Performance Characteristics of Graph500 on Large-Scale Distributed Environment

Toyotaro Suzumura[1,2,4], Koji Ueno[1,4], Hitoshi Sato[1,4]
Katsuki Fujisawa[3,4] and Satoshi Matsuoka[1.4]
[1] Tokyo Institute of Technology, [2] IBM Research – Tokyo, [3] Chuo University, [4]JST CREST

Graph500 is a new benchmark for supercomputers based on large-scale graph analysis, which is becoming an important form of analysis in many real-world applications. Graph algorithms run well on supercomputers with shared memory. For the Linpack-based supercomputer rankings, TOP500 reports that heterogeneous and distributed-memory super-computers with large numbers of GPGPUs are becoming dominant. However, the performance characteristics of large-scale graph analysis benchmarks such as Graph500 on distributed-memory supercomputers have so far received little study. This is the first report of a performance evaluation and analysis for Graph500 on a commodity-processor-based distributed-memory supercomputer. We found that the reference implementation "*replicated-csr*" based on distributed level-synchronized breadth-first search solves a large free graph problem with 2^{31} vertices and 2^{35} edges (approximately 2.15 billon vertices and 34.3 billion edges) in 3.09 seconds with 128 nodes and 3,072 cores. This equates to 11 giga-edges traversed per second. We describe the algorithms and imple-mentations of the reference implementations of Graph500, and analyze the performance characteristics with varying graph sizes and numbers of computer nodes and different implemen-tations. Our results will also contribute to the development of optimized algorithms for the coming exascale machines.

I. INTRODUCTION

Large-scale graph analysis is a hot topic in various analyses, such as for social networks, micro-blogs, protein-protein interactions, and the connectivity of the Web. The numbers of vertices in the analyzed graph networks have grown from billions to tens of billions and the edges have grown from tens of billions to hundreds of billions. Since 1994, the best known de facto ranking of the world's fastest computers is TOP500, which is based on a high performance Linpack for linear equations. As new alternative metric to Linpack, Graph500 was recently developed. Since Graph500 is a new benchmark, there have been few studies of the benchmark's characteristics. This paper gives an overview of the Graph500 benchmark and some details about the reference implementations currently available from the Graph500 website. Then we study the benchmark's theoretical perfor-mance and its actual performance on an experimental supercomputer. Currently, special-purpose custom-hardware supercomputers such as IBM's BlueGene/P and the Cray XMT dominate the top-ranked machines. However, we are focusing on a design that uses commodity hardware, the TSUBAME 2.0. Here are the main contributions of our paper:

1. Detailed analysis of Graph500 reference implementa-tions.
2. Performance analysis on the currently 5th-ranked TSUBAME 2.0 supercomputer.
3. Analysis to optimize the performance on other super-computers that use commodity processors.

Here is the organization of our paper. In Section 2, we give an overview of Graph500 by explaining the benchmark rules and the methods for generating, analyzing, and vali-dating the graphs. Section 3 is a detailed explanation of four reference implementations of distributed breadth-first search algorithms with theoretic analyses of the communication patterns and data. In Section 4, we describe our performance evaluation and give detailed profiles of three reference implementations on the TSUBAME 2.0 supercomputer while varying the number of nodes and the problem size. We review related work in Section 5 and conclude and consider future work in Section 6.

II. GRAPH500 OVERVIEW

In this section, we first summarize the Graph500 bench-mark [1] and then analyze its memory consumption.

A. *Graph500 Benchmark*

In contrast to the computation-intensive benchmark used by TOP500, Graph500 is a data-intensive benchmark. It does breadth-first searches in undirected large graphs generated by a scalable data generator based on a Kronecker graph [16]. The benchmark has two kernels: Kernel 1 constructs an undirected graph from the graph generator in a format usable by Kernel 2. The first kernel transforms the edge list (pairs of start and end vertices) to efficient data structures with sparse formats, such as CSR (Compressed Sparse Row) or CSC (Compressed Sparse Column). Then Kernel 2 does a breadth-first search of the graph from a randomly chosen source vertex in the graph. The benchmark uses the elapsed times for both kernels, but the rankings for Graph500 are determined by how large the problem is and by the through-put in numbers of edges traversed per second, TEPS (Traversed Edges Per Second). This means that the ranking

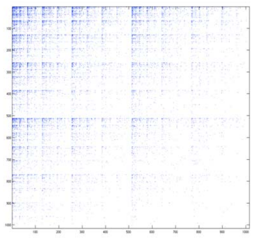

Figure 1. Visualizing the adjacent matrix of the generated Kronecker graph

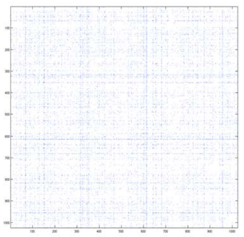

Figure 2. Visualizing the adjacent matrix of shuffled graph

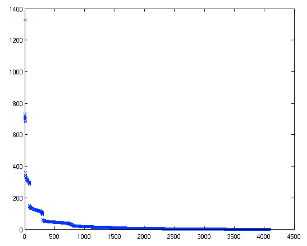

Figure 3 Degree Distribution of the Kronecker Graph

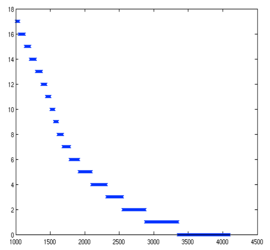

Figure 4. Distribution for degrees less than 18

results basically depend on the time for the second kernel. After both kernels have finished, there is a validation phase to check whether or not the result is correct. It becomes difficult to show that the resulting breadth first tree matches the reference result when the data set is large enough. Therefore the validation phase uses 5 validation rules. For example, the first rule is that the BFS graph is a tree and does not contain any cycles. There are five problem classes: toy, mini, small, medium, large, and huge. Each problem solves a different size graph, defined by the Scale parameter, which is the base 2 logarithm of the number of vertices. For example, the level Scale 26 for *toy* means 2^{26} and corresponds to 10^{10} bytes and occupies 17 GB of memory. The five Scale values are 26, 29, 32, 36, 39, and 42 for the five classes. The largest problem, *huge* (Scale 42), needs to handle around 1.1 PB of memory. As of this writing, Scale 38 is the largest that has been solved by the top-ranked supercomputer.

