2014 IEEE International Conference on Big Data

Parallel Breadth First Search on GPU Clusters

Zhisong Fu
SYSTAP, LLC
fuzhisong@systap.com

Harish Kumar Dasari
University of Utah
hdasari@sci.utah.edu

Bradley Bebee
SYSTAP, LLC
beebs@systap.com

Martin Berzins
University of Utah
mb@sci.utah.edu

Bryan Thompson
SYSTAP, LLC
bryan@systap.com

***Abstract*—Fast, scalable, low-cost, and low-power execution of parallel graph algorithms is important for a wide variety of commercial and public sector applications. Breadth First Search (BFS) imposes an extreme burden on memory bandwidth and network communications and has been proposed as a benchmark that may be used to evaluate current and future parallel computers. Hardware trends and manufacturing limits strongly imply that many-core devices, such as NVIDIA® GPUs and the Intel® Xeon Phi®, will become central components of such future systems.**

GPUs are well known to deliver the highest FLOPS/watt and enjoy a very significant memory bandwidth advantage over CPU architectures. Recent work has demonstrated that GPUs can deliver high performance for parallel graph algorithms and, further, that it is possible to encapsulate that capability in a manner that hides the low level details of the GPU architecture and the CUDA language but preserves the high throughput of the GPU. We extend previous research on GPUs and on scalable graph processing on supercomputers and demonstrate that a high-performance parallel graph machine can be created using commodity GPUs and networking hardware.

***Keywords*-GPU cluster, MPI, BFS, graph, parallel graph algorithm**

I. Introduction

Scalable parallel graph algorithms are critical for a large range of application domains with a vital impact on both national security and the national economy, including, among others: counter-terrorism; fraud detection; drug discovery; cyber-security; social media; logistics and supply chains; e-commerce, etc. However scalable parallel graph algorithms on large core or GPU counts is fundamentally challenging, as computational costs are relatively low compared to communications costs. Graph operations are inherently non-local, and skewed data distributions can create bottlenecks for high performance computing. Solutions based on map/reduce or requiring checkpoints to disk are relatively inflexible and 1000s of times too slow to extract the value latent in graphs in within a timely window of opportunity. Fast execution and robust scaling requires a convergence of techniques and approaches from innovative companies and the High Performance Computing (HPC) community.

Our work on graph problems is motivated by the fact that large, and often scale-free, graphs are ubiquitous in communication networks, social networks and in biological networks. These graphs are typically highly connected and have small diameters such that the frontier expands very quickly during BFS traversal as seen in Figure 1. We use the same scale-free graph generator as the Graph 500 for which there is a vertex degree distribution that follows a power law, at least asymptotically [24], [26], [27]. We focus on Breadth First Search (BFS) as this is perhaps the most challenging parallel graph problem because it has the least work per byte and the most reliance on memory bandwidth within the node while placing a severe stress on the communications network among the nodes. By demonstrating success on BFS, we hope to show that the potential of the proposed approach to a wide range of parallel graph problems.

Our research is motivated in part by Merrill *at el.* [16] who demonstrated that GPU can deliver 3 billion Traversed Edges Per Second (3 Giga-TEPS or GTEPS) across a wide range of graphs on Breadth First Search (BFS), a fundamental building block for graph algorithms. This was a more than 12× speed up over idealized multi-core CPU results. Merrill directly compared against the best published results for multi-core CPU algorithms, implemented a single-core version of the algorithms, verified performance against published single-core results, and then used idealized linear scaling to estimate multi-core performance. He found that the GPU enjoyed a speedup of at least 12× over the idealized multi-core scaling of a 3.4 GHz Intel Core i7 2600K CPU (the equivalent of 3 such 4-core CPUs). The single-GPU implementation of $MapGraph$ [1] generalizes Merrill *et al.*'s dedicated BFS solution to support a wide range of parallel graph algorithms that are expressed using the Gather Apply Scatter (GAS) abstraction [3]. This abstraction makes it easy for users to write sequential C methods and realize throughput that rivals hand-coded GPU implementations for BFS, SSSP, PageRank, and other parallel graph algorithms.

In this work, we extend the $MapGraph$ framework to operate on GPU clusters. The starting point for our design is an approach described by Checconi, et al. for the Blue Gene/Q [7]. A major challenge of our research is that GPUs are much faster than the Blue Gene/Q processors while commodity networking hardware lacks the more sophisticated communication capabilities of the Blue Gene/Q. Together, these factors can create a severe imbalance between compute and communications for bandwidth and communications constrained problems such as BFS.

Key contributions of this work include:
- A high performance implementation of parallel BFS for

978-1-4799-5666-1/14/$31.00 ©2014 IEEE

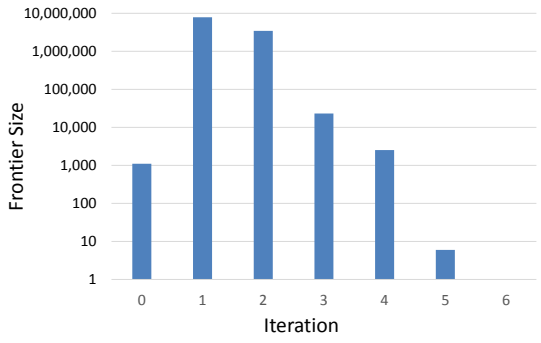

Figure 1. Frontier size during BFS traversal (scale 25)

GPU clusters with results on up to 64 GPUs and 4.3 billion directed edges.

