# A Scalable Implementation of a MapReduce-based Graph Processing Algorithm for Large-scale Heterogeneous Supercomputers

Koichi Shirahata*, Hitoshi Sato*‡, Toyotaro Suzumura*†‡ and Satoshi Matsuoka*
*Department of Mathematical and Computing Sciences, Tokyo Institute of Technology, Tokyo, Japan
†CREST, Japan Science and Technology Agency, Tokyo, Japan
‡IBM Research - Tokyo, Japan

***Abstract*—Fast processing for extremely large-scale graph is becoming increasingly important in various domains such as health care, social networks, intelligence, system biology, and electric power grids. The GIM-V algorithm based on MapReduce programing model is designed as a general graph processing method for supporting petabyte-scale graph data. On the other hand, recent large-scale data-intensive computing systems tend to employ GPU accelerators to gain good peak performance and high memory bandwidth; however, the validity of acceleration, including optimization techniques, of the GIM-V algorithm using GPUs is an open problem. To address the problem, we implemented a multi-GPU-based GIM-V application with load balance optimization between GPU devices. Our implementation extends the existing MapReduce library for supporting multi-GPU-environments using the MPI library and optimizes load balance between GPU devices by employing task scheduling-based graph partitioning. We conducted 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 $2^{30}$ (1.07 billion) vertices and $2^{34}$ (17.2 billion) edges, and 1.52 times faster than the CPU-based naive implementation with $2^{29}$ vertices and $2^{33}$ edges. We also studied the performance characteristics of our implementation and load balance optimization technique.**

***Keywords*-Large-scale Graph Processing; GPGPU; MapReduce;**

## I. INTRODUCTION

Recent emergence of extremely large-scale graphs 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 vertices and 100's millions to 100's trillions of edges, requires fast and scalable analysis by using HPC technologies. For example, a friend network in an existing social network service [1] is expressed as a graph with over 900 million vertices and over 100 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 [2] in the HPC community, which ranks supercomputers by executing a large-scale graph search problem as an instance of data-intensive supercomputing applications.

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 [4], which is an Hadoop [5]-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. [4] 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. As a motivating example, Figure 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.

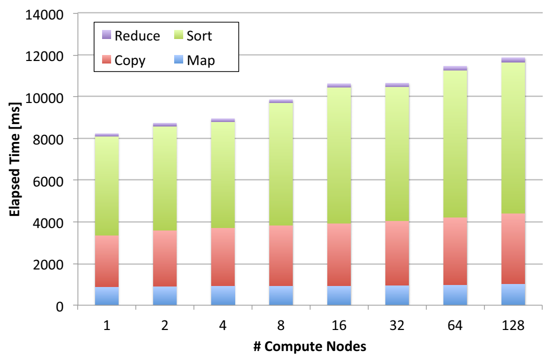

Fig. 1. Performance of CPU-based GIM-V (Scale 21 per node)

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 [6]. 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. In our earlier work, we have proposed a single naive GPU implementation of GIM-V, and we found GPU accelerates the map stage 2.7 times faster than CPU implementation [7]. However, how much multi-GPU implementation accelerates for large-scale graphs is a problem worth investigating.

To address these problems, we implemented a multi-GPU-based GIM-V application with load balance optimization between GPU devices. Our implementation extends the existing MapReduce library for supporting multi-GPU-environments using the MPI library and optimizes load balance between GPU devices by employing task scheduling-based graph partitioning. 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 our 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.

## II. BACKGROUND

In this section, we give overviews of large-scale graphs, a graph processing algorithm GIM-V, and an existing GPUMapReduce implementation called Mars.

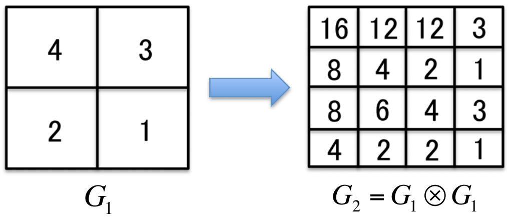

Fig. 2. Kronecker Graph

### A. Large-scale Graphs 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-low degree distributions), small-world (6 degree of separation), and clustering, etc.

Kronecker Graph [8] 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. Figure 2 shows a generation process 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), in order to represent large-scale graphs in real world.