B. Graph generation and its memory consumption

The graph generator is a Kronecker generator that implements a scale-free graph generation algorithm. We visualized a generated graph of 1,024 vertices in an adjacent format by using MATLAB. By turning off the shuffling phase of the graph generator, the adjacency matrix shown in Figure 1 indicates that the graph itself has the self-similarity expected by the Kronecker graph model. After turning on the shuffling phase, all the vertices are shuffled as shown in Figure 2. Even though all of the vertices are shuffled, the high skew produced by the Kronecker model is maintained. Figure 3 and Figure 4 show that the generated graph of 4,096 vertices is also scale free. The x-axis is the vertex ID and the y-axis is the degree (# of vertices). As shown in the graph, most of the vertices have low degrees and a small number of vertices have high degrees. The highest degree is more than 1,200. Figure 4 is the key part of Figure 3 that shows only the vertices with less than 18 degrees. This graph clearly shows that most of the vertices have the relatively small degrees expected by the Kronecker graph model.

We also estimated the memory consumption as the size increases. Since the Graph500 organization provides different reference implementations with different formats, the amount of data differs. However, with the sparse matrix format called CSR (Compressed Sparse Row), the memory consumption is E*2 + V (E = # of edges, V = # of vertices). The memory consumption function M(Scale) with the Scale parameter can be defined as M(Scale) = (2^Scale * (2*edgefactor + 1)). The E here is calculated as the product of V by the edgefactor, which is 16 in this benchmark. For example, M(32) = 2^32 * (2*16+1) * 8 (bytes) = 1.03125 TB.

Algorithm I: Level-synchronized BFS

```
for all vertex v in parallel do
| pred[v] ← -1;

pred[r] ← 0
Enqueue(CQ r)
While CQ != Empty do
|  NQ ← empty
|  for all u in NQ in parallel do
|  |  u ← Dequeue(CQ)
|  |  for each v adjacent to u in parallel do
|  |  | if pred[v] = -1 then
|  |  |  |  pred[v] ← u;
|  |  |  |  Enqueue(NQ, v)
|  swap(CQ, NQ);
```

III. Parallel Level Synchronized BFS

All of the MPI reference implementation algorithms use a "level-synchronized breadth-first search", which means that all of the vertices at a given level of the BFS tree will be processed (potentially in parallel) before any vertices from a lower level in the tree are processed. The details of the level-synchronized BFS are explained in [2][3].

Algorithm I is the abstract pseudocode for the algorithm that implements level-synchronized BFS. At any time, CQ (Current Queue) is the set of vertices that must be visited at the current level. At level 1, CQ will contain the neighbors of r, so at level 2, it will contain their neighbors (the neighboring vertices that have not been visited at levels 0 or 1). The algorithm maintains NQ (Next Queue), containing the vertices that should be visited at the next level. After visiting all of the nodes at each level, the queues CQ and NQ are swapped.

The Graph500 benchmark organization provides four implementations, *simple*, *replicated-csr*, *replicated-csc*, and *one_sided*. All of the reference implementations are based on the distributed level-synchronized breadth-first search method, but the data structures and implementation methods

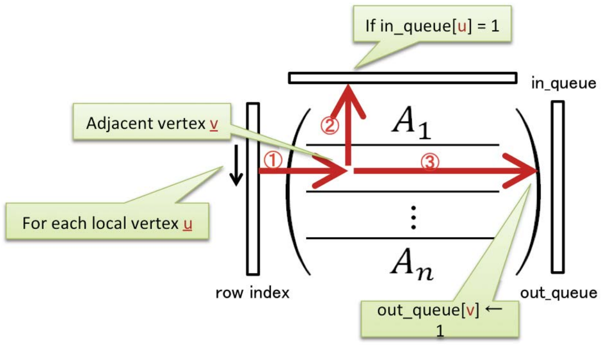

Figure 5. The *replicated-csr* implementation

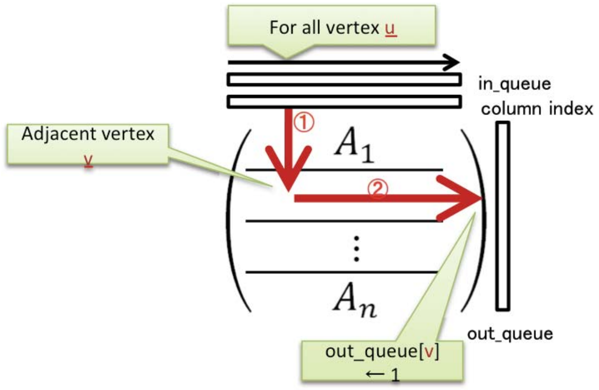

Figure 6. The *replicated-csc* implementation

differ. In the following sections, we provide a detailed explanation of each implementation except for the *one_sided* implementation and also a theoretic analysis of the memory consumption and communications characteristics in a distributed environment.

A. The simple implementation

The *simple* version is built on the parallel level-synchronized BFS. The pseudocode for the algorithm appears as Algorithm II. Initially, each MPI process has two queues, *CQ* and *NQ*, and two arrays, *pred* for a predecessor array and *visited* to track whether or not each vertex has been visited. In Line 10, it checks whether or not CQ is empty. If not, each MPI process receives one vertex and its predecessor from the other MPI processes via the asynchronous *MPI_Irecv* function in Line 11. The implementation is optimized by holding the information on whether or not each vertex has been visited in a one-dimensional bitmap array named *visited*. If a certain MPI process (A) needs to visit a target vertex that exists in another MPI process (B), then the A process needs to send a pair of a vertex ID and its parent vertex ID to the MPI process B with the asynchronous *MPI_Isend* function in Line 25. Since these transfers are too fine-grained if the process individually transfers each pair, the implementation is optimized by buffering a certain number of pairs and then sending the group. This implementation buffers 256 requests which consumes 2KB memory for each MPI process. Although the reference implementation sets this default buffer size to 256, it could be possible to optimize this size for the target environment. The pseudocode for the simple implementation shown in Algorithm II is realized in MPI. There are no actual forks or joins here. The portion of the program from “fork” to “join” is executed by multiple MPI processes in parallel. This implementation holds the graph data in the CSR (Compressed Sparse Row) format.

Analysis for communication data size:

Here we estimate the total amount of data communication. If the number of MPI processes is *n* and the number of edges is M, then the amount of data to be sent C(n,M) is calculated as

C(n, M) = A * B * C * D (bytes), where A = M*2, B = (n-1)/n, C=2, and D=8.