- A strong and weak scaling study on up to 64 GPUs with an analysis of the strengths and weaknesses of our multi-GPU approach with respect to scalability.
- Identification of a $\log p$ communication pattern that may permit strong scaling to very large clusters.
- Identification of a means to eliminate the all-to-all communication pattern used by the Blue Gene/Q [7] while allowing the parallel computation of the predecessors on each GPU.

II. Related Work

Graph traversal and graph partitioning have been studied extensively in the literature. On a distributed parallel computer architecture, scalable and efficient parallel graph algorithms require a suitable decomposition of the data and the associated work. This decomposition may be done by graph partitioning with the goal of distributing data and work evenly among processors in a way that reduces communication cost. There are many graph partitioning strategies proposed in the literature. The simplest strategy is 1D partitioning, which partitions the graph vertices into disjoint sets and assigns each set of vertices to a node as implemented in parallel in widely-used packages such as Zoltan [4] and ParMetis [5].

Vastenhouw and Bisseling [6] introduce a distributed method for parallel sparse-matrix multiplication based on 2D graph partitioning. In 2D graph partitioning, the edges are distributed among the compute nodes by arranging the edges into blocks using vertex identifier ranges. These blocks are organized into an $p \times p$ grid and mapped onto p^2 virtual processors. Each row in the grid contains all out-edges for a range of vertices. The corresponding column contains the in-edges for the same vertices. In [7], the authors implement BFS with this 2D graph partitioning algorithm on IBM Blue Gene/P and Blue Gene/Q machines using optimizations to reduce communications by 97.3% (through a "wave" propagated along the rows of the 2D processor grid to eliminate duplicate vertex updates) and also optimize for the underlying network topology. They study the weak scaling and strong scaling of their design and the effect of the graph partitions on cache locality. This approach was ranked 1st in the November 2013 Graph500 on $65,536$ Blue Gene/Q processors (http://www.graph500.org).

Many of the 1D and 2D Graph partitioning algorithms perform partitioning on the original graphs and try to minimize the edge-cuts in order to minimize the communication costs. Catalyureck [8] showed that hypergraphs more accurately model the communication cost leading to packages such as [9], [4].

Several methods and software packages are introduced in the literature to develop scalable, high performance graph algorithms on parallel architectures. In [11], the authors try to address the problem of how graph partitioning can be effectively integrated into large graph processing in the cloud environment by developing a novel graph partitioning framework. Also, Berry et al. [12], introduce the Multi-Threaded Graph Library (MTGL), generic graph query software for processing semantic graphs on multithreaded computers while Bader [13] introduces a parallel graph library (SNAP). Agarwal [25] presents results for BFS on Intel processors for up to 64 threads in a single system with comparisons against the Cray XMT, Cray MTA-2, and Blue Gene/L. Pearce [17] presents results for multi-core scaling using asynchronous methods while [18] and [19] present an approach to graph processing based on sparse matrix-vector operations. The approaches of PowerGraph [2] and GraphChi [10] have been shown to be equivalent to 2D partitioning.

As noted above, Merrill *et al.* [16] developed the first work-efficient implementation of BFS on GPUs. He developed adaptive strategies for assigning threads, warps, and CTAs to vertices and edges, and optimized frontier expansion using various heuristics to trade off time and space and obtain high throughput for algorithms with dynamic frontiers. While [16] offers the best results to date for BFS on a single GPU, Fu *et al.* offer nearly equivalent performance on BFS in MapGraph [1] using a high-level abstraction and a data-parallel runtime (the performance may be slightly higher or lower depending on the data set). Gharaibeh and Zhong present multi-GPU (single workstation) results for Page Rank (PR) and BFS [20]. Gharaibeh defines a performance model for hybrid CPU/GPU graph processing, tests that model with up to 2 GPUs using random edge cuts, and notes that aggregation can reduce communications costs. Zhong uses 1D partitioning and also tests n-hop partitioning (to increase locality through redundancy). While some of these approaches use multiple GPUs on a single node, none of these approaches scales to GPU clusters.

In addressing multi-GPU scalability we will draw on lessons learned from large-scale parallel scalability for the Uintah Software (http://www.uintah.utah.edu). Uintah runs on, and scales to, the very largest machines by using high-level abstractions, such as: (a) a domain specific abstraction for writing analytics; (b) a low-level, data-parallel runtime system with adaptive, asynchronous and latency-hiding task

execution and (c) a data warehouse on each multi-core/GPU node which abstracts away hardware specific operations required to support the movement of data [22]. In implementing BFS on GPU clusters, we combine the Uintah design philosophy with the wave communication pattern of [7] (see above) and the Mapgraph approach [1] to parallel edge expansion on GPUs.