### B. GIM-V

GIM-V (Generalized Iterative Matrix-Vector multiplication) [4] is a general expression of matrix-vector multiplication with iterative operations. 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:

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

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

As an example, here we describe the PageRank algorithm [9], which is a well-known algorithm for scoring relative importance in web pages, by using the GIM-V algorithm. 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 which is 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. In practice, we first construct a matrix M by a column-normalized adjacency matrix $E^T$ such that every column of M sums to 1. Then the next PageRank eigenvector is calculated by $p^{next} = M \times p^{cur}$, where the three operations are defined as follows:

$$\begin{aligned} &\text{combine2}(m_{i,j}, v_j) = c \times m_{i,j} \times v_j \\ &\text{combineAll}_i(x_1, \ldots, x_n) = \frac{(1-c)}{n} + \sum_{j=1}^{n} x_j \\ &\text{assign}(v_i, v_{new}) = v_{new} \end{aligned}$$

The GIM-V algorithm consists of two MapReduce stages: MapReduce Stage1 and Stage2. The MapReduce 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 is the source vertex id $i$ and the value is the partially combined result $x_j = \textit{combine2}(m_{ij}, v_j)$ . Then the output of the MapReduce Stage1 is forwarded to the input of the MapReduce Stage2. The MapReduce Stage2 combines all partial results from the MapReduce Stage1 by applying $\textit{combineAll}_i(x_j \mid j = 1 \ldots n)$, and assigns the new vector $v_{new}$ to the old vector $v_i$ by applying $\textit{assign}(v_i, \textit{combineAll}_i(x_j \mid j = 1 \ldots n))$. These two MapReduce operations are iterated until the application-specific convergence criterion is met.

### *C. Mars*

Mars [10], [11] 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 multi-GPU-based GIM-V implementation described in Section III.

Figure 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 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 [12], 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 $<\text{key offset}, \text{key size}, \text{value offset}, \text{value size}>$. 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.

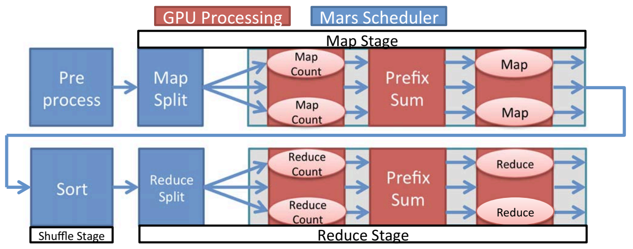

Fig. 3. Mars architecture

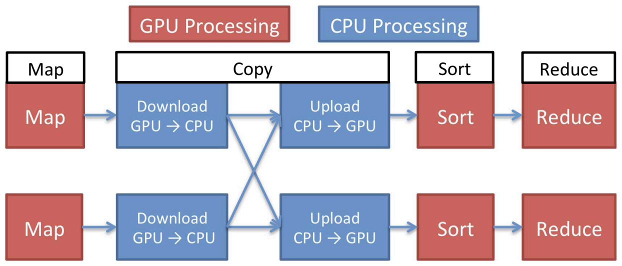

Fig. 4. Implementation overview of MapReduce on multiple nodes

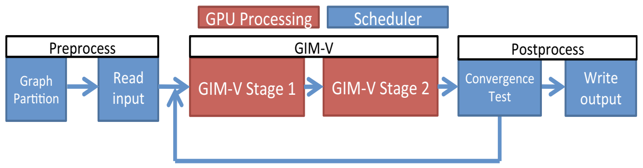

Fig. 5. GIM-V workflow

## III. A MULTI GPU IMPLEMENTATION WITH LOAD BALANCE OPTIMIZATION

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.

### A. *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 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 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.

### B. *GIM-V using Multi GPU MapReduce*

Based on the extended Mars library for multi-GPU environments, we implement the GIM-V algorithm described in Section II-B. Figure 5 shows the workflow of our GIM-V implementation, in which we connect multiple MapReduce stages as follows:

**STEP1** Preprocessing
**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 II-B. 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

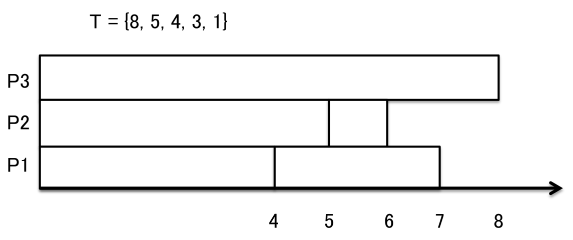

Fig. 6. An Example of LPT Schedule

Mars implementation and use fixed size payload (actual data) to reduce the amount of data transfer. Second, we change the thread allocation in 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.