Algorithm II: The *simple* implementation

```
1  for all vertex v do
2  | pred[v] ← -1;
3  | visited[v] ← 0;
4  CQ ← Empty;
5  NQ ← Empty;
6  CQ[root] ← 1;
7  fork;
8  this ← GetMyRank();
9  loop
10 | while CQ != Empty do
11 | | for each received vertex v and its predecessor u do
12 | | | if visited[v] = 0 then
13 | | | | visited[v] ← 1;
14 | | | | pred[v] ← u;
15 | | | | Enqueue(NQ, v);
16 | | u ← Dequeue(CQ);
17 | | for each vertex v adjacent to u do
18 | | | r ← GetOwner(v);
19 | | | if r = this then
20 | | | | if visited[v] = 0 then
21 | | | | | visited[v] ← 1;
22 | | | | | pred[v] ← u;
23 | | | | | Enqueue(NQ, v);
24 | | | else
25 | | | | send (v, u) to r;
26 | if new queue of all the processes is empty then
27 | | break;
28 | swap(CQ, NQ);
29 join;
```

Although the number of edges to be sent is M, the total number of edges (A) to be transferred will be twice M since the target graph is undirected. Each MPI process handles 1/n edges so all of the other MPI processes together hold (n-1)/n edges. Since each MPI process also needs to transfer the parent vertex corresponding to each vertex, C is 2. D is 8 because it is a 64 bit vertex identifier for each vertex. In summary, this implementation handles twice the number vertices of the entire graph data in the sparse matrix format.

For example, if s = 32 and n = 128 (128 MPI processes), then the total communication data volume C(128, M) is calculated as 2^32 * 16 * 2 * (127/128) * 2 * 8 = 2032 GB

(2 TB). As previously noted, the entire graph for M(32) was 1.03 TB, so the amount of data was doubled.

B. The replicated-csr implementation

The current queue (CQ) and new queue (NQ) are represented as bitmaps respectively named "in_queue" and "out_queue" in this implementation. The pseudocode is shown in Algorithm III. Both CQ and NQ are bitmap arrays in which each bit indicates whether or not the corresponding vertex exists in a queue. CQ possesses all of its vertex information as single bits. Each MPI process holds the entire graph data as a bit array called *in_queue*. This means the implementation is "replicated" since all of the graph data is replicated across all of the MPI processes. NQ (*out_queue*) for each MPI process only has the set of vertices that the specific process is responsible for. Thus the size of NQ is the number of vertices (in bits). This accounts for 1/n of CQ (where n is the number of MPI processes). Meanwhile, the *parallel* keyword in Algorithm III indicates the *for* loop is executed in parallel by multiple threads. The implementation uses an OpenMP directive that automates this parallelization.

As an optimization, this version has a data structure called *in_queue_summary* that holds summarized information for 64 (type is *ulong*) bits in CQ as 1 bit. If the corresponding 64 bits in CQ are all 0, the summary is 0. Otherwise, the summary bit is 1.

Lines 11 to 18 of Algorithm III are synchronized. First, the entire adjacency matrix of the graph is divided by n, the number of MPI processes, and each block of vertices is scattered across multiple MPI processes. Figure 5 introduces an adjacency matrix to clarify the explanation, but the actual data structures are in CSR (Compressed Sparse Row) format. Assume we focus on a specific MPI process whose rank is k. The MPI process k only processes A_k vertices. The code at Line 11 handles each local vertex, u, and then one of its adjacent vertices, v, is found in Line 13. This process is Step ① in Figure 5. In Algorithm III, the *for* loop at Line 11 is annotated as *parallel*, which indicates the OpenMP directive that enables multiple threads to process each local vertex simultaneously. Then Line 14 checks whether it was already visited by checking whether in_queue[u] equals 1 (Step ② in Figure 5). Then Line 17 marks the vertex v, the vertex adjacent to u, as visited by setting out_queue[v] to 1 (Step ③ in Figure 5). At Line 20, the out_queue bits of each MPI process are gathered by the MPI_Allgather collective operation and are combined into in_queue, which holds the latest status of the entire graph. After the collective operation is terminated, the synchronization at each level of the level-synchronized BFS is finished and it continues to the next level.

Analysis for expected performance

From the performance perspective, this implementation only looks at a set of local vertices each MPI is responsible for. Thus the execution time should be decreased as the loop from Line 7 to Line 20 is advanced becomes fewer local vertices becomes visited at higher level. This analysis is evaluated in Figure 18 of the performance evaluation section. As the drawback of this implementation, Line 14, which the adjacent vertex v is random, becomes random memory access which might be the cause the degraded performance.

Algorithm III: The *replicated-csr* implementation

```
1   for all vertex v do
2   | pred[v] ← -1;
3   | visited[v] ← 0;
4   in_queue[root] ← 1;
5   fork;
6   loop
7   while in_queue contains more than 1 vertex
8   | for all local vertex u do
9   | | out_queue[u] ← 0
10  | Synchronize;
11  | for all local vertex u parallel do // only access the local vertices
12  | | if visited[u] = 0 then
13  | | | for each v adjacent to u do
14  | | | | if in_queue[v] = 1 then // random access
15  | | | | | pred[u] = v;
16  | | | | | visited[u] = 1;
17  | | | | | out_queue[u] ← 1;
18  | | | | | break;
19  | Synchronize;
20  | in_queue ← all gather out_queue;
21  join;
```

Analysis for the amount of data exchanged:

The total amount of data bytes transmitted is calculated as C(d, n, V) = A * B * C * d = (V/(8*n))*(n-1)*n*d = V*(n-1)*d/8, where d is the level size - the distance from the root to the furthest vertex, n is the number of MPI processes, and V is the number of vertices. The value of A = V/(8*n), which represents the byte size of a bit array for the vertices held by each MPI process. Each vertex identifier is represented as 8 bits. The B = (n-1) and refers to the amount of data each MPI process needs to send to all of the other (n-1) MPI processes. Finally C = n for the total number of MPI processes. If V=2^32, d=12, n=128, and C(12, 128, 2^32), then the data transferred is 762 GB. The primary communication is done only by the *MPI_Allgather* collective operations when synchronization occurs at each level.

Analysis for vertices distribution

Assignments of vertices to MPI processes are determined by computing the module of the vertex ID for the total number of MPI processes. Thus if all of the vertex IDs are sorted in a decreasing order, a set of vertices that each MPI process needs to handle is not contiguous. This leads to an expensive collection operation in *MPI_Allgather* if the vertex ID is used for the index of CQ (*in_queue*). To avoid this, the index of CQ (in_queue) uses "*swizzeled*" vertex IDs that are translated so that the vertices each MPI process handles are made contiguous. The bits for each vertex ID are represented as <- SCALE ->||<- local ->|<- rank ->| and the swizzeled vertex ID is <- SCALE ->||<- rank ->| <- local ->|

based on reversing the local and rank bits.