III. Multi-GPU Implementation

While writing a single GPU graph-processing program is difficult due to the data-intensive graph algorithms, irregular data storage and access and dynamically varying workload, this challenge has been addressed in *MapGraph* [1]. In this paper, we extend that work to BFS on multiple GPUs using a Bulk Synchronous Parallel (BSP) approach with a synchronization point at the end of every BFS iteration.

A. Partitioning

The graphs we use are generated in parallel using the Graph 500 *Kronecker* Generator [15]. The edges are partitioned using 2D partitioning, similar to the partitioning mechanism used by [7]. Our code models each undirected edge as two directed edges. Each directed edge, consisting of a source vertex, a target vertex and an edge value, is stored on a single GPU. The adjacency matrix is partitioned in two dimensions with an equal number of vertices partitioned across each row and column. In our case, if we have p^2 GPUs, we have to partition the matrix into p partitions along both the rows and columns. We layout the source vertices along the rows and the target vertices along the columns. The global vertex identifiers are converted to local vertex identifiers, and the data are written into a file system, with each line containing an edge (row identifier, column identifier, edge weight). At run time, the CPUs read the edge list and convert it to a CSR (Compressed Sparse Row) sparse matrix. The data are then copied to the GPU's global memory, where the GPU builds a CSC (Compressed Sparse Column) sparse matrix. The CSR matrix is used during traversal. Once the traversal is complete, the CSC matrix is used to compute the predecessors from the assigned levels.

B. BFS Traversal

The algorithm for the distributed BFS traversal is described in the Figure 3. The search begins with a single search key, called starting vertex, that is communicated to all GPUs as the initial set of vertices that we should process. We call this vertex set the *global frontier*. In line 2 of Figure 3, each GPU, denoted G_{ij}, decides if the starting vertex falls in its range and sets the corresponding bit in the bitmap In_i^t, where t stands for the iteration t of BFS. In line 4 of Figure 3, all GPUs that contain the starting vertex perform a data parallel local operation (*Expand* 4) in parallel, in which they compute the 1-hop expansion of the active vertices over the local edges and produce a new frontier. The new

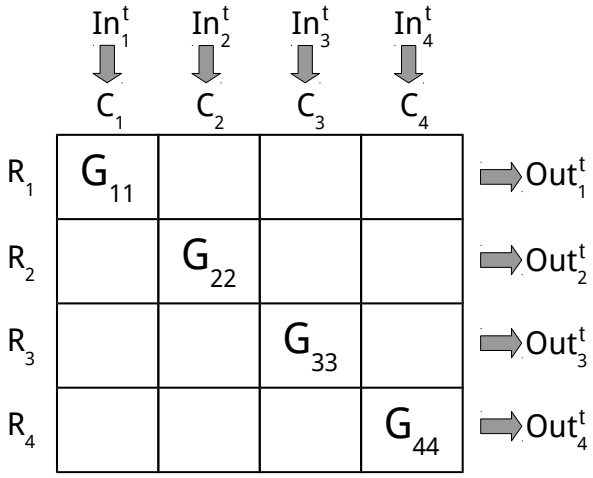

Figure 2. Multi-GPU BFS algorithm

frontier is also represented as a bitmap, denoted Out_{ij}^t. Next, the global frontier size is computed in line 5 and 6 with a Count method and a Reduce method. Then we check if the global frontier size is greater than zero. If so, we perform the Contract operation that will be describe below and the update the levels of the discovered vertices in the Expand operation. If the global frontier size is zero, we terminate the iteration and compute the predecessors from the levels 7.

The detail of the implementation of the Expand operation is shown in Figure 4. In the algorithm, line 2 first converts the bitmap to a vertex frontier list, and line 3 initializes an empty vertex frontier list. Next, each vertex in the frontier list L_{in} is assigned to a thread. Each thread then concurrently looks up in the CSR representation (consisting of RowOff and ColIdx arrays) of the graph adjacency to find the neighbors of the vertex and puts the neighbors into the new vertex frontier list L_{out}. In this process, each thread can enlist other threads to cooperatively handle its neighbors depending on its degree. This is described in detail in [1]. Finally, line 10 converts the new vertex frontier list L_{out} back to a bitmap.

C. Global Frontier Contraction and Communication

The Out_{ij}^t generated by the GPUs are contracted globally across the rows R_i of the partitions using prefix sum technique similar to the *wave* method used by the Blue Gene/Q [7]. This operation removes duplicates from the frontier that would otherwise be redundantly searched in the next iteration. It also resolves conflicts among the GPUs arising from the simultaneous discovery of the same vertex by more than one GPU by deciding which GPU will update the state associated with each vertex discovered during that iteration. The $Assigned_{ij}$ bitmap makes sure that only one GPU gets the chance to update the state associated with a particular vertex. The implementation of the Global Frontier contraction is shown in Figure 5.

