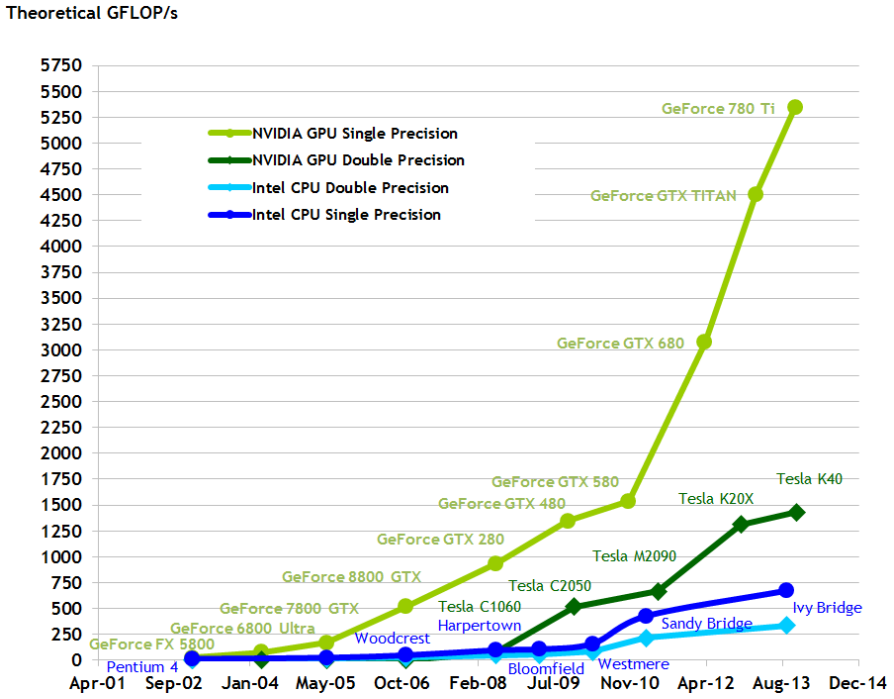

FIGURE 2.1: Floating point operations per second. Source: NVIDIA Corporation (2014) [1]

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 [1] and OpenCL [16], enables application programmers to use stream processing on non-graphics data.

GPU suits for parallel computing, since the architecture employs SIMD-based processing. GPU achieves much higher perk performance and memory bandwidth than CPU by using tens of thousands of fine-grain threads. Figure 2.1 shows the computing power of the GPU and how it compares to the CPU. The vertical axis shows the theoretical GFLOP/s (Giga Floating Point Operations per Second). The horizontal axis shows the advances in technology over the years. As can be seen from the figure, GPUs can theoretically perform 1.4 to 5.3 Trillion Floating Point Operations per Second (or 1.4 to 5.3 teraFLOPS), which is around 5 to 17 times faster compared with CPUs. The GPU is also capable of transferring large amounts of data through the PCI-Express bus. Figure 2.2 shows the memory bandwidth in GB/s of the latest NVIDIA GPU compared to the latest desktop CPUs from Intel. As can be seen from the figure, GPUs can also perform around 280 to 340 GB/s, which is around 5 to 6 times faster compared with CPUs.

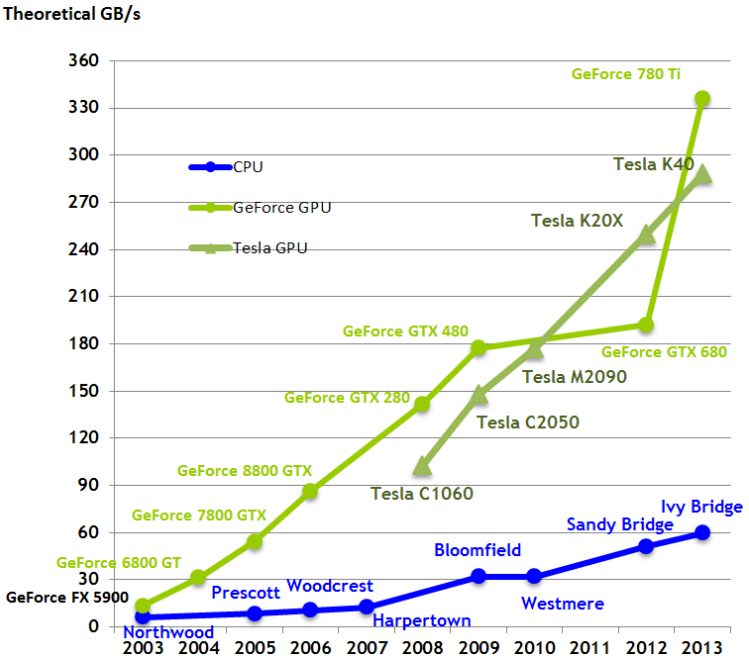

FIGURE 2.2: Memory bandwidth. Source: NVIDIA Corporation (2014) [1]

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.

Kepler architecture is currently NVIDIA 's flagship GPU replacing the Fermi architecture. The Kepler GPU was designed to be the highest performing GPU in the world. The Kepler GK110 consists of 15 SMX (streaming multiprocessor) units and six 64-bit memory controllers as shown in Figure 2.3. If we zoom into a single SMX unit, we see that each SMX unit consists of 192 single-precision CUDA cores, 64 double-precision units, 32 special function units (SFU), and 32 load/store units (LD/ST) as shown in Figure 2.4. Each SMX supports 64 KB of shared memory, and 48 KB of read-only data cache. The shared memory and the data cache are accessible to all threads executing on the same streaming multiprocessor. Access to these memory areas is highly optimized and should be favored over accessing memory in global DRAM. The SMX will

FIGURE 2.3: Kepler architecture. Source: NVIDIA Corporation (2012) [2]

schedule 32 threads in a group called a warp. Using compute capability (generation of CUDA architecture) 3.5, the GK110 GPU can schedule 64 warps per SMX for a total of 2,048 threads that can be resident in a single SMX at most at a time. Each SMX has four warp schedulers and eight instruction dispatch units (two dispatch units per warp scheduler) allowing four warps to be issued and executed concurrently on the streaming multiprocessor.

2.1.2 GPU-based Heterogeneous Supercomputers

Recent supercomputers employ many-core processors such as GPU and Intel Many Integrated Core Architecture (MIC) [8] in addition to general CPUs, since many-core processors can provide high peak performance and high memory bandwidth for applications with specific computation patterns, while CPUs offer flexibility and generality over wide-ranging classes of applications. These supercomputers are called heterogeneous supercomputers since these supercomputers employ two different types of processors. A large number of heterogeneous supercomputers have been ranked high order in terms of the TOP500 list [9]. For instance, Tianhe-2 at National Super Computer Center in Guangzhou, China, which employs Intel Xeon Phi many-core processors ranked 1st

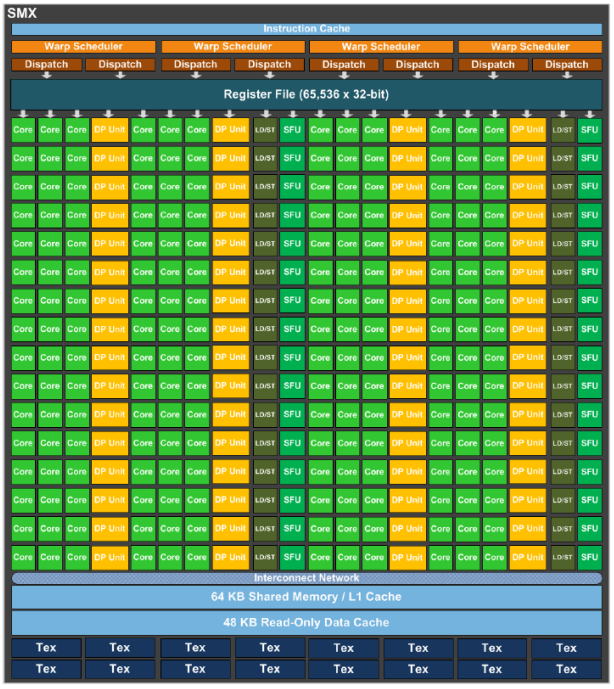

FIGURE 2.4: CUDA SMX unit. Source: NVIDIA Corporation (2012) [2]

in June 2014. As for GPU-based heterogeneous supercomputers, Titan [10] at Oak Ridge National Laboratory, United States ranked 2nd and TSUBAME2.5 [11] at Tokyo Institute of Technology, Japan ranked 13th.

2.1.3 Programming Models for GPUs

Several programming environments, such as CUDA [1] and OpenCL [16], 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.

CUDA C extends C by allowing the programmer to define C functions, called kernels, that, when called, are executed N times in parallel by N different CUDA threads, as opposed to only once like regular C functions. Each thread that executes the kernel

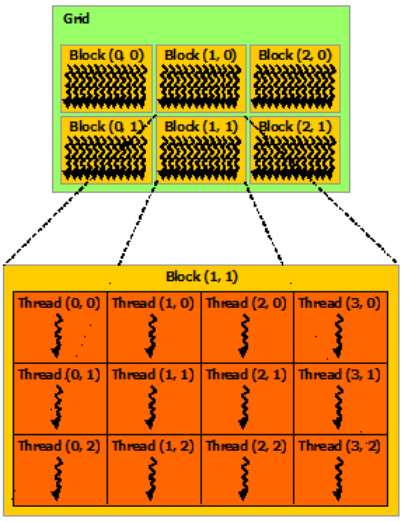

FIGURE 2.5: Grid of thread blocks. Source: NVIDIA Corporation (2014) [1]

is given a unique thread ID that is accessible within the kernel through the built-in `threadIdx` variable. For convenience, threadIdx is a 3-component vector, so that threads can be identified using a one-dimensional, two-dimensional, or three-dimensional thread index, forming a one-dimensional, two-dimensional, or three-dimensional thread block. This provides a natural way to invoke computation across the elements in a domain such as a vector, matrix, or volume. There is a limit to the number of threads per block, since all threads of a block are expected to reside on the same processor core and must share the limited memory resources of that core. On current GPUs, a thread block may contain up to 1024 threads. However, a kernel can be executed by multiple equally-shaped thread blocks, so that the total number of threads is equal to the number of threads per block times the number of blocks. Blocks are organized into a one-dimensional, two-dimensional, or three-dimensional grid of thread blocks as illustrated by Figure 2.5. The number of thread blocks in a grid is usually dictated by the size of the data being processed or the number of processors in the system, which it can greatly exceed. Each block within the grid can be identified by a one-dimensional, two-dimensional, or three-dimensional index accessible within the kernel through the built-in `blockIdx` variable. The dimension of the thread block is accessible within the kernel through the built-in `blockDim` variable.

2.2 MapReduce

This section describes overview of the MapReduce programming model, followed by summary of existing MapReduce implementations including CPU-based and GPU-based

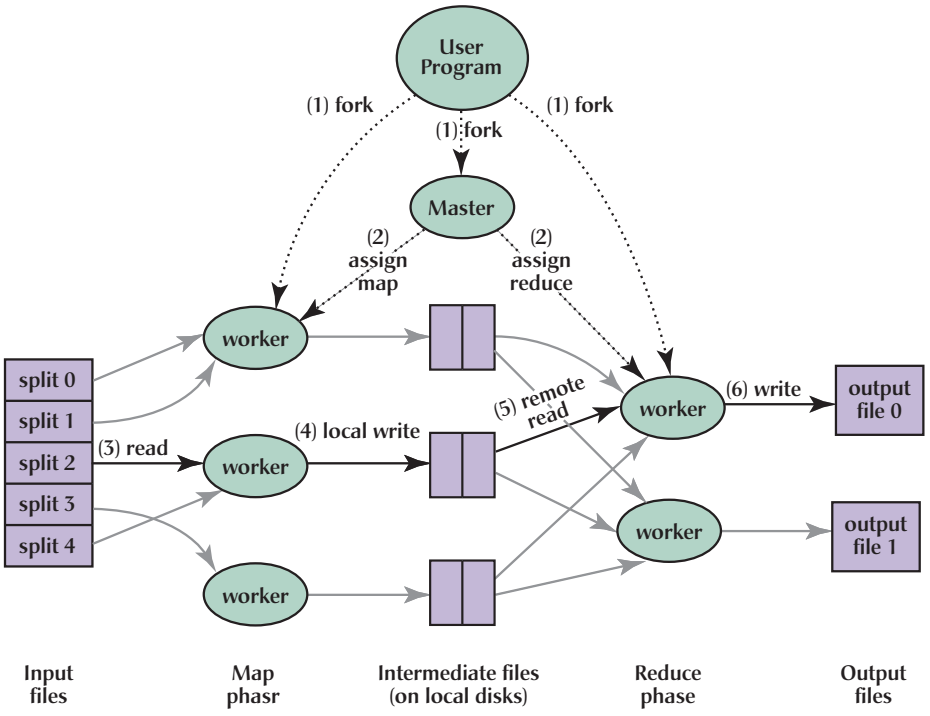

FIGURE 2.6: MapReduce workflow. Source: Dean et al. (2004) [3]

implementations.

2.2.1 MapReduce Programming Model

MapReduce [3] is a programming model with associated software toolchains proposed by Google. MapReduce is used for large data sets effectively through distributed algorithm across a cluster. MapReduce is composed of two major functions. The Map function takes in the input and emits key-value pairs that represent useful information from the input. These key-value pairs are later passes to reduce function to process the final results. The Reduce function produce zero or more outputs based on the values associated with each different key. An advantage of MapReduce is that it can handle large-scale data even when the data is larger than host memory capacity by handling memory overflow automatically. Another characteristic is that MapReduce can also handle compute node failures by applying techniques of fault tolerance. MapReduce is suitable for large-scale data processing and its implementations are widely used.

Figure 2.6 shows execution workflow of MapReduce. Firstly, input data files are divided into multiple chunks (also called splits) whose size is typically 16 - 64 MB per split.

The master assigns each map task to workers. Then, each worker reads the contents of the corresponding input split. A worker parses key/value pairs out of the input data and passes each pair to the user-defined map function. The intermediate key/value pairs produced by the map function are buffered in memory or written to local disk. After that, the intermediate key/value pairs are transferred to workers (possibly via network) who is responsible to process the key in reduce phase, which is defined by a partitioning function. Reduce workers read the transferred intermediate data sent from map workers. When a reduce worker has read all intermediate data for its partition, the worker sorts the data by the intermediate keys so that all occurrences of the same keys are grouped together. The sorting is needed since typically many different keys map to the same reduce task. If the amount of intermediate data is too large to fit in memory, an external sort is used. After the reduce worker iterates over the sorted intermediate data and for each unique intermediate key encountered, the worker passes the key and the corresponding set of intermediate values to the user-defined reduce function. The output of the reduce function is appended to a final output file for this reduce partition.

2.2.2 Existing MapReduce Implementations

Google MapReduce [3] is the original implementation, which includes a distributed file system and a MapReduce framework itself.

Hadoop [17], inspired by the original Google MapReduce, is a now-popular open-source software framework implemented in Java for storing and processing large data distributively on clusters. Hadoop is consisted of Hadoop Common, Hadoop Distributed File System (HDFS), Hadoop YARN, and Hadoop MapReduce. HDFS is a highly fault-tolerant distributed system, designed for applications with large data sets. Hadoop YARN manages the compute resources in the file system and schedule jobs. 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 three replicas). On the other hand, Hadoop MapReduce provides a MapReduce execution environment on top of HDFS, whose environment also employs master-worker model.

There exists in-memory MapReduce implementations intended higher performance compared with Hadoop. Phoenix [18] provides programming APIs and runtime systems for

shared memory systems, such as systems employing multi-core processors. As for in-memory MapReduceo on distributed memory environments, Spark [19] is a fast open-resource cluster computing framework implemented in Scala, building on top of HDFS. Spark is intended to perform faster than Hadoop by in-memory computing, and promises performance up to 100 times faster than Hadoop MapReduce in some certain applications. The main abstraction Spark provides is a resilient distributed dataset (RDD), which is a collection of elements that can be persistent in memory and operated in parallel [20]. M3R (Main-Memory Map Reduce) [21] is another in-memory MapReduce implementation implemented in X10 [22], where X10 is a language for parallel computing adopting asynchronous partitioned global address space (APGAS) programming model. M3R is also compatible with Hadoop. Java Hadoop application can be submitted to M3R through M3R/Hadoop adaptor.

MPI-based MapReduce has been studied for multi-node execution for utilizing the fast network data transfer. Tu et al. provided a MapReduce-style programming interface for users to write molecular dynamics trajectory simulations, called HiMach [23]. Hoefler et al. proposed optimizations of MapReduce on top of MPI by combining shuffle and reduce and moves into the built-in or user-defined MPI reduction operations with several limitations in input keys [24]. MapReduce with MPI point-to-point communication operations has been also proposed [25]. The MapReduce-MPI (MR-MPI) library [26] is an open-source implementation of MapReduce written for distributed-memory parallel machines on top of standard MPI massage passing for processing terabyte-scale data sets on large-scale graph algorithms. MR-MPI can handle out-of-core execution by offloading intermediate data on local disks. MR-MPI exhibits good scalability up to 1024 processors on various graph processing algorithms. Their experimental results also showed that a distributed-memory matrix-based implementation using linear algebra toolkits performs an order of magnitude faster than MR-MPI when the input data fits on CPU host memory, while MR-MPI can handle out-of-core execution.

There also exists a lot of efforts on accelerating MapReduce utilizing GPUs. As for single GPU MapReduce implementations, Mars [27] and a proposal from Catanzaro et al. [28] are generic MapReduce frameworks for a single GPU, which enables application users to implement data-intensive and computation-intensive tasks efficiently and easily on a single GPU. MapCG [29] is proposed as a MapReduce framework to provide source code level portability between CPU and GPU without using OpenCL. MapCG improved performance on a GPU by two optimizations: replacing local sort by using hash functions, and removing index counting phases by applying specialized memory allocators. Ji et al. proposed optimization techniques on MapReduce on GPU by utilizing multiple

levels of the GPU memory hierarchy [30]. Chen et al. optimized their implementation by impoving shared memory usage [31]. They also extended their implementation for an integrated architecture of the CPU and GPU, using an AMD Fusion chip as a representative example [32]. These implementations are based on CUDA for NVIDIA GPUs. Also, StreamMR has been proposed as an OpenCL MapReduce framework for not only NVIDIA GPUs but also AMD GPUs [33].

As for multi-GPU MapReduce implementations, GPMR [34] is a multi-GPU MapReduce library supporting out-of-core GPU execution on distributed computing environments. Jiang et al. presented the MATE-CG system, which is a framework similar to MapReduce, based on the generalized reduction API [35]. Mars is also extended for multi-GPU by integrating with Hadoop [36]. Farivar et al. also proposed an architecture called MITHRA together with an integration of CUDA with Hadoop for multi-GPU execution [37].

MapReduce runtimes that support iterative MapReduce computations have been studied for expanding the applicability of MapReduce to more fields such as data clustering, machine learning, and computer vision where many iterative algorithms are common. HaLoop has been proposed as an extension of Hadoop for efficient iterative computation [38], as well as static scheduler-based MapReduce runtime with iterative support has been also presented called Twister [39].

2.3 Large-scale Data Processing

This section provides basic background information of large-scale data in real world and an existing graph model, followed by large-scale data processing algorithms including large-scale graph processing algorithms and large-scale homology search algorithms in metagenomics. We also summarize MapReduce-based large-scale data processing.

2.3.1 Large-scale Data in Real World

Networks in real world, such as health care, social networks, intelligence, system biology, and electric power grid, can be modeled as a graph with millions to trillions of vertices and 100's millions to 100's trillions of edges, whose structure has the following characteristics: scale-free (power-law degree distributions), small-world (6 degree of separation), and clustering, etc.

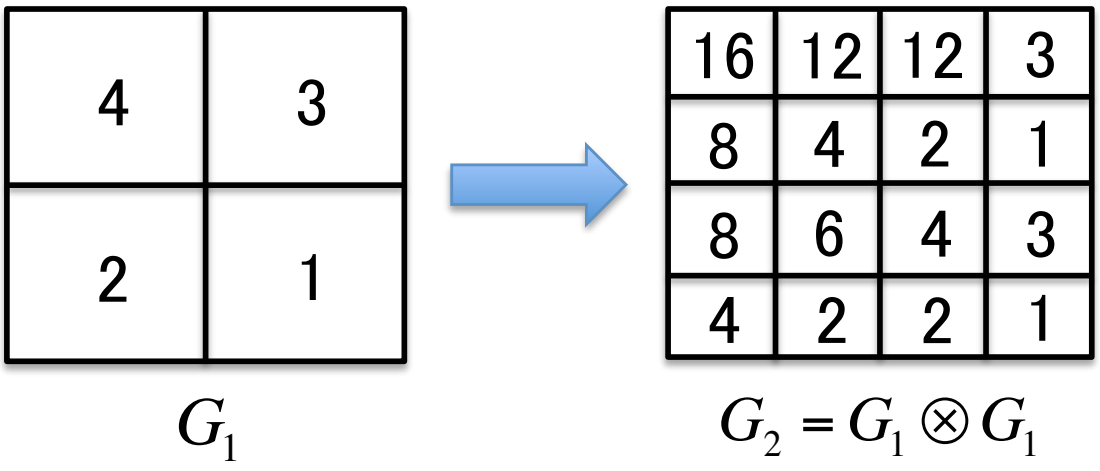

FIGURE 2.7: An example of Kronecker graph

Kronecker Graph [40] is a graph model that has similar properties to real world graphs and employs the recursive matrix (R-MAT) model. Applying Kronecker Product to a base matrix, we can generate an adjacency matrix of a Kronecker graph. Let $A = (a_{ij})$ be a $m \times n$ matrix, and $B = (b_{kj})$ be a $p \times q$ matrix. Kronecker Product $A \otimes B$ can be defined as follows:

$$A \otimes B = \begin{pmatrix} a_{11}B & \dots & a_{1n}B \\ \vdots & & \vdots \\ a_{m1}B & \dots & a_{mn}B \end{pmatrix}$$

By using this representation, we can describe a Kronecker graph G_k as follows:

$$G_k = \underbrace{G_1 \otimes G_1 \otimes \cdots \otimes G_1}_{k \ times}$$

where k denotes the number of iterations, and G_1 denotes the base matrix with v vertices and e edges. Note that G_k has v^k vertices and e^k edges; thus we can get densification with few parameters. Since the Kronecker Graph can be obtained easily by simply applying iterative product operations to a base matrix, the model is widely used (i.e., the Graph500 benchmark [41]), in order to represent large-scale graphs in real world.

Figure 2.7 shows an example of generation process of Kronecker graph. The left matrix describes G_1, a 2×2 matrix where $g_{11} = 4$, $g_{12} = 3$, $g_{21} = 2$, $g_{22} = 1$. The right matrix describes G_2, a 4×4 matrix where each element is calculated by $g_{ij} \otimes G_1$.

2.3.2 Large-scale Data Processing Applications

This section summarizes large-scale data processing applications, focusing on large-scale graph processing algorithms, followed by large-scale homology search algorithms in metagenomics.

Large-scale Graph Processing Algorithms

There exists a wide range of graph processing algorithms: breadth-first search, Shortest path, PageRank [42], connect component, minimal spanning Tree, finding graph center, bipartite matching etc. Breadth-first search is a strategy for searching a graph. Breadth-first search can be used to solve many problems such as finding all nodes within one connected component, finding the shortest path, computing maximum flow in a flow network. Breadth-first search is also used in Graph500 benchmark [41].

The Graph500 benchmark is a data processing benchmark for supercomputers and clouds launched in 2010, in addition to the original Top500 benchmark [9] which is a compute intensive benchmark. Graph500 is launched in order to guide the design of hardware architectures and software systems intended to support data intensive applications, since graph algorithms are a core part of many data analytics workloads. The Graph500 benchmark mainly consists of two kernels: Graph Construction and Breadth-First Search. Before the graph construction, the data generator will construct a list of edge tuples containing vertex identifiers. Each edge is undirected with its endpoints given in the tuple as source vertex and destination vertex. The number of input vertices and input edges are described by two parameters: SCALE and edgefactor. SCALE is the logarithm base two of the number of vertices, and edgefactor is the ratio of the graph's edge count to its vertex count. Then, the total number of vertices can be described as $N = 2^{SCALE}$ where N is the total number of vertices, and the number of edges can be described as $M = edgefactor * N$ where M is the total number of edges. The graph construction kernel may transform the edge list to any data structures, including sparse matrix formats such as Compressed Row Storage (CSR) format and linked lists, that are used for the remaining kernels. After the graph construction, a breadth-first search (BFS) of a graph starts with a single source vertex as a fundamental method on which many graph algorithms are based. After each search, run a function that ensures that the discovered breadth-first tree is correct. In order to compare the performance of Graph500 search implementations across a variety of architectures, a performance metric is adopted. In the similar manner as floating-point operations per second (FLOPS)

measured by LINPACK benchmark in Top500, a performance rate called traversed edges per second (TEPS) is defined.

Shortest path algorithms are applied to automatically find directions between physical locations, such as driving directions on road networks. The shortest path problem is categorized into two variations; single-source shortest path problem and all-pairs shortest path problem. There have been a lot of efforts on developing efficient shortest path algorithms; such as Dijkstra's algorithm, Bellman-Ford algorithm, A* algorithm for solving the single-source shortest path problem, as well as Floyd-Warshall algorithm and Johnson's algorithm for solving all-pairs shortest path problem.

PageRank is an algorithm developed and used by Google Search to rank websites in their search engine results. PageRank is a way of measuring the importance of website pages. PageRank works by counting the number and quality of links to a page to determine an estimate of how much the website is important, with an underlying assumption that more important websites are likely to have more incoming links from other websites. The PageRank computations require several iterations through the collection to adjust approximate PageRank values to more closely reflect the theoretical true value. A probability is expressed as a numeric value between 0 and 1. Let p be a PageRank eigenvector of n web pages; the PageRank algorithm satisfies the following characteristic equation:

$$p = (cE^T + (1 - c)U)p$$

where c denotes a dumping factor, set to 0.85 in typical configuration, E denotes a row-normalized adjacency matrix, and U denotes a matrix with all elements set to $1/n$. In order to acquire the next PageRank eigenvector p^{next}, we initialize p^{cur} and set all the elements to $1/n$, then we calculate $p^{next} = (cE^T + (1 - c)U)p^{cur}$. We continue the multiplication until p converges.

Graph processing implementations can be categorized into three types: pure implementation of a graph algorithm, MapReduce-based framework, and Bulk Synchronous Parallel (BSP) [43]-based framework. Pure implementation of a graph algorithm is developed for achieving high performance for a specific graph algorithm. Implementations of Graph500 [5] are instances of pure implementations of breadth first search algorithm. A graph processing framework consists of some built-in graph processing algorithms, but also provide API to build new algorithms and extend the framework. An instance of MapReduce-based graph processing framework is PEGASUS [6], a framework implemented on top of Hadoop. We also implement a MapReduce-based graph processing

framework on GPUs. As for BSP-based graph processing frameworks, Pregel [44] is proposed as a first BSP-based graph processing framework as an alternate graph processing framework to MapReduce, since not all graph algorithms can be solved with MapReduce efficiency. There exists a number of BSP-based graph processing implementations, such as Apache Giraph [45], Apache Hama [46], Apache GraphLab [47], and ScaleGraph [48]. Giraph and Hama both work on top of HDFS, and Giraph is implemented on top of Hadoop MapReduce while pure BSP framework is implemented in Hama. GraphLab is a high performance distributed graph processing framework written in C++. There also exists a disk-based large-scale graph computation framework called GraphChi [49]. ScaleGraph is a high performance distributed graph processing framework written in X10.

There are several formats to store graphs. Basically there exists three types of formats to store graphs; in a flat file as pairs of vertex id and connected vertices ids to the vertex, in a relational database using referencing tables or join tables, or using a specialized format for graphs. A flat file is typically stored as an adjacency list for sparse graph and an adjacency matrix for dense graph. Main difference between relational database and graph database is that graph database has direct pointers from a vertex to its any adjacency vertices.

There exists graph databases for storing graph dataset in specialized formats, such as Neo4j [50], GraphBase [51], InfiniteGraph [52], AllegroGraph [53], FlockDB [54]. Neo4j is a disk-based Java persistence graph database. InfiniteGraph is a distributed graph database in Java designed to handle very high throughput. FlockDB is a distributed fault-tolerant graph database for managing wide but shallow network graph, initially used by Twitter to store relationships between users. There is also a benchmarking platform for graph stores called XGDBench [55], written in X10. XGDBench supports to benchmark graph databases by generating realistic graph data and analyzing the community structures of synthetic graphs.

Large-scale Homology Search Algorithms

Homology search or alignment search in metagenomics is an approach to identify genes based upon homology with genes that are already publicly available in sequence databases by using a search algorithm. Homology search is used in the field of Metagenomics, the study of genetic material recovered directly from environmental samples for advancing knowledge in a wide variety of application domains, such as medicine, engineering, agriculture, ecology. Homology search algorithms are used as tools for life science researchers

to gain a set of high-scoring pairs from an exhaustive list of protein coding sequences similar to a given query sequence, such as the amino-acid sequence of different proteins or the nucleotides of DNA sequences.

BLAST (Basic Local Alignment Search Tool) [56, 57] is proposed as a fast homology search algorithm and its implementation is widely used as a standard homology search tool. BLAST applies a heuristic algorithm much faster than previous approaches such as a full alignment procedure using the Smith-Waterman algorithm [58] or FASTA [59].

There have been a lot of efforts for improving BLAST [60, 61]. These efforts achieve speedup from the BLAST algorithm by improving search algorithms. Some of the authors also make efforts on accelerating BLAST. GHOSTX [62] adopts the seed-extend alignment algorithm used by BLAST. GHOSTX achieved approximately 131-165 times faster than BLAST. GHOSTX finds seed that are highly similar segments between database sequences and the query sequence. Next, GHOSTX obtain alignments by extending those seeds without gaps for larger similar regions. Finally, GHOSTX make alignments by extending the seeds with gaps. In order to accelerate the seed search process, GHOSTX constructs suffix array both for the query and the database before the search. In addition, instead of fixing the length of a seed like BLAST, GHOSTX extends it till the matching score exceeds a given threshold to reduce the computation time for untapped extension while not losing the sensitivity.

There exists also an extension of GHOSTX for distributed computing environments. GHOST-MP is built on GHOSTX with MPI library for homology search on supercomputers like K computer and TSUBAME, or general PC clusters. It achieves distributed paralleling search process through a master-worker style. In GHOST-MP's algorithm, it accomplishes I/O optimization for paralleled file system by utilizing locality of database chunks to achieve high speed processing.

2.3.3 MapReduce-based Large-scale Data Processing

There exists MapReduce-based large-scale graph processing including Hadoop-based implementations and MPI-based implementations. PEGASUS [6] is a Hadoop-based graph mining system written in Java. Graph mining algorithms that PEGASUS provides include PageRank, Random Walk with Restart (RWR), connected components, degree, and radius. PEGASUS implements the GIM-V (Generalized Iterative Matrix-Vector

multiplication) graph processing algorithm, which can compute various graph processing algorithms such as PageRank, Random Walk with Restart, and Connected Components using MapReduce. MR-MPI [26] also implements several graph algorithms such as PageRank, triangle finding, connected component identification, Luby's algorithm for maximally independent sets, and single-source shortest path calculation.

There have also been a lot of efforts on MapReduce-based large-scale bioinformatic processing. CloudBLAST [63] provides MapReduce-based bioinformatics applications, which integrates Hadoop, virtual machine, and virtual network technologies to deploy the commonly used bioinformatics tool NCBI BLAST on a WAN-based test bed consisting of clusters.

Chapter 3

Related Work

This chapter summarizes prior large-scale data processing on GPUs especially focusing on graph processing on GPUs, followed by existing work on handling memory overflow including out-of-core CPU processing and out-of-core GPU processing.

3.1 Large-scale Graph Processing

In this section, we review prior work on large-scale graph processing, especially focusing on graph processing on single GPU and multiple GPUs, followed by prior local balance optimization techniques for large-scale graph processing.

3.1.1 Graph Processing on GPUs

Existing GPU-based graph processing techniques have shown that GPU accelerates performance on several graph algorithms, such as BFS [13, 64, 65, 66], PageRank [14], etc. In particular, graph processing frameworks, such as [67, 68, 69], are also accelerated by using a single GPU; however, most of these efforts focus on algorithms for a single GPU. Harish et al. [13] have solved the shortest path problem on GPUs, however, they do not achieve competitive performance compared with CPUs for scale-free graphs and real world networks in the DIMACS challenge since the distribution of the degrees follows a power law which introduces significant load imbalance. Thus, the size of processing graphs in these algorithms reaches around 10 million vertices and 60 million edges.

Several efforts focus on the use of multiple GPUs on a single node for BFS [70, 71, 72], PageRank [72, 73] etc., in which the size of graphs to process reaches around 50 million

vertices and 100 million edges. However, these techniques do not consider communication between multiple nodes nor show the scalability when the graph data size exceeds the CPU memory capacity on a single node.

As for the efforts on multi-node multi-GPU environments, GPU-based implementations of sparse matrix vector multiplication for PageRank [74, 75] and BFS [76] have been proposed. However, these implementations cannot handle GPU memory overflows due to heavy CPU-GPU data transfer overheads.

3.1.2 Load Balance Optimizations

Chhugani et al. [77] propose a work distribution approach for multi-socket platforms, whose approach ensures load-balancing while keeping cross-socket communication low on R-MAT graphs. They see a benefit of about 5-10% for their load-balancing scheme on R-MAT graphs, and as much as 30% performance improvement on stress-case graph, which is a bipartite graph where all vertices are either small or large (at alternate depths) Aydin et al. [78] achieve a reasonable load-balanced graph traversal technique by randomly shuffling all the vertex identifiers prior to partitioning, whose technique also reduces inter-processsor collective communication volume using graph partitioning.

3.2 Out-of-core Memory Management

This section describes prior work on out-of-core processing including techniques on handling memory overflow from CPU host memory and GPU device memory. We also review MapReduce on GPUs which can handle out-of-core execution.

3.2.1 Out-of-core CPU Processing

Several research efforts have explored out-of-core graph processing on CPU. As for CPU-based graph processing on a single node, several techniques, such as sequential I/O optimization [49], data placement optimization [79], and data prefetch optimization [80], have been proposed. These techniques focuses on the utilization of a single node. Thus, distributed computing environments are not supported. Pearce et al. [81] have proposed a CPU-based out-of-core large-scale graph processing technique for distributed computing environments. Their technique introduces a graph partitioning strategy and applies to their multithreaded algorithm using distributed external memory; however,

this algorithm cannot be straightforwardly applicable to GPUs, since this algorithm is highly designed for utilizing multi-core CPUs. The MapReduce [3] programming model has been proposed for processing big data applications with automatic memory/storage hierarchy encapsulation, and Hadoop [7] is one of the widely used MapReduce implementation. MR-MPI [26] is a MPI-based MapReduce implementation on CPU, which employs an out-of-core processing technique including in the sort phase after inter-node data exchanges. These MapReduce implementations are designed for CPU-based distributed environments, while our work focuses on GPU-based large-scale environments.

3.2.2 Out-of-core GPU Processing

Researchers have been working on out-of-core GPU processing algorithms in a wide range of application fields, such as BFS [66], stencil [82], rendering [83], etc. These algorithms have shown GPU accelerations by using out-of-core techniques; however, the scope of these applications is limited on specific algorithms. Out-of-core GPU sorting algorithms, such as a sample-based sorting [84] and a merge-based sorting [85], have also been studied; however, these algorithms are designed for a single node execution. These algorithms also have not well investigated load balancing issues for highly skewed data such as real world graphs. There also exists work on I/O issues from a GPU to filesystems [86]; however, they have not conducted experiments on realistic large-scale applications such as graph processing.