C. *The replicated-csc implementation*

Although the name of the implementation is similar to *replicated-csc*, its algorithm and its data format is different except for the fact that all the MPI processes has replicated information on the entire graph. The graph representation uses a sparse matrix format called CSC (Compressed Sparse Column). The pseudocode appear as Algorithm IV. Each MPI process has an *in_queue* that contains all of the vertex information as to whether a vertex exists in *in_queue*. In contrast, *out_queue* only has information on the local vertices that each MPI process is handling. In Line 6, it checks whether or not *in_queue* contains more than one vertex. If so, it loop until *in_queue* become empty. For Line 8, all of the local vertices are set to 0 which means “not visited”. In line 10, all of the vertices in *in_queue* are processed in parallel by multiple threads spawned with OpenMP. If the vertex exists in *in_queue* at Line 11, then it finds a set of local vertices adjacent to the global vertex in *in_queue*. In Line 13, if the local vertex has not been visited by each MPI process, then the predecessor of the vertex is set to be the global vertex. In Line 16, the local vertex is added to *out_queue* by setting the *out_queue* bit to 1. After all of the MPIs process Line 16, the algorithm waits until all of the MPI processes have completed the loop from Line 11 to Line 16, so all of the MPI processes can gather their own *out_queue* into *in_queue* with the *MPI_Allgather* function.

Algorithm IV: The *replicated-csc* implementation

```
1   for all vertex v do
2   | pred[v] ← -1;
3   | visited[v] ← 0;
4   in_queue[root] ← 1;
5   fork;
6   while in_queue contains more than 1 vertex
7   | for all local vertex lu in out_queue do
8   | | out_queue[lu] ← 0
9   | Synchronize;
10  | for all global vertex gu in in_queue parallel do (contiguous access)
11  | | if in_queue[gu] = 1 then
12  | | | for each local vertex v adjacent to gu do
13  | | | | if visited[v] = 0 then
14  | | | | | pred[v] = gu;
15  | | | | | visited[v] = 1;
16  | | | | | out_queue[v] ← 1;
17  | Synchronize;
18  | in_queue ← all gather out_queue;
19  join;
```

Figure 6 shows the simplified image of the *replicated_csc* algorithm. As previously described, *in_queue* holds all the information for all the vertices in the entire graph. Since the actual data representation is CSC (Compressed Sparse Column), there is also column index that indicates in which column each vertex is located. In Step 1 in Figure 6 the algorithm traverses all of the vertices, looking for the vertices that reside in the local process. If a vertex and its adjacent vertex, *v*, are local, then it adds the adjacent *v* to *out_queue*. The CSC data format uses two arrays, a column index array and a row number array. For this format, the length of the column index array is the number of vertices for the entire graph, which uses a lot of memory. To reduce the memory needed, 32 columns are grouped and represented as one index. To obtain the original column number, the row number array contains 5 bits that represents the module of the row number divided by 32. The bit sequence of the row number array is |<- local ->|<- row number mod 32 ->|. The MPI rank does not need to be stored since the MPI process knows it is a local vertex.

Analysis for expected performance

From the performance perspective, this implementation needs to look at all the global vertices to understand whether it exists in *in_queue*. In some way, this is contiguous memory access when compared to random memory access occurred in *replicated-csr*. However, this processing at Line 10 might be the cause of the degraded performance since *replicated-csr* only looks at a set of local vertices each MPI is responsible for.

IV. Performance Evaluation

We used TSUBAME 2.0, the fifth fastest supercomputer in the TOP500 list of June 2011. This section describes the architecture of TSUBAME 2.0.

A. *Overview of the TSUBAME 2.0 supercomputer*

TSUBAME 2.0 is a production supercomputer operated by Global Scientific Information and Computing Center (GSIC) at the Tokyo Institute of Technology. TSUBAME 2.0 has more than 1,400 compute nodes interconnected by high-bandwidth full-bisection-wide Infiniband fat nodes, which can differ in their local memory capacity. All of the compute nodes share a scalable storage system with a capacity of 7 PB.

Each TSUBAME 2.0 node has two Intel Westmere EP 2.93 GHz processors (Xeon X5670, 256-KB L2 cache, 12-MB L3), three NVIDIA Fermi M2050 GPUs, and 50 GB of local memory. The operating system is SUSE Linux Enterprise 11. Each node has a theoretical peak of 1.7 teraflops (TFLOPS). The main system consists of 1,408 computing nodes, and the total peak performance can reach 2.4 PFLOPS. Each of the CPUs in TSUBAME 2.0 has six physical cores and supports up to 12 hardware threads with Intel’s hyper-threading technology, thus achieving up to 76 gigaflops (GFLOPS).

The interconnect that links the 1,400 computing nodes with storage is the latest QDR Infiniband (IB) network, which has 40 Gbps of bandwidth per link. Each computing node is connected to two IB links, so the communication bandwidth for the node is about 80 times larger than a fast LAN (1 Gbps). Not only the link speed at the end-point nodes, but the network topology of the entire system heavily affects the performance for large computations. TSUBAME 2.0 uses a full-bisection fat-tree topology, which accommodates

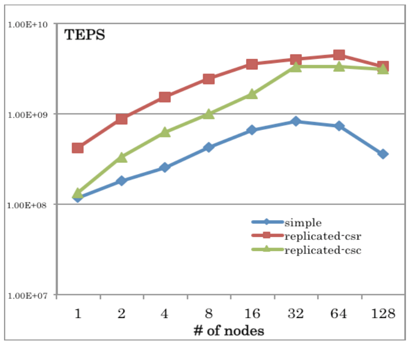

Figure 7. Strong-scaling performance comparison with OpenMPI (Scale:26)

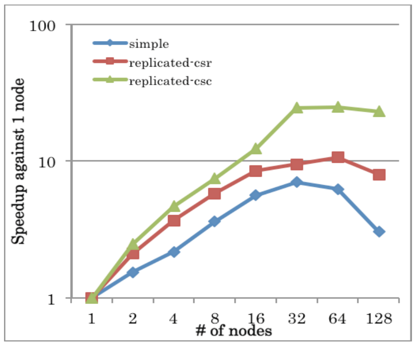

Figure 8. Strong-scaling: Speed-up ratio comparison with OpenMPI (Scale: 26)

applications that need more bandwidth than provided by such topologies as a torus or mesh. One of the key features of the TSUBAME 2.0 architecture is high-bandwidth hardware that transmits data efficiently. To sustain high rates of intra-node communications, the memory bandwidth is 32 GB/s to each CPU and 150 GB/s to each GPU. Communication between CPUs and GPUs is supported by the latest PCI-Express 2.0 x 16 technology with a bandwidth of 8 GB/s

B. *Evaluation Method*

We used GNU gcc 4.3.4 as the C compiler, which is an OpenMP-3.0-compliant implementation. For the MPI, we used OpenMPI 1.4.2 [18] and MVAPICH2 1.5.1 [17]. Both implementations are compliant with the MPI 2 specification. MVAPICH2 [17] is an optimized implementation with a high-speed network layer for Infiniband. Since TSUBAME 2.0 uses dual-link QDR Infiniband, the total bandwidth is 80 Gbps, and MVAPICH2 is suitable for this environment.