Instead of the sequential propagate communication pattern described in [7], we used a parallel scan algorithm [14]

```
1: procedure BFS(Root, Predecessor)
2:     In_ij^0 ← LocalVertex(Root)
3:     for t ← 0 do
4:         Expand(In_i^t, Out_ij^t)
5:         LocalFrontier_t ← Count(Out_ij^t)
6:         GlobalFrontier_t ← Reduce(LocalFrontier_t)
7:         if GlobalFrontier_t > 0 then
8:             Contract(Out_ij^t, Out_j^t, In_i^{t+1}, Assign_ij)
9:             UpdateLevels(Out_j^t, t, level)
10:        else
11:            UpdatePreds(Assign_ij, Preds_ij, level)
12:            break
13:        end if
14:        t++
15:    end for
16: end procedure
```

Figure 3. Distributed Breadth First Search algorithm

```
1: procedure EXPAND(In_i^t, Out_ij^t)
2:     L_in ← convert(In_i^t)
3:     L_out ← ∅
4:     for all v ∈ L_in in parallel do
5:         for i ← RowOff[v], RowOff[v + 1] do
6:             c ← ColIdx[i]
7:             L_out ← c
8:         end for
9:     end for
10:    Out_ij^t ← convert(L_out)
11: end procedure
```

Figure 4. Expand algorithm

```
1: procedure CONTRACT(Out_ij^t, Out_j^t, In_i^{t+1}, Assign_ij)
2:     Prefix_ij^t ← ExclusiveScan_j(Out_ij^t)
3:     Assigned_ij ← Assigned_ij ∪ (Out_ij^t − Prefix_ij^t)
4:     if i = p then
5:         Out_j^t ← Out_ij^t ∪ Prefix_ij^t
6:     end if
7:     Broadcast(Out_j^t, p , ROW)
8:     if i = j then
9:         In_i^{t+1} ← Out_j^t
10:    end if
11:    Broadcast(In_i^{t+1}, i, COL)
12: end procedure
```

Figure 5. Global Frontier contraction and communication algorithm

```
1: procedure UPDATELEVELS(Out_j^t, t, level)
2:     for all v ∈ Out_j^t in parallel do
3:         level[v] ← t
4:     end for
5: end procedure
```

Figure 6. Update levels

```
1: procedure UPDATEPREDS(Assigned_ij, Preds_ij, level)
2:     for all v ∈ Assigned_ij in parallel do
3:         Pred[v] ← −1
4:         for i ← ColOff[v], ColOff[v + 1] do
5:             if level[v] == level[RowIdx[i]] + 1 then
6:                 Pred[v] ← RowIdx[i]
7:             end if
8:         end for
9:     end for
10: end procedure
```

Figure 7. Predecessor update

for finding the $Prefix_{ij}^t$, which is the bitmap obtained by performing Bitwise-OR Exclusive Scan operation on Out_{ij}^t across each row j as shown in line 2 of Figure 5. There are several tree-based algorithms for parallel prefix sum. Since the bitmap union operation is very fast on the GPU, the complexity of the parallel scan is less important than the number of communication steps. Therefore, we use an algorithm having the fewest communication steps ($\log p$) [14] even though it has a lower work efficiency and does $p \log p$ bitmap unions compared to p bitmap unions for work efficient algorithms.

At each iteration t, $Assigned_{ij}$ is computed as shown in line 3 of Figure 5. In line 7, the contracted Out_j^t bitmap containing the vertices discovered in iteration t is broadcast back to the GPUs of the row R_j from the last GPU in the row. This is done to update the GPUs with the visited vertices so that they do not produce a frontier containing those vertices in the future iterations. It is also necessary for updating the levels of the vertices. The Out_j^t becomes the In_j^{t+1}. Since both Out_j^t and In_i^{t+1} are the same for the GPUs in the diagonal ($i = j$), in line 9, we copy Out_j^t to In_i^{t+1} in the diagonal GPUs and then broadcast across the columns C_i using *MPI_Broadcast* in line 11. Communicators were setup along every row and column during initialization to facilitate these parallel scan and broadcast operations.

During each BFS iteration, we must test for termination of the algorithm by checking if the global frontier size is zero as in line 6 of Figure 3. We currently test the global frontier using a *MPI_Allreduce* operation after the GPU local edge expansion and before the global frontier contraction phase. However, an unexplored optimization would overlap the termination check with the global frontier contraction and communication phase in order to hide its component in the total time.

D. Computing the Predecessors

Unlike the Blue Gene/Q [7], we do not use remote messages to update the vertex state (the predecessor and/or level) during traversal. Instead, each GPU stores the levels associated with vertices in both In_j^t frontier and Out_j^t frontier (See Figure 6 for the update level algorithm). This

information is locally known in each iteration. The level for the In_j^t frontier is available at the start of each iteration. The level for the vertices in the Out_j^t frontier is known to each GPU in a given row at the end of the iteration.