3.2.3 MapReduce on GPUs

The MapReduce model can provide out-of-core processing with simple application interfaces. There exists a generalized graph processing algorithm for the MapReduce model called GIM-V (we explain the details in Section 4.3) and its Hadoop-based implementation [6]. However, the implementation does not show good performance due to heavy overheads derived from the Hadoop framework. GPMR [34] is a multi-GPU MapReduce library supporting out-of-core GPU execution on distributed computing environments. They propose how GPUs can be applied to MapReduce model and show good scalability using some dozens of nodes. It is not clear, however, whether large-scale graph processing using MapReduce model scales well with the addition of hundreds of GPUs. Also, users need to set chunk size of input data by hand so that each chunk fits on GPU memory capacity. In addition, the sort phase in GPMR is executed on CPUs when the size of input data exceeds the capacity of the GPU memory, instead of executing on

GPUs. Besides, the performance studies on CPU vs. GPU comparison have not been sufficiently conducted, especially in the out-of-core situation.

3.2.4 Balance between Scale-up and Scale-out

Earlier work have seen effectiveness of scale-out by showing good scalability commodity clusters consisting of hundreds to thousands of machines [3, 6]. On the other hand, several prior work also show that a single scale-up machine can perform competitive with or better than scale-out clusters [87, 88]. These work suggest that the scale-up approach can perform better than the scale-out approach in terms of performance, cost, power, server density etc. However, these work do not consider the balance of scale-up and scale-out on GPU-based heterogeneous computing environments.

Chapter 4

A Scalable Implementation of a MapReduce-based Graph Processing Algorithm on GPUs

This chapter reviews previously studied graph processing techniques on GPUs and provides an in-depth analysis on their limitations. Then, we introduce target graph processing algorithm we use for MapReduce-based execution and a baseline MapReduce implementation on single GPU. Next, based on the observations, we describe our implementation of the graph processing algorithm on single GPU, followed by multi-GPU extension and optimization techniques including load balance optimization among GPUs. At the end, we analyze experimental results, and observe 1.52x performance improvement on 256 nodes with 768 GPUs over performance on 6144 hyper-threaded CPU cores, and 186.6x improvement over an existing popular CPU-based MapReduce implementation, using TSUBAME2.0.

4.1 Motivation

As we described in Section 1.1, recent emergence of extremely large-scale graphs in various application fields, which typically consists of millions to trillions of vertices and 100's millions to 100's trillions of edges, requires fast and scalable analysis by using HPC technologies. MapReduce [3] is a successful programing model for efficient, scalable, and massive data processing in clouds with large-scale commodity compute clusters, which conceals elaborate efforts in distributed systems such as communication between

thousands of nodes, data management for petabyte-scale large data volumes, and fault tolerance. MapReduce is also applied to graph processing with petabyte-scale data; PEGASUS [6], which is an Hadoop [7]-based peta-scale graph mining system that employs the GIM-V (Generalized Iterative Matrix-Vector multiplication) algorithm, has been proposed. GIM-V enables users to describe important graph algorithms, such as PageRank, Random Walk, and Connected Component, without any difficulties in distributed systems. Kang et al. [6] have reported that GIM-V exhibits good scalability in a compute cluster; however, such CPU-based implementation introduces significant performance overheads when we increase the size of a graph.

On the other hand, recent supercomputers employ commodity graphics processing units (GPUs) in addition to compute nodes with general purpose CPUs, since GPUs can provide high peak performance and memory bandwidth for applications with specific computation patterns, while CPUs offer flexibility and generality over wide-ranging classes of applications. This tendency is applied not only to HPC supercomputers, but cloud data centers as well. For example, Amazon EC2 provides Cluster GPU Instances as a GPU-based compute cluster [12]. In such environments, large-scale graph processing is also considered as an important kernel application. Although GPU-based heterogeneous compute clusters are also possible environments for GIM-V-based graph applications, how much the application can be accelerated is an open problem, especially in terms of the performance in the map, reduce, and sort stages. Moreover, we have to consider overflow of graph data from a single GPU memory when applying the GIM-V algorithm to large-scale graphs. Using multiple GPU devices may relax the overflow situation; however, even in such cases, load balance optimization techniques between GPU devices are required for efficient execution of the application.

4.2 Introduction to Graph Processing on GPU

In this section, we describe the existing GPU-based graph processing techniques and point out the issues on processing large-scale graphs using GPUs.

4.2.1 Existing Graph Processing Techniques on GPU

Existing GPU-based graph processing techniques have shown that GPU accelerates performance on several graph algorithms, such as BFS [13, 64, 65, 66], PageRank [14], etc. In particular, graph processing frameworks, such as [67, 68, 69], are also accelerated

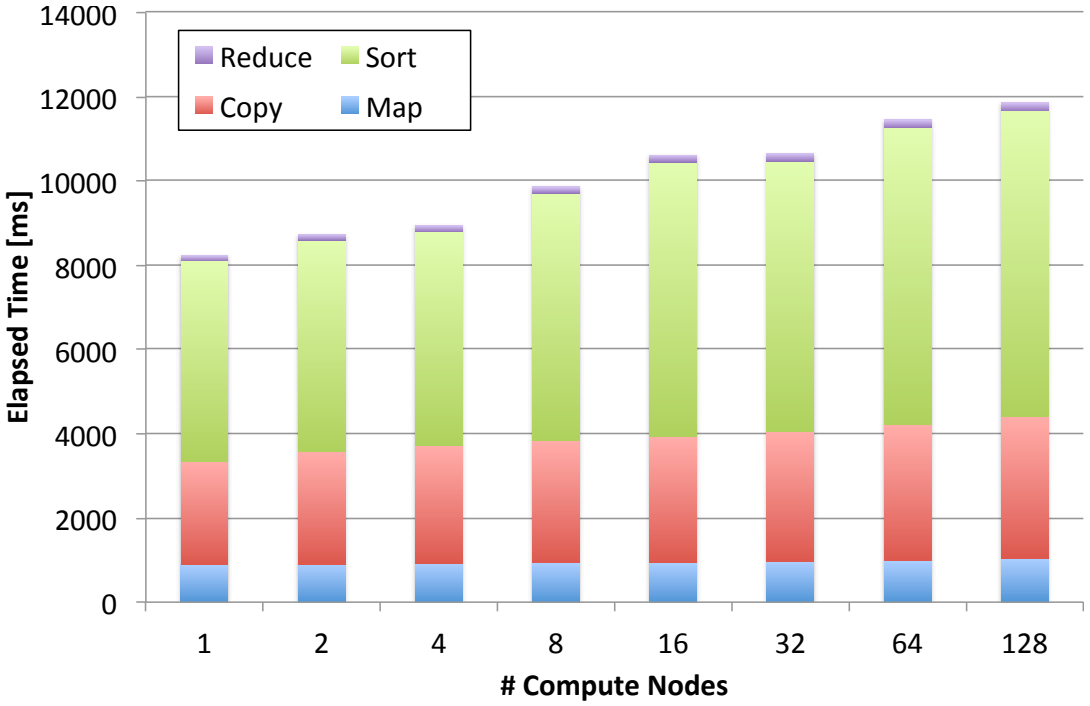

FIGURE 4.1: Performance of MapReduce-based graph processing on CPUs (SCALE 21 per node)

by using a single GPU; however, most of these efforts focus on algorithms for a single GPU. Thus, the size of processing graphs in these algorithms reaches around 10 million vertices and 60 million edges.

Several efforts focus on the use of multiple GPUs on a single node for BFS [70, 71, 72], PageRank [72, 73] etc., in which the size of graphs to process reaches around 50 million vertices and 100 million edges. However, these techniques do not consider communication between multiple nodes nor show the scalability when the graph data size exceeds the CPU memory capacity on a single node.

As for the efforts on multi-node multi-GPU environments, GPU-based implementations of sparse matrix vector multiplication for PageRank [74, 75] and BFS [76] have been proposed. However, these implementations cannot handle GPU memory overflows due to heavy CPU-GPU data transfer overheads.

4.2.2 Issues on Processing Large-scale Graphs on GPUs

Although GPU-based heterogeneous compute clusters are applicable for graph processing applications, how much the application can be accelerated is an open problem. Specifically, It is unclear how much each computation phases in MapReduce-based graph processing on GPU in terms of the performance in the map, reduce, and shuffle stages. As a motivating example, Figure 4.1 shows the execution time of the CPU-based GIM-V implementation when we vary the size of a graph. In this figure, the shuffle stage, where the output of map stage is sorted and forwarded to the input of the reduce stage, is divided into two stages: copy and sort. After the map stage is finished, the output of the map stage is hashed and then transferred via MPI between multiple processes in the copy stage, then the received output is sorted by each node in the sort stage. Here we see significant performance overheads in the map and sort stages, whose overheads may affect performance of graph processing with further large size even if we run the program using multiple compute nodes.

Another significant issue on processing large graphs on GPUs is considered that how to manage graph data whose size exceeds the capacity of GPU memory with minimal performance overheads. As explained in the previous section, GPU memory generally has the smaller capacity than CPU memory, and computation on GPUs requires to transfer data between CPU memory and GPU memory. Thus, when we naively apply the graph algorithms to GPUs, data transfers dominantly disturb efficient graph processing. In particular, when the size of the graphs exceeds the capacity of device memory on GPUs, the number of data transfers drastically increases for executing dependent graph kernels. As a motivating example, Figure 4.2 shows an adjacency matrix and sorted vertex distribution of a Kronecker graph in SCALE 14 with $A = 0.57, B = 0.19, C = 0.19, D = 0.05$ generated by the graph generator included in Graph500 reference implementation. The top of Figure 4.2, the x-axis indicates source vertex index and the y-axis indicates destination vertex index for edges. Plots indicate that the graph contains edges with associated source and destination vertices. The bottom of Figure 4.2, the x-axis indicates vertex index and the y-axis indicates the number of edges connected with. Because of the nature of the Kronecker graph, the adjacency matrix and the vertex distribution indicate the graph is composed of a highly skewed edge distribution where some vertices are connected with large number of vertices while the other vertices are connected with a few number of vertices.

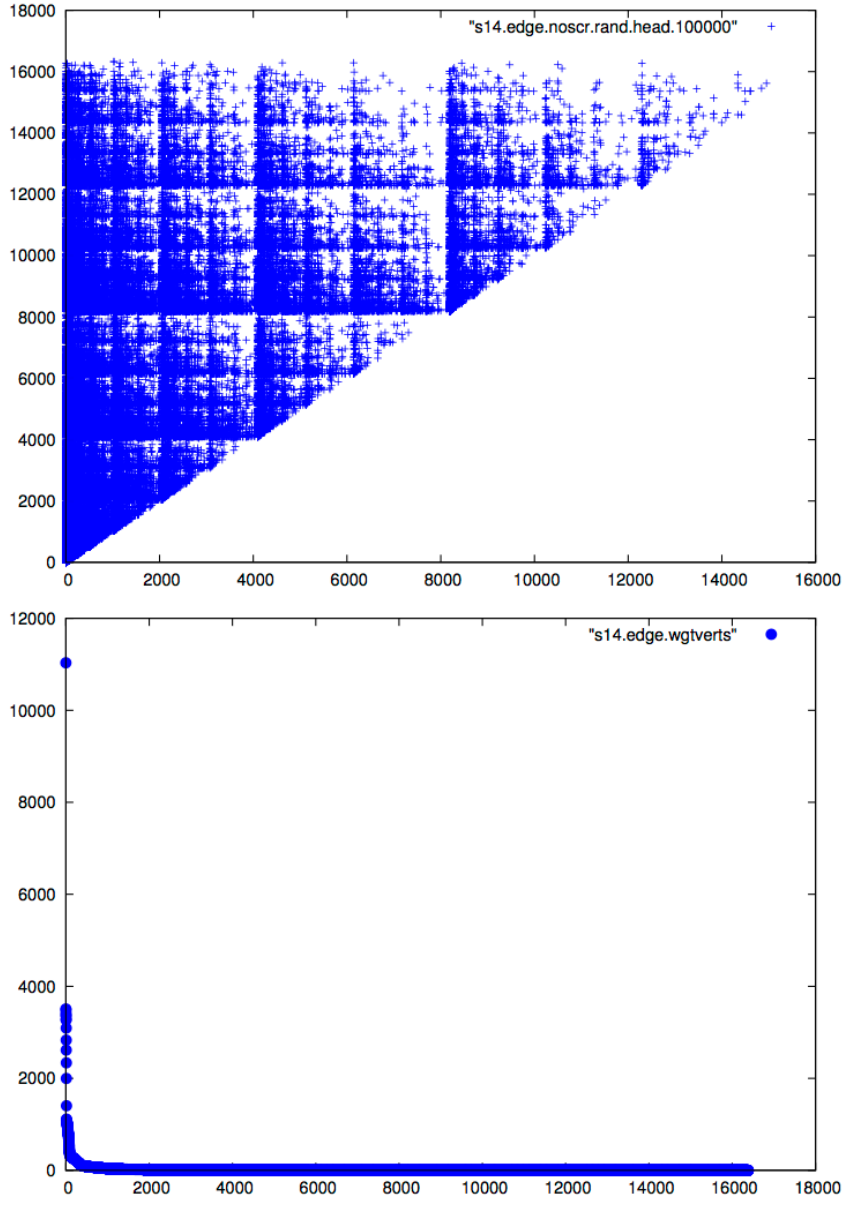

FIGURE 4.2: (Top) Adjacency matrix of a Kronecker graph (SCALE 14), (Bottom) Sorted vertex distribution of a Kronecker graph (SCALE 14)

4.3 GIM-V: Target Graph Processing Algorithm

GIM-V (Generalized Iterative Matrix-Vector multiplication) [6] is a general expression of matrix-vector multiplication with iterative operations for MapReduce-based large-scale graph processing. Let $M = (m_{i,j})$ be a matrix of size $n \times n$, and $v = (v_i)$ be a vector of size n, where $i, j \in \{1, ..., n\}$. Matrix-vector multiplication is described as follows:

$$M \times v = v' \text{ where } v'_i = \sum_{j=1}^{n} m_{i,j} v_j$$

Here the above expression is described by using three operators: *combine2*, *combineAll*, and *assign*:

combine2: Multiply $m_{i,j}$ and v_j.

combineAll: Sum the results of *combine2* for vertex i.

assign: Update v_i to the new result v'_i.

By introducing the operator $\times_G$, we can define the GIM-V algorithm as follows:

$$\begin{aligned} &v' = M \times_G v \\ &\text{where } v'_i = assign(v_i, combineAll_i(\{x_j \mid j = 1..n, \\ &\qquad\qquad \text{and } x_j = combine2(m_{i,j}, v_j)\})) \end{aligned}$$

We iterate the above operation until satisfying a convergence condition defined by graph algorithms such as PageRank, Random Walk, Connected Component, etc. We can describe these graph algorithms by defining the above three operators.

As an example, here we describe the PageRank algorithm [42], which is a well-known algorithm for scoring the relative importance in web pages, Let p be a PageRank eigenvector of n web pages; the PageRank algorithm satisfies the following characteristic equation:

$$p = (cE^T + (1 - c)U)p$$

where c denotes a dumping factor, set to 0.85 in typical configuration, E denotes a row-normalized adjacency matrix, and U denotes a matrix with all elements set to $1/n$. In order to acquire the next PageRank eigenvector p^{next}, we initialize p^{cur} and set all the elements to $1/n$, then we calculate $p^{next} = (cE^T + (1 - c)U)p^{cur}$. We continue the

Algorithm 1 GIM-V Stage 1. Source: Kang et al. (2009) [6]

Require: Matrix $M = \{(id_{src}, (id_{dst}, mval))\}$,
Vector $V = \{(id, vval)\}$
Ensure: Partial vector $V' =$
$\{(id_{src}, \texttt{combine2}(mval, vval))\}$

```
1: Stage1-Map(Key k, Value v);
2: if (k,v) is of type V then
3:    Output(k,v);            //(k: id, v: vval)
4: else if (k,v) is of type M then
5:    (id_dst, mval) ← v;
6:    Output(id_dst, (k, mval));          //(k: id_src)
7: end if
8: Stage1-Reduce(Key k, Value v[1..m]);
9: saved_kv ← [ ];
10: saved_v ← [ ];
11: for all v ∈ v[1..m] do
12:    if (k,v) is of type V then
13:       saved_v ← v;
14:       Output(k, ("old", saved_v));
15:    else if (k,v) is of type M then
16:       Add v to saved_kv          //(v:(id_src, mval))
17:    end if
18: end for
19: for all (id'_src, mval' ∈ saved_kv) do
20:    Output(id'_src, ("new", combine2(mval', saved_v)))
21: end for
```

multiplication until p converges. Based on the above descriptions, we can define the three operations as follows:

$$combine2(m_{i,j}, v_j) = c \times m_{i,j} \times v_j$$
$$combineAll_i(x_1, \ldots, x_n) = \frac{(1-c)}{n} + \sum_{j=1}^{n} x_j$$
$$assign(v_i, v_{new}) = v_{new}$$

The GIM-V algorithm can be implemented using two MapReduce stages: GIM-V Stage1 and Stage2, whose pseudo codes are shown in Algorithm 1 and 2. In these algorithms, id represents an index of vertices, and id_{src} and id_{dst} represent a source index and a destination index of edges respectively. The GIM-V Stage1 performs the *combine2* operation by combining m_{ij} of the matrix M and v_j of the vector v, and outputs key-value pairs, where the key denotes the source vertex id i and the value denotes the partially combined result $x_j = combine2(m_{ij}, v_j)$. Then the output of the GIM-V Stage1

Algorithm 2 GIM-V Stage 2. Source: Kang et al. (2009) [6]

Require: Partial vector $V' = \{(id_{src}, vval')\}$
Ensure: Result vector $V = \{(id_{src}, vval)\}$
1: `Stage2-Map(Key k, Value v);`
2: Output(k, v);
3: `Stage2-Reduce(Key k, Value v[1..m]);`
4: $new\_v \leftarrow [\ ]$;
5: $old\_v \leftarrow [\ ]$;
6: **for all** $v \in v[1..m]$ **do**
7: $(tag, v') \leftarrow v$;
8: **if** $tag ==$ "*old*" **then**
9: $old\_v \leftarrow v'$;
10: **else if** $tag ==$ "*new*" **then**
11: Add v' to new_v;
12: **end if**
13: **end for**
14: Output$(k,$ `assign`$(old\_v,$ `combineAll`$_k(new\_v)))$;

is forwarded to the input of the GIM-V Stage2. The GIM-V Stage2 combines all partial results from the GIM-V Stage1 by applying $combineAll_i(x_j \mid j = 1 \ldots n)$, and assigns the new vector v_{new} to the old vector v_i by applying $assign(v_i, combineAll_i(x_j \mid j = 1 \ldots n))$. These two MapReduce operations are iterated until the application-specific convergence criterion is met.

4.4 Mars: a MapReduce Implementation on Single GPU

Mars [27, 36] is a library-based MapReduce framework for a single GPU device, whose library provides similar APIs to CPU-based MapReduce frameworks. By writing map and reduce operations on top of CUDA kernels through these APIs, users can run MapReduce programs on a GPU device. We use Mars as a base of our GPU-based GIM-V implementation described in Section 4.5 and 4.6.

Figure 4.3 shows the overview of the original Mars library. Similar to the existing MapReduce frameworks, Mars basically has two stages: map and reduce. Before starting the map stage, Mars reads input data from secondary storage and converts the input data to key-value pairs as a preprocessing step.

In the map stage, the split operator firstly divides the input key-value pairs into multiple fragments such that the number of fragments is equal to that of GPU threads. Next each GPU thread calculates the number and the size of intermediate records to allocate

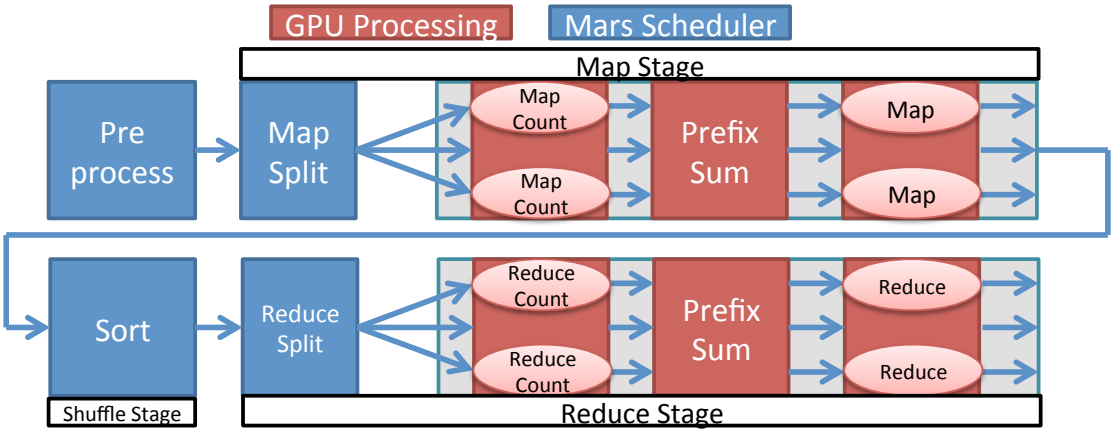

FIGURE 4.3: Mars architecture

memory region on a GPU device. Then the runtime executes GPU-based PrefixSum in order to get the size and the position of the output. After the preparation, a CPU process allocates a buffer in GPU device memory and invokes a GPU kernel, in which each thread executes a map function defined by users.

After the map stage is finished, the intermediate key-value pairs are sorted so that the pairs with the same key are stored consecutively. We call the stage between the map and reduce stages, the shuffle stage. Mars uses GPU-based bitonic sort [89], whose time complexity is $O(nlog^2(n))$, in the shuffle stage, since bitonic sort can efficiently utilize the parallelism of GPU.

Then, in the reduce stage, the split operator divides the sorted key-value pairs into multiple fragments of similar size, whose pairs with the same key belong to the same fragment. Note that the number of fragments is equal to that of GPU threads. After the reduce stage is finished, the output of key-value pairs from all GPU threads are finally merged into a single buffer.

Mars runtime automatically assigns key-value pairs to each thread by a scheduler running on a CPU process and invokes massively parallel GPU threads for map and reduce tasks, respectively. In order to avoid conflicts in concurrent writes to an output buffer by GPU threads, Mars employs a lock-free scheme that manages concurrent writes among different threads and enables Mars to accurately execute massively parallel GPU threads while reducing synchronization overheads.

Mars employs array data structure for input, intermediate, and output records, each of which has three arrays for key, value, and index. The index array consists of an entry of $<$ key offset, key size, value offset, value size $>$. In order to get a key-value pair in key

and value arrays, we need to access the index array to get the offset and the size of the corresponding key-value instance.

4.5 Single GPU Implementation

This section introduces our proposal of an implementation of GIM-V algorithm on MapReduce using a single GPU.

4.5.1 Basic Idea

In order to implement the GIM-V algorithm for GPU environments, we implement the GIM-V algorithm on the existing Mars library that supports MapReduce execution on a single GPU environment. Our implementation employs CUDA and C++, although the original PEGASUS library is written in Java. This section describes the details of our implementation.

4.5.2 GIM-V on Single-GPU MapReduce

Based on the Mars library for GPU environments, we implement the GIM-V algorithm described in Section 4.3. Figure 4.4 shows the workflow of our GIM-V implementation, in which we connect multiple MapReduce stages as follows:

STEP1 Preprocessing (Reading input)

STEP2 MapReduce Stage1 (*combine2*)

STEP3 MapReduce Stage2 (*combineAll* and *assign*)

STEP4 Convergence test

STEP5 Next iteration if not converged (go to **STEP2**)

First, the application reads an edge list and generates initial vertex vectors in the preprocessing step. Next, we conduct two MapReduce stages as described in Section 4.3. Namely, the MapReduce Stage1 performs the *combine2* operation and forwards the result to the input of the MapReduce Stage2. Then the MapReduce Stage2 performs the *combineAll* and *assign* operations. Finally, we conduct a convergence test.

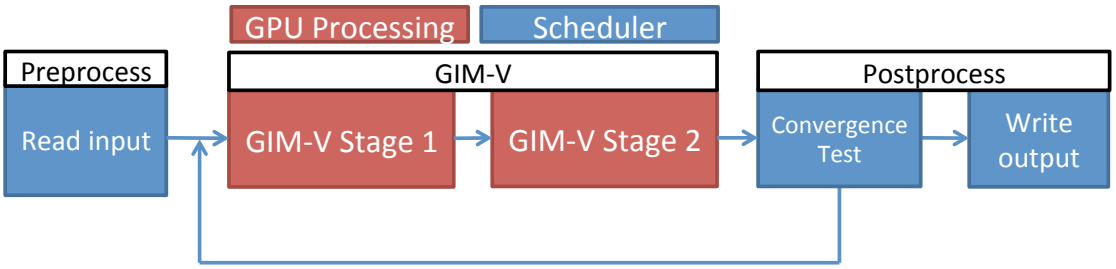

FIGURE 4.4: GIM-V workflow

4.6 Multi-GPU Extension

This section describes multi-GPU extension of GIM-V on GPU MapReduce from a single GPU MapReduce-based GIM-V provided in Section 4.5. We first extend the single GPU MapReduce implementation to multi-GPU using MPI, then we implement GIM-V on our multi-GPU MapReduce extension. We also apply several optimization techniques including a task-scheduling-based load balance optimization.

4.6.1 Basic Idea

In order to implement the GIM-V algorithm for multi GPU environments, we apply the following techniques to the existing Mars library that supports MapReduce execution on a single GPU environment:

- We extend Mars to run on a multi-GPU environment using MPI.
- We implement the GIM-V algorithm on our muti-GPU-based Mars library.
- We apply a load balance optimization technique based on task scheduling to our multi-GPU-based GIM-V application.

This section describes the details of our implementation and optimization technique.

4.6.2 Mars Extension for Supporting multi-GPU devices

In order to enable the existing Mars library to run on a multi-GPU environment, we extend the shuffle layer of the original Mars library to support inter-process communication. Figure 4.5 shows the overview of our extended Mars implementation, in which we divide the shuffle stage into two parts: data transfer between processes (*copy*) and sorting in a single process (*sort*). First, each process sends the outputs of the map stage to

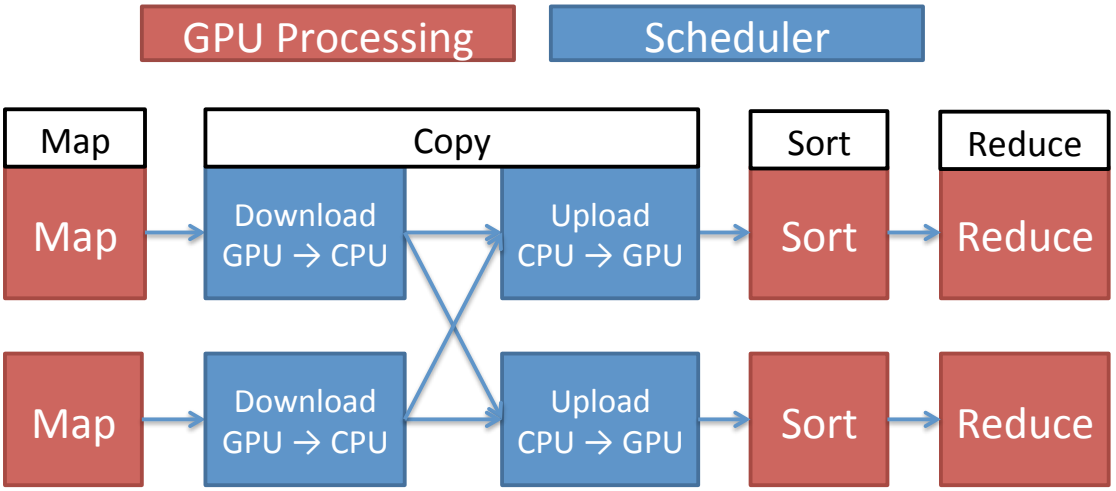

FIGURE 4.5: Implementation overview of MapReduce on multiple nodes

destination processes that are determined by hash values generated from corresponding keys. Our implementation basically determines a hash value by the remainder acquired by dividing a key by the number of processes; however, when we use graph partitioning for load balance optimization, we determine a hash value by the partition id acquired by graph partitioning. After sending the outputs, each destination process receives the outputs of the map stage and conducts sorting of the received outputs. We employ the `MPI_Alltoallv` function for implementing this feature.

We also implement a parallel I/O feature using MPI-IO in order to improve I/O throughput between host memory and secondary storage. When the application reads an input edge list, each process sets the start and end positions to the range that the process is responsible for conducting I/O operations. Then, the processes read the part of the input edge list in parallel. Note that the parallel I/O not only reduces the time for I/O operations but enables the application to handle large-scale graph data whose size extends memory capacity on a single node.

We assign input data onto multi-GPU statically so that we can avoid additional overhead of data movements among GPUs. Intermediate data after the copy stage is also assigned to GPUs statically. We further discuss advantages and disadvantages of our task scheduling strategy in Section 6.1.3.

4.6.3 GIM-V on Multi-GPU MapReduce

Based on the extended Mars library for multi-GPU environments, we implement the GIM-V algorithm described in Section 4.3. Figure 4.6 shows the workflow of our GIM-V

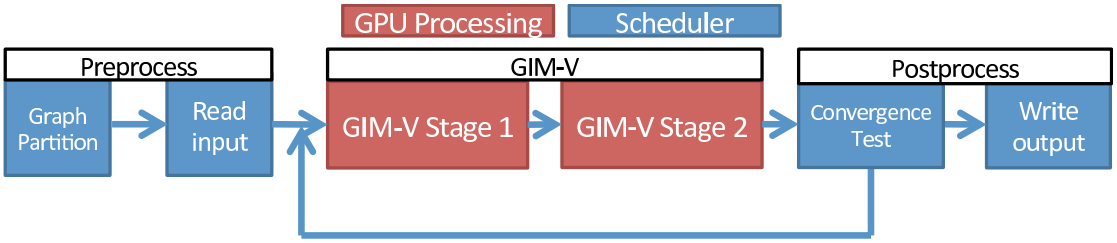

FIGURE 4.6: GIM-V workflow on multi-GPU Mars

implementation, in which we connect multiple MapReduce stages as follows:

STEP1 Preprocessing (Graph partitioning and reading input data)

STEP2 MapReduce Stage1 (*combine2*)

STEP3 MapReduce Stage2 (*combineAll* and *assign*)

STEP4 Convergence test

STEP5 Next iteration if not converged (go to **STEP2**)

First, the application reads an edge list and generates initial vertex vectors in the preprocessing step. When we apply graph partitioning, we divide a graph to sub-graphs after reading the edge list, and each process holds a part of the edge list. Next, we conduct two MapReduce stages as described in Section 4.3. Namely, the MapReduce Stage1 performs the *combine2* operation and forwards the result to the input of the MapReduce Stage2. Then the MapReduce Stage2 performs the *combineAll* and *assign* operations. Finally, we conduct a convergence test that employs two detection mechanism phases. In the first phase of the test, each process sums the number of vertices that meet the convergence condition after finishing the reduce stage. Then the master process aggregates the number of the converged vertices from the processes using the `MPI_Allreduce` function. We compare the aggregated value with the total number of vertices in the graph. If the values differ, we iterate MapReduce operations (MapReduce Stage1 and Stage2); otherwise we terminate the GIM-V algorithm.

We apply several optimization techniques to the original Mars implementation for scalable GIM-V processing. First, we change data structure of Mars, since Mars has metadata (size) in addition to payload (actual data) of key-value pairs, whose structure introduces heavy performance degradation and wastes memory. We eliminate metadata (size) from the original Mars implementation and use fixed size payload (actual data) to reduce the amount of data transfer. Second, we change the thread allocation in

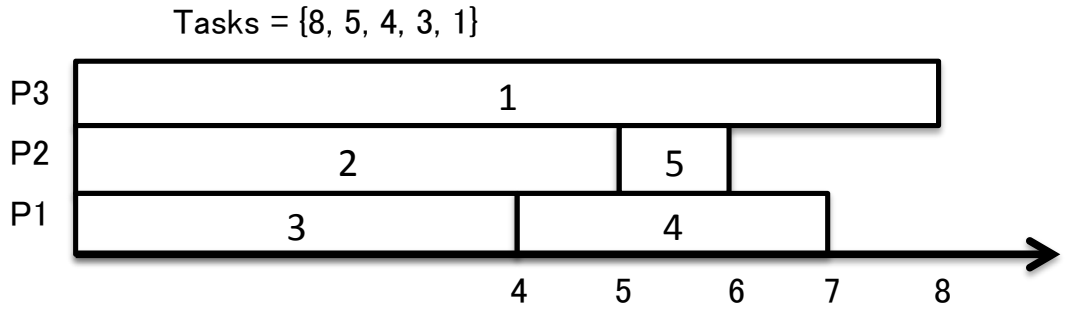

FIGURE 4.7: An example of the LPT schedule

the reduce stage. The original Mars implementation introduces significant performance overheads in the reduce stage due to lack of massive parallelization of CUDA threads; that is, Mars assigns a single CUDA thread to a reduce operation for values to a single key in the reduce stage, whose situation introduces inefficient reduce execution. Our implementation allocates multiple CUDA threads to a single reduce operation in *combine2* in MapReduce Stage1.

4.7 Optimization Techniques

This section describes optimization techniques on top of our multi-GPU-MapReduce-based GIM-V provided in Section 4.6. We first introduce a task-scheduling-based load balance optimization, followed by implementation techniques of a GIM-V-based graph algorithm on GPU.

4.7.1 Load Balance Optimization

We use a task scheduling-based graph partitioning, called Longest Processing Time (LPT) schedule [90], for load balance optimization. The purpose of graph partitioning here is to reduce the load imbalance between GPU devices by partitioning a graph into several sub-graphs to minimize the maximum number of edges and vertices, and by distributing the partitioned sub-graphs to each process as the preprocessing step. Our implementation describes a graph as an edge list, in which each edge consists of a pair of source and destination vertices. The straightforward allocation without load balancing may introduce performance overheads in proportion to the imbalance of the number of incoming and outgoing edges. In contrast, the optimized allocation with load balancing minimizes the difference of the number of edges to handle on a GPU device.

Several heuristic load balancing algorithms exist [91], [92], [90] since obtaining optimal solution for the problem is considered to be NP-complete [91]. LPT is an $O(nlogn)$ heuristic algorithm for homogeneous processors; whose algorithm produces a schedule close to the optimal and has the termination time obtained by assigned jobs in decreasing order of the execution times. Graham et al. [90] have reported that the termination time of LPT is at most $\frac{4}{3}$ of the optimal time. Here we show an example of LPT in Figure 4.7. We assume three processors, P1, P2, and P3, and five tasks, whose sizes are 8, 5, 4, 3, and 1. The axis indicates the amount of assigned tasks on each processor. LPT assigns tasks to a processor which contains least amount of tasks, from task1(8) to task5(1). The figure shows the final state after all the tasks has been assigned, where each bar indicates a task and each number inside the bars indicates the order in which the task assigned. By applying LPT to the GIM-V algorithm, we obtain a near optimal partitioning result.

4.7.2 Implementation of a GIM-V-based Graph Algorithm on GPU

We demonstrate an implementation of the PageRank algorithm on top of the GPU-based MapReduce framework with our proposed out-of-core GPU memory management technique. We implement two stages of MapReduce (*Map1-Reduce1* and *Map2-Reduce2* phases) based on the GIM-V algorithm explained in Section 4.3. First, *Map1* phase simply passes input key-value pairs to *Reduce1* phase. Next, the *Reduce1* phase conducts the *combine2* operation. Then, *Map2* phase simply passes the results of key-value pairs to *Reduce2* phase. Finally, the *Reduce2* phase conducts the *combineAll* and *assign* operations. In the *Reduce1* and *Reduce2* phases, we apply the warp-based thread assignments onto key-value scans: lines 11 to 18 and lines 19 to 21 in Algorithm 1 for the *Reduce1* phase, and lines 6 to 13 in Algorithm 2 for the *Reduce2* phase. We use shared memory for efficient warp-based key-value scans. We also apply warp shuffle operations to the *combineAll* operations for fast reduction. The warp shuffle operation is a new feature of the NVIDIA Kepler compute architecture.