### C. Load Balance Optimization

We use a task scheduling-based graph partitioning, called Longest Processing Time (LPT) schedule [13], 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 [14], [15], [13] since obtaining optimal solution for the problem is considered to be NP-complete [14]. 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. [13] 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 6. We assume three processors, P1, P2, and P3, and five tasks, whose sizes are 8, 5, 4, 3, and 1. The LPT assigns tasks to each processor from task1(8) to task5(1). By applying LPT to the GIM-V algorithm, we obtain a near optimal partitioning result.

## IV. Experiments

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.

### A. 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-low 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 [16], [17]. On top of the GIM-V algorithm, we implemented PageRank as an application for the experiments.

### B. Experimental Environment

We use 256 compute nodes of the TSUBAME2.0 supercomputer [18] 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.

### C. 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 7 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 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 7 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 8 shows the performance breakdown on SCALE 28 at 128 nodes in Figure 7. 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 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.

## *D. Performance Comparison with Naive GPU Implementation*

We compare the performance differences between naive GPU implementation and our optimized implementation. Figure 9 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 III-B; 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.

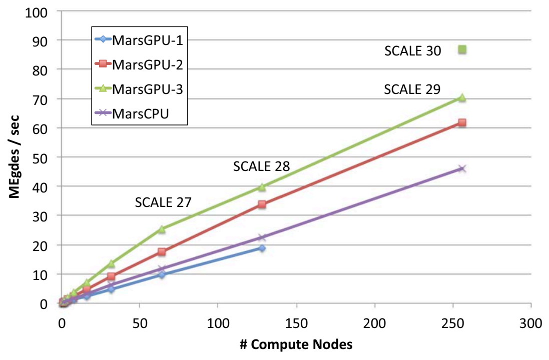

Fig. 7. Weak scaling performance on MarsGPU and MarsCPU (Scale 21 per node)

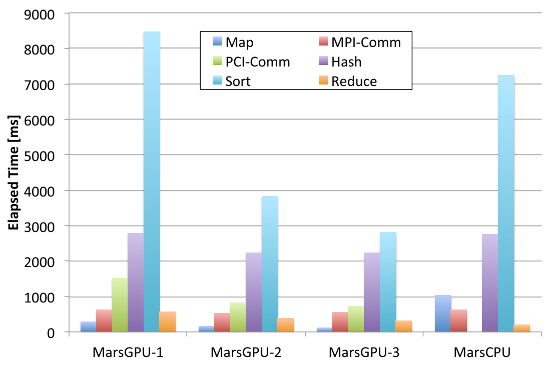

Fig. 8. Performance breakdown on MarsGPU and MarsCPU (Scale 28,128 nodes)

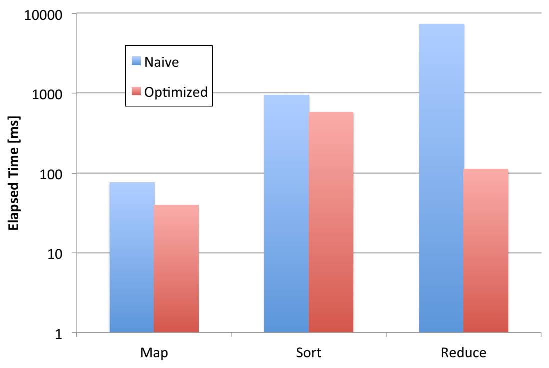

Fig. 9. Performance comparison with naive GPU implementation on MarsGPU-3 (Scale 26,128 nodes)

## *E. 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 III-C, 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 10 shows the weak-scaling results of the simulation. Each node has the constant problem size, SCALE 19. The $x$ axis denotes the

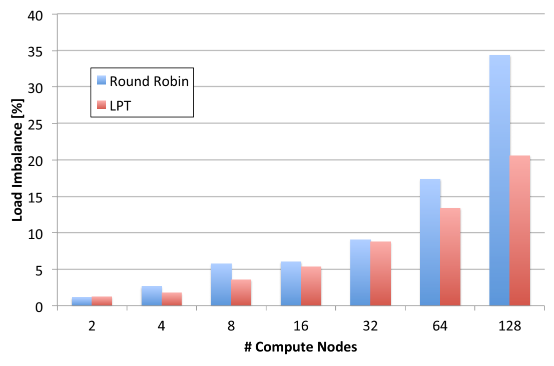

Fig. 10. Load balance: Round robin vs. LPT (Scale 19 per node)