Once the BFS traversal terminates, the predecessors are computed in parallel by all GPUs without any further communications. The algorithm for the predecessor computation is available in Figure 7, where $ColOff$ and $RowIdx$ are the CSC index. For each GPU, all vertices discovered by that GPU (that is, those vertices that correspond to a set bit in $Assigned_{ij}$ bitmap) are resolved to a source vertex at the previous level on that GPU. Specifically, each vertex v_i first looks up in the level array to find its own level l_i. Next, it accesses its adjacency list and find the levels of its neighbors. For neighbors with level l_i-1, this vertex writes the neighbor identifier in a temporary array. Otherwise, it writes -1 in the array. Finally, we do a reduction on the temporary array to find the maximum and set it as the predecessor of the vertex.

This procedure is entirely local to each GPU. When it terminates, the GPUs have discovered a valid predecessor tree. On each GPU, the elements of the predecessor array will either be a special value (-1) indicating that the predecessor was not discovered by that GPU or the vertex identifier of a valid predecessor. The local predecessor arrays can then be reduced to a single global array on a single GPU.

IV. Evaluation

Two fundamental measures of scalability for a parallel code running on a parallel computer are strong and weak scaling. As strong scaling has a constant problem size and increasing processor or core count, it measures how well a parallel code can solve a fixed size problem as the size of the parallel computer is increased. In contrast, weak scaling has a fixed problem size per core per processor and so measures the ability of a parallel machine and a parallel code to solve larger versions of the same problem in the same amount of time.

To evaluate our approach, we conduct an empirical study of the strong and weak scaling behavior on a GPU compute cluster with up to 64 GPUs. This scalability study provides empirical evidence that GPU compute clusters can be successfully applied to very large parallel graph problems. As noted in [7], weak scaling is not expected for this design. This is because the frontier message size grows with the number of vertices while the work per GPU is roughly constant given a fixed number of edges per GPU.

A. Test Environment

The test environment is a GPU cluster hosted at the Scientific Computing and Imaging (SCI) Institute in Utah with 32 nodes and 64 GPUs. Each node of the cluster has two Intel Xeon CPUs, 64 GB of RAM, and two NVIDIA K20 GPUs. Each GPU is equipped with 5GB DDR5 memory and has a peak single precision performance of 3.52 Tflops, a peak memory bandwidth of 208 GB/sec and supports PCIe Gen 2. ECC was enabled for all runs. The 32 nodes of the cluster are connected with the Mellanox InfiniBand SX6025 switches that provide up to thirty-six 56Gb/s full bi-directional bandwidth per port. The nodes were configured using CentOS 6.5. The MPI distribution is MVAPICH2-GDR and the CUDA version is 5.5. This configuration allows us to use GPUDirect, thus eliminating copying of data between the GPU and the CPU. Instead, MPI requests are managed using the PCIe connections between the GPU and the InfiniBand port paired with each GPU. This substantially decreases the latency and increases the throughput of MPI messages. The CPUs and the CPU memory are not relied on in these experiments except to coordinate the activity on the GPUs. All data is stored in the GPU memory, and all computation is performed on the GPUs.

B. Graph 500 Graph Generator

In order to provide comparable benchmark results we use the Graph 500 [15] generator, which is based on a Kronecker generator similar to the Recursive MATrix (R-MAT) scale-free graph generation algorithm described by [24]. The graph has 2^{SCALE} vertices based on inputs of *SCALE* and *EDGEFACTOR*, which defined the ratio of the number of edges of the graph to the number of vertices. The *EDGEFACTOR* was 16 for all generated graphs. The adjacency matrix data structure for the graph edges is then produced and is recursively partitioned into four equal sized partitions and edges are added to these partitions one at a time; each edge chooses one of the four partitions with probabilities $A = 0.57$, $B = 0.19$, $C = 0.19$, $D = 1-(A+B+C) = 0.05$. For each condition, we randomly selected 5 vertices with non-zero adjacency lists (excluding self-loops). The average of those 5 runs is reported. The observed variability among those runs was very small. All experimental runs visit the vast majority of the edges in the graph. This is because the generated scale-free graphs have a very large connected component that encompasses most of the vertices.

C. Validation

Each GPU extracts the predecessors for those vertices that are marked in its *Assigned* bitmap. After conversion to global vertex identifiers and adjustment by $+1$, the existence of a predecessor in the predecessor array is denoted by *PredecessorVertex*$+1$ and the rest of the vertices in the precesssor array are assigned 0. We then perform a *MPI_Reduce* across the rows R_i to the nodes in the column C_1 to find all the predecessors of the vertices of that row. Then we perform a *MPI_Allgather* across nodes in the column C_1 to build a lookup table for the predecessors.

To validate the results, the predecessor information is used to test if the BFS tree has cycles. In the nodes in column C_1, we create a temporary list of global vertices that were in the partition of vertices for the row R_i during the BFS and replace each visited vertex with its predecessor in a loop until its predecessor is the source vertex of BFS. If the number of iterations of the loop reaches the number of vertices in the graph, it means that there are loops in the BFS graph, and validation fails.

Next, the directed edges of the graph are traversed to verify that each predecessor vertex is in the BFS tree. For each edge, we also verify that the destination vertex is in the BFS tree at a level not greater than *x+1*, where *x* is the level assigned to the source vertex.

