A Portable, High-Level Graph Analytics Framework Targeting Distributed, Heterogeneous Systems

Robert Searles*, Stephen Herbein*, Sunita Chandrasekaran
University of Delaware
Department of Computer and Information Sciences
Newark, DE
{rsearles, sherbein, schandra}@udel.edu

***Abstract*—As the HPC and Big Data communities continue to converge, heterogeneous and distributed systems are becoming commonplace. In order to take advantage of the immense computing power of these systems, distributing data efficiently and leveraging specialized hardware (e.g. accelerators) is critical. MapReduce is a popular paradigm that provides automatic data distribution to the programmer. CUDA and OpenCL are some of the most popular frameworks for leveraging accelerators (specifically GPUs) on heterogeneous systems.**

In this paper, we develop a portable, high-level framework using a popular MapReduce framework, Apache Spark, in conjunction with CUDA and OpenCL in order to simultaneously take advantage of automatic data distribution and specialized hardware present on each node of our HPC systems. Using our framework, we accelerate two real-world, compute and data intensive, graph analytics applications: a function call graph similarity application and a triangle enumeration subroutine. We demonstrate linear scalability on the call graph similarity application as well as an exploration of the triangle enumeration parameter space. We show that our method yields a portable solution that can be used to leverage almost any legacy, current, or next-generation HPC or cloud-based system.

I. Introduction

The use of high performance computing (HPC) resources and cloud-based systems is widespread and steadily increasing. Today's HPC systems are comprised of many nodes, each containing heterogeneous computing resources. In order to effectively tap into the power of such machines, programmers need to structure their applications so that they can distribute tasks across these nodes, while also optimizing for single-node performance in order to get every last ounce of performance that each node has to offer. This paper aims to tackle these challenges by bridging the gap between Big Data and HPC technologies. Our high-level framework uses Big Data's MapReduce framework, Apache Spark [1], to distribute tasks across nodes in our cluster. We combine this with two high-level HPC frameworks, CUDA and OpenCL [2], [3], to make use of the accelerators present on each machine in our cluster in order to achieve optimal performance. This combination of technologies allows our applications to scale to distributed, heterogeneous systems of any size. In addition, our framework enables applications to be both backwards compatible and forwards compatible (i.e., they can efficiently run on any legacy, current, or next-generation hardware).

*Robert Searles and Stephen Herbein contributed equally to this work.

WACCPD16; Salt Lake City, Utah, USA; November 2016
978-1-5090-6152-5/16/$31.00 2016 IEEE

Our approach is fundamentally different from the widely used MPI+X techniques used to program HPC systems (e.g., MPI+CUDA and MPI+OpenACC). MPI allows the programmer to explicitly move data, which in some cases results in higher efficiency, but in many cases it can be cumbersome, since all data movement and memory management is handled manually [4]. Additionally, MPI code is not natively fault-tolerant. The decision to use Apache Spark allows us to automatically distribute data across nodes in our system, providing us with a portable, fault-tolerant solution that requires much less development overhead. This allows the programmer to focus the majority of their efforts on parallelization and optimization of their algorithms. To that end, we combine Spark with the popular GPU computing frameworks CUDA and OpenCL. Spark exclusively handles the distribution of data across nodes, and it performs the computation directly when no GPU is present. When a GPU is present, node-local computations are performed using CUDA or OpenCL kernels. We demonstrate this technique on two real-world applications: the Fast Subtree Kernel (FSK) and triangle enumeration [5].

FSK is a modified graph kernel that takes directed trees (graphs with no cycles) as input and returns a normalized measure of their overall similarity. This compute-bound kernel can be used in a variety of applications, but for the purposes of this paper, it will be used for program characterization. In this paper, we use FSK to examine the pairwise similarities of call graphs generated from decompiled binary applications. The resulting kernel matrix can be used as an input to a Support Vector Machine (SVM) to generate a model that will detect malicious behavior in applications (malware), as well as classify detected malware by type [6]. We call these varying types of malicious programs "malware families".

Triangle enumeration is a data-bound graph operation that counts the number of triangles (i.e., a cycle on three vertices) in a graph. Although by itself triangle enumeration is a simple operation, it is used as a subroutine in many larger graph analyses such as spam detection, community detection, and web link recommendation. Accelerating triangle enumeration leads to a speed up in these larger graph analyses, much like accelerating BLAS operations speeds up traditional scientific applications. In this paper, we enumerate the triangles in Erdős-Rényi (ER) random graphs [7].

The major contributions of this paper are as follows:

- We propose a high-level framework for combining MapReduce and accelerators to run applications portably and scalably on heterogeneous HPC and cloud-based systems.
- We adapt a compute-bound HPC application, FSK, and demonstrate linear scalability with our solution.
- We adapt a data-bound Big Data application, triangle enumeration, and demonstrate performance optimization through parameter tuning.

The rest of the paper is organized as follows: Section II provides a high-level overview of the applications we will use to demonstrate the effectiveness of our solution on both compute-bound and data-bound applications. Section III describes the methodology used to adapt these applications to run on HPC and cloud-based systems using our proposed solution. Section IV presents our results. We discuss related work in Section V, and we conclude in Section VI.