4.8 Performance Analysis

In this section we present performance results for our multi-GPU-MapReduce-based large-scale graph processing using GIM-V. We compare performance of our GPU-MapReduce-based implementation with our CPU-MapReduce-based implementation as well as Hadoop;

an existing popular CPU-based MapReduce implementation. We first analyze the results of single GPU performance by comparing with Hadoop and our CPU-based implementation, then analyze multi-GPU performance. We investigate the speedup from our CPU-based MapReduce implementation, performance breakdown, effectiveness of optimizations for GPU utilization, effectiveness of the load balance optimization using Kronecker graphs as well as Twitter graphs, and the speedup from Hadoop.

4.8.1 Single GPU Performance

We conducted performance studies of our GPU implementation of the GIM-V algorithm to answer the question; comparison with a CPU-based implementation. We also include a comparison of our GIM-V implementation with the original Hadoop-based implementation using PEGASUS.

In our experiments, we use artificial Kronecker graphs, which are characterized by *SCALE* and *edge factor* parameters, to represent real world networks with scale-free and power-law distribution properties. Note that *SCALE* denotes the base 2 logarithm of the number of vertices, and *edge factor* denotes a parameter to represent the total number of edges as *edge factor* $\times 2^{SCALE}$. We set the *edge factor* parameter to 16 in our experimental setting. Our experiments use the Kronecker graph generator in the Graph500 reference implementation [41] [93] to generate adjacency matrices of the Kronecker graphs.

We use single compute node, in which a machine has 2 processors of Intel(R) Core(TM) i7-3930K 3.20GHz (6 cores) CPU running in hyper-threading mode, 16.3GB of main memory, 1 device of NVIDIA Tesla C2050 GPU with 3GB of memory connnected via a PCI-Express 2.0 $\times$ 16 bus, running Scientific Linux release 6.1. We use GCC 4.4.6 for the CPU implementation, and CUDA driver 4.1 and CUDA runtime 4.1 for the GPU implementation.

Comparison with Hadoop-based GIM-V implementation

First, we compare our GPU-based GIM-V implementation with PEGASUS, which is the Hadoop-based original GIM-V implementation, using single compute node. Here we use Hadoop version 0.21.0 and HDFS for the underlying Hadoop's file system. Figure 4.8 shows the elapsed time of one iteration in the GIM-V algorithm, where the x-axis denotes the size of the input graph in SCALE and the y-axis denotes the elapsed time

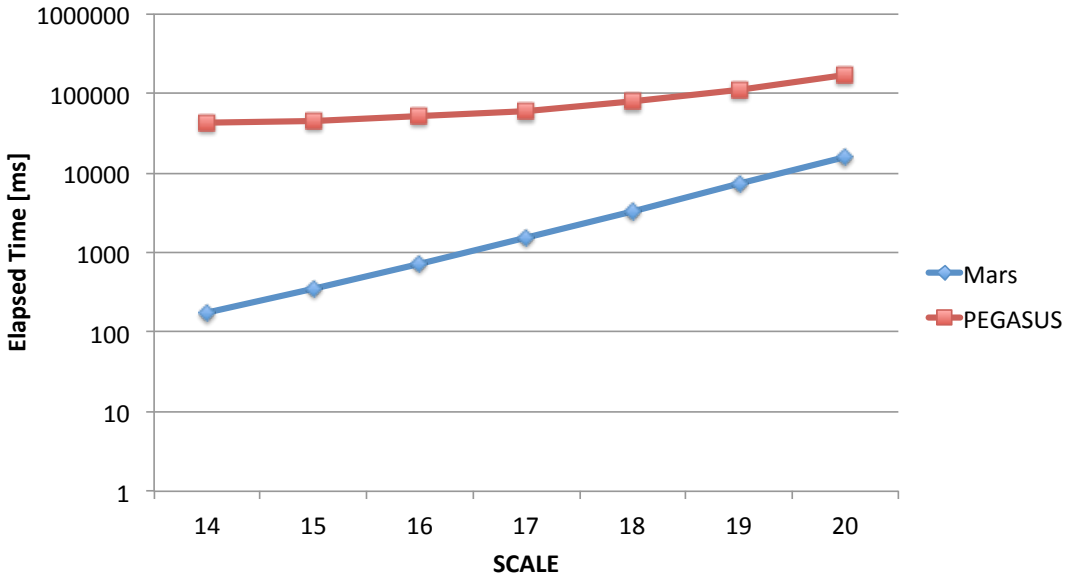

FIGURE 4.8: Mars vs PEGASUS: Elapsed time of an iteration in GIM-V

of an iteration in the GIM-V algorithm in the logarithm scale. We see that our implementation exhibits 8.80 to 39.0 times faster than PEGASUS. The main reason of this significant performance improvement is mainly derived from acceleration by GPUs in the map stages. Figure 4.9 shows the performance comparison in map and reduce stages between our GPU-based implementation and PEGASUS, where the x-axis denotes the size of graph in SCALE and the y-axis denotes the elapsed time of an iteration in the GIM-V algorithm in the logarithm scale. We see that the map stage in our implementation is highly accelerated by GPU. Besides, PEGASUS conducts I/O operations from/to secondary storage in every map and reduce stage, while our implementation only conducts read I/O operations in the preprocessing step and write I/O operations in the end of the computation to output final data to secondary storage. Our implementation forwards output data of the reduce stage to the input of the next map stage by keeping the output on GPU device memory, when the iterative operation continues.

Comparison with CPU implementation

We compare our implementation with a CPU-based implementation to investigate the validity of GPU acceleration. To compare the performance difference between GPU-based implementation and CPU-based one, we also implement the GIM-V algorithm for CPU environments, whose implementation employs a parallelization technique using the POSIX thread library, in addition to our GPU-based implementation. We use the POSIX

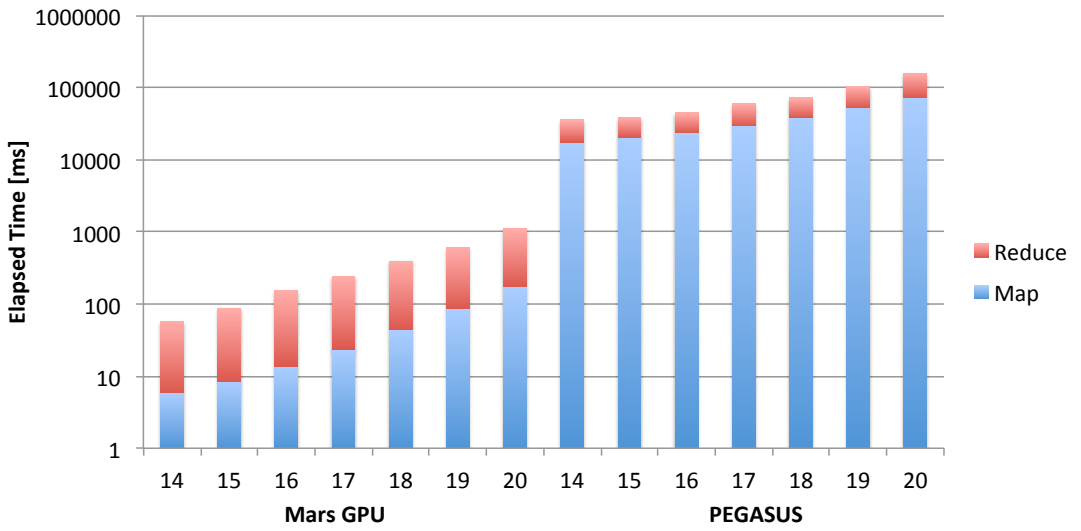

FIGURE 4.9: Mars vs PEGASUS: Breakdown of an iteration in GIM-V

thread library to parallelize map and reduce tasks, and shuffle in a single node. We implement parallelization of map and reduce tasks in straightforward manner by using a simple fork-join model. Shuffle is implemented by quick sort using a work-pile model that suits for the divide and conquer algorithm. Besides the above implementation, our CPU-based implementation optimizes fetching patterns of cache line in map and reduce tasks to avoid cache line conflict which is caused by simultaneous accesses to the same cache line by processes running on different CPU cores.

Figure 4.10 shows the performance results of our CPU and GPU implementations, where the x-axis denotes the size of input graph in SCALE and the y-axis denotes the elapsed time in each stage. Here we describe the GPU-based implementation as *MarsGPU*, and the CPU-based implementation as *MarsCPU*. We use single node in both *MarsGPU* and *MarsCPU* experiments; we use 12 threads in a node running on hyper-threaded cores in *MarsCPU*.

The results in Figure 4.10 exhibit significant performance improvement in map stage; the elapsed time for the map stage in *MarsGPU* achieves 2.72 times faster than that of *MarsCPU*. The reason of this performance improvement is considered that the map stage consists of simple instructions and the input graph data for the stages are comparatively well-balanced, which suites for highly parallelized architecture of GPU.

On the other hand, *MarsGPU* introduces significant performance overheads in the reduce stages compared with *MarsCPU*. The reason for the overheads is derived from the characteristics of the graph; since we use Kronecker graphs for the experiments, the

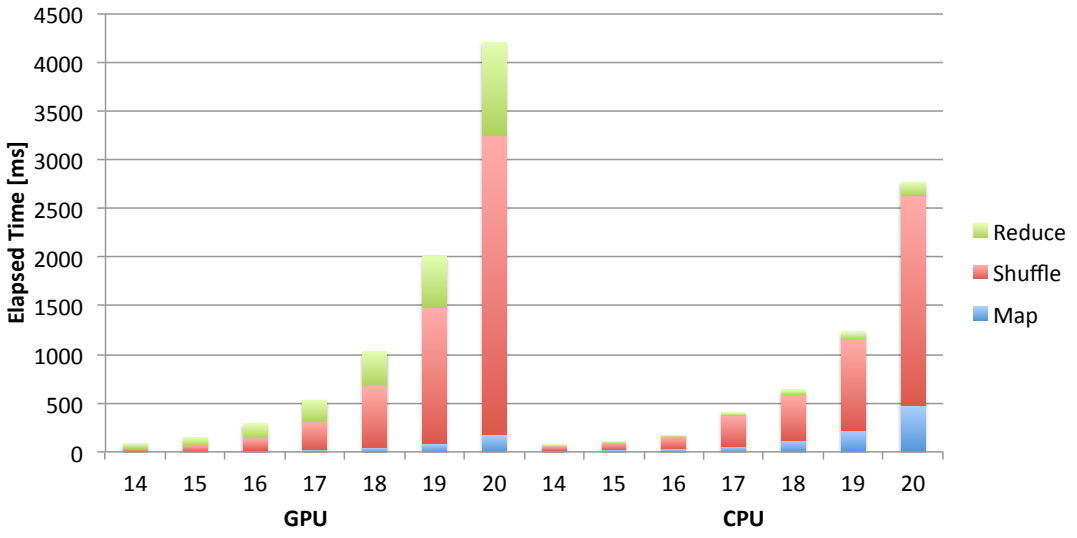

FIGURE 4.10: Mars on GPU and CPU

characteristics of which include the property that few vertices hold a large number of edges, workload imbalance among threads in a GPU device is introduced.

In summary, the above results exhibit that our GPU-based implementation can improve performance of the GIM-V algorithm in map stage. It is also possible to improve the performance of our implementation further by executing the map stage on GPUs and the reduce stage on CPUs.

4.8.2 Multi-GPU Performance

We conducted performance studies of our multi-GPU implementation of the GIM-V algorithm to determine the following: scalability of our GPU-based implementation, performance comparison with a CPU-based implementation, and validity of load balance optimization. We also compare our GIM-V implementation with the original Hadoop-based implementation using PEGASUS.

Evaluation Method

Our experiments use artificial Kronecker graphs, which are characterized by *SCALE* and *edge_factor* parameters, to represent real world networks with scale-free and power-law distribution properties. Note that *SCALE* denotes the base 2 logarithm of the number of vertices, and *edge_factor* denotes a parameter to represent the total number of edges as $edge\_factor \times 2^{SCALE}$. We generate adjacency matrices of the Kronecker graphs with

edge_factor 16 by the Graph500 reference implementation [41, 93]. On top of the GIM-V algorithm, we implemented PageRank as an application for the experiments.

Experimental Environment

We use 256 compute nodes of the TSUBAME2.0 supercomputer [94] located at Tokyo Institute of Technology; each of the machines has 2 processors of Intel Xeon X5670 2.93GHz (6 cores) CPU running in hyper-threading mode, 54GB of DDR3 main memory, 3 devices of NVIDIA Tesla M2050 GPU, each of which has 3GB of discrete GDDR5 memory, and connects to a PCI-Express 2.0 $\times$ 16 bus, and 2 cards of QDR Infiniband HBA (40 Gbps) connected to the dual rail interconnect network with full bisection fat tree, and runs SUSE Linux Enterprise 11 SP1. Files are stored on the Lustre file system (version 1.8), which is configured with 2 MDSs and 8 OSSs with 104 OSTs connected to the IB network. We use Open MPI version 1.4.2 with GNU GCC 4.3.4 for the MPI implementation, and CUDA driver 4.1 and CUDA runtime 4.0 for the GPU implementation.

Comparison with CPU implementation

We compare our GPU-based implementation with a CPU-based implementation to investigate the validity of GPU acceleration. To do so, we also implemented the GIM-V algorithm for multi CPU environments, whose implementation employs a hybrid parallelization technique using MPI and POSIX threads. We use MPI to parallelize the code among compute nodes similar to our multi-GPU implementation, whereas we use POSIX threads to parallelize map, reduce, and sort in a single node. Our implementation parallelizes map and reduce tasks in straightforward manner by using a simple fork-join model. Sorting is implemented by quick sort using a work-pile model that suits well for the divide and conquer algorithm. Besides the above implementation, our CPU-based implementation optimizes fetching patterns of cache line in map and reduce tasks to avoid cache line conflict which is caused by simultaneous accesses to the same cache line by threads running on different CPU cores.

Figure 4.11 shows the weak-scaling performance results of our CPU- and GPU-based implementations, where the x-axis denotes the number of compute nodes and the y-axis denotes the performance in ME/s (mega edges per second) in each stage. Each node has the constant problem size, SCALE 21. Here we describe the GPU-based implementation as *MarsGPU* or *MarsGPU-n* where n denotes the number of GPUs

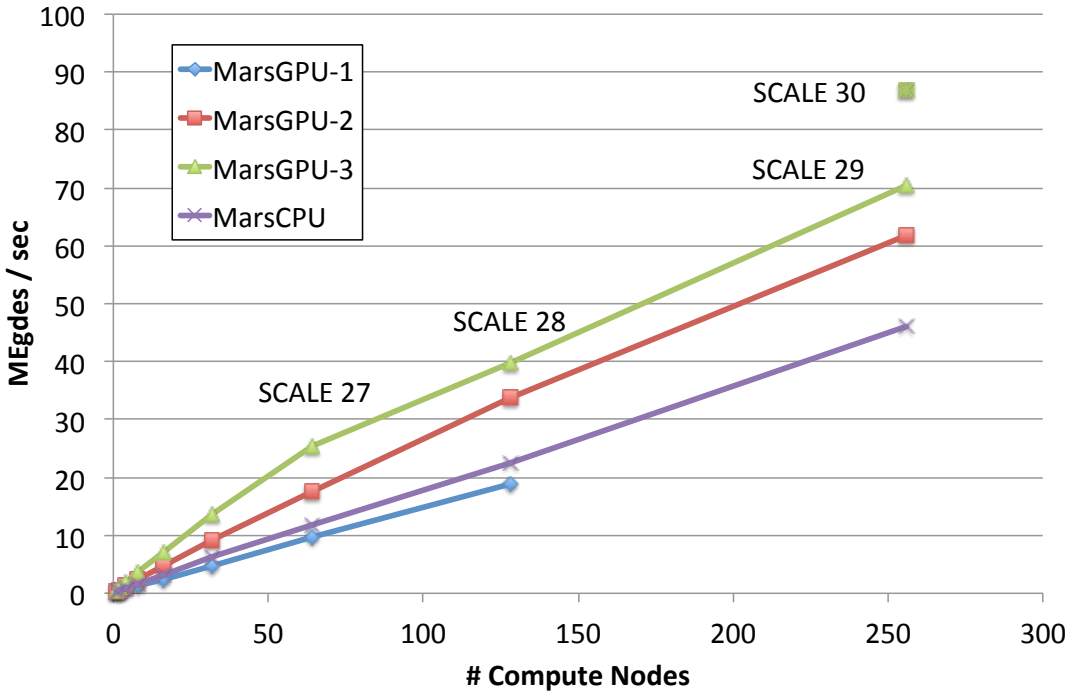

FIGURE 4.11: Weak scaling performance on MarsGPU and MarsCPU (SCALE 21 per node)

per node, and the CPU-based implementation as *MarsCPU*. We use 256 nodes in both *MarsGPU* and *MarsCPU* experiments; we vary the number of GPUs per node from 1 to 3 in *MarsGPU*, while we use 12 threads per node running on hyper-threaded cores using 1 socket in *MarsCPU*. Note that the result of 256 nodes on *MarsGPU-1* is not listed since the input size is too large to fit into the amount of GPU memory. Figure 4.11 also shows the performance of *MarsGPU-3* on SCALE 30 with 256 nodes (SCALE 22 per node). We achieved 87.04 ME/s on SCALE 30 with 256 nodes (6144 hyper-threaded CPU cores, 768 GPUs). The results also exhibit 1.52 times performance improvement in a single iteration on SCALE 29 with 256 nodes.

Figure 4.12 shows the performance breakdown on SCALE 28 at 128 nodes in Figure 4.11. The y-axis denotes the elapsed time in milliseconds. We divide the results of the copy stage into Hash, MPI-comm, and PCI-comm phases; each denotes the time for determining destination for the next reduce stage using a hash function, the time for communication via `MPI_Alltoallv` function, and the time for data transfer between GPU and CPU devices, respectively. Note that PCI-comm includes both data transfers between CPU and GPU devices, i.e., data transfer from GPU to CPU at the start of the copy stage and data transfer from CPU to GPU at the end of the copy stage. The results exhibit that the elapsed time for the map and sort stages in *MarsGPU-3* with 128 nodes

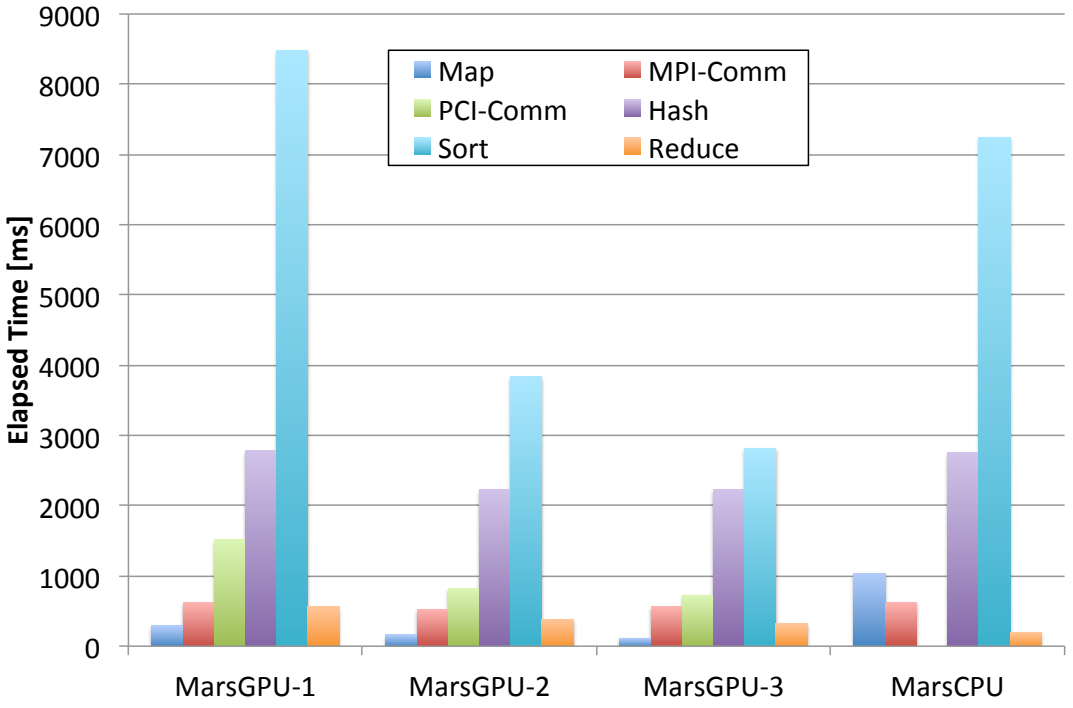

FIGURE 4.12: Performance breakdown on MarsGPU and MarsCPU (SCALE 28,128 nodes)

achieves 8.93 times and 2.58 times faster than those of *MarsCPU* respectively. The reason for this performance improvement is that the map, sort, and reduce stages consist of simple instructions and the input graph data for the stages are comparatively well-balanced, whose configuration suites for highly parallelized architecture of GPU. On the other hand, we observe that *MarsGPU* introduces significant overheads in PCI-Comm, since *MarsGPU* has to transfer data between CPU and GPU devices at the start of the map stage, the copy stage, and at the end of the reduce stage.

Performance Comparison with Naive GPU Implementation

We compare the performance differences between naive GPU implementation and our optimized implementation. Figure 4.13 shows the elapsed time of the map, sort, and reduce stages on *MarsGPU-3* on SCALE 26 with 128 nodes in milliseconds in the logarithm scale.

This figure shows that our optimized implementation performs better than the naive implementation; 1.92 times in map, 1.64 times in sort, and 66.8 times in reduce. This performance benefits come from our optimization techniques described in Section 4.6.3;

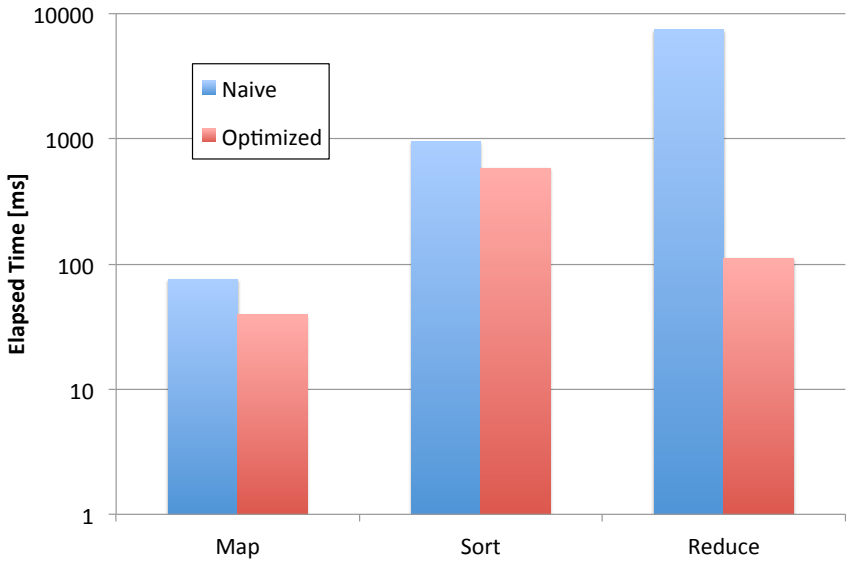

FIGURE 4.13: Performance comparison with naive GPU implementation on MarsGPU-3 (SCALE 26,128 nodes)

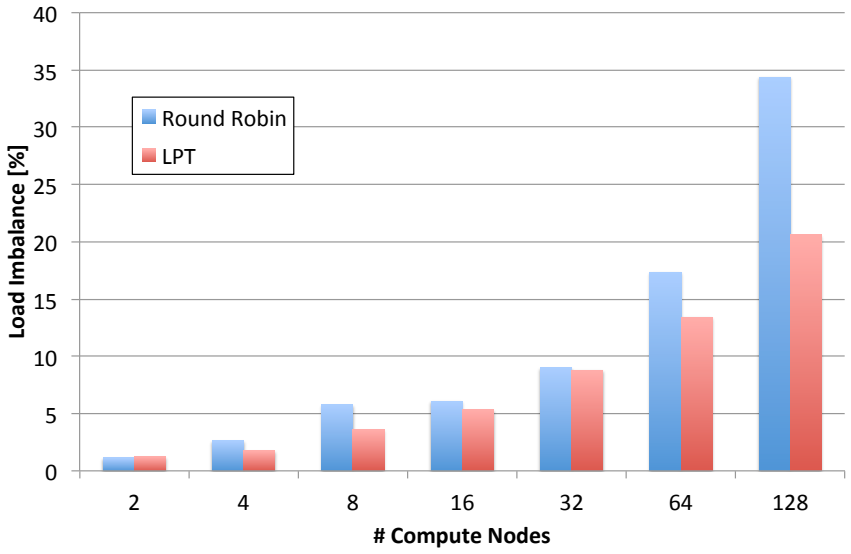

FIGURE 4.14: Load balance: Round robin vs. LPT (SCALE 19 per node)

first, the change of the data structure improves memory access performance and reduces waste memory consumption, and second, the change of the thread allocation also improves performance in the reduce stage.

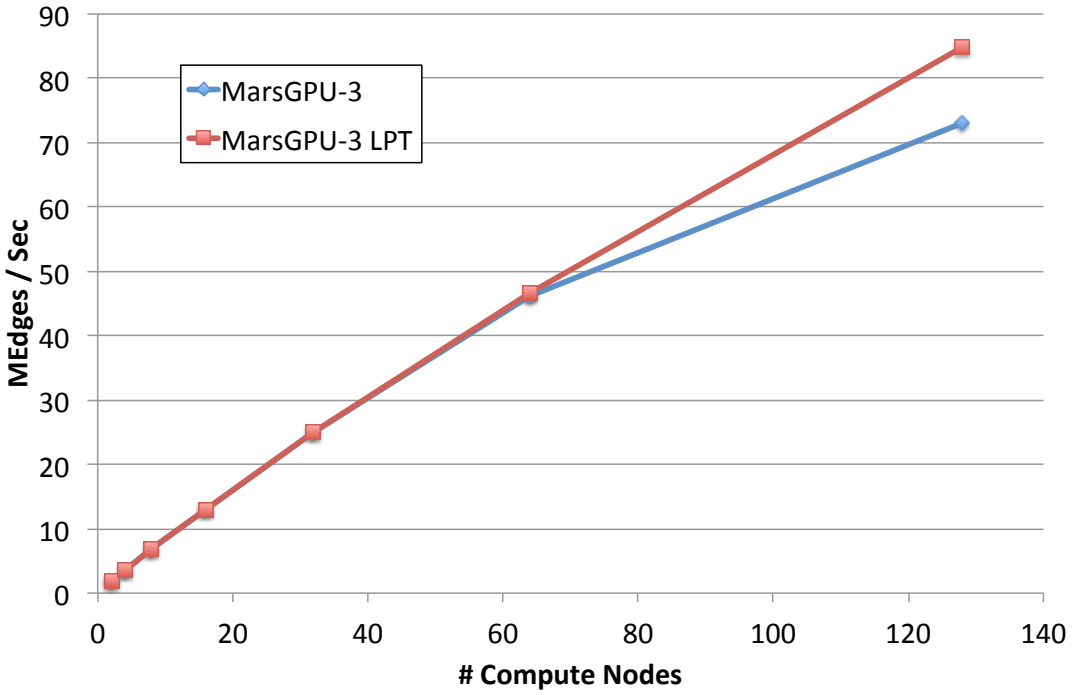

FIGURE 4.15: Weak scaling performance on MarsGPU-3: Round robin vs. LPT (SCALE 21 per node)

Performance Comparison with Load Balancing Algorithm

First, we compare two load balance techniques, naive partitioning (Round robin) and partitioning based on load balance (LPT) described in Section 4.7.1, based on simulation. Here we define load imbalance as the ratio of maximum and average amount of task each GPU handle, which is calculated by the percentage of the difference between maximum and average amounts divided by average amount. Figure 4.14 shows the weak-scaling results of the simulation. Each node has the constant problem size, SCALE 19. The x-axis denotes the number of compute nodes and the y denotes the load balance in percentage. We observe performance improvement by using optimized partitioning: 13.8% better on SCALE 26 at 128 nodes. In other cases, however, we cannot see significant performance differences; only 3.98% for SCALE 25 at 64 nodes. The result means that the input graphs generated from the Graph500 implementation are relatively well-balanced without any optimized partitioning.

Next, we compare the weak-scaling performance of our GPU-based GIM-V implementation with load balancing techniques. Figure 4.15 shows the comparison of the results between *MarsGPU-3* with Round robin partitioning and *MarsGPU-3* with LPT-based partitioning. The results exhibit that the performance is almost equal for each plot

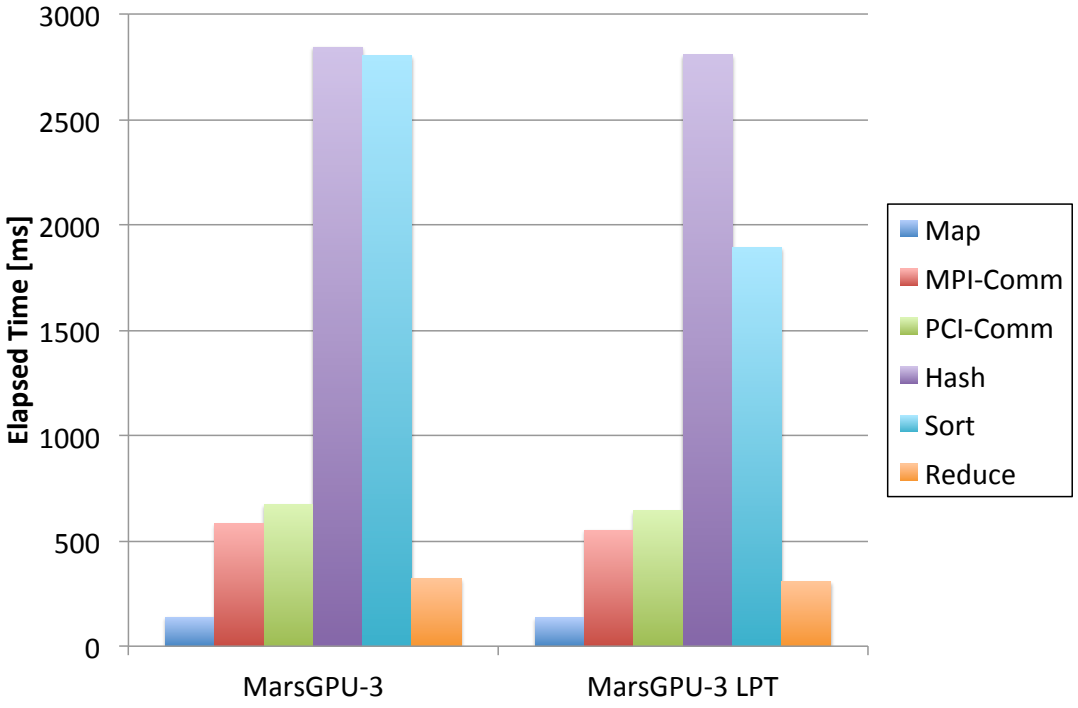

FIGURE 4.16: Performance breakdown on MarsGPU-3: Round robin vs. LPT (SCALE 28, 128 nodes)

except for 128 nodes. This is because the input graphs are relatively well-balanced as shown in Figure 4.14. However, In the case of 128 nodes, we see some performance improvement. Here Figure 4.16 shows the performance breakdown on 128 nodes from Figure 4.15. The results exhibit that the performance improvement is derived from the sort stage. Since we use the bitonic sort algorithm in our current implementation, the algorithm sorts power-of-two key-value pairs, the number of which has to be equal to or larger than the number of input key-value pairs. Therefore, the number of key-value pairs to sort can vary by the number of input key-value pairs. This situation in the sort stage introduces the performance differences as we see in Figure 4.16.

We also compare the load-balance approaches on different types of graphs. Here we use four types of graphs; (1) a Kronecker graph with A = 0.57, B = 0.19, C = 0.19, D = 0.05 (default parameters used in the Graph500 benchmark), (2) a Kronecker graph with A = 0.80, B = 0.05, C = 0.05, D = 0.10, and (3) a random graph. We compare weak scaling performance with non-scrambled graphs, scrambled graphs, and graphs using the LPT scheduling. Note that the scrambled graphs are the graphs whose vertex indices are randomized as preprocessing. Note that we use one GPU per node for these experiments. We use TSUBAME-KFC as a computing environment and we use TSUBAME 2.5 for the fourth graph. A node on TSUBAME-KFC contains 2 sockets of Intel Xeon E5-2620

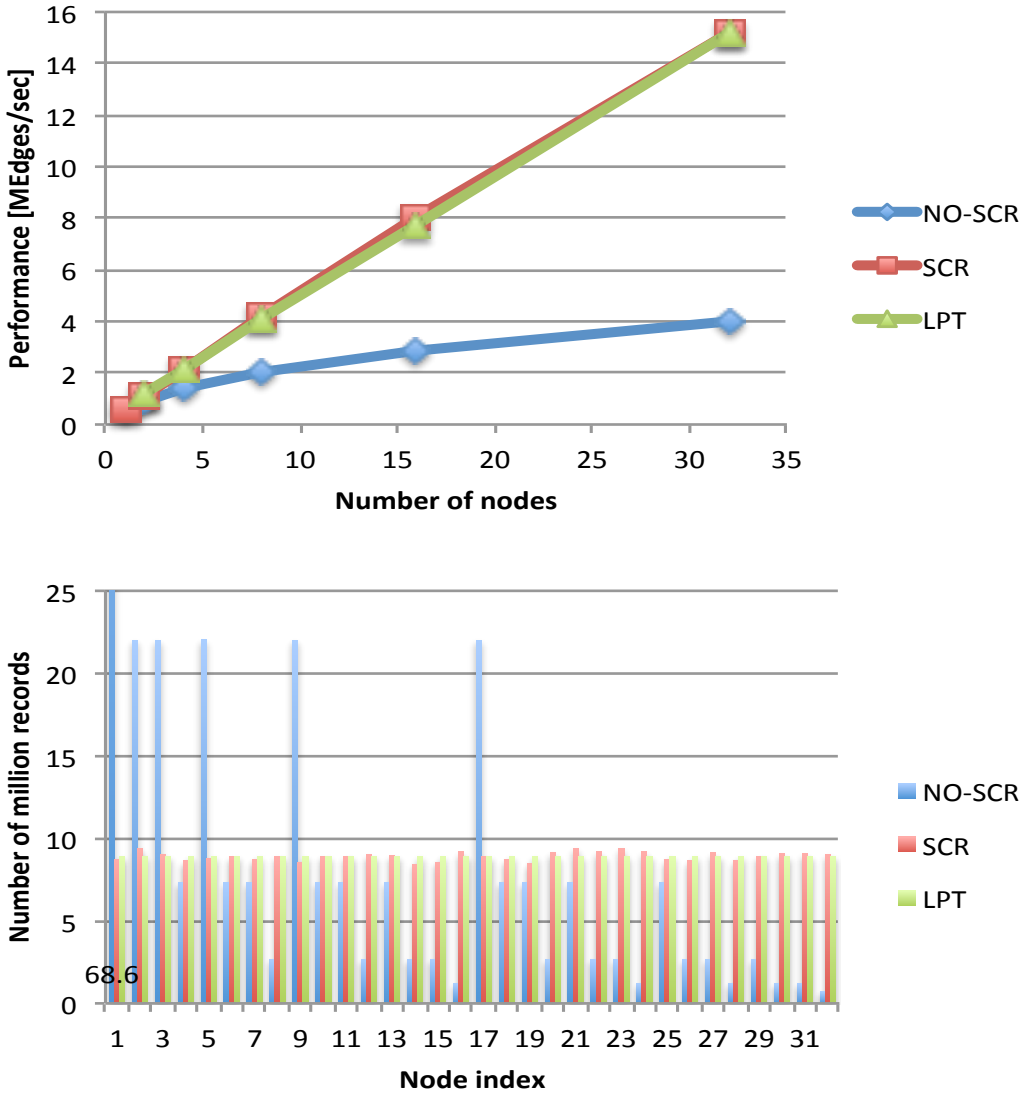

FIGURE 4.17: (Top) Comparative performance on TSUBAME-KFC (Weak scaling, SCALE 19 per node, A = 0.57, B = 0.19, C = 0.19, D = 0.05), (Bottom) Comparative graph data distribution on 32 nodes of TSUBAME-KFC

v2 (Ivy Bridge EP, 2.10GHz, 6 cores) CPU, 64GB of DDR3 main memory, 4 devices of NVIDIA Tesla K20X GPU with 6GB of discrete GDDR5 memory connected to PCI-Express 2.0 × 16 buses, and 1 card of FDR InfiniBand HBA (56Gbps) connected to a single rail interconnect network, and runs on CentOS release 6.4. We use Open MPI 1.7.2 with GNU GCC 4.4.7 for the MPI implementation, and CUDA driver 5.5 and CUDA runtime 5.5 for the GPU implementation.