V. RESULTS

Our initial scalability runs demonstrated the underlying promise of our approach. However, they also illustrated the fundamental challenge in the algorithm used for the global frontier aggregation. This step aggregates the local frontiers from the GPUs and forms the global frontier, which is represented as a bitmap. We used the wave strategy described the in paper [7], but our analysis shows that the impact of this algorithm on both our results, and on those of the Blue Gene/Q, is considerable.

In an $p \times p$ array of processors, the wave algorithm does p communication steps per frontier operation followed by a broadcast across $2p$ processors. As we increase p, the decrease in the message size is not sufficient to offset the p communication steps and the overall costs do not decrease sufficiently fast across each of these stages to ensure that the calculation scales. The fundamental issue lies in the p communication steps per frontier operation, which is sequential across the p processors on a given row of the $p \times p$ processor array, but parallel across the different rows. This p cost is one of the primary effects that limits the strong scaling efficiency. Our solution adopts an approach based on pairwise exchanges that has a communication cost of $\log p$ compared to the original cost of p. This change in the communication pattern provided a 30% gain in TEPS and improved the strong scaling efficiency from 44% to 48% on the same graphs and hardware. (The Blue Gene/Q has a strong scaling efficiency of 44% as measured on a cluster sizes of 512, 1024, 2048, and 4096 nodes.)

The main challenge in weak scaling is ensuring that the communications costs do not grow appreciably in a manner that is proportional to the overall problem size. In the design considered here, the communications costs is theoretically $C_{comm} = \epsilon S \log p$, where ϵ is a constant, and S denotes the bitmap size. Therefore, when the number of GPUs in a weak scaling study increases by a factor of 4, the bitmap size S grows by a factor of 2, and communications costs grow by a factor of $2\frac{\log 2p}{\log p}$. This is illustrated by Figure 8, which breaks down the costs in a central iteration of the BFS traversal for the weak scaling study. Figure 8 shows that the communications costs grow significantly while the computation costs per GPU remain very nearly constant. In order to have weak scaling, the message size would need to be constant, which it is not. Thus, while we do not achieve weak scaling in the normal sense, we would not expect to do so with this design. The weak scaling results in Table II are consistent with the Blue Gene/Q [7].

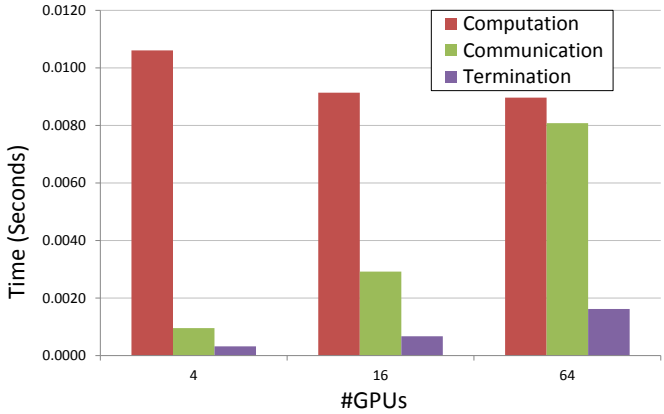

Figure 8. Communication and computation costs during BFS traversal for iteration 2 (weak scaling)

A. Strong and Weak Scalability Tests

For strong scaling study, we generate a scale 25 graph, which has 2^{25} vertices and 2^{30} directed edges. We then evaluate our BFS implementation for this graph on 16, 25, 36, 49, and 64 GPUs, respectively. For the strong scaling study we have the same data for each condition. Therefore, we use the same set of randomly selected starting vertices for all conditions. This makes it possible to directly compare the performance for the same graph traversal on different numbers of GPUs. The average performance is shown in Table I and Figure 9. As noted above, our implementation has a strong scaling of 48% compared to the Blue Gene/Q with a strong scaling of 44%. This is attributable to the log(p) communication pattern for propagate.

GPUs	GTEPS	BFS Time (s)
16	15.2	0.071
25	18.2	0.059
36	20.5	0.053
49	21.8	0.049
64	22.7	0.047

Table I
STRONG SCALING RESULTS

For weak scaling study, we generated a series of graphs of different scales. Each GPU has 2^{26} directed edges. In order to have the same number of edges per GPU, we used a power of 4 for the number of GPUs.

GPUs	Scale	Vertices	Directed Edges	BFS Time (s)	GTEPS
1	21	2,097,152	67,108,864	0.0254	2.5
4	23	8,388,608	268,435,456	0.0429	6.3
16	25	33,554,432	1,073,741,824	0.0715	15.0
64	27	134,217,728	4,294,967,296	0.1478	29.1

Table II
WEAK SCALING RESULTS

Figure 9. Strong scaling results

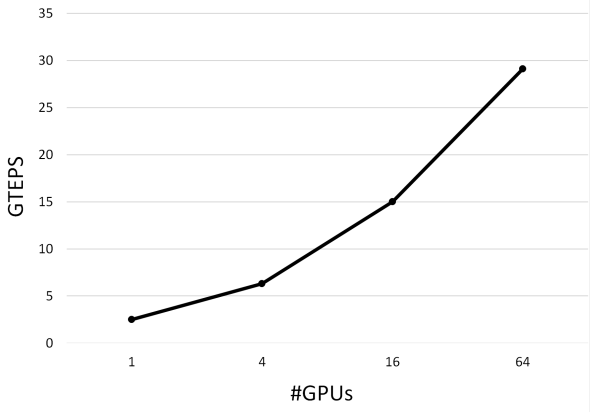

Figure 10. Weak scaling results

Performance for weak scaling is presented in Table II and Figure 10. Unlike the strong scaling study, a different starting vertex was chosen for each condition because a different scale graph is used in each condition for a weak scaling study.