Since *replicated-csr* and *replicated-csc* implementations use MPI and OpenMP, we need appropriate numbers of MPI ranks (processes) and OpenMP threads. Each node has dual sockets for two 6-core Intel Westmere processors with the HyperThread enabled, so the number of hardware threads is 24. To determine how many MPI ranks and OpenMP threads to spawn for maximal throughput, we tested different values as shown in Table 1. The Scale was 22 and the implementation was replicated-csr. The results showed that 2 MPI processes and 12 threads per node achieved the best performance. Therefore we used this configuration for all of the benchmarks except *simple*, since *simple* is implemented without OpenMP and only runs as one thread/core. For the *simple* implementation, we spawned 12 MPI processes per node.

Table 1 TEPS with varying numbers of MPI ranks (processes) and OpenMP threads

# of MPI ranks	16	8	8	8	4	4	2	2	1	1
OpenMP Threads	1	1	2	3	3	6	6	12	12	24
Total Threads	16	8	16	24	12	24	12	24	12	24
TEPS (E+08)	3.28	1.62	3.08	4.00	2.58	4.21	2.49	4.45	2.46	4.13

C. *Performance Analysis for Strong Scaling*

Figure 7 shows the throughput in TEPS in a strong-scaling manner when using OpenMPI. Since this is a strong-scaling evaluation, the problem size has a constant scale (Scale 26), and the number of nodes varies from 1 (24 cores) up to 128 nodes (3,072 cores). The Scale was set to 26 since this is the largest size that one node (24 cores, 52 GB RAM) can handle. We used three versions of reference implementations, *simple*, *replicated-csr*, and *replicated-csc*. Figure 8 shows the ratio of acceleration of each node against 1 node. As shown in Figure 7 and Figure 8, *replicated-csc* shows the best throughput (TEPS) among the three versions. The scalability is almost linear with the number of nodes up to 32 nodes. The acceleration ratio with 32 nodes against 1 node is 24.65 for *replicated-csc*, 9.46 for *replicated-csr*, and 7.05 for *simple*. However, since the problem is relatively small for the large-scale environments with 64 and 128 nodes, the throughput and acceleration were effectively saturated at 32 nodes.

D. *Performance Analysis for Weak Scaling*

To study the performance characteristics, we also conducted the experiments in a weak-scaling manner by fixing the number of vertices at 1 node. This weak-scaling evaluation shows how the linear scalability is achieved and how much of the performance degradation is due to the communication and level synchronization among the multiple MPI processes and other causes. The number of nodes used for this experiment was 1, 2, 4, 8, 16, 32, and 64 nodes. Since each node has 24 cores with HyperThread enabled, 64 nodes represent 1,536 cores. For the graph size, we tested two versions of Scale in each node, Scales 24 and 26 (Experiments I and II, respectively). Based on the formula for memory consumption in Section II, Scale 24 uses 4.43 GB per node and Scale 26 needs 17.71 GB to store all of the vertices and edges in the CSC format. The physical memory of each node was 52 GB.

The Scales for Experiment I were 24, 25, 26, 27, 28, 29, and 30, respectively, for 1, 2, 4, 8, 16, 32, and 64 nodes. The Scales for Experiment II were 26, 27, 28, 29, 30, 31, and 32 for 1, 2, 4, 8, 16, 32, and 64 nodes (**Due to the program*

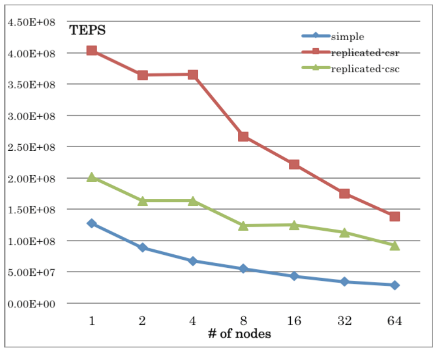

Figure 9. Weak-scaling performance with OpenMPI (Scale 24 per node)

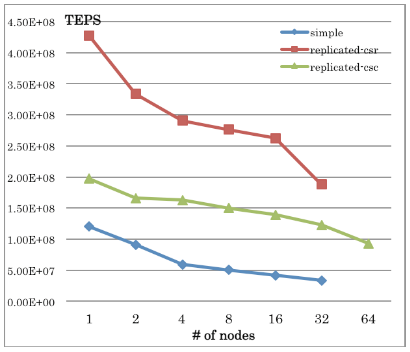

Figure 10. Weak-scaling performance with OpenMPI (Scale 26 per node)

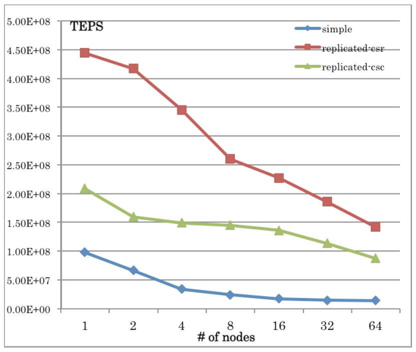

Figure 11. Weak-scaling performance with MVAPICH2 (Scale 24 per node)

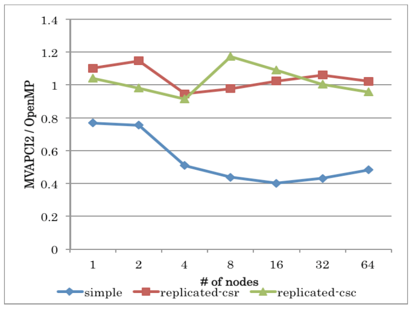

Figure 12. Comparison between OpenMPI and MVAPICH2 in weak-scaling (Scale 24 per node)

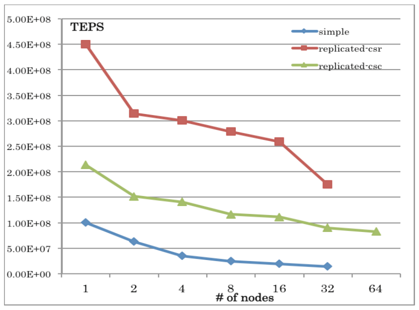

Figure 13. Weak-scaling performance with MVAPICH2 (Scale 26 per node)