The top of Figure 4.17 shows the results of comparative performance of the first three graphs respectively. Also, the bottom of Figure 4.17 shows the results of comparative

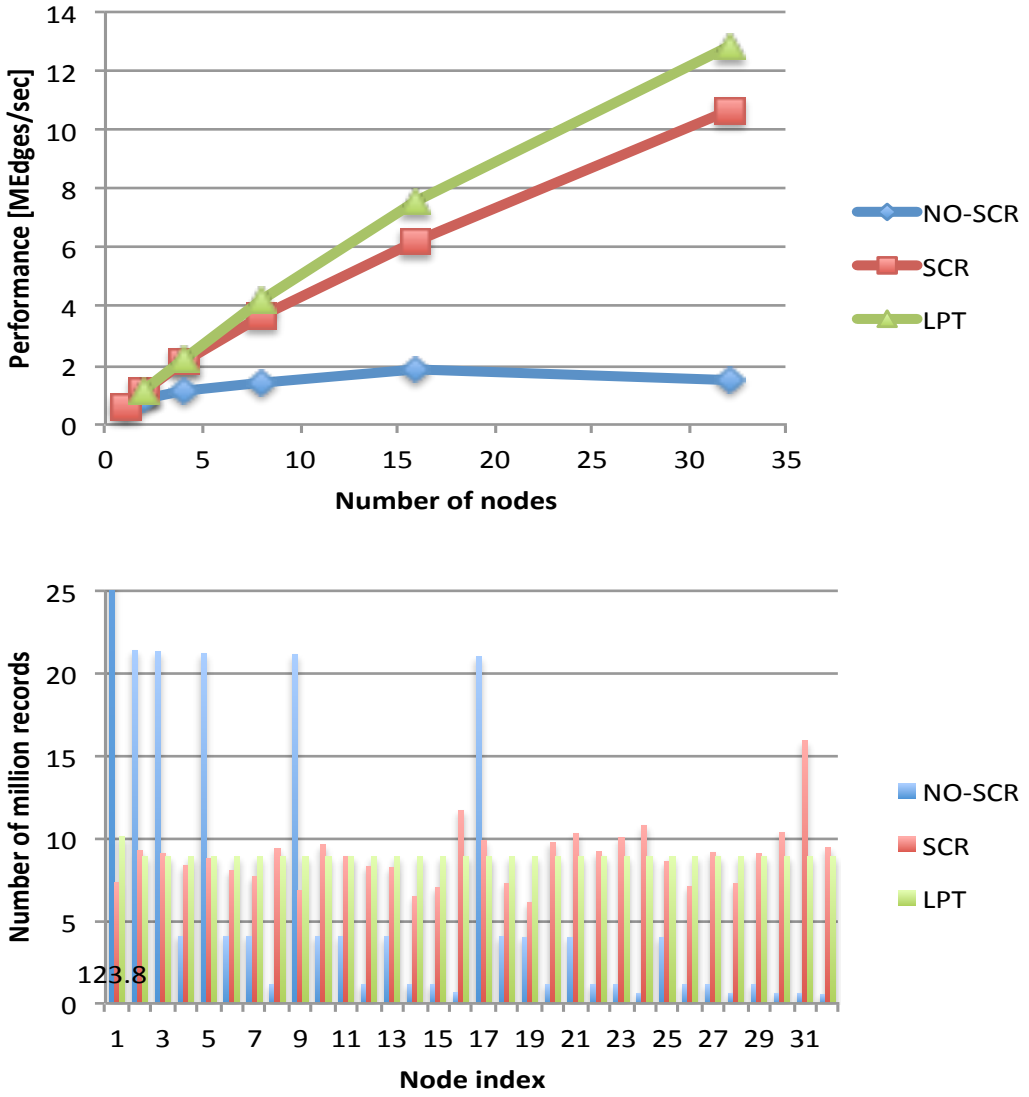

FIGURE 4.18: (Top) Comparative performance on TSUBAME-KFC (Weak scaling, SCALE 19 per node, A = 0.80, B = 0.05, C = 0.05, D = 0.10), (Bottom) Comparative graph data distribution on 32 nodes of TSUBAME-KFC

graph data distribution on 32 nodes of the first three graphs respectively. These distribution represent data distribution (i.e. number of records on each node) after Shuffle operation in GIM-V Stage 1. Note that the distributions depend on source index of edge data; i.e. a node which handles high degree vertex index receives large number of records. For all the three results, the results indicate non-scrambled graphs exhibit heavy performance degradation. This degradation is due to load-imbalance stemmed from highly skewed edges, as we see in Figure 4.2 in non-scrambled graphs. We also see that the results show similar performance between scrambled graphs and graphs

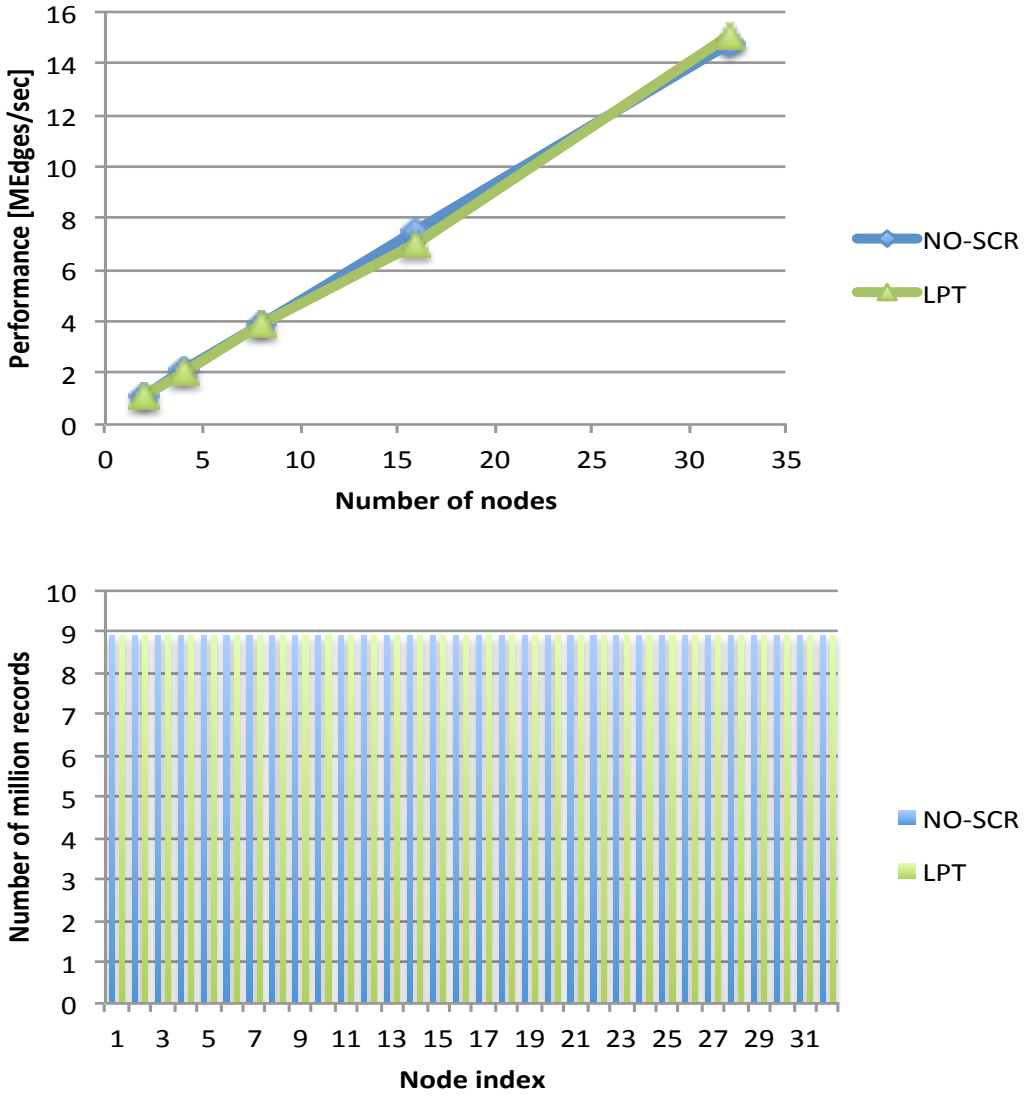

FIGURE 4.19: (Top) Comparative performance on TSUBAME-KFC (Weak scaling, SCALE 19 per node, random graph), (Bottom) Comparative graph data distribution on 32 nodes of TSUBAME-KFC

using LPT in the Kronecker graph with A = 0.57, B = 0.19, C = 0.19, D = 0.05 and the random graph, while performance improvement using LPT in the Kronecker graph with A = 0.80, B = 0.05, C = 0.05, D = 0.10. The results show 20.6% performance improvement with LPT on 32 nodes compared with the scrambled graph. We consider a possible reason of this results is that the scrambled graphs consist of relatively even amount of edges for each node in the Kronecker graph with A = 0.57, B = 0.19, C = 0.19, D = 0.05 and the random graph, while there still is workload imbalance in the Kronecker graph with A = 0.80, B = 0.05, C = 0.05, D = 0.10. In fact, the graph

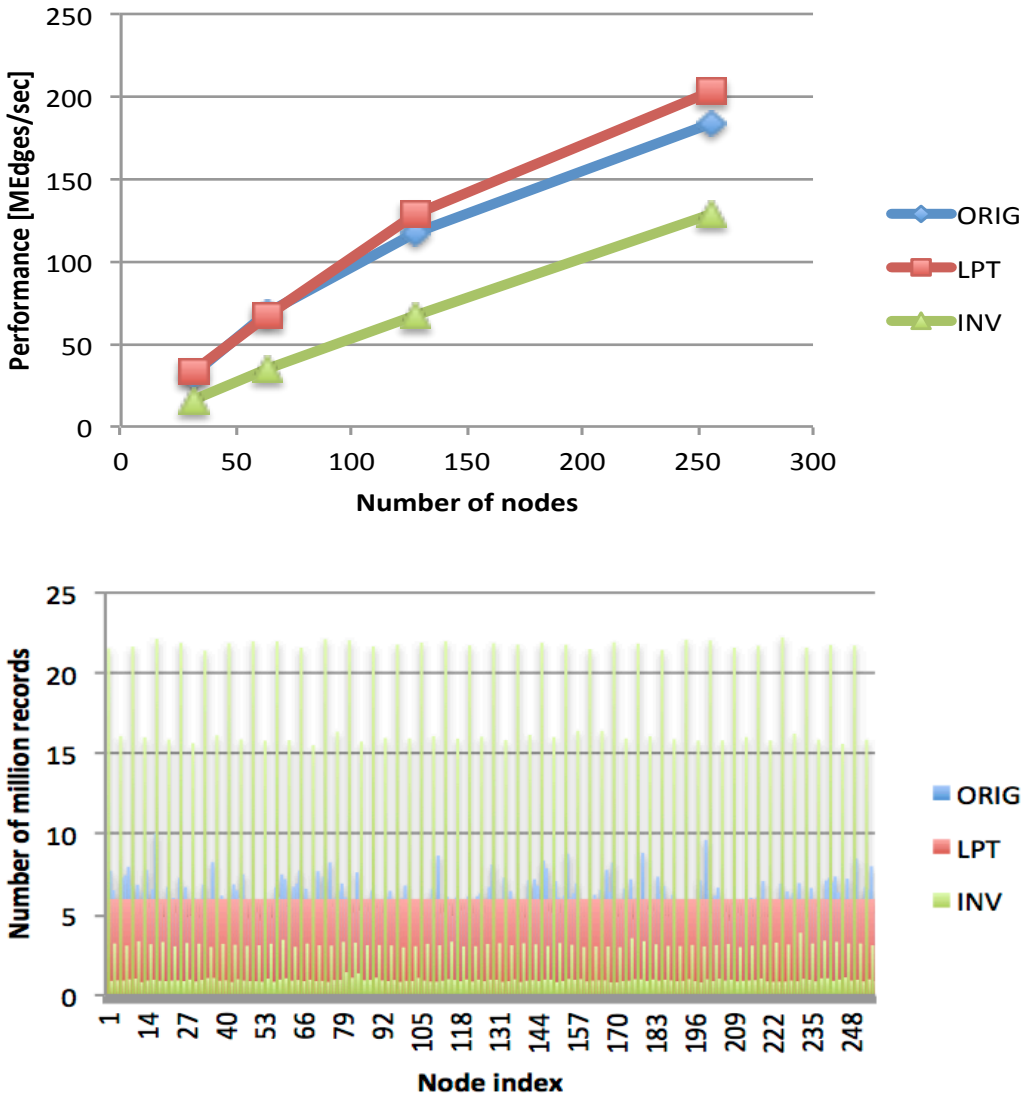

FIGURE 4.20: (Top) Comparative performance on TSUBAME 2.5 (Strong scaling, Twitter friendship graph), (Bottom) Comparative graph data distribution on 256 nodes of TSUBAME 2.5

data distribution results indicate that the scrambled graphs and the graphs using LPT exhibit similar in the Kronecker graph with A = 0.57, B = 0.19, C = 0.19, D = 0.05 and the random graph, while exhibit load imbalance with scrambled graphs in the Kronecker graph with A = 0.80, B = 0.05, C = 0.05, D = 0.10.

We also conduct experiments with a real world graph data, using a Twitter friendship graph with 61.6 million vertices and 1.47 billion edges. We compare strong scaling performance with the original (non-scrambled) graph, the graph using the LPT scheduling, and the inverse graph in which sources and destinations in all edges are swapped (we

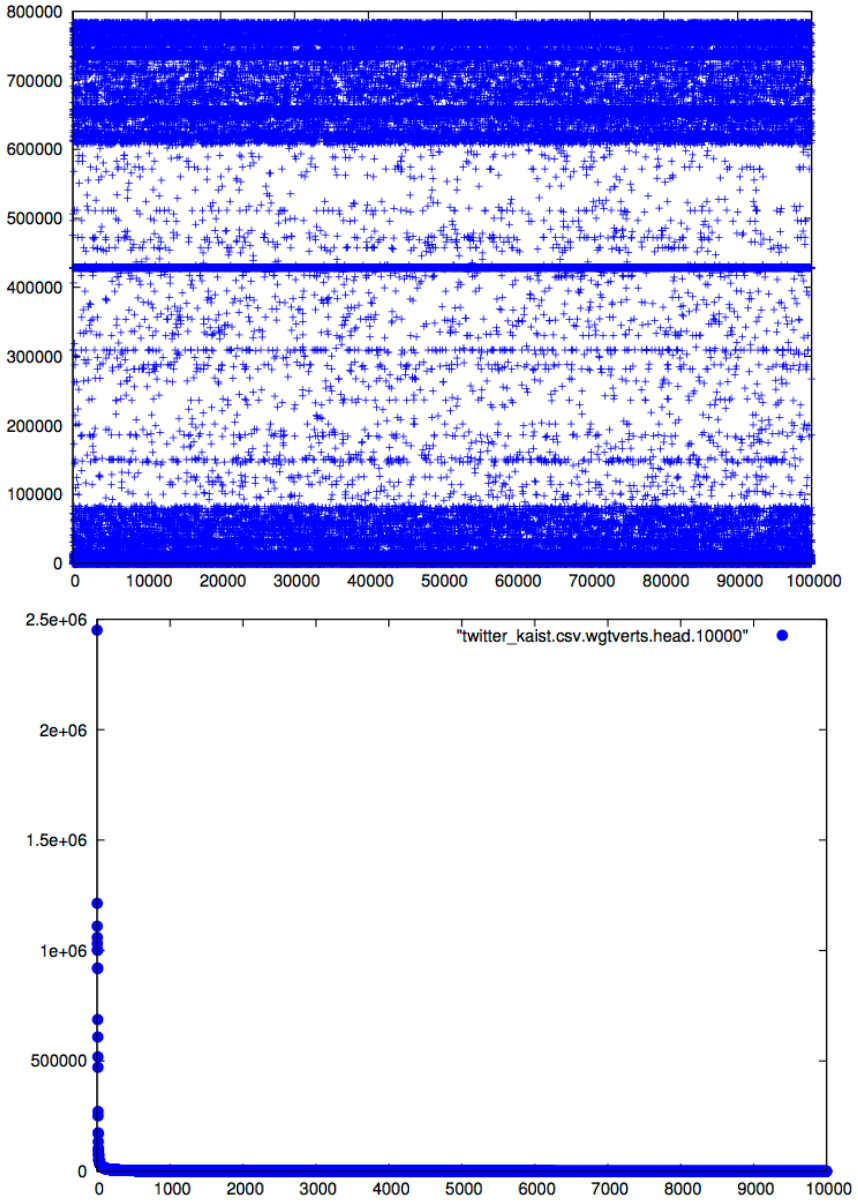

FIGURE 4.21: (Top) Adjacency matrix of Twitter friendship graph), (Bottom) Sorted vertex distribution of Twitter friendship graph

call the third graph inverse graph). Note that we use one GPU per node for these experiments. The top of Figure 4.20 shows the results of comparative performance of the three types of configurations on TSUBAME 2.5, as well as the results of comparative graph data distribution on 256 nodes of TSUBAME 2.5. The results show that LPT performs the best; 1.11x faster compared with the original graph and 1.58x faster compared with the inverse graph. The performance improvement of using LPT derives from the fact that load imbalance of the other two graphs is improved by the LPT scheduling,

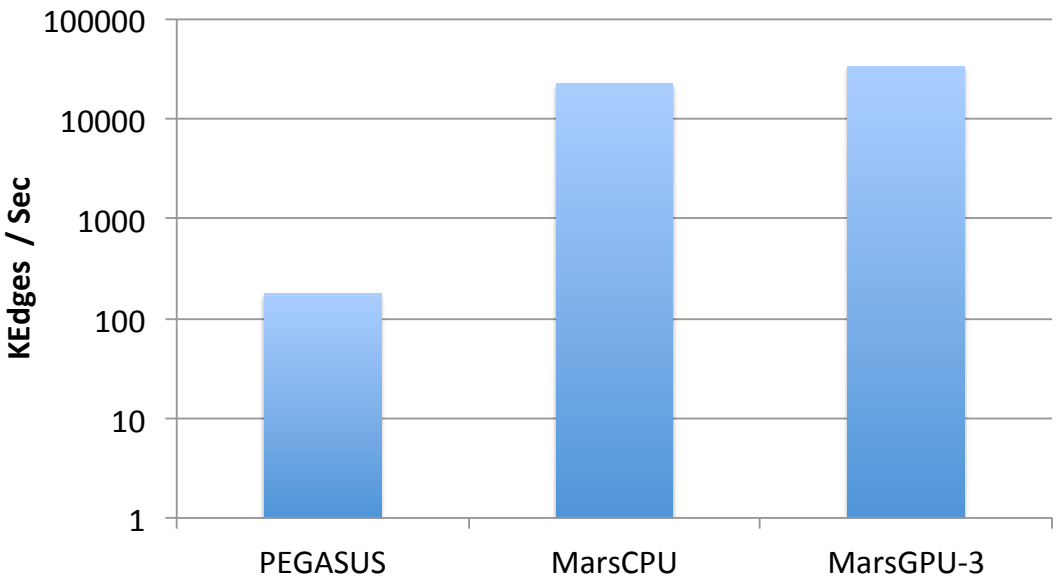

FIGURE 4.22: Performance Comparison with PEGASUS (SCALE 27, 128 nodes)

as the bottom of Figure 4.20 indicates. The results also show that the inverse graph suffers much heavier load imbalance compared with the original graph. In order to investigate the cause of the work load difference, we see the structure of the graph. The top of Figure 4.21 shows the adjacency matrix and the sorted vertex distribution of the Twitter friendship graph. The figure shows scale-free property similar to the artificial Kronecker graphs, while the structure of adjacency matrix is different. This adjacency matrix shows that some destination vertices receive a large number of edges. On the other hand, source vertices are relatively evenly connected with the some destination vertices. This evenly distributed source vertices result in little load imbalance of the earlier performance results of the original graph. On the other hand, inverse graph suffers much heavier load imbalance due to the skewed destination vertex distribution.

Comparison with Hadoop-based GIM-V implementation

Finally, we compare our multi GPU-based GIM-V implementation with PEGASUS, which is the Hadoop-based original GIM-V implementation, using up to 128 compute nodes. Here we use Hadoop version 0.21.0 and Lustre for the underlying Hadoop's file system. Figure 4.22 shows the performance of one iteration in the GIM-V algorithm, where the x-axis denotes the size of the input graph in SCALE and the y-axis denotes the performance in KE/s in the logarithm scale. We see that our implementation exhibits 186.6 times performance improvement than PEGASUS on SCALE 27 with 128 nodes. The main reason of this significant performance improvement is derived from the

underlying differences in implementation; Mars and Hadoop. PEGASUS conducts I/O operations from/to secondary storage in every map and reduce stage, while our implementation only conducts read I/O operations in the preprocessing step and write I/O operations in the end of the computation to output final data to secondary storage. Our implementation forwards output data of the reduce stage in MapReduce Stage2 to the input of the next map stage by keeping the output on CPU memory, when the iterative operation continues.

4.9 Summary

We introduce a GIM-V implementation with load balance optimization for multi GPU environments and conduct performance studies using our implementation on the TSUBAME2.0 supercomputer using 256 nodes (6144 hyper-threaded CPU cores, 768 GPUs). The results exhibit that our GPU-based implementation performed 87.04 ME/s on SCALE 30, and 1.52 times faster than the CPU-based native implementation on SCALE 29. We also show the effectiveness of our load balance optimization techniques especially on highly skewed graphs. The results also show that our implementation exhibits 186.6 times performance improvement than PEGASUS on SCALE 27 with 128 nodes.

Chapter 5

Out-of-core GPU Memory Management for MapReduce-based Large-scale Graph Processing

We have so far discussed issues and restrictions on large-scale graph processing on GPU-based heterogeneous supercomputers due to uncertainty of efficiency using GPU and memory management of hundreds of GPUs for large-scale graph processing. Another significant challenge with large-scale graph processing on GPU-based heterogeneous supercomputers is handling memory overflow from GPU memory. This chapter first reviews prior proposed techniques on out-of-core processing on CPUs and GPUs, then advantages and limitations of the techniques are thoroughly analyzed and are followed by a description of novel techniques to out-of-core GPU memory management for MapReduce-based large-scale graph processing. We analyze experimental results on TSUBAME 2.5 using 1024 nodes (12288 CPU cores, 3072 GPUs) reveal that our GPU-based implementation performs 2.10x faster than the CPU-based implementation on 12288 CPU cores, using a graph with 17.18 billion vertices and 274.9 billion edges.

5.1 Motivation

Extremely large-scale graphs recently emerge in various application fields, such as health care, social networks, system biology, and electric power grids, etc., typically consisting

of millions to trillions of vertices and edges. These large-scale graphs require fast and scalable analysis using HPC technologies by fully exploiting performance of recent supercomputers. On the other hand, modern supercomputers employ commodity graphics processing units (GPUs) in addition to general purpose CPUs, since GPU-based heterogeneous supercomputers continue to attract attention due to their high peak performance and high power efficiency. In chapter 4, we have proposed a distributed multi-GPU implementation of a MapReduce-based graph processing algorithm, where we show that our multi-GPU-based PageRank implementation performs faster compared with a multi-core CPU-based implementation on the TSUBAME2.0 supercomputer [94] using 256 nodes and 768 GPUs.

Although GPU-based heterogeneous supercomputers are suitable for graph applications, the capacity of device memory on GPUs limits scalable large-scale graph processing, since GPUs typically have smaller memory capacity than the CPU hosts. For example, the TSUBAME2.5 supercomputer [11] employs 1408 compute nodes, each of which equips 3 GPU devices and 2 CPU sockets, where the capacity of device memory on each GPU is 6GB, while that of CPU host memory is 54GB. Thus, in order to process larger-scale graphs whose size exceeds the capacity of GPU memory, data management techniques for handling GPU memory overflows are required. However, such out-of-core GPU data management techniques with detailed performance studies for large-scale graph processing are not well investigated. Furthermore, even if we apply the out-of-core GPU data management techniques, which execution approaches to use, only the device memory on GPUs (scale-out) or offload partial graph data to the secondary CPU memory (scale-up) on a multi-node environment, in terms of graph application's performance and its power efficiency, is considered another important issue.

5.2 Introduction to Out-of-core Processing

This section summarizes prior proposed techniques on out-of-core processing on CPUs, followed by prior efforts on out-of-core GPU processing, and memory overflow handling on MapReduce-based processing on GPUs.

5.2.1 Out-of-core CPU Processing

Several research efforts have explored out-of-core graph processing on CPU. As for CPU-based graph processing on a single node, several techniques, such as sequential I/O optimization [49], data placement optimization [79], and data prefetch optimization [80],

have been proposed. These techniques focuses on the utilization of a single node. Thus, distributed computing environments are not supported. Pearce et al. [81] have proposed a CPU-based out-of-core large-scale graph processing technique for distributed computing environments. Their technique introduces a graph partitioning strategy and applies to their multithreaded algorithm using distributed external memory; however, this algorithm cannot be straightforwardly applicable to GPUs, since this algorithm is highly designed for utilizing multi-core CPUs. The MapReduce [3] programming model has been proposed for processing big data applications with automatic memory/storage hierarchy encapsulation, and Hadoop [7] is one of the widely used MapReduce implementation. MR-MPI [26] is a MPI-based MapReduce implementation on CPU, which employs an out-of-core processing technique including in the sort phase after inter-node data exchanges. These MapReduce implementations are designed for CPU-based distributed environments, while our work focuses on GPU-based large-scale environments.

5.2.2 Out-of-core GPU Processing

Researchers have been working on out-of-core GPU processing algorithms in a wide range of application fields, such as BFS [66], stencil [82], rendering [83], etc. These algorithms have shown GPU accelerations by using out-of-core techniques; however, the scope of these applications is limited on specific algorithms. Out-of-core GPU sorting algorithms, such as a sample-based sorting [84] and a merge-based sorting [85], have also been studied; however, these algorithms are designed for a single node execution. These algorithms also have not well investigated load balancing issues for highly skewed data such as real world graphs. There also exists work on I/O issues from a GPU to filesystems [86]; however, they have not conducted experiments on realistic large-scale applications such as graph processing.

5.2.3 MapReduce-based Out-of-core Processing on GPUs

The MapReduce model can provide out-of-core processing with simple application interfaces. There exists a generalized graph processing algorithm for the MapReduce model called GIM-V (we explain the details in Section 4.3) and its Hadoop-based implementation [6]. However, the implementation does not show good performance due to heavy overheads derived from the Hadoop framework. In chapter 4, we proposed a distributed multi-GPU-MapReduce-based graph processing implementation and showed that our multi-GPU-based PageRank implementation outperforms the Hadoop-based implementation considerably on TSUBAME2.0. GPMR [34] is a multi-GPU MapReduce library

supporting out-of-core GPU execution on distributed computing environments. However, the sort phase in GPMR is executed on CPUs when the size of input data exceeds the capacity of the GPU memory, instead of executing on GPUs. Besides, the performance studies on CPU vs. GPU comparison have not been sufficiently conducted, especially in the out-of-core situation.

5.2.4 Issues on Out-of-core GPU memory management

One of the significant issues for processing large graphs on GPUs is considered that how to manage graph data whose size exceeds the capacity of GPU memory with minimal performance overheads. As explained in the previous section, GPU memory generally has the smaller capacity than CPU memory, and computation on GPUs requires to transfer data between CPU memory and GPU memory. Thus, when we naively apply the graph algorithms to GPUs, data transfers dominantly disturb efficient graph processing. In particular, when the size of the graphs exceeds the capacity of device memory on GPUs, the number of data transfers drastically increases for executing dependent graph kernels.

We can certainly overcome the GPU capacity limitation problem by using multi-node multi-GPU environments. Indeed, several existing efforts have shown good weak-scaling performance of graph processing on GPU-based large-scale environments; however, these techniques still have the limit on the size of the graphs below the device memory capacity on GPUs. Although out-of-core GPU memory management techniques may help solving the problem by utilizing secondary host memory volumes, best approaches with detailed performance studies whether we should use only the device memory on GPUs (scale-out) or offload partial graph data to the secondary CPU memory (scale-up) on a multi-node environment are not well investigated in terms of graph application's performance and power efficiency. Moreover, optimization techniques for out-of-core GPU memory management techniques to achieve good weak scalability on large-scale environments should be investigated, since the graph algorithms generally include irregular data accesses to sparse data sets, whose situations introduce significant performance overheads and disturb scalable large-scale graph processing.

5.3 Out-of-core GPU Memory Management

This section describes our proposed techniques on out-of-core GPU memory management for MapReduce-based large-scale graph processing. We first summarize basic idea

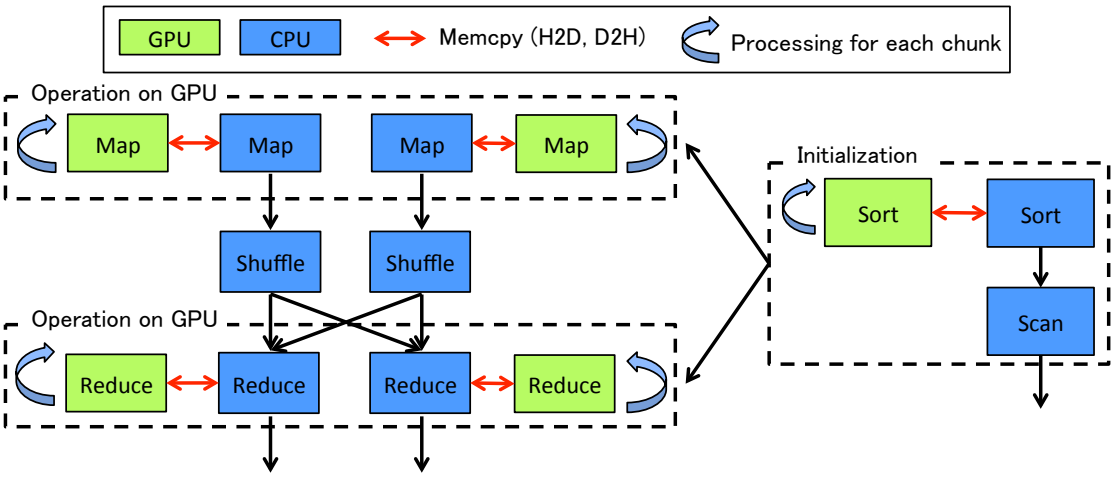

FIGURE 5.1: Overview of our out-of-core multi-GPU MapReduce framework. The dashed boxes on the left side represent operations initialized by the dashed box on the right side.

of our techniques, followed by a description of two novel techniques; stream-based GPU memory management on MapReduce and optimized sample-based out-of-core GPU sorting.

5.3.1 Basic Idea

Our out-of-core GPU memory management technique is designed on top of the MapReduce model, since MapReduce can transparently encapsulate memory hierarchies by providing automatic memory management from the system. Before describing our proposed data management technique, we introduce the target multi-GPU MapReduce framework.

Figure 5.1 shows an overview of the framework. The basic architecture of the framework remains the same as our proposal in chapter 4, nevertheless we use a different implementation here. We firstly read key-value pairs as input data from a distributed file system to CPU memory on multiple nodes and keep the data on CPU memory. Next, we sort and reorder the input key-value pairs by key to obtain a set of values for a key. Then, users call *Map*, *Shuffle*, or *Reduce* operations based on user-specific application workflow. Note that we may skip the sorting process for the *Map* operations. When *Map* or *Reduce* operations are called, the input data are processed on GPUs inside the framework with user-provided operations. When *Shuffle* operations are called, the input data are exchanged between multiple nodes based on system-provided or user-provided splitters by using MPI all-to-all communications. Finally, output data are transferred

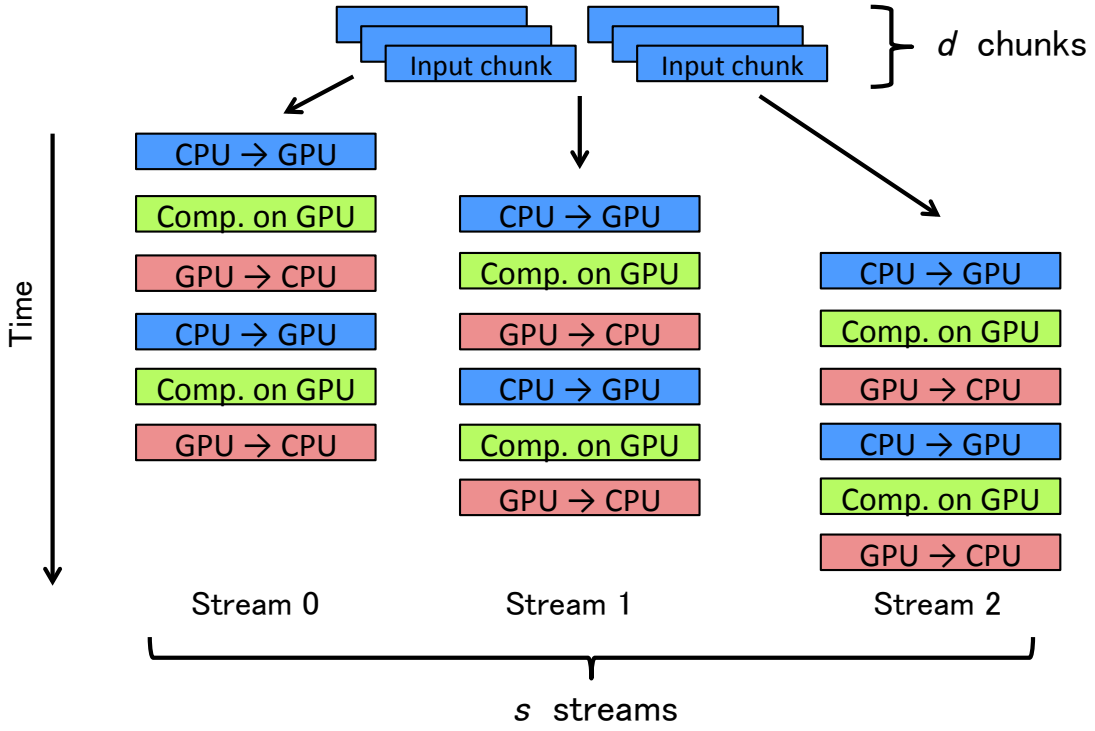

FIGURE 5.2: Overview of our stream-based out-of-core GPU memory management. The upper boxes represent input chunks on CPU memory to be processed. The lower three types of boxes represent data transfers from CPU memory to GPU memory, computations on GPUs, and data transfers from GPU memory to CPU memory, respectively.

onto the CPU memory on each node. Our framework is flexibly designed so that the users can define multiple *Map* and *Reduce* operations and call *Map*, *Shuffle*, and *Reduce* operations in an arbitrary order. The users can also write applications with iterative computations by writing loop syntax with user-provided convergence criteria.