B. Performance at maximum scale

The largest graph on which we report here is a scale 27 ($2^{27} \approx 134$ million vertices and $2^{32} \approx 4.3$ billion directed edges). On the 64 GPU cluster, this graph has 2^{26} (67,108,864) directed edges per GPU. While we can fit larger graphs onto the GPU, the next larger scale graph (scale 28) would have twice as many directed edges per GPU which would exceed the on device RAM (5GB) for the K20 GPU.

The full edge-to-edge traversal of this 4.3 billion edge graph took an average of 0.14784 seconds and traversed 4,294,925,646 (4.3B) edges for an effective average traversal rate of 29.1 GTEPS. Note that 41,650 edges were not visited because they were not part of the connected component for the starting vertices.

VI. Conclusions and Future Work

The fundamental understanding that we now have of the scalability properties of the algorithm is perhaps as important as the software implementation and results obtained. It is now possible for us to go beyond present scaling by considering alternative approaches such as those described above, optimizations such as the topology folding described in [7] and graph compression [28] [29], and using pipelined processing with multiple graph partitions per GPU to overlap the computation with the communication in order to hide the delay effects of the communication. Moreover, we have shown that there is no fundamental roadblock to scalability at levels that meets the requirements of many users.

Since our submission, two works have recently emerged that are related to our research. Ueno [30] (December 2013) describes an approach using CPUs to coordinate communications and GPUs to perform the edge expansion and reports scaling up to 4096 GPUs on the Japanese TSUBAME 2.0 supercomputer. Bisson [31] appeared as a pre-print after our submission and describes a pure GPU approach on a Cray XC30 supercomputer with scaling up to 4096 nodes. We will compare to these results in a future publication.

While we have shown that it is possible to achieve a scalable solution for challenging BFS graph problems on multiple GPU nodes, the next decade presents enormous challenges when it comes to developing software that is portable across different architectures. The anticipated rapid increase in the number of cores available together with the desire to reduce the power consumption and concerns about resilience at large scales is introducing a rapid phase of innovation in both hardware and software. There are many different competing design paradigms ranging from low power chips, to GPU designs, to the Intel Xeon Phi. For example, GPU architectures themselves are changing very quickly with the possibility of increased compute power, vertically stacked memory and much improved GPU-to-GPU communications, and continued gains in FLOPS/watt.

One solution is to use a layered software approach that adapts to such rapid evolution in hardware and systems. In such an approach, the aim is to automatically translate and inject user specific algorithms into data-parallel kernels, such as the compute intensive parts of the GPU code presently running today. This code is then executed using a distributed runtime system (similar to Uintah) that makes use of all available node-to-node (e.g. GPU to GPU) optimized communications.

The important aspect of such a design is that it involves layers of abstraction that are sufficiently rich to encompass future architecture developments. Our future challenge will be to combine this layered approach with achievable performance for challenging applications such as that considered here.

Acknowledgment

This work was (partially) funded by the DARPA XDATA program under AFRL Contract #FA8750-13-C-0002. This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract

No. D14PC00029. The authors would like to thank Dr. White, NVIDIA, and the MVAPICH group at Ohio State University for their support of this work.

REFERENCES

[1] Zhisong Fu, Michael Personick and Bryan Thompson MapGraph: A High Level API for Fast Development of High Performance Graph Analytics on GPUs in Proceedings of Grades2014 Workshop at SIGMOD/PODS Conference Snowbird Utah, 2014.

[2] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: distributed graph-parallel computation on natural graphs. In Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation (OSDI'12). USENIX Association, Berkeley, CA, USA, 17-30.

[3] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. Hellerstein. Distributed GraphLab: A framework for machine learning and data mining in the cloud. Proceedings of the VLDB Endowment 5, no. 8 (2012): 716-727.

[4] K. Devine, E. Boman, R. Heaphy, R. Bisseling, and U. Catalyurek. "Parallel hypergraph partitioning for scientific computing." In Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International, pp. 10-pp. IEEE, 2006.

[5] G. Karypis and V. Kumar, Parallel Multilevel k-way Partitioning Scheme for Irregular Graphs Siam Review 41, no. 2 (1999): 278-300.

[6] B. Vastenhouw, and R. Bisseling. A Two-Dimensional Data Distribution Method for Parallel Sparse Matrix-Vector Multiplication. SIAM Review, Vol. 47, No. 1 : pp. 67-95, 2005.