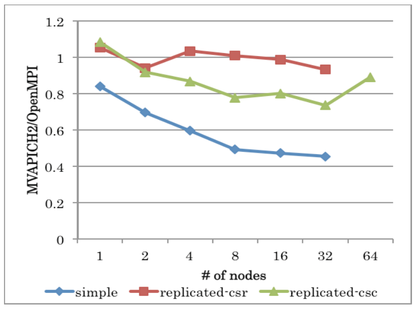

Figure 14. Comparison between OpenMPI and MVAPICH2 in weak-scaling (Scale 26 per node)

fault and unstable result, we could not show the result for 128 nodes in two experiments). Figure 9 shows the results of Experiment I (Scale 24 per node) and Figure 10 shows the results of Experiment II (Scale 26 per node). The *replicated-csr* version shows a relatively high score (more than 8 GE/s) in Experiment I with 1, 2, and 4 nodes, but the throughput drop from 8 GE/s with 1 node to 6 GE/s with 2 nodes in Experiment II. The *replicated-csc* implementation is relatively stable near 6 GE/s, though there is some degradation from 32 nodes and up. Overall, the throughput of *replicated-csr* is better than *replicated-csc*. Figure 11 is the weak-scaling performance with Scale 24 and MVAPICH2 and Figure 12 shows the comparison between OpenMPI and MVAPICH2. Figure 12 shows that OpenMPI and MVAPICH2 are nearly equal with the replication-based algorithm that has 2 MPI processes per node. However, MVAPICH2 shows relatively poor performance over OpenMPI, since 12 MPI processes per node are spawned and the communication overhead becomes dominant. Figure 13 and Figure 14 correspond to Figures 11 and 12, except for using Scale 26 in each node. As previously described, MVAPICH2 is optimized for the Infiniband network, so this result is not what we expected.

E. In depth Analysis - Profiling Message Size

Hereafter we describe in-depth analysis such as message size, execution time breakdown, and graph generation/construction/validation time. For this profiling, we only tested OpenMPI as the MPI implementation.

Figure 15 shows the communication data size at each level of the level-synchronized BFS for the simple implementation. Since all of the reference implementations use a level-synchronized approach, the actual level size was 8 in the case of weak-scaling experiment II. For the simple implementation, each vertex and its parent vertex is sent to the other MPI process via *MPI_Isend*. The x-axis is level and the y-axis is the total data size transferred by the *MPI_Isend* function in the simple implementation. Three lines are shown, *sim-29-8* with Scale 29 and 8 nodes, *sim-30-16* with Scale 30 and 16 nodes, and *sim-31-32* with Scale 31 and 32 nodes. As shown in the graph, the message size and the number of calls to *MPI_Isend* increases logarithmically from level 1 to level 4, and then decreases from level 4 to level 8.

In contrast, the replication-based implementations including *replicated-csr* and *replicated-csc* have different message transfer characteristics. Figure 16 shows the aggregated message size when each MPI process aggregates the out_queue data into in_queue by using the *MPI_Allgather* collection operation. The *replicated-csr* and *replicated-csc* implementations show the same communication patterns. As shown in Figure 16, the total message size increases linearly. When 64 nodes and 128 MPI processes are used, the total aggregated message size for the # of MPI processes can reach 485 gigabytes.

F. In depth Analysis – Profiling Execution Time at Each Level

Figure 17 and Figure 18 shows the execution time at each level of the level-synchronized BFS. The x-axis is the level number (from 1 to 8 in this experiment) and the y-axis is the

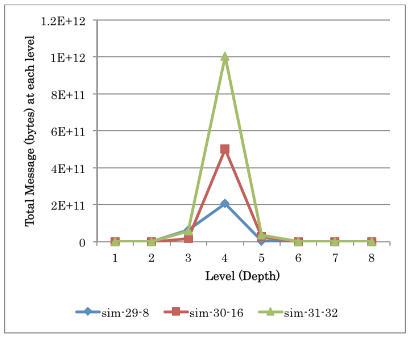

Figure 15. Aggregated message size at each level of the simple implementation

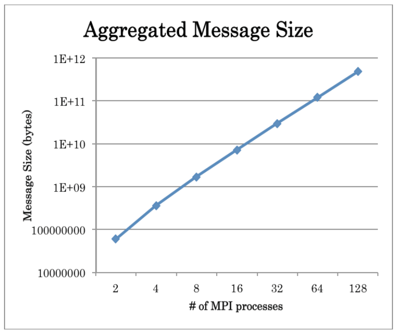

Figure 16. Aggregated message size in bytes of replicated-based implementations (Experiment II)

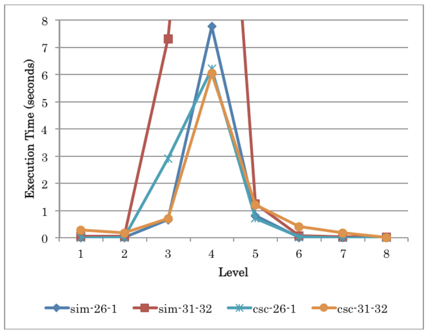

Figure 17. Execution time at each level of the level-synchronized BFS (simple and replicated-csc)

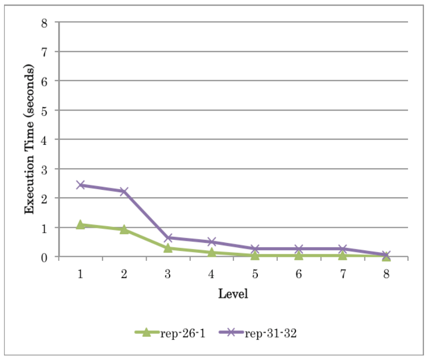

Figure 18 Execution time at each level of the level-synchronized BFS (replicated-csr)

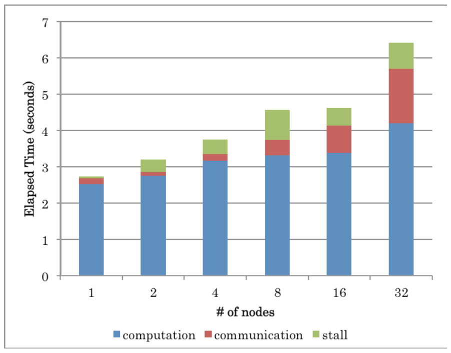

Figure 19. Execution breakdown for *replicated-csr* implementation (Experiment II)

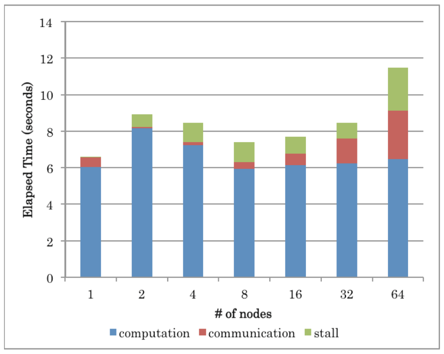

Figure 20. Execution breakdown for *replicated-csc* implementation (Experiment II)

execution time in seconds for each MPI process at each level. The first part of the label in the graph, sim, rep, or csr means *simple*, *replicated-csc*, or *replicated-csr*, respectively. The first number of the label is the problem size, Scale. The second number is the number of nodes. As shown in Figure 17, the execution time of level 4 takes longer time than other levels and decreases to small execution time for the simple and replicated-csc implementations such as sim-26-1, sim-31-32, csc-26-1, and csc-31-32. However, shown in Figure 18, the execution time for the replicated-csr implementation has different characteristics. As the level goes up, the execution time is decreased shown in Figure 18.