The above technique includes a significant limitation that the framework cannot handle GPU memory overflows. Thus, we simply extend the above framework based on two straightforward ideas, streaming processing on GPUs and GPU-based external sorting. By dividing input data into multiple chunks and by processing each chunk one by one in a stream, we apply overlapping techniques between computation and data transfer for hiding data transfer overheads as much as possible.

5.3.2 Stream-based GPU MapReduce Processing

Figure 5.2 shows an overview of our streaming processing technique for GPU-based *Map* and *Reduce* operations. In order to optimize data transfer between CPU and GPU, we overlap three operations: data transfer from CPU memory to GPU memory,

Map and *Reduce* operations on GPU, and data transfer from GPU memory to CPU memory, otherwise we suffer additional CPU-GPU data transfer overheads for each *Map* or *Reduce* operation. Note that our stream-based memory management provides additional benefits that hide CPU-GPU data transfer from *Map* and *Reduce* operations on the GPU even if the size of input data fits the capacity of GPU memory. The detailed instructions of our stream-based CPU-GPU memory management technique are shown as follows:

STEP1: Divide input key-value data into d chunks evenly, where d denotes the number of chunks. We determine the number of chunks dynamically so that each chunk fits on the GPU memory capacity.

STEP2: Create s CUDA streams, where s denotes the number of streams, and allocate s buffers on GPU for the chunks of the input key-value data; a single buffer is linked to a single CUDA stream.

STEP3: Repeat streaming processing d times; transferring a chunk of the input key-value data from CPU to a buffer on GPU, running *Map* and *Reduce* operations on GPU, and transferring output from the buffer on GPU to CPU. These three operations are overlapped using asynchronous function calls (i.e. *cudaMemcpyAsync* function with pinned memory).

We set the s parameter to three by default in order to overlap the above three operations. We dynamically update the d parameter to fit the size of input data chunks on the capacity of GPU device memory.

5.3.3 Out-of-core GPU Sorting

We introduce a GPU sorting implementation to the framework for handling GPU memory overflows. The implementation consists of a combination of existing GPU-based out-of-core and in-core algorithms. As for out-of-core GPU sorting, we employ an existing sample-based out-of-core sorting algorithm for GPUs proposed by Ye et al. [84], while as for in-core GPU sorting, we employ the radix sort algorithm based on the Thrust library [95]. Out-of-core GPU sorting is conducted when the size of input data exceeds the GPU memory capacity. Otherwise, in-core GPU sorting is conducted.

Figure 5.3 shows an overview of the out-of-core GPU sorting algorithm. Sample-based parallel sorting uses $t-1$ samples as splitters to partition the input data set into several

data chunks, where t denotes the number of sample points. The chunks can be put on GPU memory by considering the size of chunks and the capacity of GPU memory. We dynamically determine the number of chunks by checking the available amount of memory and the input data size at the beginning. If the input data size is too large to fit on the GPU memory, our framework divides the input data into d chunks based on the available GPU memory capacity and the input data size. The detailed instructions of the out-of-core GPU sorting algorithm are shown as follows:

STEP1: Randomly select c keys as sample candidates from input keys on CPU host memory.

STEP2: Sort the c sample candidates. Then, pick the $(k+1) \cdot c/d \,^{th}$ sample points and set the points to $t[k]$, where $k \in [0, d-1]$. Here, d denotes the number of chunks. We set $t[d-1]$ to the maximum limit value on the host memory.

STEP3: Divide the input data set into d chunks on the host, each of which contains n/d elements evenly, where n denotes the number of input keys.

STEP4: Copy each chunk onto GPU memory, sort each chunk using the in-core sorting algorithm, and split each chunk into d buckets using splitters based on the sample points on GPU.

STEP5: Swap the buckets among chunks on the host, so that elements in the $(i+1)^{th}$ chunk are no smaller than those in the i^{th} chunk.

STEP6: Copy each chunk onto GPU and sort one by one using the in-core sorting algorithm on GPU.

Our out-of-core sorting algorithm differs from the existing out-of-core GPU sorting proposed by Ye al. [84] in that we present less CPU-GPU data transfer overheads by simplifying the data dividing strategy than the existing algorithm, since we observe good load balance when we set c to larger numbers than 1000.

We implement a stream-based overlapping feature for GPU sorting and CPU-GPU data transfers, whose instructions are shown in Step 4 and 6. Thrust uses default CUDA stream and does not presently have a mechanism to control execution streams. In order to overlap with the default stream, we create multiple streams by *cudaStreamCreateWithFlags* with *cudaStreamNonBlocking* flag, a new feature enabled from CUDA 5.0. *cudaStreamNonBlocking* flag enables overlapping with the default stream. In Step 5, we also implement pointer-based swapping algorithm with low-overheads. Pointers of buckets are swapped instead of the payloads.

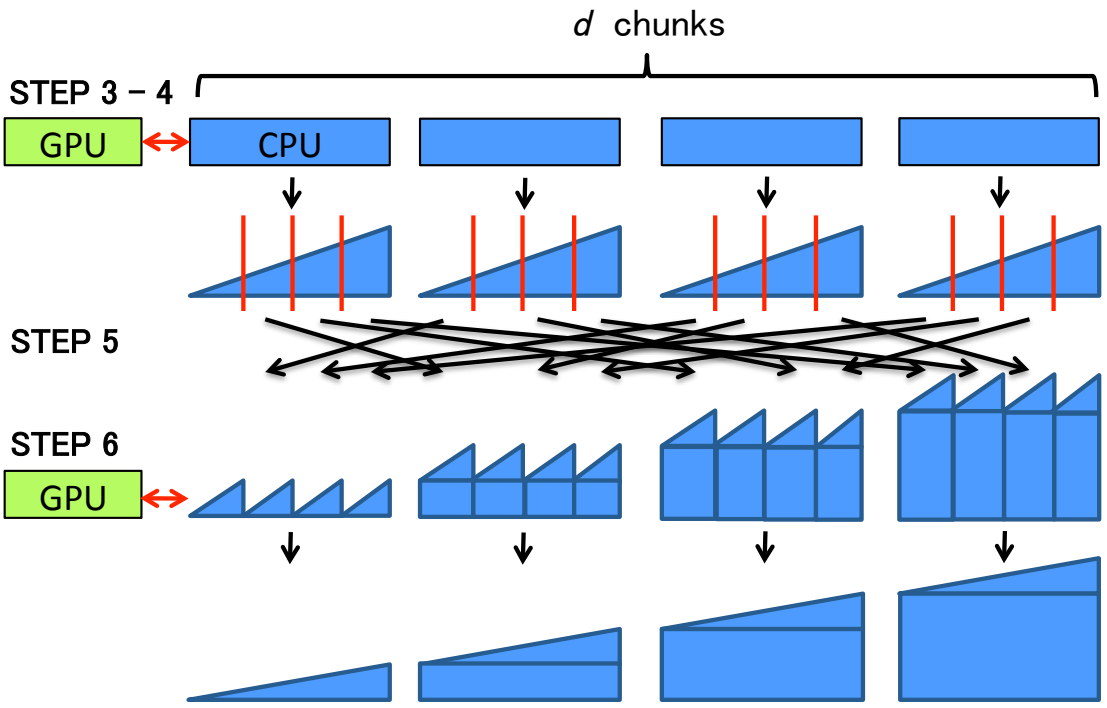

FIGURE 5.3: Overview of the out-of-core GPU sorting algorithm. Blue boxes represent operations called on CPUs and green boxes represent operations running on GPUs. Red vertical bars represent splitters based on sample points.

5.4 Optimization Techniques

In order to achieve good weak-scaling performance on large-scale GPU-based heterogeneous supercomputers, we apply several optimization techniques to the framework with our out-of-core GPU memory management technique. Here we describe the details of the optimization techniques.

5.4.1 Data Structure

We employ a compact data structure similar to CSR (Compressed Sparse Row) for sparse matrix formats, which consists of an array of unique keys, values, and indices of first values for unique keys, for compressing redundant data and for achieving efficient *Map* and *Reduce* processing. For instance, if a Kronecker graph in the Graph500 benchmark is given as input data, we can expect to compress duplicate keys to around 1/16, since the graph includes 16 edges per vertex on average. We firstly reorder input key-value pairs by using the out-of-core GPU sorting algorithm. Then, we apply the scan (prefix sum) operation to the sorted keys in order to calculate indices of first values for unique keys. Finally, we compact the duplicated keys by using the unique operation.

5.4.2 Shuffle

We implement a *Shuffle* operation for redistributing intermediate data onto each node based on a system-provided or user-defined splitter function. We implement range-based and hash-based splitters as system-provided splitters. We provide a default splitter as the range-based splitter, where each node takes charge of a range of the number of data. Instead users can also implement customized splitters. We observe the range-based splitter performs good load balance for skewed graphs generated in the same way as the Graph500 benchmark by randomizing vertex indices. Although load balance depends on the input graph structure, our *Shuffle* operation can extend to other splitters by customization according to the input graph structure.

5.4.3 Thread Assignment Policy on GPU

We apply a thread assignment optimization on a GPU for handling the skew of vertex degrees on large-scale graph processing. We consider the following three strategies for assigning threads onto vertices and edges.

1. **Thread-based Assignment:** Assign one thread per vertex.
2. **Warp-based Assignment:** Assign one warp per vertex. The warp size on recent GPUs is set to 32.
3. **Thread Block-based Assignment:** Assign one thread block per vertex. The thread block size on recent GPUs (e.g. NVIDIA Tesla K20X) is set to 1024.

The strategies 2) and 3) are expected to achieve good performance on GPUs by utilizing massive amounts of threads; however, these strategies require to write CUDA-specific descriptions, such as `threadIdx`, `blockDim` etc., in the user-defined *Map* and *Reduce* operations. On the other hand, the strategy 1) can work on both CPUs and GPUs without any special descriptions. We employ the strategy 2) since the warp size is expected to be close to the average number of edges per vertex for wide range of graphs. For example, in graphs used in the Graph500 benchmark, the average number of edges per vertex is set to 16. As another example in real world graphs, the average number of edges per vertex in the Facebook friend network reaches around 130. For graphs with a large average number of edges, the strategy 3) is expected to achieve good performance. We set the thread block size as $(ws, max\_tbs/ws, 1)$ and the grid size as $(nv/blockDim.x, 1, 1)$ for the strategy 2), where ws denotes the warp size, max_tbs denotes the maximum number of threads per thread block, and nv denotes the number of vertices per GPU.

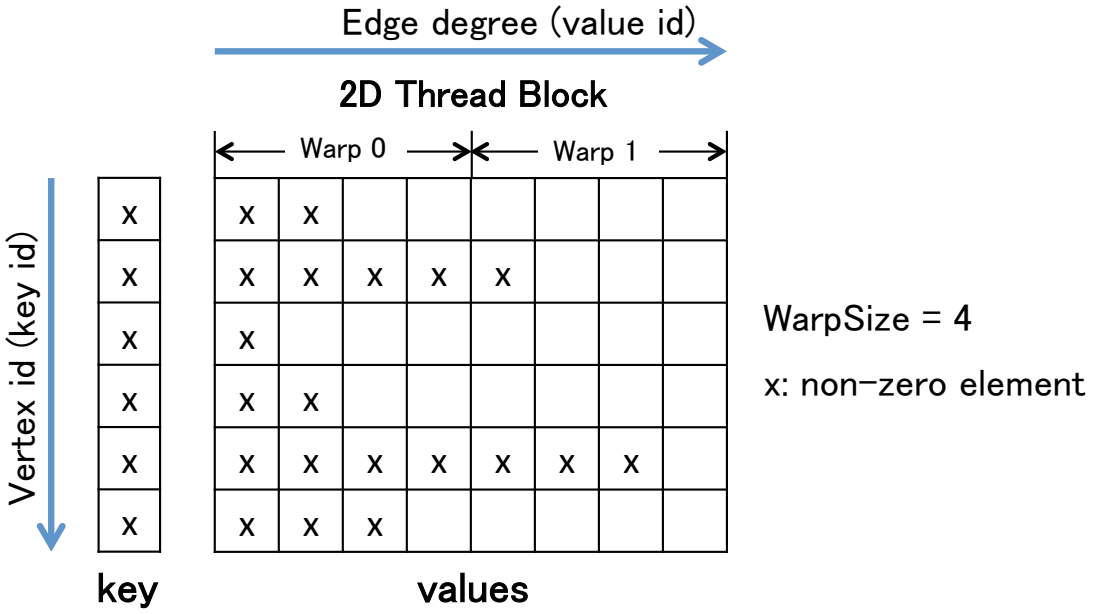

FIGURE 5.4: Warp-based thread assignment onto 2D thread block on GPU. The mesh in the left side represents keys, and the mesh in the right side represents values corresponding to each key. Each warp is assigned to a portion of values corresponding to a key. The warp is assigned multiple times to values whose length is larger than the warp size.

5.5 Performance Analysis

In order to understand the efficiency of our out-of-core GPU memory management technique for GPU-MapReduce-based graph processing, we run a PageRank application based on the GIM-V algorithm on the TSUBAME2.5 supercomputer [11]. TSUBAME2.5 mainly consists of 1408 compute nodes, each of which has 2 sockets of Intel Xeon X5670 (Westmere EP, 2.93GHz, 6 cores) CPU, 54GB of DDR3 main memory, 3 devices of NVIDIA Tesla K20X GPU with 6GB of discrete GDDR5 memory connected to PCI-Express 2.0 × 16 buses, and 2 cards of QDR InfiniBand HBA (40Gbps) connected to the dual rail interconnect network with full bisection fat tree, and runs on SUSE Linux Enterprise 11 SP1. We use up to 1024 compute nodes of TSUBAME2.5 in the experiments. We use Kronecker graphs generated in the same way as Graph500 benchmark, using the recursive matrix (R-MAT) procedure with the following initiator parameters: (A, B, C, D) = (0.57, 0.19, 0.19, 0.05) and an average vertex degree of 16. We describe the size of the graphs as SCALE, the logarithm base two of their number of vertices. We use Open MPI 1.4.2 with GNU GCC 4.3.4 for the MPI implementation, and CUDA driver 5.0, CUDA runtime 5.0, and thrust 1.7.0 for the GPU implementation.

5.5.1 Comparison with CPU-based implementation

We compare our proposed GPU-based implementation with a CPU-based implementation in order to investigate the efficiency of GPU acceleration when the size of graph exceeds the capacity of device memory on GPUs. In order to make fair comparisons, we extend our GPU-based implementation to support multi-node multi-CPU environments as well. Our implementation employs a hybrid parallelization technique using MPI and OpenMP. MPI is used for parallelization between compute nodes (or processes) in the same way as our GPU-based implementation, whereas OpenMP is used for parallelization of *Map*, *Reduce*, and *Sort* operations inside a single node (or process). Our implementation parallelizes the *Map* and *Reduce* operations in a straightforward manner by using a simple fork-join model. In the *Sort* operation, we use OpenMP-based parallel sorting in the Thrust library. We use Thrust's OpenMP sorting instead of parallel STL sorting, since parallel STL sorting is not compatible with the CUDA compiler which we use in the CPU-based implementation.

Figure 5.5 shows the results of the weak-scaling performance of our CPU- and GPU-based implementations on TSUBAME2.5, where the x-axis denotes the number of compute nodes and the y-axis denotes the performance in ME/s (million edges per second) in each stage. Each node has the constant problem size: SCALE 23 for running on 1 CPU and 1 GPU and SCALE 24 for running on 2 CPUs, 2 GPUs, and 3 GPUs. Note that the size of graphs in the configurations exceeds the capacity of device memory on the GPUs. For example, the size of a SCALE 23 graph exceeds the capacity of device memory on a GPU, and the size of a SCALE 24 graph also exceeds the aggregate capacity of device memory on 2 and 3 GPUs. Here we describe the GPU-based implementation as *nGPU(s)*, where n denotes the number of GPU devices per node, and the CPU-based implementation as *mCPU(s)*, where m denotes the number of CPU sockets per node. We use up to 1024 nodes in both *nGPU(s)* and *mCPU(s)* experiments; we vary the number of GPUs per node from 1 to 3, while we use 12 threads per node using 1 or 2 socket(s) in *mCPU(s)*. We see that our implementation on *3GPUs* achieves 2.81 GE/s (billion edges per second) on SCALE 34 on 1024 nodes (12288 CPU cores and 3072 GPUs). The results also exhibit 2.10x performance improvement compared with *2CPUs* on SCALE 34 on 1024 nodes.

Figure 5.6 shows the performance breakdown on SCALE 31 on 256 nodes, where the y-axis denotes the elapsed time in milliseconds. We divide a single GIM-V iteration into five phases; *Map*, *Shuffle*, *Reduce*, *Sort*, and *Others*. The *Map* and *Reduce* phases include the time for *Map* and *Reduce* kernel executions and CPU-GPU data transfer.

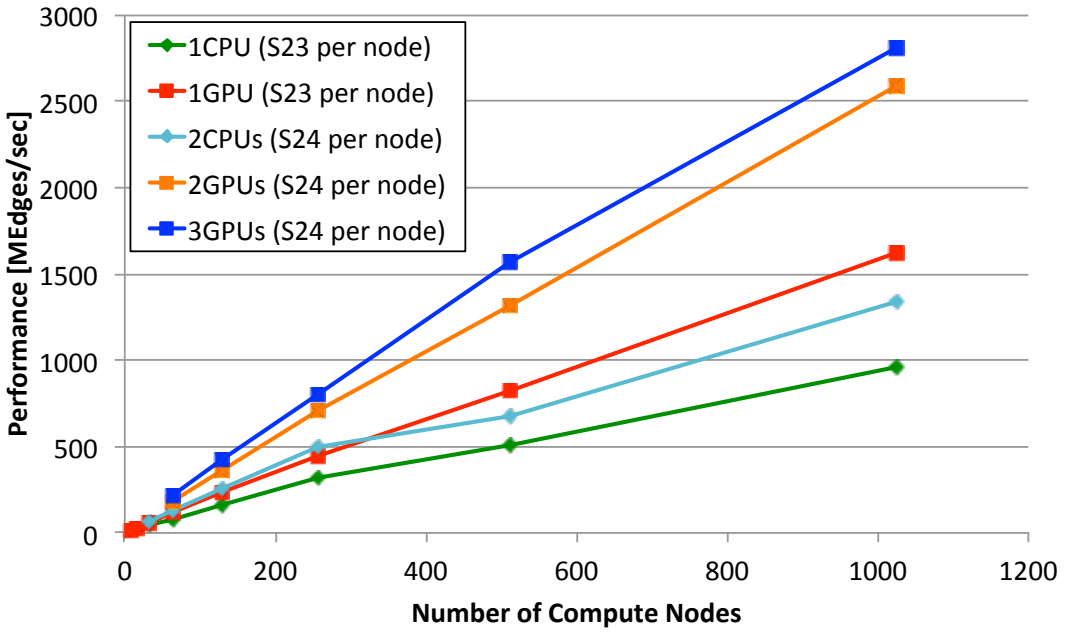

FIGURE 5.5: Results of weak scaling performance, where SCALE 23 for running on 1 CPU and 1 GPU per node and SCALE 24 for running on 2 CPUs, 2 GPUs, and 3 GPUs per node.

Shuffle phase includes the time for inter-node data transfer and its preparation. *Sort* phase includes the time for sorting in each *Map*, *Shuffle*, and Reduce phase. *Others* includes the time for the rest of the *Map*, *Shuffle*, *Reduce*, and *Sort* phases. The results exhibit that the elapsed times for the *Map*, *Reduce*, and *Sort* phases on *3GPUs* achieve 1.41x, 1.49x, and 4.95x faster than those on *2CPUs* respectively. The reason for this performance improvement is considered that our implementation hides CPU-GPU data transfer overheads efficiently in *Map*, *Reduce*, and *Sort* phases by stream-based asynchronous computation.

We analyze further breakdown of the *Map* and *Reduce* phases in a single GIM-V iteration. Figure 5.7 shows the results, where *Map n* and *Reduce n* denotes each *Map* or *Reduce* phase in GIM-V Stage *n*. As we see in Algorithm 1 and 2 in Section 4.3, the *Map1* and *Map2* operations only pass input vertices or edges data to the next phase, the *Reduce1* operation combines a vertex and connecting edges for all vertices and passes to the next phase, and the *Reduce2* operation combines all edges connecting to a vertex into an updated vertex for all vertices. Thus, we expect the *Reduce1* and *Reduce2* operations to be accelerated compared to the *Map1* or *Map2* operations by using GPUs, since the *Reduce1* and *Reduce2* operations include actual computation kernels as opposed to the *Map1* and *Map2* operations. The results in Figure 5.7 indicates that the *Map1* and *Map2* operations are accelerated 1.41x, the *Reduce1* operation is accelerated

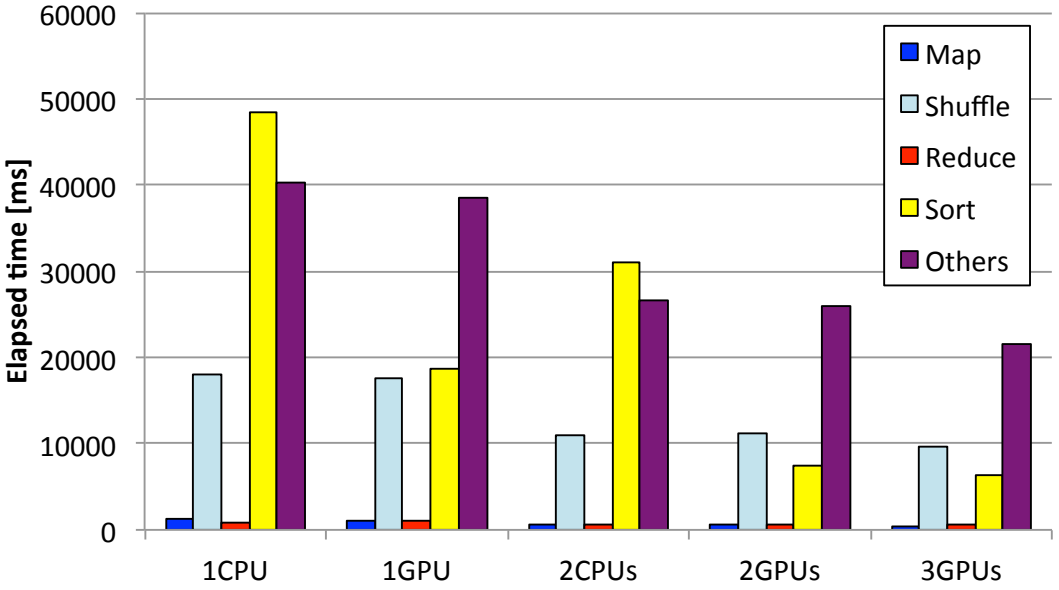

FIGURE 5.6: Results of performance breakdown on SCALE 31 using 256 nodes.

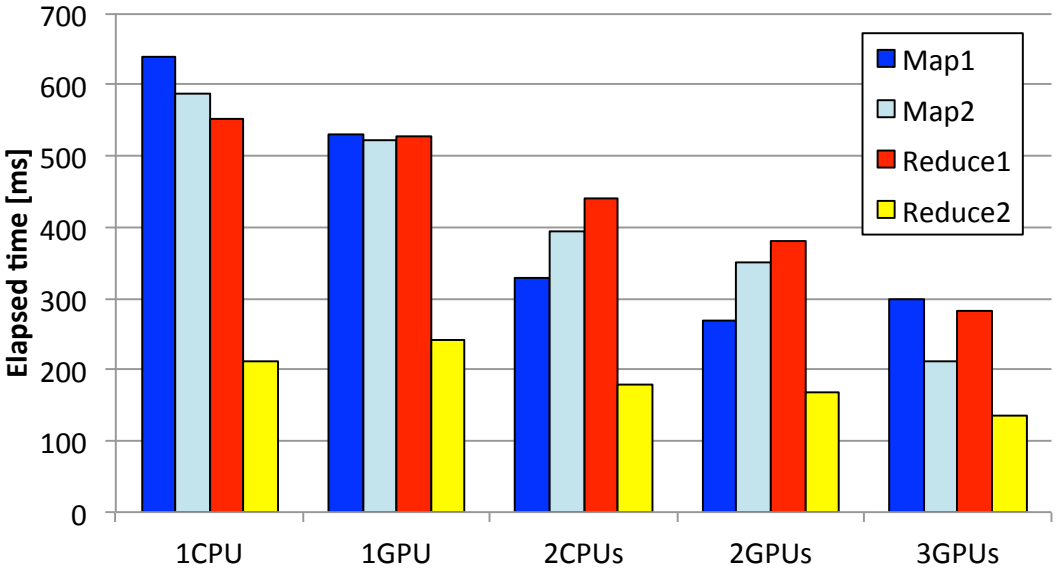

FIGURE 5.7: Results of *Map* and *Reduce* phases on SCALE 31 using 256 nodes.

1.56x, and the *Reduce2* operation is accelerated 1.33x respectively by using 3 GPUs per node compared with 2 CPUs per node. As expected, the *Reduce1* operation is more accelerated than the *Map1* and *Map2* operations; however, we also see that the *Reduce2* operation is not accelerated as much as the other phases. We consider the result comes from not fully overlapping CPU-GPU communication, since the computation time in *Reduce2* operation is not sufficiently large.

5.5.2 Results of Out-of-core GPU Sorting

In order to investigate the efficiency of the out-of-core GPU sorting technique explained in Section 5.3.3, we compare the performance of our our-of-core GPU sorting implementation with STL sort and Thrust OpenMP sort on TSUBAME2.5 using a single node. The objective of this experiment is to understand the effectiveness of the use of GPUs in the *Sort* operation when the input key-value data exceeds the capacity of GPU memory. Figure 5.8 shows the results, where the x-axis denotes the input number of key-value pairs in millions and the y-axis denotes the sorting rate on key-value pairs in millions per second. Note that the blue vertical bar between 100 and 150 on the x-axis denotes the border that the size of input data exceeds the capacity of device memory on a GPU when the input data increase. The results exhibit that our implementation performs 2.53x faster than STL sort at 285 million of the input key-value pairs. These results indicate that GPU can accelerate sorting performance even though the size of input data exceeds the memory capacity on a GPU; however, we also see the performance degradation in our implementation when the size of input data becomes large. This degradation is largely caused by the nature of the out-of-core GPU sorting algorithm; We conduct the in-core GPU sorting when the size of input data fits the capacity of device memory on the GPU, while conducting the out-of-core GPU sorting only when the input data exceeds the GPU memory capacity. The out-of-core GPU sorting algorithm introduces several additional instructions, such as multiple repetitions of chunk-based in-core GPU sorting and data transfers between CPU and GPU compared with the in-core GPU sorting; however, GPU sorting still has performance benefits even for large data sets that exceed the capacity of GPU device memory.

5.5.3 Balance between Scale-up and Scale-out

In order to investigate execution approaches whether we should use only device memory on GPUs (scale-out) or offload partial graph data to secondary CPU memory (scale-up) on a multi-node environment, we conduct performance studies on the balance of the number of compute nodes and GPUs per node. We vary the number of GPUs per node from 1 to 3 and use two patterns of the number of nodes: 512 and 1024. Then we set the three configurations: a) 1 GPU per node on 1024 nodes (1024 GPUs in total), b) 2 GPUs per node on 512 nodes (1024 GPUs in total), and c) 3 GPUs per node on 512 nodes (1536 GPUs in total), and compare the edge scan performance of each configuration. Figure 5.9 shows the results of the experiment, where the x-axis denotes the size of graphs in SCALE and the y-axis denotes the performance in ME/s (million

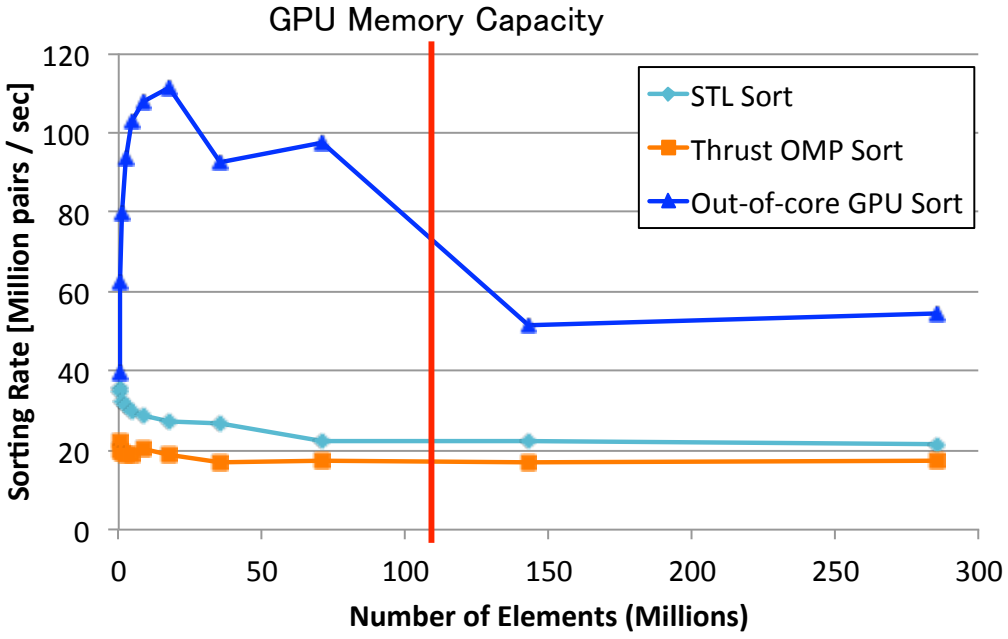

FIGURE 5.8: Results of out-of-core GPU sorting.

edges per second). We see that the configurations b) and c) exhibit 0.81x and 0.97x of the performance compared with the configuration a). The results indicate that we can obtain competitive performance results when we use a large number of GPUs per node in a small number of nodes.

Furthermore, we also investigate the power efficiency on scale-up and scale-out approaches on TSUBAME-KFC, each of which has 2 sockets of Intel Xeon E5-2620 v2 (Ivy Bridge EP, 2.10GHz, 6 cores) CPU, 64GB of DDR3 main memory, 4 devices of NVIDIA Tesla K20X GPU with 6GB of discrete GDDR5 memory connected to PCI-Express 2.0 × 16 buses, and 1 card of FDR InfiniBand HBA (56Gbps) connected to a single rail interconnect network, and runs on CentOS release 6.4. We use Open MPI 1.7.2 with GNU GCC 4.4.7 for the MPI implementation, and CUDA driver 5.5 and CUDA runtime 5.5 for the GPU implementation. We use a SCALE 27 graph and measure the elapsed time and the mean power consumption using the GPUs. Figure 5.10 shows the results of the performance and the power efficiency using three configurations: d). 32 nodes with 1 GPU per node, e). 16 nodes with 2 GPUs per node, and f). 8 nodes with 4 GPUs per node. Note that the three configurations use the same number of GPUs (i.e. 32 GPUs) in total. The results show that the simple scale-out approach d) performs the best in the three configurations in edge scan performance. On the other hand, the scale-up approaches e) and f) perform better power efficiency than the scale-out strategy, by 1.53x and 1.71x respectively. These results suggest that

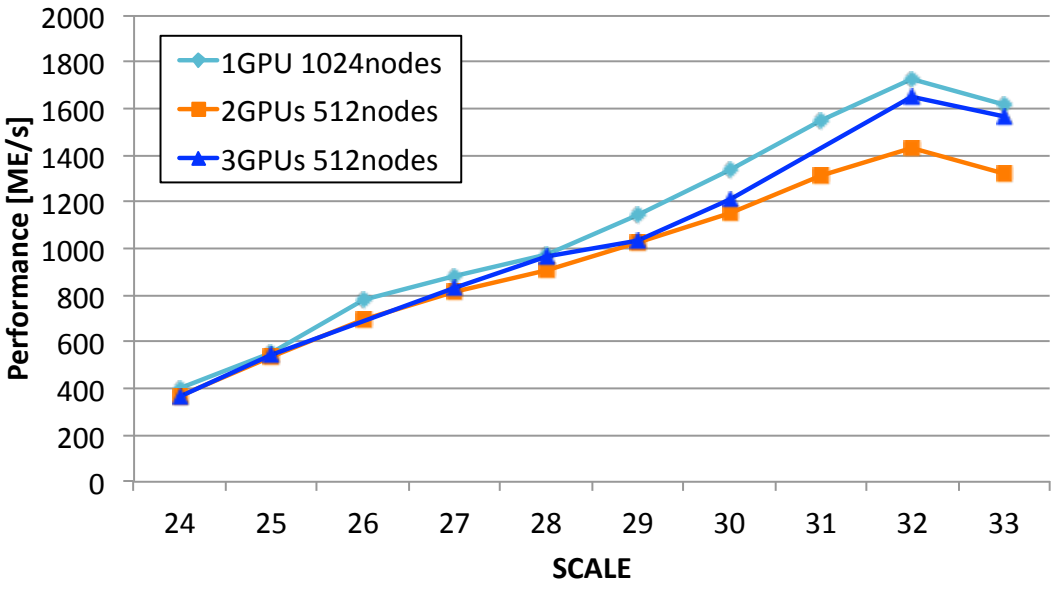

FIGURE 5.9: Results of performance of scale-up and scale-out strategies on TSUBAME2.5.

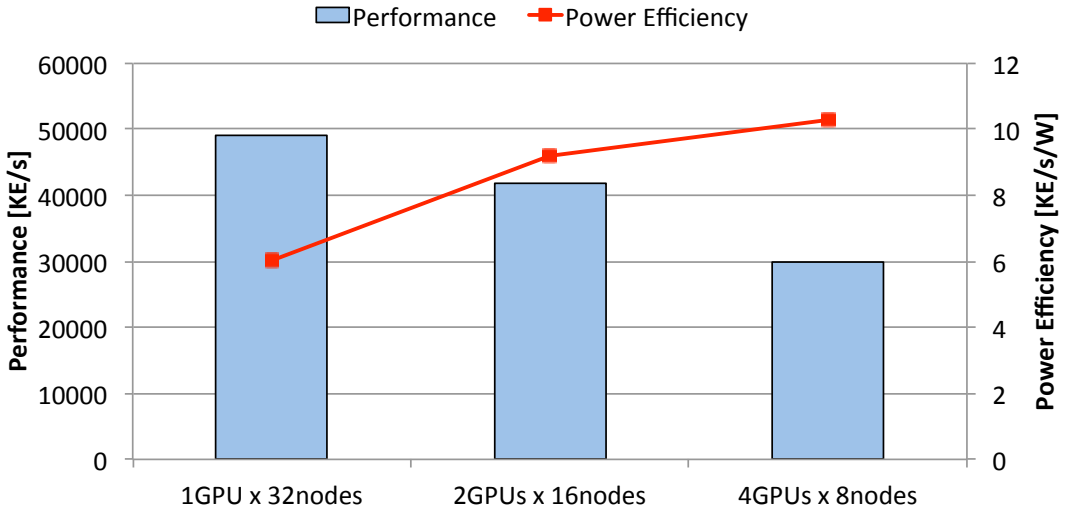

FIGURE 5.10: Results of the performance and the power efficiency on scale-up and scale-out strategies on TSUBAME-KFC.

the scale-up approach should be considered as an option for the architectures of next generation supercomputers, since the power efficiency is considered as one of the most important problems for future large-scale computing environments.