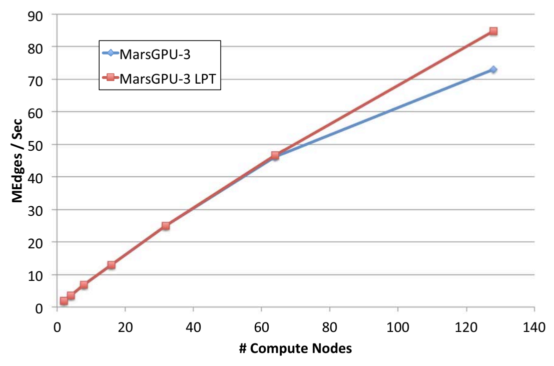

Fig. 11. Weak scaling performance on MarsGPU-3: Round robin vs. LPT (Scale 21 per node)

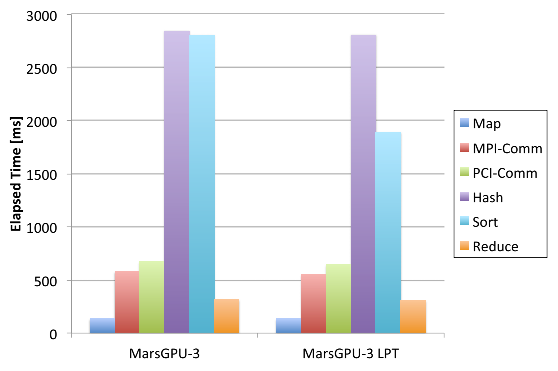

Fig. 12. Performance breakdown on MarsGPU-3: Round robin vs. LPT (Scale 28, 128 nodes)

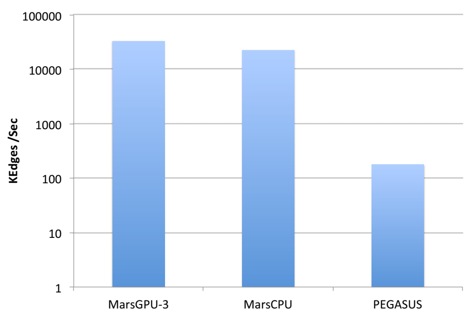

Fig. 13. Performance Comparison with PEGASUS (SCALE 27, 128 nodes)

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 11 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 except for 128 nodes. This is because the input graphs are relatively well-balanced as shown in Figure 10. However, In the case of 128 nodes, we see some performance improvement. Here Figure 12 shows the performance breakdown on 128 nodes from Figure 11. 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 12.

### F. 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 13 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.

## V. RELATED WORK

Harish et al. [19] 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. We implement the MapReduce-based graph processing implementation which runs on GPUs, and even with billions of edges, our GPU-based implementation shows overall better performance than the CPU-based one.

There is a lot of research related to MapReduce on multi cores [21] or multi GPUs [20], [22]. They propose how recent processors 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. We show the good scalability of MapReduce-based graph processing using hundreds of GPUs on the large-scale heterogeneous supercomputer.

Chhugani et al. [24] propose a work distribution approach for multi-socket platforms, whose approach ensures load-balancing while keeping cross-socket communication low on R-MAT graphs. Aydin et al. [25] 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. We propose a load balancing technique based on task scheduling on top of a multi-GPU MapReduce system and show this technique improves the performance.

## VI. CONCLUSIONS

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. For future work, we are planning to include data management of memory hierarchy using FLASH devices for larger scale graphs than the sum of memory each GPU owns.

## ACKNOWLEDGEMENT

The part of this work was supported by JST, CREST, and JSPS KAKENHI Grant Number 23700031.

## REFERENCES

[1] "Facebook." [Online]. Available: http://www.facebook.com

[2] J. A. Ang, B. W. Barrett, K. B. Wheeler, and R. C. Murphy, "Introducing the graph 500," May 2010.

[3] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *OSDI '04, Sixth Symposium on Operating System Design and Implementation*, pp. 137–150, 2004.