We believe the *csc* and *simple* implementation have similar performance characteristics because the *csc* implementation needs to find all of the vertices in *in_queue* that contain all the global vertex information by checking the corresponding bit in the *in_queue* array. It then checks whether or not the adjacent vertices of an existing vertex were visited. Thus, when *in_queue* contains more vertices at higher level, its processing time increases. This means the *csc* implementation has similar performance characteristics as the processing time is increasing from level 1 up to level 4, and then starts decreasing. In contrast, the *csr* implementation only checks whether adjacent "local" vertices are unvisited in_queue, and thus as the number of unvisited vertices decreases, the processing time also decreases.

G. In depth Analysis – Profiling Computation, Communication, and Stall Times

Figure 19 and Figure 20 show the execution time breakdowns in three phases, computation, communication, and stall times for the *replicated-csr* and *replicated-csc* implementations, respectively. For 64 nodes, we could not obtain the profiling data in Figure 19 since it needs to solve the Scale 32 and it crashes due to the memory error and the validation error. We also could not the profiling data for the *simple* implementation just because it takes too much time to profile massive amount of MPI processes (16 processes per node) with profiling mode.

The stall time reflects the time for level synchronization. Both graphs show the stall and communication time is increasingly dominant as the number of the nodes increases from 1 to 32 or 64. This profiling result clearly explains why performance degradation is observed in the weak-scaling experiments shown in Figure 9 and Figure 10.

The execution time in Figure 19 is increased as the number of nodes is increased. We assume that this might be because the cache hit ratio for *in_queue* of the *replicated-csr* implementation becomes lower. However further study is needed for more quantitative proof with using *Oprofile* or other profiling tools. Meanwhile, Figure 20 shows that the computation time is almost constant, which is expected since this is weak-scaling experiment.

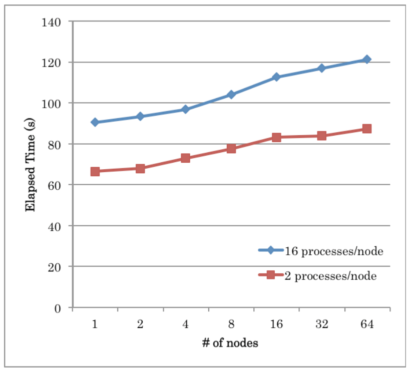

Figure 21. Graph Generation Time (seconds) (Weak-scaling, Scale 26 per node)

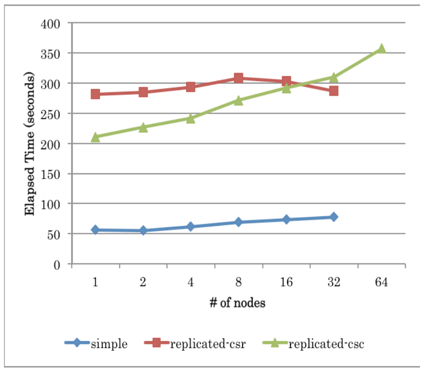

Figure 22. Graph Construction Time (seconds) (Weak-scaling, Scale 26 per node)

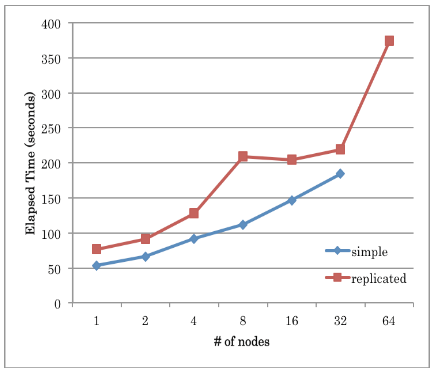

Figure 23. Validation Time (seconds) (Weak-scaling, Scale 26 per node)

H. In depth Analysis – Graph Generation, Construction, and Validation

Figures 21, 22, and 23 showed the elapsed times for graph generation, graph construction, and validation. These three phases are not relevant for the TEPS scores, but for iterative experiments to understand the performance of optimized implementations, it is important to understand how long they take. The initial time for graph generation with the Kronecker graph generator is shown in Figure 21. The x-axis is the number of nodes and the y-axis is the elapsed time in seconds. The three reference implementations use the same generation logic, so we only show two versions. Since the simple implementation is not multi-threaded, it spawns 16 MPI processes, while the others use 2 MPI processes. We are comparing the elapsed time for 2 and 16 MPI processes in a weak-scaling manner for Scale 26. As shown in the graph, 2 processes outperform 16 processes due to the communication overhead among large numbers of MPI processes.

Figure 22 shows the time for graph construction. This graph construction is the data conversion from the edge list to the efficient data format such as CSR and CSC used for Kernel 2. The *simple* implementation also uses the CSR format as the *replicated-csr* implementation. The graph construction phase has fewer parallelization phase helped by the OpenMP multi-threading. However, since 16 MPI processes needs to be spawned off for the *simple* implementation due to its single threading, it automatically parallelizes the processing by MPI. However, the replicated-based implementations only use 2 MPI processes because BFS is parallelized by multi-threading. For such a reason, the *simple* implementation greatly outperforms the *replicated-csr* implementation that only uses 2 MPI processes.

For the validation phase, Figure 23 compares *simple* and *replicated* since both *replicated-csc* and *replicated-csr* use the same validation logic. The weak-scaling result in the graph shows the time is increasing with the number of nodes due to the communication overhead.

I. Analysis Summary

Our thorough study reveals the performance characteristics of reference implementations on large-scale distributed environment using the TSUBAME2 supercomputer. We could only tested at maximum 64 nodes (1536 cores) for weak-scaling due to the upper limit of physical memory size, but still the profiling results, especially the execution breakdown shows that communication is getting dominant with larger nodes. This study shows that the reference replicated-based implementations can not be used for larger nodes such as 128, 256, 512, and 1024 nodes due to its communication overhead shown in Figure 19 and Figure 20. Currently the *simple* implementation shows bad performance since it runs with single thread and is not parallelized by OpenMP as the replicated-based implementations, but it has potentiality for obtaining linear scalability in such a larger environment when we design an optimization method based on the findings in this paper.

V. Related Work

Yoo [4] presents a distributed BFS scheme that scales on the IBM BlueGene/L with 32,768 nodes. Their optimization differs in the scalable use of memory in that they use 2D (edge) partitioning of the graph instead of conventional 1D (vertex) partitioning to reduce the communication overhead. Since BlueGene/L has a 3D torus network, they developed efficient collective communication functions for that network. Bader [3] describes the performance of optimized parallel BFS algorithms on multithreaded architectures such as the Cray MTA-2.