5.6 Summary

We proposed an out-of-core GPU memory management technique for large-scale MapReduce-based graph applications. The proposed technique handles memory overflows from GPUs by automatically dividing graph data into multiple chunks and overlaps CPU-GPU data transfer overheads as much as possible. Our experimental results on TSUBAME 2.5 using 1024 nodes (12288 CPU cores, 3072 GPUs) exhibit that our GPU-based implementation performs 2.10x faster than the CPU-based implementation on a graph with 17.18 billion vertices and 274.9 billion edges. We reveal that our GPU-based approach with out-of-core GPU data management can accelerate *Map* and *Reduce* phases by fully overlapping CPU-GPU data transfer and by applying several optimizations. We also show that scale-up approach performs better power efficiency than simple scale-out approach.

Chapter 6

Discussion

This chapter first describes discussion on applicability of the proposed techniques on MapReduce-based large-scale data processing on large-scale heterogeneous supercomputers in Section 6.1. Then, we also introduce another application case study of performance analysis of a MapReduce-based homology search algorithm on metagenomics for discussing advantage and disadvantage of our MapReduce implementation based on comparative implementation and performance with existing MapReduce implementations in Section 6.2.

6.1 Applicable Scope of the Proposed Techniques

This section describes additional discussion on applicability of the proposed techniques for MapReduce-based large-scale data processing on GPU-based heterogeneous supercomputers. We first discuss advantage and disadvantage of MapReduce-based large-scale data processing including comparison with other programming models including domain-specific systems such as other graph processing systems in Section 6.1.1. We then describe additional discussion of applicability of the proposed techniques on MapReduce-based large-scale data processing on GPUs in Section 6.1.2. Also, we discuss advantages and disadvantages of scheduling strategy of our proposed MapReduce implementation, especially in terms of static task scheduling used in our implementation compared with dynamic task scheduling used in other MapReduce implementations such as Hadoop in Section 6.1.3. At the end, we discuss applicability of our proposed techniques to future architecture in Section 6.1.4.

6.1.1 Pros and Cons of MapReduce-based Large-scale Data Processing

We discuss advantage and disadvantage of MapReduce-based large-scale data processing including comparison with other programming models and systems.

Advantages of MapReduce-based Large-scale Data Processing

An advantage of MapReduce-based large-scale data processing is that MapReduce is applicable to a wide range of applications including both data-intensive and compute-intensive applications. The data-intensive applications on MapReduce include large-scale graph processing and large-scale text mining. The compute-intensive applications on MapReduce include large-scale sequence alignment and large-scale clustering.

In addition to the wide applicable scope, MapReduce includes another advantage that MapReduce can handle scalable and hierarchical data management automatically. As for scalability, MapReduce handles multi-node execution automatically by distributing input data onto multiple memories before processing then processing multiple data on multiple nodes in parallel. MapReduce also handles data exchange among a large number of memories for aggregating data from and to different nodes. When we use many-core accelerators such as GPUs, we can use multiple GPUs in parallel as well.

As for hierarchical data management, MapReduce can also handle data movement between multiple memories and local disks on a single node for automatic out-of-core data management and fault tolerance. For example, data movement between CPU host memory and a local disk enables large-scale data processing whose data size exceeds CPU host memory capacity. When we use many-core accelerators such as GPUs, we can also handle out-of-core memory of accelerators by moving data from memory on an accelerator to CPU host memory.

Disadvantages of MapReduce-based Large-scale Data Processing

On the other hand, MapReduce also includes disadvantages: suffering performance overhead from the MapReduce programming model itself. Firstly, MapReduce includes communication overhead in Shuffle phase. Shuffle phase includes redundant data copy and synchronization compared with other systems such as domain specific programming models and systems. Although Shuffle phase conducts all to all data exchange operation on whole data among all nodes and synchronization among all the nodes, an application

may require only one to one synchronization or partial data exchange such as boundary data exchange in stencil computation.

Secondly, MapReduce may also incur redundant data movement overhead for certain applications. MapReduce programming model requires whole data to read while an application may only require partial data to read. For example, in breadth first search computation, only frontier vertices and corresponding edges are required to be read while the MapReduce programming model reads whole vertices and edges. On the other hand, certain other graph processing applications are well suited to MapReduce, which explores all paths in parallel such as PageRank, Random Walk with Restart, Connected Component, Diameter Estimation.

MapReduce also includes computation overhead from the MapReduce programming model. Some phases of MapReduce may be redundant compared with other systems depending on application characteristics. For example, a Map phase of GIM-V computation is redundant since the phase only passes input data for subsequent Reduce phase. However, this computation overhead can be eliminated by skipping redundant phases. For example, we can skip a Map phase in GIM-V.

Applicable Scope of MapReduce-based Large-scale Data Processing by Comparing with BSP

In terms of comparison with other programming models and systems, the BSP (Bulk Synchronous Parallel) model is a parallel computation model introduced by Valliant [96]. The BSP model is a programming model which designs parallel algorithms for distributed computing environments. BSP algorithms proceed in supersteps in each of which processors receive input at the beginning, perform some computation asynchronously, and communicate any output from a processor to another processor among the processors at the end. Barrier synchronization is used at the end of every superstep to synchronize all the processors.

There exists BSP-based systems for large-scale graph processing such as Pregel by Google [44] and Apache Hama [97]. Pregel develops certain graph processing applications such as PageRank, single source shortest paths (SSSP), bipartite matching, and a semi-clustering algorithm. Hama also develops certain BSP-based applications such as matrix inversion, PageRank, breadth first search. An advantage of BSP-based graph processing is that BSP keeps vertices and edges on the process that performs computation and uses network transfers only for massages to be used in the destination process,

while MapReduce requires passing the entire state of the graph from one stage to the next.

Earlier work revealed that BSP can be more effective than plain MapReduce in certain sets of graph processing applications as recognized theoretically in [98] and empirically in [97, 99]. Pace analyses the relationship between MapReduce and BSP including simulating BSP on MapReduce model with cost modeling of BSP and MapReduce [98]. They introduce theorems including a theorem simulating BSP on MapReduce, and the theorem reveals that MapReduce can implement BFS efficiently only if the order of the cost of storing local data over all supersteps is equal to the communication cost not including the cost of reading the input and writing the output. This theorem is based on the fact that BSP and MapReduce are different in that BSP can store local data while MapReduce clears the primary local memory and the data is stored in global memory.

Based on the theorem, They also give a few examples including sorting, dense matrix multiplication, and breadth first search to determine whether MapReduce should be used to implement BSP, and prove that sorting and dense matrix multiplication can be implemented in MapReduce efficiently while breadth first search cannot since the I/O cost of local data is larger than the communication cost. According to the theorem, a set of graph traversal algorithms such as breadth first search and single source shortest path cannot be implemented in MapReduce efficiently since these algorithms store frontier and neighbor vertices while transfer only neighbor vertices therefore the cost of storing local data is larger than the communication cost without I/O.

Their work, however, do not consider memory overflow for extremely large-scale data processing. when we consider memory overflow for processing large-scale data, MapReduce can handle out-of-core execution while BSP cannot. When we consider handling memory overflow, BSP has to offload local data onto external memory. Hutchinson proposed the EM-BSP model, which is an extension of the BSP model to include secondary local memories in [100]. In the case of EM-BSP, a set of graph traversal algorithms such as breadth first search are still inefficient in MapReduce since these algorithms only processes frontier and their neighbor vertices in a single step. However, certain graph processing algorithms such as PageRank and betweenness centrality in EM-BSP can be implemented in MapReduce efficiently, since these algorithms store all the vertices which each processor is responsible and transfer all the neighbor vertices.

6.1.2 Applicable Scope of the Proposed Techniques on GPUs

We have proposed techniques for MapReduce-based large-scale data processing on GPU-based heterogeneous supercomputers including techniques for scalability to hundreds to thousands of GPUs and hierarchical GPU memory management for handling out-of-core GPU memory by offloading data from GPU device memory to CPU host memory. We have used the PageRank graph processing application with Kronecker graphs as an instance of large-scale data processing which includes similar properties with real world large-scale data processing.

Criteria for Determining Whether to use GPU

In this section, we discuss applicability of the proposed techniques especially focusing on applicable scope that GPU can accelerate over CPU. First, we discuss basic criteria when to use GPU. Basic idea is that we should use GPU when running on GPU is faster than running on CPU. We consider how we can define the performance on GPU. In the proposed techniques we apply a chunk-based overlapping technique between computation on GPU and data transfer between CPU and GPU. When we overlap the three operations, the performance is dominated by one of the three operations. If elapsed time of computation on GPU is longer than elapsed time of data transfer from CPU to GPU or from GPU to CPU, the performance is dominated by the computation on GPU. Otherwise, the performance is dominated by the data transfer from CPU to GPU or from GPU to CPU. We should run on GPU when the dominating performance on GPU is better than the performance on CPU. For example, if the performance on GPU is dominated by computation and the computation is faster than CPU, we should run on GPU and otherwise we should run on CPU.

Required Properties for Acceleration by using GPU

Whether computation on GPU is faster than on CPU depends on application characteristics such as computational intensity, memory access pattern, processing data size. Although finding whether GPU performs faster than CPU before comparing these actual performance is difficult, we further discuss the applicable scope of utilizing GPU including memory access pattern and processing data size studies in the following.

The first required property of the application for accelerating by using GPU is the SIMD computational pattern. The SIMD (Single instruction, multiple data) computational

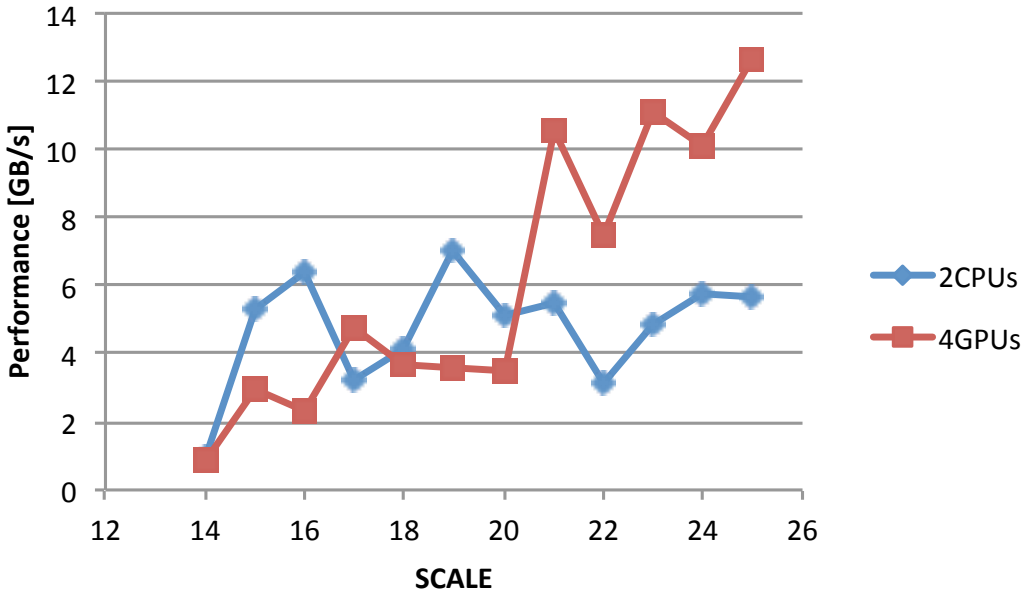

FIGURE 6.1: Comparison on different graph data size with CPU on a single node of TSUBAME-KFC

pattern is the computational pattern that can be performed the same operation on multiple data points simultaneously. The SIMD computational pattern is well suited to GPU since computation on GPU can be highly accelerated by applying the same operation on multiple data in parallel.

The second property to be accelerated on GPU is large input data size enough to hide latency on GPU computation. GPU has a property that GPU can hide latency of data movement on a GPU by switching groups of threads (i.e. warps) and GPU can hide the latency when the input data size is large enough to overlap data movement and computation on the GPU.

We conduct performance experiments for understanding how much input data size affects performance on GPU. We compare performance on 4GPUs with 2CPUs on Kronecker graphs with (a, b, c, d) = (0.57, 0.19, 0.19, 0.05) of different sizes from SCALE 14 to SCALE 25 on a single node of TSUBAME-KFC. We apply overlapping of computation on GPU and CPU-GPU data transfer and set the number of chunk to 4. Figure 6.1 shows results of performance on the different graph sizes. The x-axis indicates input graph data size in SCALE and the y-axis indicates performance of throughput in Gigabytes per second. Note that the input data do not fit on the GPU memories with equal to or larger than SCALE 24. The results exhibit that 4GPUs performs 2.23x faster than 2CPUs and 12.68 GB/s on SCALE 25. On the other hand, 4GPUs perform around 3

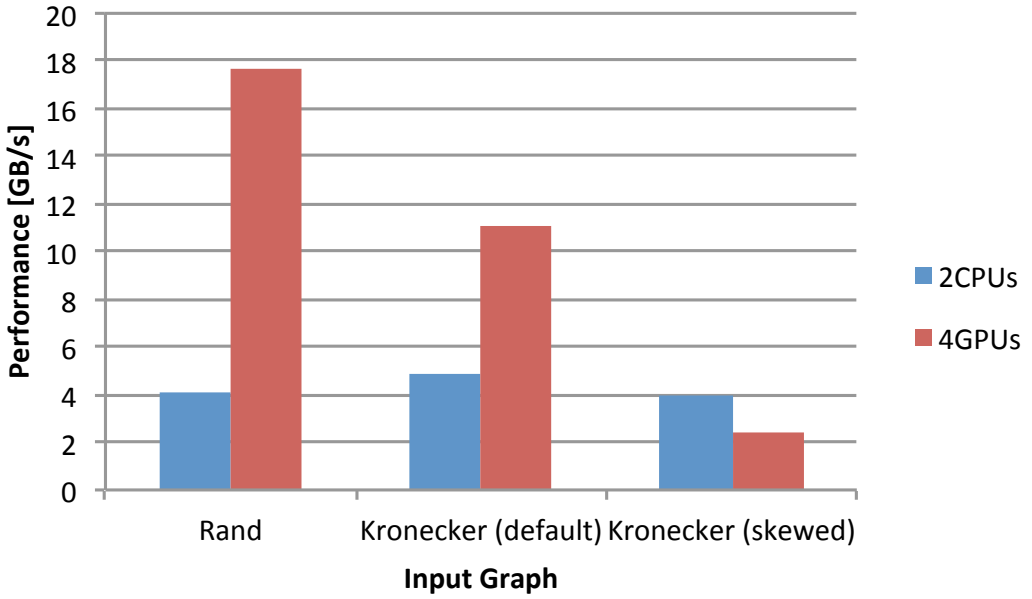

FIGURE 6.2: Comparison on different graph data type with CPU on a single node of TSUBAME-KFC using SCALE 23

GB/s when input graph data size is smaller than SCALE 20. These results indicate that the performance on GPU also highly depends on input data size. The results indicate we should use graph data size over SCALE 21; which corresponds to around 570 MB on 4GPUs which is equivalent to around 140 MB per GPU.

The third property to be accelerated on GPU is that input graph data contains low data skew. Data skew indicates that input data consists of imbalanced data distribution which corresponds to a small number of keys include large number of corresponding values while the other keys include small number of corresponding values. For example, Kronecker graphs typically consist of skew edge lists. GPU is suitable for low data skew since GPU computes multiple data in parallel with multiple threads and skewed data incurs load imbalance among threads. More precisely, GPU computes with a unit of warp (i.e. typically 32 threads) and load imbalance among the threads lead to performance overhead since GPU is suitable for parallel computation while is not suitable for sequential operation.

In order to understand how much data skew affects performance on GPU, we conduct comparative performance experiments with CPU on different data. We use three graph data; random graph data, Kronecker graph with (a, b, c, d) = (0.57, 0.19, 0.19, 0.05) (we call this graph Kronecker default), and a highly skewed Kronecker graph with (a, b, c, d) = (0.80, 0.05, 0.05, 0.10) (we call this graph Kronecker skewed) and compare

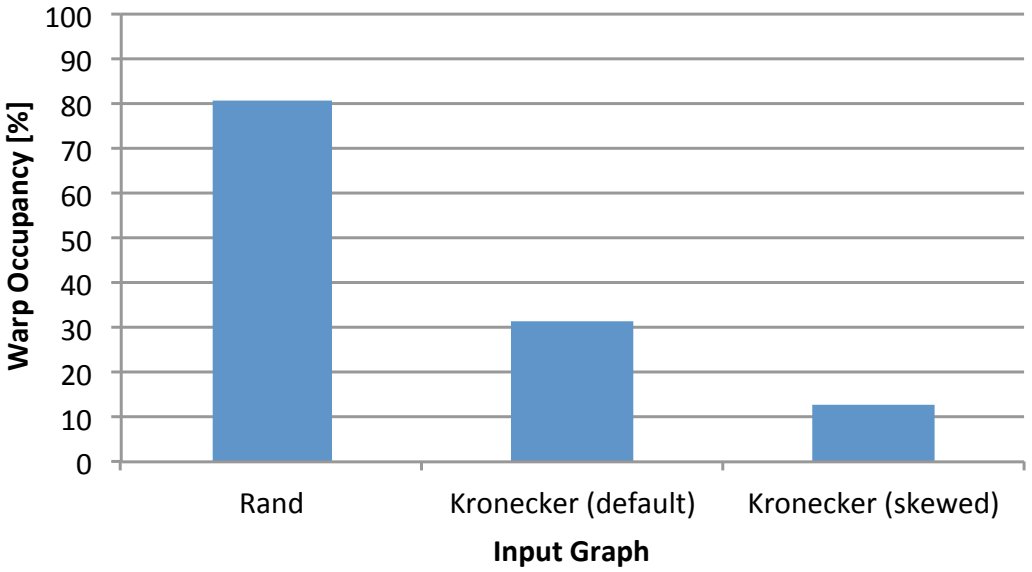

FIGURE 6.3: Achieved warp occupancy on different graph data type on 1GPU of TSUBAME-KFC using SCALE 21

performance on 4 GPUs with 2CPUs on a single node of TSUBAME-KFC. We apply overlapping of computation on GPU and CPU-GPU data transfer and set the number of chunk to 4. We use problem size of SCALE 23 and we compare performance of Combine2 operation on Reduce Stage 1 in GIM-V-based PageRank computation.

Figure 6.2 shows results of performance on the three graph types. The x-axis indicates input graph type and the y-axis indicates performance of throughput in Gigabytes per second. The results exhibit that 4GPUs perform 4.32x faster than 2CPU and 17.6 GB/s on the random graph. This throughput indicates that 4GPUs perform close to the maximum bandwidth of CPU-GPU data transfer rate through PCI-E. On the other hand, the results on the Kronecker graphs exhibit that 4GPUs perform slower than the case of the random graph; 4GPUs perform 2.26x faster than 2CPU and 8.31 GB/s on Kronecker default, and 0.60x of 2CPU and 2.38 GB/s on Kronecker skewed. Also, Figure 6.3 shows results of achieved warp occupancy on the three graph types using SCALE 21 on 1GPU of TSUBAME-KFC. The x-axis indicates input graph type and the y-axis indicates achieved warp occupancy in percentage. The results exhibit that warp occupancy is highly correlated with the performance in Figure 6.2. These results indicate that the performance on GPU highly depends on the input graph data structure. This data dependency on GPUs derives from the fact that some vertices are connected with thousands to millions of vertices as we have seen in Figure 4.2 and Figure 4.21, while a single warp (e.g. 32 threads in the case of Tesla K20X) is assigned to a vertex and

edges connected with the vertex in our implementation as we presented in Section 5.4.3. We consider the main reason of the data dependency derives from the characteristics of GPU computation: GPU is suitable for coalesced memory access and is not suitable for work load imbalance among threads. However, we consider this limitation can be overcome and perform even faster than the current performance on the random graph, by applying packing techniques for SpMV such as the padded JDR format [101] to the vertices and edges sorted by the degree of the vertices.

Limitations to Applications with Variable Key/Value Lengths

In terms of our proposed techniques, we assume types of key and value are fixed length types such as int or float. Our proposed techniques including task-scheduling-based load balance optimization, out-of-core GPU memory management, and optimization techniques for GPUs including the external GPU sorting and warp-based thread mapping on GPU assume that key and value types are fixed length. Applications with variable length of key or value such as variable sizes of string keys or values in text processing are not applicable to the proposed techniques and applying these types to the proposed techniques will be future work.

Application examples of large-scale data processing with fixed length key and value include machine learning applications such as k-means clustering, linear regression, and principal component analysis as described in [102], and graph processing applications such as PageRank, random walk with restart, connected components, diameter estimation as described in [6], minimum spanning trees, maximal matchings, approximate weighted matchings, approximate vertex and edge covers and minimum cuts as described in [103], as well as graph-based approaches for DNA sequence assembly and the analysis of protein-protein interaction networks as described in [104].

On the other hand, application examples of large-scale data processing using web crawling with variable length include word count from [3], and inverted index, similarity score, page view count, page view rank as described in [27]. These applications use crawled web data of HTML files or informations such as URL, IP, Cookie.

6.1.3 Comparison of Static and Dynamic Task Scheduling

In this section, we compare the two task scheduling strategy in MapReduce: static and dynamic task scheduling. In chapter 4 and 5, we have proposed a multi-GPU-MapReduce-based large-scale graph processing where we have extended an existing

single GPU MapReduce implementation to multi-GPU. Mars is based on static task scheduling since GPUs do not support dynamic thread scheduling [27]. Therefore, our multi-GPU-based implementation adopts static task scheduling as well across multiple GPUs, though we assign multiple chunks dynamically on each GPU. Twister also adopts static task scheduling for supporting catchable map/reduce tasks during multiple MapReduce iterations by fixing the cached locations [39].

On the other hand, other CPU-based MapReduce implementations such as Google's original MapReduce [3] and Hadoop [7] adopt dynamic task scheduling for improving load balancing and speeds up recovery when a worker fails by spreading tasks out across other worker machines [3].

Pros and Cons of Static Task Scheduling in MapReduce

The static task scheduling in MapReduce has advantages and disadvantages compared with the dynamic task scheduling. An advantage of the static task scheduling in MapReduce is low scheduling overhead of data movement across machines, especially in the Map phase. In the static task scheduling in our implementation, each worker reads chunks of input data which the worker is responsible to at the beginning of execution then starts map tasks as soon as finishing reading the input data onto CPU host memory and no additional data movement across machines until the Shuffle phase starts. In addition, the static task scheduling is suitable for our proposal of out-of-core GPU memory management since the proposed stream-based overlapping between computation and communication on a GPU works efficiently by assigning multiple chunks on each GPU and processing each chunk in a streaming manner. Also, the static task scheduling is beneficial for iterative computations since the static task scheduling enables the locations of the map/reduce tasks to be remain fixed for caching local data through multiple iterations.

On the other hand, the static task scheduling also includes an disadvantage compared with the dynamic task scheduling. The static task scheduling may lead load imbalance among the workers after the Shuffle phase, since value length of each key which the Reduce phase receives depends on input data structure, algorithms of the Map phase, and splitters used in the Shuffle phase. We have seen large-scale graph data has scale-free property (power-law degree distribution) and consists of highly skewed value lengths in the Reduce phases in GIM-V as described in Section 2.3.1. Although we have minimized the load imbalance after the Shuffle phase by introducing a task scheduling-based load

balancing in Section 4.7.1, additional preprocessing is required for applying the load balance optimization.

Pros and Cons of Dynamic Task Scheduling in MapReduce

The dynamic task scheduling also has advantages and disadvantages. An advantage is that the dynamic task scheduling can achieve relatively balancing work loads among workers well without preprocessing by setting chunk size of each task as small as all the workers can receive tasks. The dynamic load balancing works efficiently especially when the number of worker machines is small relative to input data size since the chunk size can be set relatively large for distributing all the workers, which results in small number of tasks in total and less overhead of task scheduling. The dynamic task scheduling is also effective for hybrid task scheduling such as CPU-GPU hybrid task scheduling [105].

The dynamic task scheduling, however, also has disadvantages. First, the dynamic task scheduling incurs additional overhead of data movement across machines as well as task scheduling overhead of communicating between master and worker machines, compared with the static task scheduling. Also, the dynamic task scheduling may suffer load imbalance when setting the chunk size of a task large relative to the number of workers, since some workers may receive smaller number of tasks or even may not receive any task due to the small number of tasks in total. Parameter surveying for setting the chunk size as well as setting other parameters including the number of slots each worker handles is required for achieving optimal performance.

We further compare our static scheduling-based MapReduce implementation empirically with Hadoop and Spark, popular dynamic task scheduling-based existing MapReduce implementations, in Section 6.2.

6.1.4 Applicability of the Proposed Techniques to Future Architectures

This section describes discussion on applicability of our proposed techniques on GPU-based heterogeneous supercomputers to architectures to be developed in the future. We firstly discuss performance bottlenecks on current architectures based on the results of our proposals, followed by additional discussion on future architectures including possible factors to improve and to affect performance. We then also discuss balance of scale-up and scale-out approaches on future architectures.

Performance Bottlenecks on Present Architectures

We have analyzed performance of our proposed MapReduce implementation including performance breakdown in the previous chapters. The results have shown that the Shuffle phase including local sort affects performance significantly such as in Figure 4.12 in chapter 4 and in Figure 5.6 in chapter 5. The results have shown that the Shuffle phase and local sort take around 31.0% and 19.1% of the total execution time on 768 GPUs (3GPUs per node using 256 nodes) in Figure 5.6. We have also seen that data transfer between CPU and GPU through PCI-E bandwidth affects performance on GPUs in map, reduce, and local sort operations, since PCI-E bandwidth is around 30x slower compared with bandwidth inside a GPU. Also, in our current implementation, Each process reads input data using MPI-IO before starting computation and writes output on its local disk as we have been introduced in Section 4.6.2. In this implementation, since disk I/O is conducted separately from computation and is not overlapped, disk I/O bandwidth also affects performance.

Possible Factors to Affect Performance on Future Architectures

The improvement of CPU-GPU bus interconnect bandwidth can make it possible to improve the performance of map, reduce, and local sort operations, with the next generation of interconnects such as PCIe4.0 and NVIDIA NVLink. Also, I/O bandwidth improvement between CPU and local disks can improve the performance of reading input and writing output, with Non-Volatile Memory (NVM) such as PCI-E-attached flash memory.

As the number of nodes increases, performance gap between computational performance and network communication may increase. For example, on TSUBAME, TSUBAME1.2 performs 69.3TB/s of memory bandwidth on 680 Tesla S1070 GPUs and 2GB/s of network bisection bandwidth on 170 compute nodes, while TSUBAME2.5 performs 1056TB/s of memory bandwidth on 4224 Tesla K20X GPUs and 8GB/s of network bisection bandwidth on 1408 compute nodes, whose performance gap between bandwidth on GPUs and network bandwidth gets 3.81x larger. A possible reason of this performance gap is that the number of nodes and performance per GPU increase more rapidly than the increase of network bandwidth. Therefore, network bandwidth can be a possible performance bottleneck on future architectures.

Balance Between Scale-up and Scale-out Approaches

Since using larger number of nodes may affect performance by limited network bandwidth, we consider using fewer number of nodes by assigning larger data per node can improve performance. Therefore, optimizing balance between scale-up and scale-out approaches in terms of using larger data per node (scale-up) and increasing the number of node (scale-out) can improve performance based on balance between bandwidth of offloading to external memory such as NVM and network bandwidth. On present architecture, scale-out works more efficiently as the results have shown in Figure 5.9 in Section 5.5.3, since the network bandwidth is relatively better than the PCI-E bandwidth. On future architecture, scale-up may work more efficiently when I/O bandwidth and PCI-E bandwidth increases relative to network bandwidth by the new generations of bus interconnects such as NVLink and NVM.

6.2 Performance Analysis of a MapReduce-based Homology Search Algorithm

In order to understand efficiency of our proposed MapReduce implementation especially in terms of task scheduling we discussed in Section 6.1.3, we conduct empirical performance analysis based on comparison with existing MapReduce implementations on CPUs. Although we have revealed that our proposed MapReduce implementation achieves significantly better performance over Hadoop in Section 4.8.2, we further conduct in-depth analysis. In this section, we implement a MapReduce-based homology search algorithm in metagenomics as another application case study.

6.2.1 Motivation

Homology search to be used in emerging bioinformatics problems such as metagenomics is of increasing importance and challenge as its application area grows more broadly while the computational complexity is increasing. One way to cope with the increasing complexity is to utilize massively parallel data processing. Required dataset for homology search in metagenomics consists of queries and database, each of whose size will reach Gigabytes to Terabytes, and total data size to compute will grow to product of these two datasets (i.e. Exabytes to Zettabytes). BLAST [56, 57] is proposed as a basis of homology search algorithms and there have been a lot of efforts on improving the algorithm. Earlier work by some of the authors have devised novel algorithms such as

GHOSTX [62] and extend the algorithm to distributed computing environments. Their work has demonstrated their implementation scales well on existing supercomputers including TSUBAME2.0 [94] and K computer [106], but the master-worker parallelization to enumerate and schedule for data processing was done with their privately developed MPI-based master-worker framework called GHOST-MP.

An alternative to using GHOSTX is to utilize the now-popular big data software substrates, such as MapReduce with abundant associated software tool-chains, but it is unclear how to apply MapReduce to extremely large-scale homology search in an efficient way. Firstly, It is not obvious how to design and implement homology search algorithms onto the MapReduce model. Specifically, how to handle two different dataset called queries and database which homology search algorithms receive using MapReduce is not straightforward. Secondly, performance characteristics of MapReduce-based implementations of homology search should be considered in order to achieve high performance homology search.

By converting the GHOSTX master-worker data processing pipeline to accommodate MapReduce, and benchmarking them on a variety of high performance MapReduce incarnations including Hadoop [17], Spark [19], and our proposed implementation, we attempt to characterize the appropriateness of MapReduce as a generic framework for metagenomics that embody extremely resource consuming requirements for both compute and data. We consider two different MapReduce-based designs of homology search considering data allocation of queries and database. Then we implement one of the designs onto Hadoop, Spark, and ours, and conduct performance analysis on real world dataset in metagenomics. We also compare our MapReduce-based implementations with GHOST-MP, an existing distributed implementation of GHOSTX on MPI-based master-worker framework.

6.2.2 Introduction to Homology Search

Homology search or alignment search is an approach to identify genes based upon homology with genes that are already publicly available in sequence databases by using a search algorithm. Homology search is used in the field of Metagenomics, the study of genetic material recovered directly from environmental samples for advancing knowledge in a wide variety of application domains, such as medicine, engineering, agriculture, ecology. Homology search algorithms are used as tools for life science researchers to gain a set of high-scoring pairs from an exhaustive list of protein coding sequences similar

to a given query sequence, such as the amino-acid sequence of different proteins or the nucleotides of DNA sequences.

BLAST (Basic Local Alignment Search Tool) [56, 57] is proposed as a fast homology search algorithm and its implementation is widely used as a standard homology search tool. BLAST applies a heuristic algorithm much faster than previous approaches such as a full alignment procedure using the Smith-Waterman algorithm [58] or FASTA [59]. Figure 6.4 shows an overview of BLAST workflow. Firstly, BLAST finds seeds that are substring of database sequences similar to the substrings of a query sequence. Then, BLAST makes alignments by extending those seeds without gaps, and then similar, nearby seeds are brought together by a chain filter. Finally, BLAST makes alignments from seeds with gaps.

There have been a lot of efforts for improving BLAST [60, 61]. These efforts achieve speedup from the BLAST algorithm by improving search algorithms. GHOSTX [62] adopts the seed-extend alignment algorithm used by BLAST. GHOSTX achieved approximately 131-165 times faster than BLAST. GHOSTX finds seed that are highly similar segments between database sequences and the query sequence. Next, GHOSTX obtain alignments by extending those seeds without gaps for larger similar regions. Finally, GHOSTX make alignments by extending the seeds with gaps. In order to accelerate the seed search process, GHOSTX constructs suffix array both for the query and the database before the search. In addition, instead of fixing the length of a seed like BLAST, GHOSTX extends it till the matching score exceeds a given threshold to reduce the computation time for untapped extension while not losing the sensitivity.

There exists also an extension of GHOSTX for distributed computing environments. GHOST-MP is built on GHOSTX with MPI library for homology search on supercomputers like K computer and TSUBAME, or general PC clusters. It achieves distributed paralleling search process through a master-worker style. In GHOST-MP's algorithm, it accomplishes I/O optimization for paralleled file system by utilizing locality of database chunks to achieve high speed processing.

6.2.3 Large-scale Bioinformatic Applications

MapReduce-based bioinformatics implementations have been studied [107, 108, 109, 63, 110, 111, 112, 113]. Their work indicate a wide range of applications using MapReduce related to bioinformatics as well as show high scalability on clusters and clouds using existing MapReduce implementations such as Hadoop. Their work focus on introducing

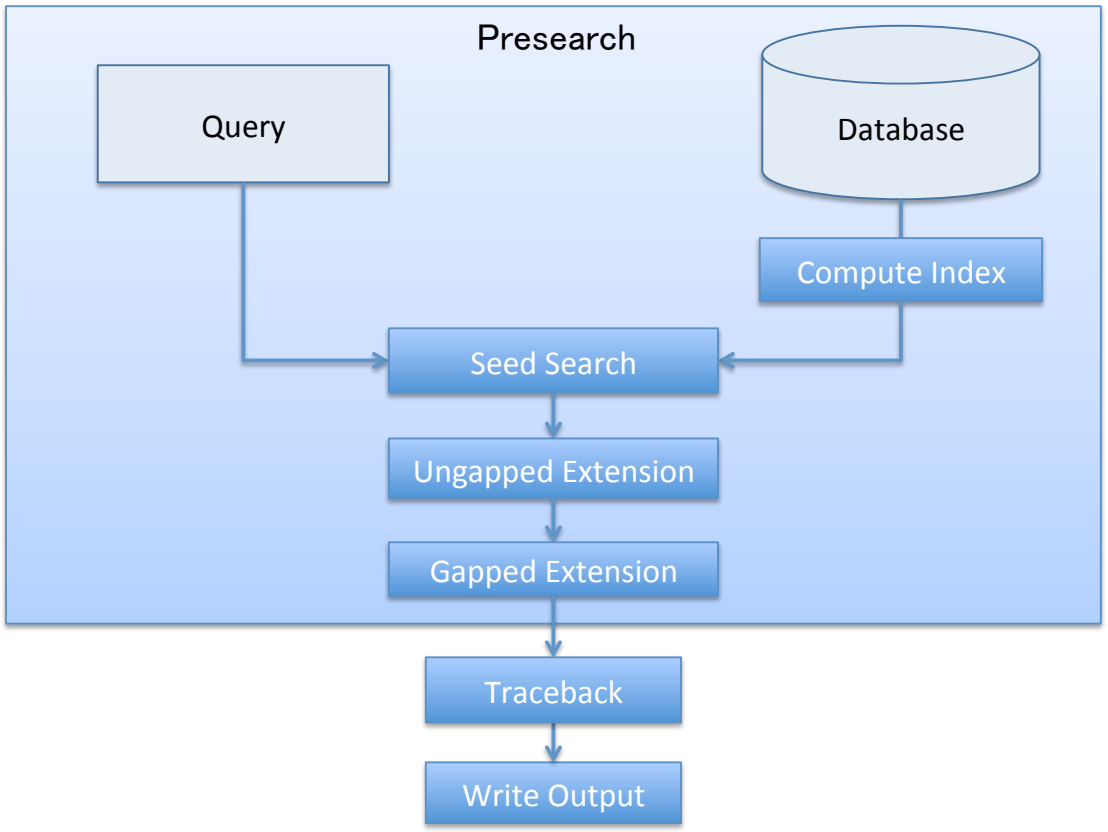

FIGURE 6.4: Workflow of homology search

algorithms or demonstrating scalability on relatively small number of nodes. However, our work focus on high performance and scalable homology search using MapReduce on large-scale computing environment such as supercomputers and analyze high performance MapReduce implementations.