[7] Fabio Checconi, Fabrizio Petrini, Jeremiah Willcock, Andrew Lumsdaine, Anamitra Roy Choudhury, and Yogish Sabharwal. Breaking the speed and scalability barriers for graph exploration on distributed-memory machines. In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '12). IEEE Computer Society Press, Los Alamitos, CA, USA, , Article 13, 12 pages.

[8] U. Catalyurek and C. Aykanat. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. IEEE Trans. Parallel Dist. Systems, 10(7):673-693, 1999.

[9] U. V. Catalyurek and C. Aykanat. PaToH: A Multilevel Hypergraph Partitioning Tool, Version 3.0. Bilkent University, Department of Computer Engineering, Ankara, 06533 Turkey, 1999.

[10] A. Kyrola, G. Blelloch, and C Guestrin. GraphChi: Large-scale graph computation on just a PC. In Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI), pp. 31-46. 2012.

[11] R. Chen, M. Yang, X. Weng, B. Choi, B. He, and X. Li. Improving large graph processing on partitioned graphs in the cloud. In Proc. of the Third ACM Symposium on Cloud Computing (SoCC '12), 2012.

[12] J. Berry, B. Hendrickson, S. Kahan, P. Konecny, Software and Algorithms for Graph Queries on Multithreaded Architectures. IPDPS, pp.495, 2007 IEEE International Parallel and Distributed Processing Symposium, 2007.

[13] D. Bader and K. Madduri, Snap, small-world network analysis and partitioning: An open-source parallel graph framework for the exploration of large-scale networks. in IPDPS, pp. 12., 2008

[14] W. Hillis and G. Steele Jr. Data parallel algorithms. Communications of the ACM 29.12 (1986): 1170-1183.

[15] Graph 500 Benchmark. http://graph500.org

[16] Merrill, Duane, Garland, Michael and Grimshaw, Andrew, Scalable GPU Graph Traversal, Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2012, isbn 978-1-4503-1160-1, New Orleans, Louisiana, USA,117–128, ACM, NY, USA

[17] R. Pearce, M. Gokhale, and N. Amato. Multithreaded asynchronous graph traversal for in-memory and semi-external memory. In Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1-11. IEEE Computer Society, 2010.

[18] A. Buluç, and J. Gilbert. The Combinatorial BLAS: Design, implementation, and applications. International Journal of High Performance Computing Applications 25, no. 4 (2011): 496-509.

[19] A. Lugowski, D. Alber, A. Buluç, J. Gilbert, S. Reinhardt, Y. Teng, and A. Waranis. A Flexible Open-Source Toolbox for Scalable Complex Graph Analysis. In SDM, vol. 12, pp. 930-941. 2012.

[20] A. Gharaibeh, L. Costa, E. Santos-Neto, and M. Ripeanu. A yoke of oxen and a thousand chickens for heavy lifting graph processing. In Proceedings of the 21st international conference on Parallel architectures and compilation techniques, pp. 345-354. ACM, 2012.

[21] Jianlong Zhong and Bingsheng He, Medusa: Simplified Graph Processing on GPUs, IEEE Transactions on Parallel and Distributed Systems,99,2013, IEEE Computer Society,Los Alamitos, CA, USA

[22] Q. Meng, M. Berzins. Scalable Large-scale Fluid-structure Interaction Solvers in the Uintah Framework via Hybrid Task-based Parallelism Algorithms, Concurrency and Computation, 2014.

[23] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. Hellerstein. Graphlab: A new framework for parallel machine learning. arXiv preprint arXiv:1006.4990 (2010).

[24] D. Chakrabarti, Y. Zhan, and C. Faloutsos, R-MAT: A Recursive Model for Graph Mining, in SIAM Data Mining 2004, Orlando, Florida, USA

[25] V. Agarwal, F. Petrini, D. Pasetto, and D. Bader. Scalable graph exploration on multicore processors. In Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1-11. IEEE Computer Society, 2010.

[26] Chakrabarti, Deepayan, and Christos Faloutsos. "Graph mining: Laws, generators, and algorithms." ACM Computing Surveys (CSUR) 38.1 (2006): 2.

[27] Seshadhri, C., Ali Pinar, and Tamara G. Kolda. "An in-depth study of stochastic Kronecker graphs." Data Mining (ICDM), 2011 IEEE 11th International Conference on. IEEE, 2011.

[28] Blandford, D., Blelloch, G., and Kash, I. "Compact Representations of Separable Graphs." in Proc. ACM-SIAM Symposium on Discrete Algorithms, 2003.

[29] Kaczmarski, K., Przymus, P., and Paweł Rzążewski, P. "Improving High-Performance GPU Graph Traversal with Compression." GPUs In Databases, ADBIS workshop on., Springer, 2014.

[30] Ueno, Koji, and Toyotaro Suzumura. "Parallel distributed breadth first search on GPU." High Performance Computing (HiPC), 2013 20th International Conference on. IEEE, Dec. 2013.

[31] Bisson, Mauro, Massimo Bernaschi, and Enrico Mastrostefano. "Parallel Distributed Breadth First Search on the Kepler Architecture." arXiv preprint arXiv:1408.1605 (2014).