Agarwal [2] proposes an efficient and scalable BFS algorithm for commodity multicore processors such as the 8-core Intel Nehalem EX processor. With the 4-socket Nehalem EX (Xeon 7560, 2.26 Ghz, 32 cores, 64 threads with Hyper-Threading), they ran 2.4 times faster than a Cray XMT with 128 processors when exploring a random graph with 64 million vertices and 512 million edges, and 5 times faster than 256 BlueGene/L processors on a graph with an average degree of 50. The performance impact of their proposed optimization algorithm was tested only on a single node, but it would be worthwhile to extend their proposed algorithm to larger machines with commodity multicore processors, such as TSUBAME 2.0.

Harish [10] devised a method of accelerating single-source shortest path problems with GPGPUs. Their GPGPU-based method solves the breadth-first search problem in approximately 1 second for 10 million vertices of a randomized graph where each vertex has 6 edges on average. However, the paper concluded that the GPGPU-method does not does not match the CPU-based implementation for scale-free graphs such as the road network of the 9th DIMACS implementation challenge, since the distribution of degrees follows a power law in which some vertices have much higher degrees than others. However since the top-ranked supercomputers in TOP500 are equipped with GPGPUs for compute-intensive applications, it would be worthwhile to pursue the optimization of Graph500 by exploiting GPGPUs.

VI. Concluding Remarks and Future Work

In this paper we gave an overview of the internals of some Graph500 reference implementations and how they actually scale on a supercomputer build from commodity x86 processors. The maximum throughput in TEPS (Traversed Edges Per Second) was nearly 8 GE/s for the Scale 34 problem.

The findings in this paper offer opportunities for developing optimized implementations. We believe our results will help in the development of optimized algorithms for upcoming exascale machines.

For future work, we will develop optimized implementations based from the findings described in this paper. We hope to obtain Graph500 scores for larger problems and for larger numbers of nodes. TSUBAME 2.0 is characterized as a typical recent supercomputer with a large number of GPGPU accelerators that can contribute to high FLOPS results. Because Graph500 is not a compute-intensive benchmark but a more I/O intensive benchmark, it will be challenging to leverage its GPU capabilities. The technical challenges of exploiting the massive amounts of GPGPU power for the Graph500 benchmark await us.

References

[1] Graph500 : http://www.graph500.org/

[2] Virat Agarwal, Fabrizio Petrini, Davide Pasetto, and David A. Bader. 2010. Scalable Graph Exploration on Multicore Processors. In Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '10). IEEE Computer Society, Washington, DC, USA, 1-11

[3] David A. Bader and Kamesh Madduri. 2006. Designing Multithreaded Algorithms for Breadth-First Search and st-connectivity on the Cray MTA-2. In Proceedings of the 2006 International Conference on Parallel Processing (ICPP '06). IEEE Computer Society, Washington, DC, USA, 523-530

[4] Andy Yoo, Edmond Chow, Keith Henderson, William McLendon, Bruce Hendrickson, and Umit Catalyurek. 2005. A Scalable Distributed Parallel Breadth-First Search Algorithm on BlueGene/L. In Proceedings of the 2005 ACM/IEEE conference on Supercomputing (SC '05). IEEE Computer Society, Washington, DC, USA, 25-.

[5] D.A. Bader, J. Feo, J. Gilbert, J. Kepner, D. Koester, E. Loh, K. Madduri, W. Mann, and Theresa Meuse, HPCS Scalable Synthetic Compact Applications #2 Graph Analysis (SSCA#2 v2.2 Specification), 5 September 2007.

[6] D. Chakrabarti, Y. Zhan, and C. Faloutsos, R-MAT: A recursive model for graph mining, SIAM Data Mining 2004.

[7] Bader, D., Cong, G., and Feo, J. 2005. On the architectural requirements for efficient execution of graph algorithms. In Proc. 34th Int'l Conf. on Parallel Processing (ICPP). IEEE Computer Society, Oslo, Norway.

[8] K. Madduri, D.A. Bader, J.W. Berry, and J.R. Crobak, ``Parallel Shortest Path Algorithms for Solving Large-Scale Instances," 9th DIMACS Implementation Challenge -- The Shortest Path Problem, DIMACS Center, Rutgers University, Piscataway, NJ, November 13-14, 2006.

[9] Richard C. Murphy, Jonathan Berry, William McLendon, Bruce Hendrickson, Douglas Gregor, and Andrew Lumsdaine, "DFS: A Simple to Write Yet Difficult to Execute Benchmark,", IEEE International Symposium on Workload Characterizations 2006 (IISWC06), San Jose, CA, 25-27 October 2006.

[10] Pawan Harish and P. J. Narayanan. 2007. Accelerating large graph algorithms on the GPU using CUDA. In Proceedings of the 14th international conference on High performance computing (HiPC'07), Srinivas Aluru, Manish Parashar, Ramamurthy Badrinath, and Viktor K. Prasanna (Eds.). Springer-Verlag, Berlin, Heidelberg, 197-208.

[11] Daniele Paolo Scarpazza, Oreste Villa, and Fabrizio Petrini. 2008. Efficient Breadth-First Search on the Cell/BE Processor. IEEE Trans. Parallel Distrib. Syst. 19, 10 (October 2008), 1381-1395.

[12] Douglas Gregor and Andrew Lumsdaine. 2005. Lifting sequential graph algorithms for distributed-memory parallel computation. SIGPLAN Not. 40, 10 (October 2005), 423-437.

[13] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In Proceedings of the 2010 international conference on Management of data (SIGMOD '10). ACM, New York, NY, USA, 135-146.

[14] U. Kang, Charalampos E. Tsourakakis, and Christos Faloutsos. 2009. PEGASUS: A Peta-Scale Graph Mining System Implementation and Observations. In Proceedings of the 2009 Ninth IEEE International Conference on Data Mining (ICDM '09). IEEE Computer Society, Washington, DC, USA, 229-238.

[15] Toshio Endo, Akira Nukada, Satoshi Matsuoka, and Naoya Maruyama. Linpack Evaluation on a Supercomputer with Heterogeneous Accelerators. In IEEE International Parallel & Distributed Processing Symposium (IPDPS 2010).

[16] J. Leskovec, D. Chakrabarti, J. Kleinberg, and C. Faloutsos, "Realistic, mathematically tractable graph generation and evolution, using kronecker multiplication," in Conf. on Principles and Practice of Knowledge Discovery in Databases, 2005.

[17] MVAPICH2: http://mvapich.cse.ohio-state.edu/

[18] OpenMPI : http://www.open-mpi.org/