K MapReduce (KMR) [114] is a MPI-based MapReduce implementation for large-scale supercomputers such as K computer. KMR optimizes shuffle operation by collective communication utilizing interconnect on K computer. Their work also conducted experiments using GHOST-MP by replacing master-worker tasking library in GHOST-MP with KMR. Although their work achieved high communication and I/O performance on K computer, they did not compare with other existing MapReduce implementation. We compare multiple MapReduce implementations and investigate high performance MapReduce-based homology search.

There have also been efforts on MPI-based parallelization of bioinformatics applications. mpiBLAST [115] is a MPI-based parallelization of BLAST that achieves high scalability by optimizing allocation of database. mpiBLAST applys database segmentation which

distributes a chunk of database to each node and let each node searches a unique portion of database. While mpiBLAST is high optimized for BLAST, our work focus on MapReduce-based high performance homology search since MapReduce is more widely used framework and can handle memory overflow and compute node failures.

6.2.4 Designs of a Homology Search Algorithm on MapReduce

We describe how to design homology search on MapReduce. Our main idea is to parallelize query data onto multiple Mappers. We consider two different designs based on how to assign query data and database onto worker nodes. On the two designs, query data is distributed onto the worker nodes on both designs while database allocation strategies are different. Note that we assume computing environments equip local disk on each compute node.

MapReduce-based Design with Database Replication

We describe a design of homology search on MapReduce using database replication. Query data is distributed on worker nodes while database is replicated among the worker nodes. Figure 6.5 describes how MapReduce works on the design. First, input query data files are copied to a distributed file system (e.g. HDFS) and the database file is replicated onto local disk on each compute node. After putting query and database, a client submits a job with a MapReduce application binary. A homology search application is called in map function of the MapReduce application. After submitting the application, each Mapper runs the homology search application with a split of query data and whole database for each map function the Mapper calls. A Mapper emits outputs of homology search for each query. Whole set of results from map functions is simply the final result.

This database replication design is useful when the size of database is small, since the result of each query is directly computed using whole database for each query. When the whole database can fit on local disk on each node, runtime can utilize locality of database. On the other hand, when the size of database is large, not only it may not fit on local disks but also parallelization efficiency may decrease because of the reduction in the locality of the database.

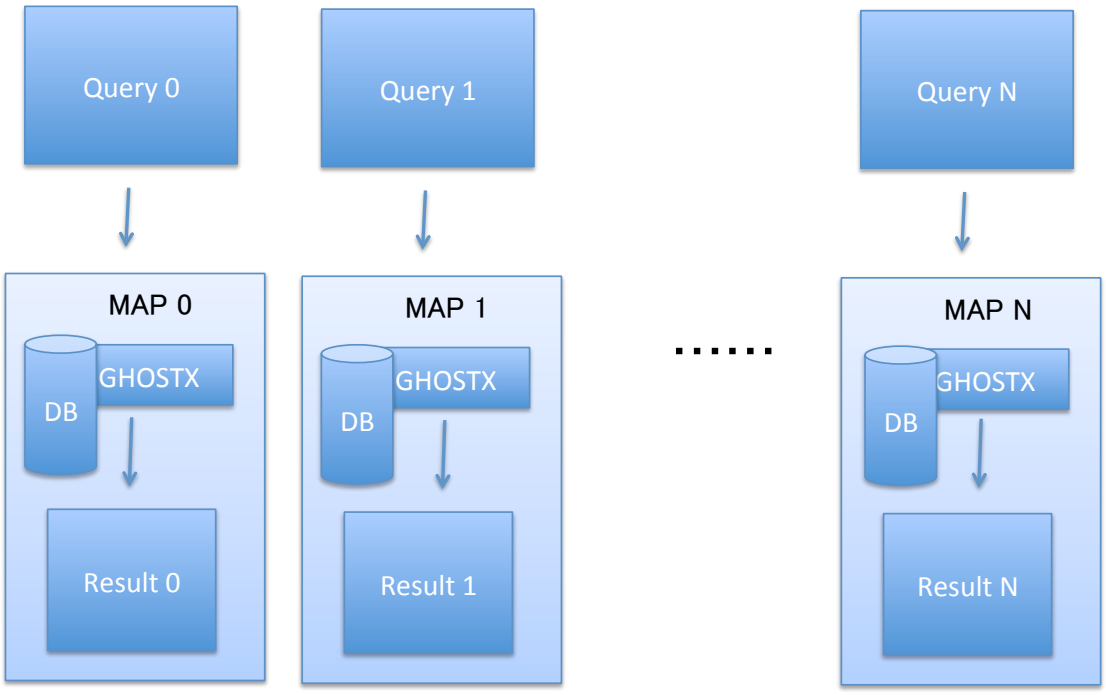

FIGURE 6.5: Design of homology search with replicated database

MapReduce-based Design with Database Distribution

We consider another design that distributes database as well as query data. Query data is distributed on worker nodes and database is also distributed on the worker nodes. Figure 6.6 describes how the design works. First, input query data files are copied to distributed file system in the same way as the database replication design. Database is split to multiple chunks and each chunk is distributed on each node. These chunks can be also replicated to multiple nodes when the number of nodes is larger than the number of chunks. After putting query and database, a client submits a job with a MapReduce application binary. While a homology search application is called in each map function in similar way as the database replication design, result of each map function is different in that the result is a partial search result with a chunk of database. The results of Mappers are passed to Reducers and the Reducers merge the partial search results into a final search result for each query.

An advantage of this database distribution design is that the task granularity is smaller which can result in better parallelization efficiency. The number of tasks (i.e. the number of map function calls) with this database distribution design is larger than the database replication design since the database is divided to multiple chunks and each chunk is assigned to a Mapper. Having large number of tasks might not always be good; locality

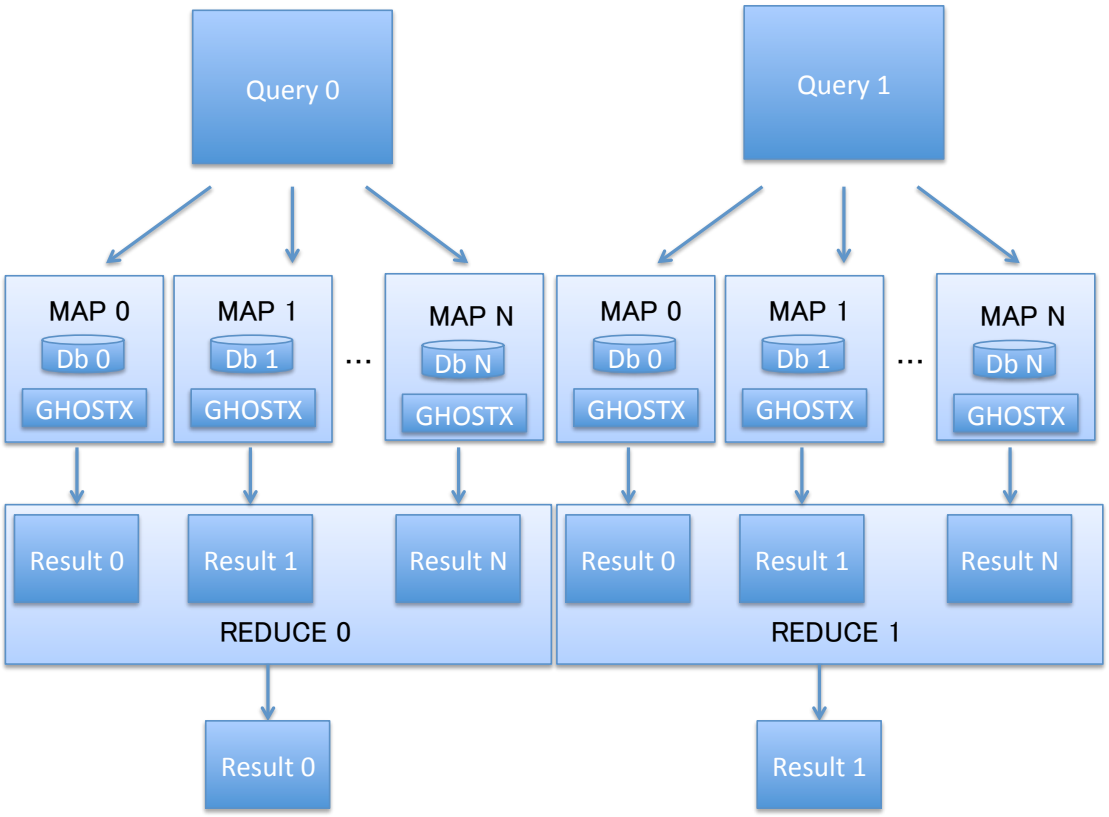

FIGURE 6.6: Design of homology search with distributed database

of database may become worse since each map function requires a specific chunk, which may result in multiple movements of chunks among worker nodes.

6.2.5 Implementations of a Homology Search Algorithm on MapReduce

We implement MapReduce-based homology search on existing multiple MapReduce implementations. We use GHOSTX as a sequential implementation and extend it onto the MapReduce model. We implement the database replication design described in Section 6.2.4 on Hadoop and Spark.

Implementation on Hadoop

In order to use GHOSTX on top of Hadoop, we need a way to call C++ from Java since GHOSTX is written in C++ while Hadoop is written in Java. There are several ways for calling GHOSTX from Hadoop, including Hadoop Pipes, Hadoop Streaming, and Java Native Interface. Hadoop Pipes is a library that allows C++ source code to

```
hadoop pipes\
  -D hadoop.pipes.java.recordreader=true\
  -D hadoop.pipes.java.recordwriter=true\
  -files [db_files]\
  -input [input_dir]\
  -output [output_dir]\
  -inputformat WholeFileInputFormat\
  -program ghostmr
```

FIGURE 6.7: Calling GHOSTX from Hadoop Pipes. `ghostmr` is the compiled binary program incorporated original GHOSTX with a Hadoop Pipes application.

be used for Mapper and Reducer code. Hadoop Pipes provides C++ API of map and reduce functions and users write the functions in C++ according to input and output formats provided by Hadoop. Hadoop Streaming is a more generic API that allows programs written in any language to be used as Mapper and Reducer implementations. While Hadoop Pipes and Hadoop Streaming are similar in that they split the application code into a separate process, they are different in that Hadoop Pipes uses serialization to covert the types into bytes that are sent to the process via socket, while Hadoop Streaming uses Unix standard streams as the interface. Java Native Interface (JNI) is a programming framework that enables Java code running in Java Virtual Machine (JVM) to call native applications and libraries written in other language such as C++. We select Hadoop Pipes since it provides closer interface with Java-based Mapper and Reducer. We modify the interface of original GHOSTX program so that Mapper can call GHOSTX program and setting query and database files through HDFS.

In order to distribute query and database files, we use different approaches for each dataset. As for query files, we use HDFS in a standard way for distributing multiple query files onto local disks on each node. We distribute the query files by the following command; `hdfs dfs -put [query_files] [input_dir]`. On the other hand, we do not distribute but copy the same database files onto each node since the database files are identical among all the nodes. To do this, we use `-files` option provided by Hadoop Pipes which copies specified files to cluster. As for query files, we need to avoid splitting them since the design of replicated database assigns one whole query file per Mapper, and Hadoop splits input data into lines and assign each line per map function by default. In order to disable splitting a query file into multiple splits, we implement WholeFileInputFormat for Hadoop Pipes based on [17]. We pass the customized input format to Hadoop Pipes by using `-inputformat` option. We run our GHOSTX on Hadoop by the following command described in Figure 6.7.

```
spark-submit\
  --class "GhostMR"\
  --master yarn-client\
  --num-executors [num_nodes]\
  --executor-cores [num_threads]\
  --files [db_files]\
  --jars lib/hadoop-mapreduce-client-core-[ver].jar\
  ghostmr.jar
```

FIGURE 6.8: Calling GHOSTX from Spark. `ghostmr.jar` is the compiled bytecode incorporated original GHOSTX with a Spark application.

Implementation on Spark

As with the case of Hadoop, we need a way for calling C++ from Scala since GHOSTX is written in C++ while Spark is written in Scala. Spark provides resilient distributed dataset (RDD) `pipe()` operation, which pipes each partition of RDD through a shell command in the same way as Unix pipe operation. RDD `pipe()` operation receives RDD input and sends output through Unix standard input and output. We apply GHOSTX to the `pipe()` operation, by simply executing GHOSTX binary program in `pipe()`.

In order to pass input files to Spark, we assign query files through HDFS and assign database files by copying to local disks on each node. In order to assign query files through HDFS to Spark, we put the query files to HDFS before running the application. We need to avoid splitting them since the Map-only design assigns one whole query file per Mapper as with the case of Hadoop. In order to disable splitting a query file into multiple splits, we apply WholeFileInputFormat for Spark. We pass the customized input format to Spark by using `-jars` option with the jar file including WholeFileInputFormat. During running the application, it reads the query files from HDFS using `SparkContext.textFile()` method onto a RDD, then the RDD passes the query files to `pipe()`. As for database files, we copy them using `--files` option provided by Spark similar to Hadoop. Figure 6.8 describes the actual command for submitting GHOSTX on Spark.

Implementation on Our MapReduce Framework

We use CPU-based implementation of our MapReduce to call GHOSTX, since GHOSTX is implemented for CPU. We integrate GHOSTX directly into map function on our implementation, since our implementation is implemented in C++ for CPU implementation and GHOSTX is also implemented in C++. Although our implementation has a feature

```
# query   database   output
query.0   database   output.0
query.1   database   output.1
...
query.n   database   output.n
```

FIGURE 6.9: An example of table file for GHOSTX on our implementation.

```
$ mpirun -n [num_nodes] -hostfile [host_file]\
  ghostmr -t [table_file]
```

FIGURE 6.10: Calling GHOSTX from our implementation. `ghostmr` is the compiled binary incorporated original GHOSTX with an application on our implementation.

to run multiple map tasks in parallel using OpenMP, we do not use the feature and call one map task per node since GHOSTX itself can be run using OpenMP.

In order to pass input files to our MapReduce, we use table file that consists of tuples of query file name, database name, and output file name in the similar way as GHOST-MP. A table file is consisting of tab-delaminated tuples of query, database, and output files per line, and each tuple will be passed to a worker node in runtime. An example of table file is described in Figure 6.9. In Figure 6.9, different query files and output files are specified per line, while a whole database file is specified on all lines, since we apply the replicated database design where the query files are distributed and the identical database file is replicated. Our MapReduce reads the table file at the beginning of execution then input query files are assigned onto multiple nodes according to the table file. We run our GHOSTX on our implementation by the following command described in Figure 6.10.

6.2.6 Performance Analysis

In order to understand performance characteristics of MapReduce implementations, we conduct comparative performance experiments. We compare the elapsed time of homology search using existing MapReduce implementations as well as a MPI-based master worker implementation in order to investigate effectiveness of MapReduce-based implementation. We conduct data size scaling using different datasets as well as scaling of using multiple compute nodes. We use 13MB and 130MB of query data which is reduced from originally 1.1GB of query data named `SRS014107` obtained from Data Analysis and Coordination Center for Human Microbiome Project website (`http://www.hmpdacc.org/`) [116]. We use 1.1GB of FASTA database which is reduced from originally 30GB of database named `nr` obtained on November 4th, 2014 from The National Center for

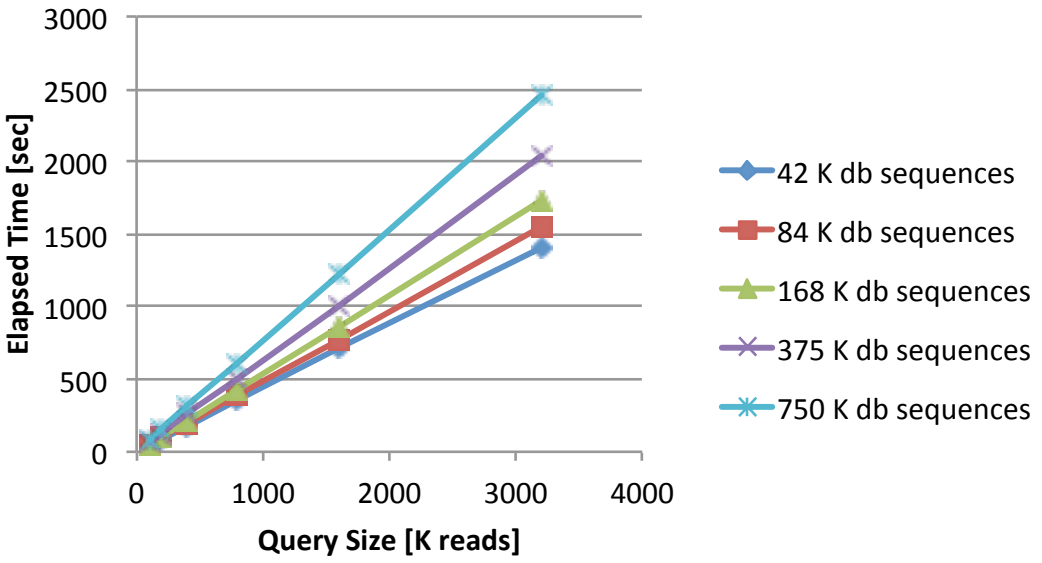

FIGURE 6.11: Elapsed time of query size scaling on single node

Biotechnology Information website (`http://www.ncbi.nlm.nih.gov/`) [117]. Note that we split input query files into 10MB of smaller files before putting them to HDFS for Hadoop and Spark, since we use WholeFileInputFormat as we described in Section 6.2.5. Note that we do not include the elapsed time of database construction nor the time of data placement to local disk or HDFS.

We use TSUBAME-KFC as a computing environment. A node on TSUBAME-KFC contains 2 sockets of Intel Xeon E5-2620 v2 (Ivy Bridge EP, 2.10GHz, 6 cores) CPU, 64GB of DDR3 main memory, 4 devices of NVIDIA Tesla K20X GPU with 6GB of discrete GDDR5 memory connected to PCI-Express 2.0 $\times$ 16 buses, and 1 card of FDR InfiniBand HBA (56Gbps) connected to a single rail interconnect network, and runs on CentOS release 6.4. We use Open MPI 1.7.2 with GNU GCC 4.4.7 for the MPI implementation. We use Hadoop version 2.4.1, Spark version 1.1.0, and GHOST-MP version 1.2.1. We use YARN scheduler on Hadoop and Spark. We use OpenMP for GHOSTX and GHOST-MP using 24 threads per node and use local SSD for placing query data and database as well as for writing output results. We build GHOST-MP with original configuration, without defining CHUNK and IOMASTER parameters. We use one worker process per node for GHOST-MP and set optional parameters to be equal to that of GHOSTX. We do not apply OpenMP parallelization for Hadoop and Spark, since the YARN scheduler may assign multiple tasks onto each node according to free resource on each node.

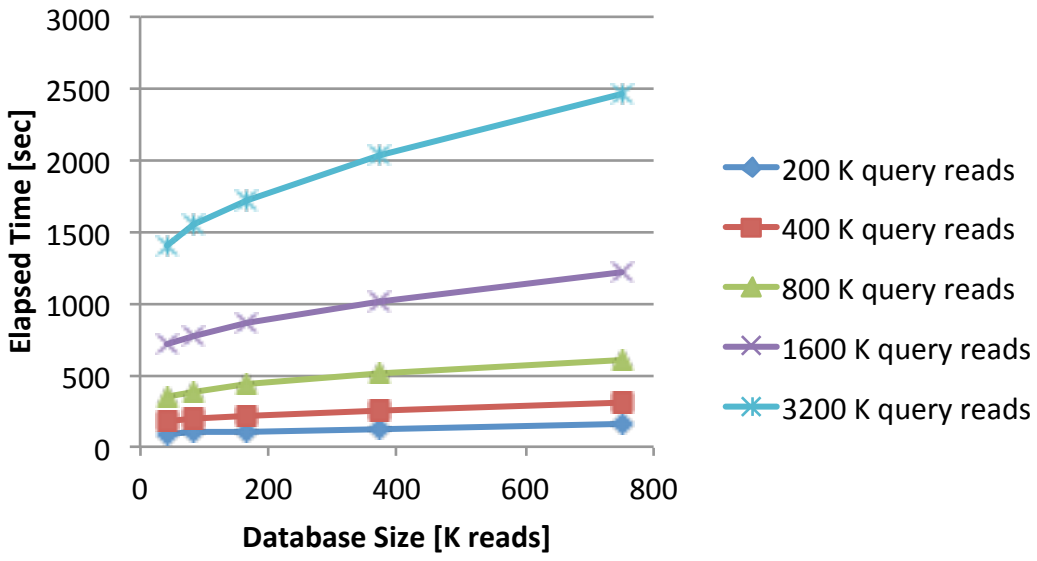

FIGURE 6.12: Elapsed time of database size scaling on single node

Data Size Scaling

First we conduct data size scaling of GHOSTX using single node with different datasets. We conduct two types of data size scaling; query size scaling with different database size, and database size scaling with different query size. Figure 6.11 shows the performance results of query data size scaling. The x-axis indicates query data size and the y-axis indicates elapsed time of homology search. Each line indicates elapsed time on different query size with five sets of fixed database sizes. The results show that the elapsed time increases in proportion to query size. On the other hand, Figure 6.12 shows the elapsed time of database size scaling. The x-axis indicates database size and the y-axis indicates elapsed time of homology search. Each line indicates elapsed time on different database size with five sets of fixed query sizes. The results show that the elapsed time does not increase in proportion to database size, as opposed to the query size scaling results. When we consider multiple node scaling, this unproportional database size scaling would result in poor scaling of distributing db, since dividing database into smaller chunks is not considered to scale linearly. On the other hand, distributing query would scale well, since dividing query size into smaller chunks is considered to scale near linearly. Therefore, we employ performance analysis on the replicated database design which we introduced in Section 6.2.4 in the following subsections.

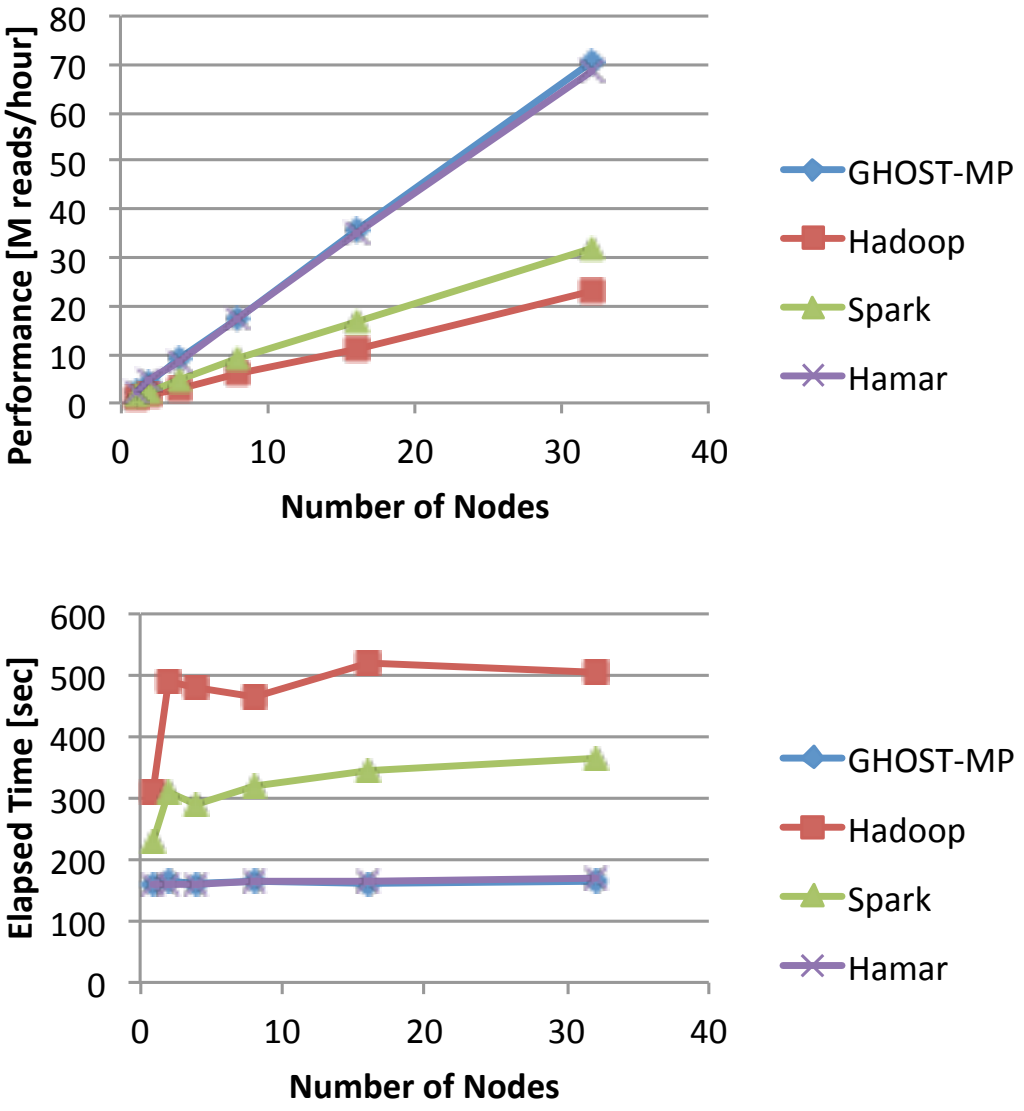

FIGURE 6.13: (Top) Performance of weak scaling, (Bottom) Elapsed time of weak scaling, with 13MB of query per node and 1.1GB of database

Weak Scaling

We also conduct weak scaling experiments on the different implementations of the replicated database design using up to 32 nodes. We fix the size of database to 1.1GB and use two different query sizes: 13MB per node and 130MB per node. Figure 6.13 shows the performance and elapsed time of weak scaling using 13MB of query size per node. Also, Figure 6.14 shows the performance and elapsed time of weak scaling using 130MB of query size per node. The x-axis indicates the number of nodes and the y-axis indicates

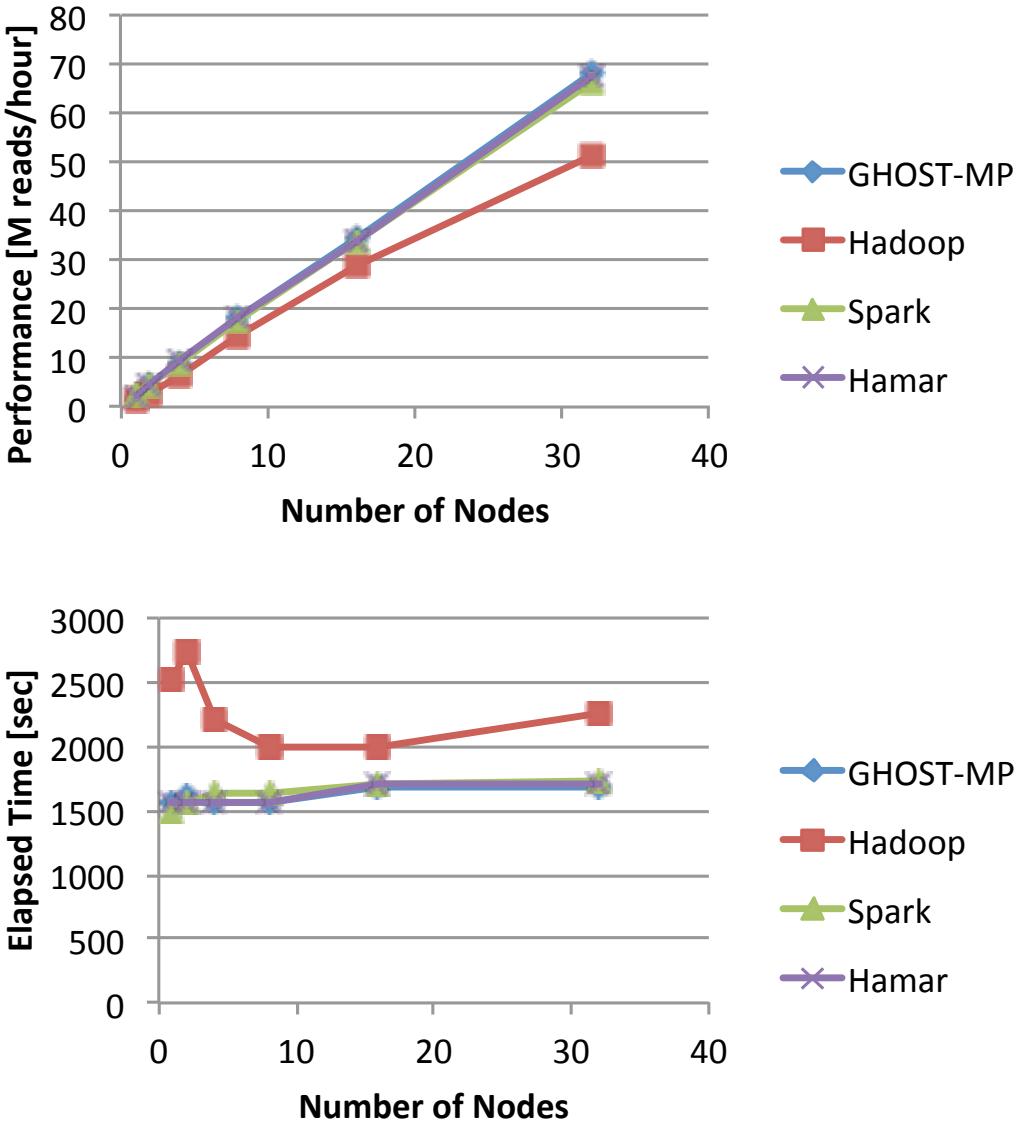

FIGURE 6.14: (Top) Performance of weak scaling, (Bottom) Elapsed time of weak scaling, with 130MB of query per node and 1.1GB of database

millions of query reads per hour on the top of Figure 6.13 and Figure 6.14, and elapsed time in second on the bottom of Figure 6.13 and Figure 6.14.

The results indicate that all the implementations exhibit good scalability. We consider the results comes from the facts that homology search mainly consists of computational and I/O operations as well as the application includes little communication since computation of each query is independent of other queries. Another possible reason is that the implementations have little possibility to suffer load imbalance since workload we

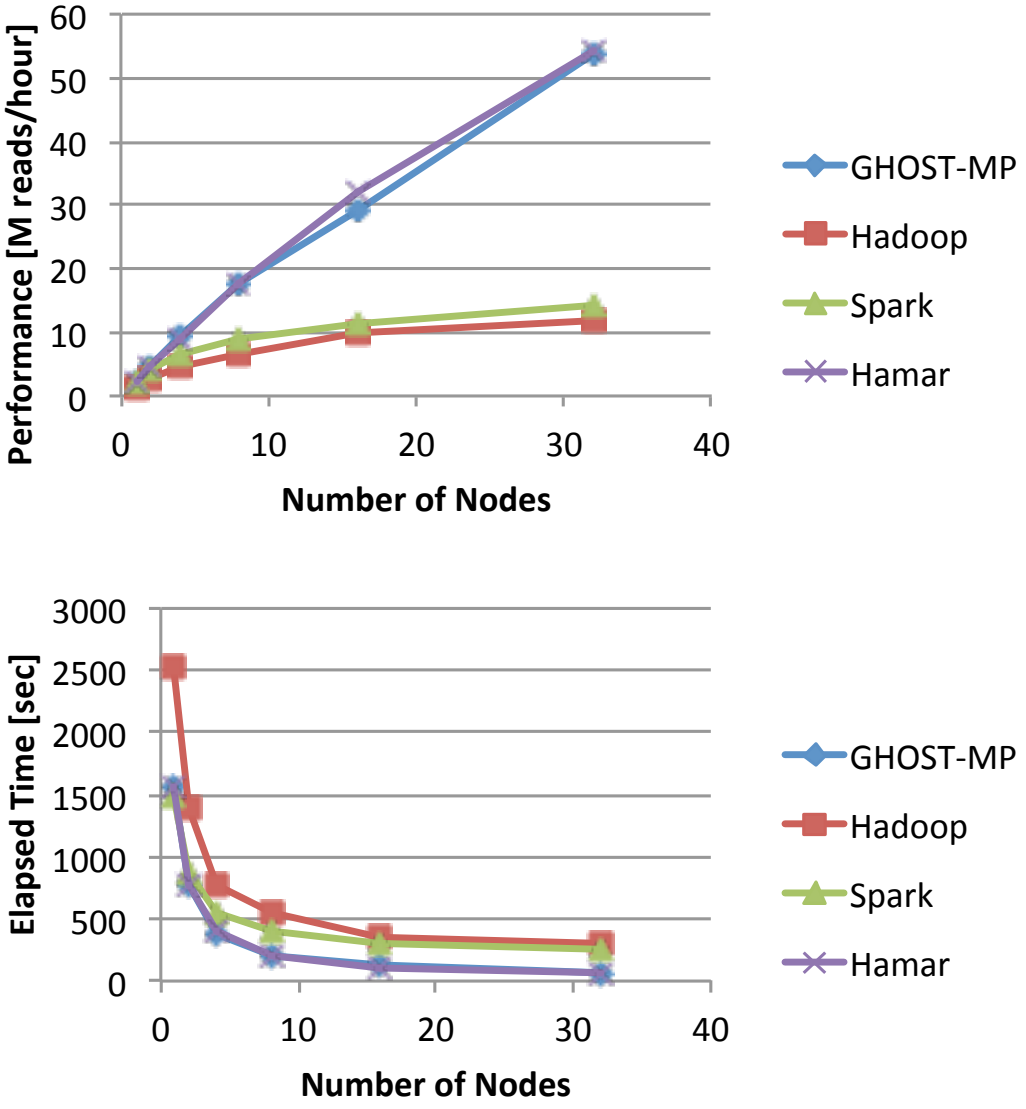

FIGURE 6.15: (Top) Performance of strong scaling, (Bottom) Elapsed time of strong scaling, with 130MB of query and 1.1GB of database

use is well balanced. The results also show that our implementation exhibits comparable performance with GHOST-MP, and the performance of Spark and Hadoop highly depends on the query size; i.e. these implementations perform similar with 130MB of query while slower with 13MB of query. Our implementation performs 2.16x and 2.99x faster than Spark and Hadoop on 13MB query respectively. A possible reason of this query data size dependency is that these two implementations suffer overhead of dynamic task scheduling and involved data movements onto multiple nodes through HDFS. On the other hand, our implementation does not suffer the scheduling and data movement overheads since our implementation assigns query data onto multiple nodes evenly at

the beginning statically in the similar way as GHOST-MP. The bottom of Figure 6.13 and Figure 6.14 also indicates elapsed time increases significantly when using two nodes on Hadoop and Spark. We consider a possible reason of this time increase is additional task scheduling overhead of YARN by using multiple nodes.

Strong Scaling

We also conduct strong scaling experiments using up to 32 nodes. We fix the size of database to 1.1GB. Figure 6.15 shows the performance and elapsed time of strong scaling using 130MB of query. The x-axis indicates the number of nodes and the y-axis indicates millions of query reads per hour on the top of Figure 6.15 and elapsed time in second on the top of Figure 6.15.

The results show that all the implementations scale well on small number of nodes, while our implementation exhibits better performance compared with Spark and Hadoop on larger number of nodes; 3.83x and 4.54x faster on 32 nodes respectively. The results also show that our implementation exhibits similar performance with GHOST-MP. This performance degradation on Spark and Hadoop derives from the fact that the query size per node gets smaller as the number of nodes increases then dynamic task scheduling and involved data movement overheads get larger ratio out of the total elapsed time. On the other hand, our implementation scales better since our implementation assigns equivalent amount of query data onto multiple nodes statically in the similar way as GHOST-MP. We further investigate performance characteristics of the three implementations by understanding the resource usage.