[4] U. Kang, C. E. Tsourakakis, and C. Faloutsos, "PEGASUS: A Peta-Scale Graph Mining System- Implementation and Observations," 2009.

[5] A. Bialecki, M. Cordova, D. Cutting, and O. O'Malley, "Hadoop: a framework for running applications on large clusters built of commodity hardware," 2005. [Online]. Available: http://lucene.apache.org/hadoop

[6] "Amazon elastic compute cloud (amazon ec2)." [Online]. Available: http://aws.amazon.com/ec2/

[7] K. Shirahata, H. Sato, T. Suzumura, and S. Matsuoka, "A GPU Implementation of Generalized Graph Processing Algorithm GIM-V," in *2012 International Workshop on Parallel Algorithm and Parallel Software (IWPAPS12)*, September 2012.

[8] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, and Z. Ghahramani, "Kronecker graphs: An approach to modeling networks," *J. Mach. Learn. Res.*, vol. 11, pp. 985–1042, Mar. 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1756006.1756039

[9] S. Brin and L. Page, "The Anatomy of a Large-Scale Hypertextual Web Search Engine," in *Seventh International World-Wide Web Conference (WWW 1998)*, 1998. [Online]. Available: http://ilpubs.stanford.edu:8090/361/

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

[11] W. Fang, B. He, Q. Luo, and N. K. Govindaraju, "Mars: Accelerating MapReduce with Graphics Processors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, pp. 608–620, 2011.

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

[13] R. L. Graham, "Bounds on multiprocessing anomalies and related packing algorithms," in *Proceedings of the May 16-18, 1972, spring joint computer conference*, ser. AFIPS '72 (Spring). New York, NY, USA: ACM, 1972, pp. 205–217. [Online]. Available: http://doi.acm.org/10.1145/1478873.1478901

[14] J. Bruno, E. G. Coffman, Jr., and R. Sethi, "Scheduling independent tasks to reduce mean finishing time," *Commun. ACM*, vol. 17, no. 7, pp. 382–387, Jul. 1974. [Online]. Available: http://doi.acm.org/10.1145/361011.361064

[15] C. Chekuri, S. Khanna, and A. Zhu, "Algorithms for minimizing weighted flow time," in *Proceedings of the thirty-third annual ACM symposium on Theory of computing*, ser. STOC '01. New York, NY, USA: ACM, 2001, pp. 84–93. [Online]. Available: http://doi.acm.org/10.1145/380752.380778

[16] D. A. Bader, J. Berry, S. Kahan, R. Murphy, and E. J. Riedy. The graph 500 list. Graph500.org. [Online]. Available: http://www.graph500.org/

[17] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, and Z. Ghahramani, "Kronecker graphs: An approach to modeling networks," *J. Mach. Learn. Res.*, vol. 11, pp. 985–1042, March 2010. [Online]. Available: http://portal.acm.org/citation.cfm?id=1756006.1756039

[18] S. Matsuoka, T. Endo, N. Maruyama, H. Sato, and S. Takizawa, "The total picture of tsubame2.0," *Tsubame e-Science Journal*, vol. 1, pp. 2 – 4, 2010.

[19] P. 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*, ser. HiPC'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 197–208. [Online]. Available: http://portal.acm.org/citation.cfm?id=1782174.1782200

[20] J. A. Stuart and J. D. Owens, "Multi-GPU MapReduce on GPU Clusters," in *Proceedings of the 25th IEEE International Parallel and Distributed Processing Symposium*. IEEE, May 2011.

[21] S. J. Plimpton and K. D. Devine, "MapReduce in MPI for Large-scale Graph Algorithms," *Parallel Computing*, vol. In Press,, no. February, pp. 1–39, 2011. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0167819111000172

[22] B. Catanzaro, N. Sundaram, and K. Keutzer, "A map reduce framework for programming graphics processors," in *In Workshop on Software Tools for MultiCore Systems*, 2008.

[23] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 28th ACM symposium on Principles of distributed computing*, ser. PODC '09. New York, NY, USA: ACM, 2009, pp. 6–6.

[24] J. Chhugani, N. Satish, C. 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*, may 2012, pp. 378 –389.

[25] A. Buluç and K. Madduri, "Parallel breadth-first search on distributed memory systems," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. New York, NY, USA: ACM, 2011, pp. 65:1–65:12. [Online]. Available: http://doi.acm.org/10.1145/2063384.2063471