Resource Usage

In order to understand performance characteristics of MapReduce-based homology search implementations, we investigate the resource usage of CPU, disk I/O, and network. We conduct the experiments on 32 nodes with 13MB of query per node, and 500MB of database, using `dstat` command on a single node to get CPU usage, the amount of read/write on local disk, and the amount of send/receive over network.

Figure 6.16 shows usage of CPU and read/write on Hadoop, Spark, and our implementation. The x-axis indicates elapsed time in second and the y-axes indicate CPU usage in percentage and the amount of read/write on local disk in million bytes per second. s Figure 6.17 shows network usage on Hadoop, Spark, and our implementation. The x-axis indicates elapsed time in second and the y-axis indicates the amount of send/receive

over network in million bytes per second. The results exhibit that Hadoop and Spark conduct significant amount of I/O and network data transfer at the begging while our implementation does not. We consider these additional I/O and network data transfer on Hadoop and Spark derive from dynamic resource scheduling of YARN which moves significant amount of data among multiple nodes. The results also exhibit high CPU usage in the middle on all the implementations. This high CPU usage derives from homology search operations using OpenMP in GHOSTX. When we compare elapsed time during high CPU usage, our implementation takes smaller elapsed time than Hadoop and Spark; our implementation takes around 65 seconds while Hadoop and Spark take 150 seconds and 140 seconds respectively. We consider this elapsed time difference derives from task scheduling strategies of the YARN scheduler and the static scheduler on our implementation, since we observe YARN assigns multiple tasks on a node while our implementation assigns single task per node equivalently. At the end of execution, we see disk write caused by write output operation in GHOSTX. We also observe significant amount of elapsed time after the write output operation even on our implementation, which indicates there exists some amount of load imbalance among compute nodes. We consider this time difference derives from the fact that search time of a query varies by query sequence in GHOSTX.

6.2.7 Summary

In order to understand performance characteristics of MapReduce implementations on a compute-intensive large-scale data processing application, we present MapReduce-based designs and implementations of a homology search algorithm. We conduct comparative performance analysis of existing widely used MapReduce implementations as well as comparison with an existing MPI-based master-worker implementation of a homology search algorithm. The results indicate that our implementation performs not only good weak scaling but also good strong scaling and outperforms existing widely used MapReduce implementations significantly.

6.3 Summary

This chapter described discussion on applicability of the proposed techniques on MapReduce-based large-scale data processing on GPU-based heterogeneous supercomputers. We summarized applicable scope of our proposed techniques by reviewing advantages and disadvantages of MapReduce-based large-scale data processing, required properties for

acceleration by using GPUs, followed by comparison of the task scheduling strategy in our proposed implementation with dynamic task scheduling used in popular MapReduce implementations. Then, we also discussed applicability of our proposed techniques to future architectures by summarizing bottlenecks on present architectures and possible bottlenecks on future architectures.

We also introduce another application case study: performance analysis of a MapReduce-based homology search algorithm on metagenomics for discussing advantage and disadvantage of our MapReduce implementation based on comparative performance analysis with Hadoop and Spark. Hadoop and Spark apply dynamic task scheduling while our implementation applies static task scheduling. We show that our implementation outperforms existing widely used MapReduce implementations significantly, mainly by little overhead of task scheduling.

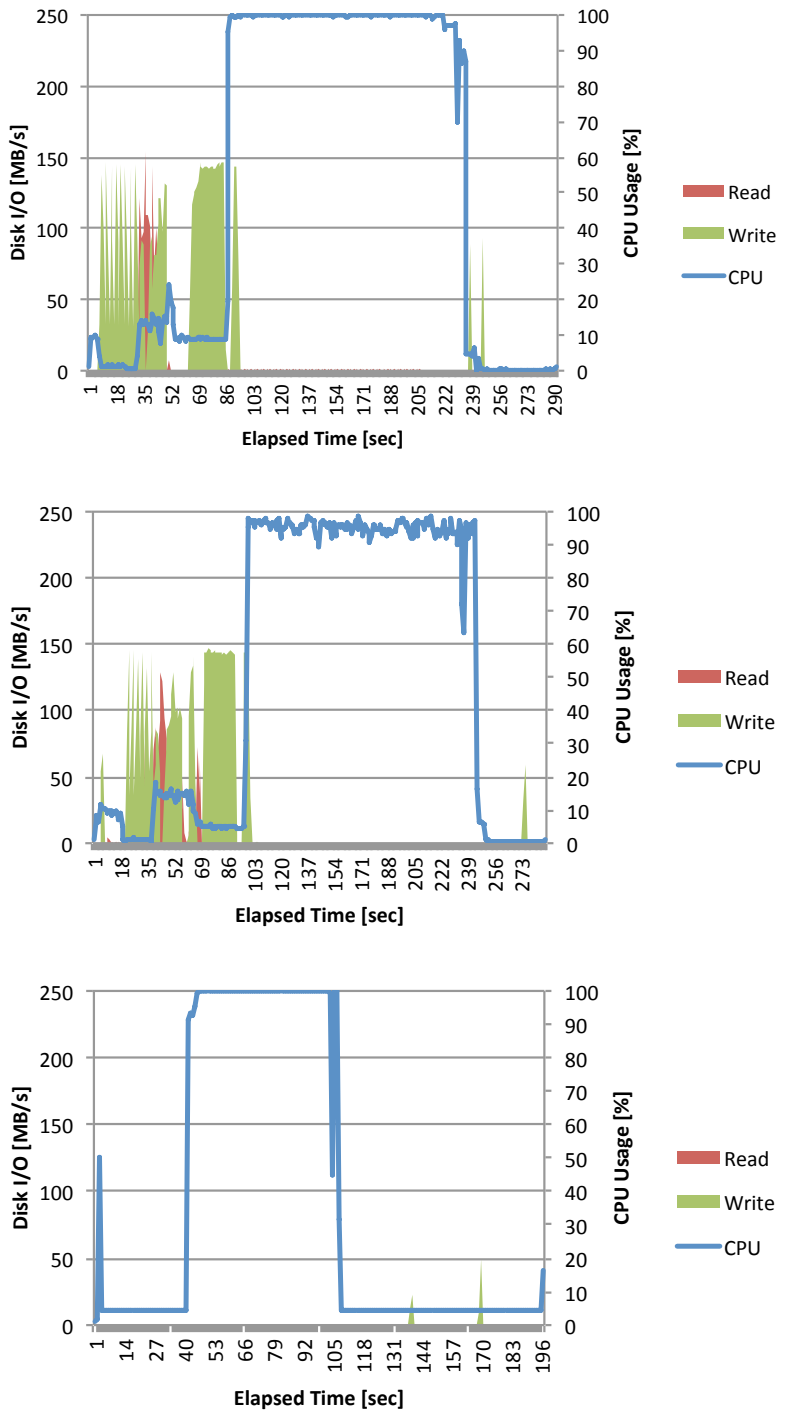

FIGURE 6.16: Resource usage of CPU and disk I/O on a node out of 32 nodes in total, using 13MB of query per node and 500MB of database (Top: Hadoop, Middle: Spark, Bottom: our implementation).

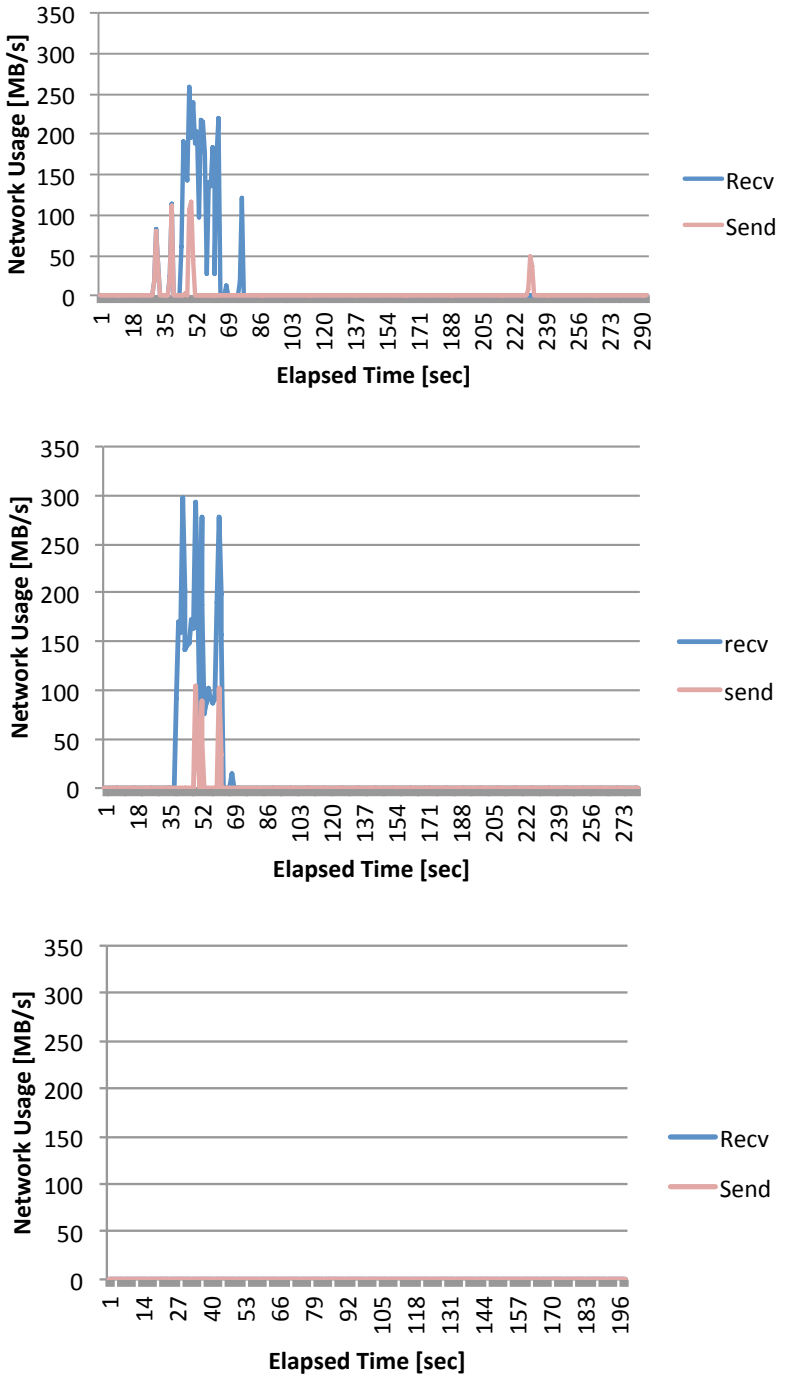

FIGURE 6.17: Network resource usage on a node out of 32 nodes in total, using 13MB of query per node and 500MB of database (Left: Hadoop, Middle: Spark, Right: our implementation).

Chapter 7

Conclusion

7.1 Conclusion

We propose scalable and hierarchical multi-GPU MapReduce-based large-scale data processing techniques for GPU-based heterogeneous supercomputers. Firstly, we introduce a GIM-V implementation with load balance optimization for multi GPU environments and conduct performance studies using our implementation on the TSUBAME2.0 supercomputer using 256 nodes (6144 hyper-threaded CPU cores, 768 GPUs). The results exhibit that our GPU-based implementation performed 87.04 ME/s on SCALE 30, and 1.52 times faster than the CPU-based native implementation on SCALE 29. We also show an approach for optimizing load balance.

Secondly, we proposed an out-of-core GPU memory management technique for large-scale MapReduce-based graph applications. The proposed technique handles memory overflows from GPUs by automatically dividing graph data into multiple chunks and overlaps CPU-GPU data transfer overheads as much as possible. Our experimental results on TSUBAME 2.5 using 1024 nodes (12288 CPU cores, 3072 GPUs) exhibit that our GPU-based implementation performs 2.10x faster than the CPU-based implementation on a graph with 17.18 billion vertices and 274.9 billion edges. We reveal that our GPU-based approach with out-of-core GPU data management can accelerate *Map* and *Reduce* phases by fully overlapping CPU-GPU data transfer and by applying several optimizations. We also show that scale-up approach performs better power efficiency than simple scale-out approach.

We also summarized applicable scope of our proposed techniques by reviewing advantages and disadvantages of MapReduce-based large-scale data processing, required properties for acceleration by using GPUs, followed by comparison of the task scheduling strategy in our proposed implementation with dynamic task scheduling used in popular MapReduce implementations. Then, we also discussed applicability of our proposed techniques to future architectures by summarizing bottlenecks on present architectures and possible bottlenecks on future architectures. Then, we also introduce another application case study: performance analysis of a MapReduce-based homology search algorithm on metagenomics for discussing advantage and disadvantage of our MapReduce implementation based on comparative performance analysis with Hadoop and Spark. We show that our implementation outperforms existing Hadoop and Spark significantly, mainly by little overhead of task scheduling.

7.2 Future Work

Our future work includes the use of Non-Volatile Memory (NVM) such as flash for handling the larger size of data than the CPU memory capacity. We consider investigating efficient hierarchical memory management techniques that utilize three-level memory layers including GPU device memory, CPU host memory, and Non-Volatile Memory. Also the balance between scale-up approach using NVM and scale-out approach using network is worth investigating for utilizing I/O and network performance efficiently.

We also consider applying fault tolerance techniques, such as data replication technique incorporated in Hadoop, or checkpoint/restart, which has been studied in a lot of researchers for resilient large-scale computing.

Bibliography

[1] NVIDIA Corporation. CUDA C programming guide. 2014. http://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[2] NVIDIA Corporation. NVIDIA 's next generation CUDA compute architecture: Kepler GK110. 2012. http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf.

[3] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. *OSDI '04, Sixth Symposium on Operating System Design and Implementation*, pages 137–150, 2004.

[4] Facebook. http://www.facebook.com.

[5] James Alfred Ang, Brian W. Barrett, Kyle Bruce Wheeler, and Richard C Murphy. Introducing the Graph 500, May 2010.

[6] U. Kang, Charalampos E. Tsourakakis, and Christos Faloutsos. PEGASUS: A peta-scale graph mining system implementation and observations. In *Proceedings of the 9th IEEE International Conference on Data Mining*, ICDM '09, pages 229–238, Washington, DC, USA, 2009.

[7] A. Bialecki, M. Cordova, D. Cutting, and O. O'Malley. Hadoop: a framework for running applications on large clusters built of commodity hardware.

[8] James Jeffers and James Reinders. *Intel Xeon Phi coprocessor high performance programming*. Newnes, 2013.

[9] Jack J Dongarra, Hans W Meuer, Erich Strohmaier, et al. TOP500 supercomputer sites. *Supercomputer*, 13:89–111, 1997.

[10] Introducing Titan. https://www.olcf.ornl.gov/titan/.

[11] Satoshi Matsuoka. The TSUBAME2.5 evolution. *Tsubame e-Science Journal*, 10:2 – 8, 2013. http://www.gsic.titech.ac.jp/en/TSUBAME_ESJ.

[12] Amazon elastic compute cloud (Amazon EC2). `http://aws.amazon.com/ec2/`.

[13] Pawan Harish and P. J. Narayanan. Accelerating large graph algorithms on the GPU using CUDA. In *Proceedings of the 14th International Conference on High Performance Computing*, HiPC'07, pages 197–208, Berlin, Heidelberg, 2007. Springer-Verlag.

[14] Tianji Wu, Bo Wang, Yi Shan, Feng Yan, Yu Wang, and Ningyi Xu. Efficient PageRank and SpMV computation on AMD GPUs. In *39th International Conference on Parallel Processing*, ICPP '10, pages 81–89, Sept 2010.

[15] John D.Owens, Mike Houston, David Luebke, Simon Green, John E.Stone, and James C.Phillips. GPU computing. *Proc IEEE*, 96(5):879–899, 2008.

[16] Khronos Group Open Computing Language. `http://www.khronos.org/opencl/`.

[17] Tom White. *Hadoop: the definitive guide: the definitive guide.* " O'Reilly Media, Inc.", 2009.

[18] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for multi-core and multiprocessor systems. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 13 –24, feb. 2007.

[19] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 10–10, 2010.

[20] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[21] Avraham Shinnar, David Cunningham, Vijay Saraswat, and Benjamin Herta. M3R: increased performance for in-memory Hadoop jobs. *Proceedings of the VLDB Endowment*, 5(12):1736–1747, 2012.

[22] X10: Performance and productivity at scale. `http://x10-lang.org/`.

[23] Tiankai Tu, Charles A Rendleman, David W Borhani, Ron O Dror, Justin Gullingsrud, MO Jensen, John L Klepeis, Paul Maragakis, Patrick Miller, Kate A

Stafford, et al. A scalable parallel framework for analyzing terascale molecular dynamics simulation trajectories. In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pages 1–12. IEEE, 2008.

[24] T. Hoefler, A. Lumsdaine, and J. Dongarra. Towards efficient MapReduce using MPI. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 16th European PVM/MPI Users' Group Meeting*. Springer, Sep. 2009.

[25] Yu-Fan Ho, Sih-Wei Chen, Chang-Yi Chen, Yung-Ching Hsu, and Pangfeng Liu. A Mapreduce programming framework using message passing. In *International Computer Symposium (ICS)*, pages 883 –888, dec. 2010.

[26] Steven J. Plimpton and Karen D. Devine. MapReduce in MPI for large-scale graph algorithms. *Parallel Computing*, 37(9):610–632, September 2011.

[27] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K.Govindaraju, and Tuyong Wang. Mars: A MapReduce framework on graphics processors. *Parallel Architectures and Compilation Techniques*, pages 260–269, 2008.

[28] Bryan Catanzaro, Narayanan Sundaram, and Kurt Keutzer. A Map Reduce Framework for Programming Graphics Processors. *Workshop on Software Tools for MultiCore Systems (STMCS)*, 2008.

[29] Chuntao Hong, Dehao Chen, Wenguang Chen, Weimin Zheng, and Haibo Lin. MapCG: Writing parallel program portable between CPU and GPU. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 217–226, New York, NY, USA, 2010. ACM.

[30] Feng Ji and Xiaosong Ma. Using shared memory to accelerate MapReduce on graphics processing units. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 805–816. IEEE, 2011.

[31] Linchuan Chen and Gagan Agrawal. Optimizing MapReduce for GPUs with effective shared memory usage. In *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '12, pages 199–210, New York, NY, USA, 2012. ACM.

[32] Linchuan Chen, Xin Huo, and Gagan Agrawal. Accelerating MapReduce on a coupled CPU-GPU architecture. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 25:1–25:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[33] Marwa Elteir, Heshan Lin, Wu-chun Feng, and Tom Scogland. StreamMR: an optimized MapReduce framework for AMD GPUs. In *Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on*, pages 364–371. IEEE, 2011.

[34] Jeff A. Stuart and John D. Owens. Multi-GPU MapReduce on GPU clusters. In *Proceedings of the 25th IEEE International Parallel and Distributed Processing Symposium*, IPDPS '11, May 2011.

[35] Wei Jiang and Gagan Agrawal. MATE-CG: A map reduce-like framework for accelerating data-intensive computations on heterogeneous clusters. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 644–655. IEEE, 2012.

[36] Wenbin Fang, Bingsheng He, Qiong Luo, and Naga K. Govindaraju. Mars: Accelerating MapReduce with graphics processors. *IEEE Transactions on Parallel and Distributed Systems*, 22:608–620, 2011.

[37] R. Farivar, A. Verma, E.M. Chan, and R.H. Campbell. MITHRA: Multiple data independent tasks on a heterogeneous resource architecture. In *Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on*, pages 1–10, Aug 2009.

[38] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D. Ernst. HaLoop: efficient iterative data processing on large clusters. *Proc. VLDB Endow.*, 3:285–296, September 2010.

[39] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. Twister: a runtime for iterative MapReduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 810–818, New York, NY, USA, 2010. ACM.

[40] Jure Leskovec, Deepayan Chakrabarti, Jon Kleinberg, Christos Faloutsos, and Zoubin Ghahramani. Kronecker graphs: An approach to modeling networks. *The Journal of Machine Learning Research*, 11:985–1042, March 2010.

[41] David A. Bader, Jonathan Berry, Simon Kahan, Richard Murphy, and E. Jason Riedy. The Graph 500 list. `http://www.graph500.org/`.

[42] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *Proceedings of the 7th International World-Wide Web Conference*, WWW '98, 1998.

[43] Alexandros V Gerbessiotis and Leslie G Valiant. Direct bulk-synchronous parallel algorithms. *Journal of parallel and distributed computing*, 22(2):251–267, 1994.

[44] Grzegorz Malewicz, Matthew H. Austern, Aart J.C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*, PODC '09, pages 6–6, New York, NY, USA, 2009. ACM.

[45] Ching Avery. Giraph: Large-scale graph processing infrastruction on Hadoop. *Proceedings of Hadoop Summit. Santa Clara, USA:[sn]*, 2011.

[46] Apache Hama. `http://hama.apache.org`.

[47] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M Hellerstein. Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1006.4990*, 2010.

[48] Miyuru Dayarathna, Charuwat Houngkaew, and Toyotaro Suzumura. Introducing ScaleGraph: an X10 library for billion scale graph analytics. In *Proceedings of the 2012 ACM SIGPLAN X10 Workshop*, page 6. ACM, 2012.

[49] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. GraphChi: Large-scale graph computation on just a PC. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI '12, pages 31–46, 2012.

[50] Jim Webber. A programmatic introduction to Neo4j. In *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*, pages 217–218. ACM, 2012.

[51] Donald Ervin Knuth, Donald Ervin Knuth, and Donald Ervin Knuth. *The Stanford GraphBase: a platform for combinatorial computing*, volume 37. Addison-Wesley Reading, 1993.

[52] RFVD Lans. InfiniteGraph: Extending business, social and government intelligence with graph analytics. *The analysis*, 2010.

[53] Jans Aasman. Allegro graph: RDF triple database. Technical report, Technical report. Franz Incorporated, 2006. ur l: http://www. franz. com/agraph/allegrograph/(visited on 10/14/2013)(cited on pp. 52, 54), 2006.

[54] Robey Pointer, N Kallen, E Ceaser, and J Kalucki. Introducing flockdb, 2010.

[55] Miyuru Dayarathna and Toyotaro Suzumura. XGDBench: A benchmarking platform for graph stores in exascale clouds. In *Proceedings of the 2012 IEEE 4th International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 363–370. IEEE Computer Society, 2012.

[56] Stephen F Altschul, Warren Gish, Webb Miller, Eugene W Myers, and David J Lipman. Basic local alignment search tool. *Journal of molecular biology*, 215(3):403–410, 1990.

[57] Stephen F Altschul, Thomas L Madden, Alejandro A Schäffer, Jinghui Zhang, Zheng Zhang, Webb Miller, and David J Lipman. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic acids research*, 25(17):3389–3402, 1997.

[58] T.F. Smith and M.S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195 – 197, 1981.

[59] David J Lipman and William R Pearson. Rapid and sensitive protein similarity searches. *Science*, 227(4693):1435–1441, 1985.

[60] W James Kent. BLAT—the BLAST-like alignment tool. *Genome research*, 12(4):656–664, 2002.

[61] Bin Ma, John Tromp, and Ming Li. PatternHunter: faster and more sensitive homology search. *Bioinformatics*, 18(3):440–445, 2002.

[62] Shuji Suzuki, Masanori Kakuta, Takashi Ishida, and Yutaka Akiyama. GHOSTX: An improved sequence homology search algorithm using a query suffix array and a database suffix array. *PloS one*, 9(8):e103833, 2014.

[63] Zhen Meng, Jianhui Li, Yunchun Zhou, Qi Liu, Yong Liu, and Wei Cao. bCloudBLAST: An efficient mapreduce program for bioinformatics applications. In *Biomedical Engineering and Informatics (BMEI), 2011 4th International Conference on*, volume 4, pages 2072–2076. IEEE, 2011.

[64] Lijuan Luo, Martin Wong, and Wen-mei Hwu. An effective GPU implementation of breadth-first search. In *Proceedings of the 47th Design Automation Conference*, DAC '10, pages 52–55, 2010.

[65] Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. Accelerating CUDA graph algorithms at maximum warp. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 267–276, 2011.

[66] Stefan Edelkamp and Damian Sulewski. External memory breadth-first search with delayed duplicate detection on the GPU. In *Model Checking and Artificial Intelligence*, volume 6572 of *Lecture Notes in Computer Science*, pages 12–31. Springer Berlin Heidelberg, 2011.

[67] J. Zhong and B. He. Medusa: Simplified graph processing on GPUs. *IEEE Transactions on Parallel and Distributed Systems*, 25(6):1543–1552, June 2014.

[68] Erich Elsen. Speeding up GraphLab using CUDA. In *GPU Technology Conference*, 2014.

[69] Yangzihao Wang. High-performance graph primitives on the GPU: Design and implementation of Gunrock. In *GPU Technology Conference*, 2014.

[70] Duane Merrill, Michael Garland, and Andrew Grimshaw. Scalable GPU graph traversal. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 117–128, 2012.

[71] Sungpack Hong, T. Oguntebi, and K. Olukotun. Efficient parallel graph exploration on multi-core CPU and GPU. In *Proceedings of the 20th International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, pages 78–88, Oct 2011.

[72] Abdullah Gharaibeh, Lauro Beltrao Costa, Elizeu Santos-Neto, and Matei Ripeanu. On graphs, GPUs, and blind dating: A workload to processor matchmaking quest. In *Proceedings of the IEEE 27th International Symposium on Parallel and Distributed Processing*, IPDPS '13, pages 851–862, 2013.

[73] Nhat Tan Duong, Quang Anh Pham Nguyen, Anh Tu Nguyen, and Huu-Duc Nguyen. Parallel PageRank computation using GPUs. In *Proceedings of the Third Symposium on Information and Communication Technology*, SoICT '12, pages 223–230, 2012.

[74] Xintian Yang, Srinivasan Parthasarathy, and P. Sadayappan. Fast sparse matrix-vector multiplication on GPUs: Implications for graph mining. *Proceedings of the VLDB Endowment*, 4(4):231–242, January 2011.

[75] A. Rungsawang and B. Manaskasemsak. Fast PageRank computation on a GPU cluster. In *Proceedings of the 20th Euromicro International Conference onParallel, Distributed and Network-Based Processing*, PDP '12, pages 450–456, Feb 2012.

[76] Koji Ueno and Toyotaro Suzumura. Parallel distributed breadth first search on GPU. In *Proceedings of the 20th IEEE conference on High Performance Computing*, HiPC '13, pages 314–323, Dec 2013.

[77] J. Chhugani, N. Satish, Changkyu Kim, J. Sewall, and P. Dubey. Fast and efficient graph traversal algorithm for CPUs: Maximizing single-node efficiency. In *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 378 –389, may 2012.

[78] Aydin Buluç and Kamesh Madduri. Parallel breadth-first search on distributed memory systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 65:1–65:12, New York, NY, USA, 2011. ACM.

[79] Keita Iwabuchi, Hitoshi Sato, Ryo Mizote, Yasui Yuichiro, Katsuki Fujisawa, and Satoshi Matsuoka. Hybrid BFS approach using semi-external memory. In *IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW)*, May 2014.

[80] Karthik Nilakant, Valentin Dalibard, Amitabha Roy, and Eiko Yoneki. PrefEdge: SSD Prefetcher for Large-Scale Graph Traversal. In *7th ACM International Systems and Storage Conference*, 2014.

[81] R. Pearce, M. Gokhale, and N.M. Amato. Scaling techniques for massive scale-free graphs in distributed (external) memory. In *Proceedings of the IEEE 27th International Symposium on Parallel Distributed Processing*, IPDPS '13, pages 825–836, May 2013.

[82] Guanghao Jin, T. Endo, and S. Matsuoka. A parallel optimization method for stencil computation on the domain that is bigger than memory capacity of GPUs. In *Proceedings of the IEEE International Conference on Cluster 2013*, CLUSTER '13, pages 1–8, Sept 2013.

[83] Rui Wang, Yuchi Huo, Yazhen Yuan, Kun Zhou, Wei Hua, and Hujun Bao. GPU-based out-of-core many-lights rendering. *ACM Transactions on Graphics*, 32(6):210:1–210:10, November 2013.

[84] Yin Ye, Zhihui Du, and David A. Bader. GPUMemSort: A high performance graphic co-processors sorting algorithm for large scale in-memory data. *GSTF International Journal on Computing*, pages 1(2):23–28, 2011.

[85] H. Peters, O. Schulz-Hildebrandt, and N. Luttenberger. Parallel external sorting for CUDA-enabled GPUs with load balancing and low transfer overhead. In *IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1–8, April 2010.

[86] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. GPUfs: Integrating a file system with GPUs. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 485–498, 2013.

[87] Raja Appuswamy, Christos Gkantsidis, Dushyanth Narayanan, Orion Hodson, and Antony Rowstron. Scale-up vs scale-out for Hadoop: Time to rethink? In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 20:1–20:13, New York, NY, USA, 2013. ACM.

[88] M. Michael, J.E. Moreira, D. Shiloach, and R.W. Wisniewski. Scale-up x scale-out: A case study using Nutch/Lucene. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8, March 2007.

[89] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha. GPUTeraSort: high performance graphics coprocessor sorting for large database management. *SIGMOD*, 2006.

[90] R. L. Graham. Bounds on multiprocessing anomalies and related packing algorithms. In *Proceedings of the May 16-18, 1972, spring joint computer conference*, AFIPS '72 (Spring), pages 205–217, New York, NY, USA, 1972. ACM.

[91] J. Bruno, E. G. Coffman, Jr., and R. Sethi. Scheduling independent tasks to reduce mean finishing time. *Commun. ACM*, 17(7):382–387, July 1974.

[92] Chandra Chekuri, Sanjeev Khanna, and An Zhu. Algorithms for minimizing weighted flow time. In *Proceedings of the thirty-third annual ACM symposium on Theory of computing*, STOC '01, pages 84–93, New York, NY, USA, 2001. ACM.

[93] Jure Leskovec, Deepayan Chakrabarti, Jon Kleinberg, Christos Faloutsos, and Zoubin Ghahramani. Kronecker graphs: An approach to modeling networks. *J. Mach. Learn. Res.*, 11:985–1042, March 2010.

[94] Satoshi Matsuoka, Toshio Endo, Naoya Maruyama, Hitoshi Sato, and Shin'ichiro Takizawa. The total picture of TSUBAME2.0. *Tsubame e-Science Journal*, 1:2 – 4, 2010.

[95] Nathan Bell and Jared Hoberock. Thrust: A productivity-oriented library for CUDA. *GPU Computing Gems*, 7, 2011.

[96] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[97] Sangwon Seo, Edward J Yoon, Jaehong Kim, Seongwook Jin, Jin-Soo Kim, and Seungryoul Maeng. Hama: An efficient matrix computation with the MapReduce framework. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 721–726. IEEE, 2010.

[98] Matthew Felice Pace. BSP vs MapReduce. *Procedia Computer Science*, 9:246–255, 2012.

[99] Tomasz Kajdanowicz, Przemyslaw Kazienko, and Wojciech Indyk. Parallel processing of large graphs. *Future Generation Computer Systems*, 32:324–337, 2014.

[100] David Alexander Hutchinson. *Parallel algorithms in external memory.* PhD thesis, Carleton University, 2000.

[101] Moritz Kreutzer, Georg Hager, Gerhard Wellein, Holger Fehske, Achim Basermann, and Alan R Bishop. Sparse matrix-vector multiplication on GPGPU clusters: A new storage format and a scalable implementation. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 1696–1702. IEEE, 2012.

[102] Gary Bradski, Cheng-Tao Chu, Andrew Ng, Kunle Olukotun, Sang Kyun Kim, Yi-An Lin, and YuanYuan Yu. [map-reduce.

[103] Silvio Lattanzi, Benjamin Moseley, Siddharth Suri, and Sergei Vassilvitskii. Filtering: a method for solving graph problems in MapReduce. In *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*, pages 85–94. ACM, 2011.

[104] Jimmy Lin and Michael Schatz. Design patterns for efficient graph algorithms in MapReduce. In *Proceedings of the Eighth Workshop on Mining and Learning with Graphs*, MLG '10, pages 78–85, New York, NY, USA, 2010. ACM.

[105] Koichi Shirahata, Hitoshi Sato, and Satoshi Matsuoka. Hybrid map task scheduling for GPU-based heterogeneous clusters. In *The 1st International Workshop on Theory and Practice of MapReduce (MAPRED'2010)*, 2010.

[106] Keiji Yamamoto, Atsuya Uno, Hitoshi Murai, Toshiyuki Tsukamoto, Fumiyoshi Shoji, Shuji Matsui, Ryuichi Sekizawa, Fumichika Sueyasu, Hiroshi Uchiyama, Mitsuo Okamoto, et al. The K computer operations: Experiences and statistics. *Procedia Computer Science*, 29:576–585, 2014.

[107] Ronald C Taylor. An overview of the Hadoop/MapReduce/HBase framework and its current applications in bioinformatics. *BMC bioinformatics*, 11(Suppl 12):S1, 2010.

[108] Massimo Gaggero, Simone Leo, Simone Manca, Federico Santoni, Omar Schiaratura, Gianluigi Zanetti, Edificio CRS, and Sardegna Ricerche. Parallelizing bioinformatics applications with MapReduce. *Cloud Computing and Its Applications*, pages 22–23, 2008.

[109] Andréa Matsunaga, Maurício Tsugawa, and José Fortes. CloudBLAST: Combining MapReduce and virtualization on distributed resources for bioinformatics applications. In *eScience, 2008. eScience'08. IEEE Fourth International Conference on*, pages 222–229. IEEE, 2008.

[110] Xiao-liang Yang, Yu-long Liu, Chun-feng Yuan, and Yi-hua Huang. Parallelization of BLAST with MapReduce for long sequence alignment. In *Parallel Architectures, Algorithms and Programming (PAAP), 2011 Fourth International Symposium on*, pages 241–246. IEEE, 2011.

[111] Freddie Sunarso, Srikumar Venugopal, and Federico Lauro. Scalable protein sequence similarity search using locality-sensitive hashing and MapReduce. *arXiv preprint arXiv:1310.0883*, 2013.

[112] Simone Leo, Federico Santoni, and Gianluigi Zanetti. Biodoop: bioinformatics on Hadoop. In *Parallel Processing Workshops, 2009. ICPPW'09. International Conference on*, pages 415–422. IEEE, 2009.

[113] Mingming Sun, Xuehai Zhou, Feng Yang, Kun Lu, and Dong Dai. Bwasw-Cloud: Efficient sequence alignment algorithm for two big data with MapReduce. In *Applications of Digital Information and Web Technologies (ICADIWT), 2014 Fifth International Conference on the*, pages 213–218. IEEE, 2014.

[114] Motohiko Matsuda, Naoya Maruyama, and Shin'ichiro Takizawa. K MapReduce: A scalable tool for data-processing and search/ensemble applications on large-scale supercomputers. In *Cluster Computing (CLUSTER), 2013 IEEE International Conference on*, pages 1–8. IEEE, 2013.

[115] Aaron Darling, Lucas Carey, and Wu-chun Feng. The design, implementation, and evaluation of mpiBLAST. *Proceedings of ClusterWorld*, 2003, 2003.

[116] Peter J Turnbaugh, Ruth E Ley, Micah Hamady, Claire Fraser-Liggett, Rob Knight, and Jeffrey I Gordon. The human microbiome project: exploring the microbial part of ourselves in a changing world. *Nature*, 449(7164):804, 2007.

[117] David L Wheeler, Tanya Barrett, Dennis A Benson, Stephen H Bryant, Kathi Canese, Vyacheslav Chetvernin, Deanna M Church, Michael DiCuccio, Ron Edgar, Scott Federhen, et al. Database resources of the national center for biotechnology information. *Nucleic acids research*, 35(suppl 1):D5–D12, 2007.