II. Application Descriptions

Typically, the HPC and Big Data communities each have their own types of applications: compute-bound applications and data-bound applications, respectively. As these two communities continue to converge, we see a growing number of applications containing elements of both. While some framework are designed with one particular application category in mind, our framework was designed to support applications in either category as well as applications that share elements from both. To demonstrate the robustness of our high-level framework, we selected one compute-bound HPC application and one data-bound Big Data application. We demonstrate the different techniques and parameters used to adapt and tune both types of applications on heterogeneous systems effectively. The following section describes the two real-world applications used in this paper.

A. Fast Subtree Kernel

Many real-world systems and problems can be modeled as graphs, and these graphs can be used for a variety of applications, such as complex network analysis, data mining, and even cybersecurity. For our compute-bound HPC application, we examine an existing graph kernel, the Fast Subtree Kernel (FSK). FSK is a specialized graph kernel that takes directed trees (graphs with no cycles) as input and tells us how similar they are. We will use FSK to characterize potentially malicious binary applications by representing applications as graphs, pruning those graphs into trees by removing any cycles found within them, and constructing a matrix that represents the pairwise similarity of all the binary applications in our dataset. This resulting similarity matrix can be used as input to an SVM, which will generate a classification model from our matrix. This model can then be used to classify applications as malicious (belonging to one or more categories of malware) or benign.

FSK takes encoded trees as input. An encoded tree is simply a tree that explicitly represents all possible subtrees in addition to single nodes. In our case, we are starting with function call graphs generated from a tool called Radare2 [8], which decompiles binaries and returns a call graph representing the function call hierarchy of the application in question. In these graphs, nodes represent functions and edges represent function calls. For example, if function A calls function B, there will be an edge going from A to B. In addition to these relationships, we represent nodes as feature vectors used to count the types of instructions present in a given function. Each entry in the feature vector represents a type of instruction, and the value at each entry is simply a count of the number of times that instruction is used within the given function. This information allows us to examine individual functions in an application, while preserving the structural characteristics of that application. However, FSK examines subtrees in addition to single nodes, so we must first encode these trees by constructing representations of all the subtrees found in them. In order to do this, we simply combine the feature vectors of the nodes that make up a particular subtree by performing an element-wise sum on the feature vectors. Our graphs are then represented as a list of these feature vectors. Once this has been done, we can start using FSK to construct our similarity matrix.

The creation of the similarity matrix has two main components, both of which are embarrassingly parallel. The first component is the pairwise comparisons of all the graphs in a dataset. Since these comparisons are independent of each other, they can be done in parallel. In addition to this coarse-grained component, the examination of a given pair of graphs (A and B respectively) can be performed in parallel as well. All feature vectors of graph A must be compared with all feature vectors of graph B, and vice-versa. The comparisons in this fine-grained portion of FSK measure the distance between feature vectors and considers them similar if their distance is below a predetermined threshold, $delta$. Distance between two feature vectors (m and n) of length L is calculated as follows:

$$distance = \frac{\sum_{i=0}^{L-1} |m_i - n_i|}{max(max(m,n),1)} \tag{1}$$

This calculated distance is then divided by the feature vector length in order to normalize its value to lie between 0 and 1 as shown below.

$$normalized\_distance = \frac{distance}{L} \tag{2}$$

We keep a count of the total number of similar feature vectors (sim_count) in a given pair of graphs (A and B), and we calculate their overall similarity using the following formula:

$$\frac{sim\_count}{len(A) + len(B)} \tag{3}$$

The resulting similarity values are then stored in the similarity matrix we construct, which can be analyzed as-is or fed to an SVM to generate a classification model. In Subsection III-A, we describe how we implement FSK using our high-level framework.

B. Triangle Enumeration

For our data-bound Big Data application, we examine an existing graph operation, triangle enumeration. Triangle enumeration is the process of counting the number of triangles in a graph. For this paper, we consider only undirected graphs, but the techniques we describe are easily extensible to support directed graphs. A triangle is a set of three vertices where each pair of vertices share an edge (i.e., a complete subgraph on 3 vertices or a cycle on 3 vertices). Figure 1 shows a graph containing 2 triangles (which are highlighted).

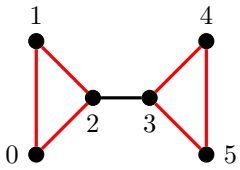


Fig. 1: This graph contains 2 triangles (highlighted in red).

Triangle enumeration is used in many ways. It is used in calculating the cluster coefficient and transitivity ratio of a graph; both of which are a measure of how clustered together nodes in a graph are. It is also used as a subroutine in graph algorithms for spam detection, community detection, and web link recommendation. Its prevalence in so many metrics and algorithms demonstrates its relevance and usefulness in our study. Accelerating triangle enumeration is of key importance to accelerating these larger graph algorithms, much like accelerating BLAS operations is important to accelerating scientific simulations.

There are two existing parallel triangle enumeration methods. The first method involves computing A^3, where A is the adjacency matrix of the graph, as described in [9]. Each cell (i, j) within A^3 represents the number of possible walks of length three from node i to node j. Thus, summing the values along the diagonal (i.e., the trace of the matrix) gives the number of triangles in the graph (after we divide by six to eliminate the overcounting caused by the symmetry of the triangles). This method is parallelized by performing the computation of A^3 in parallel, typically with the use of either a CPU or GPU BLAS library. For most graphs, this methods performs well, but for small or sparse graphs, the overhead of transferring to and from the GPU outweighs the benefits of parallelism. In addition, this method is limited by the amount of memory on a single node or GPU.

The second method involves uses the edge list of the graph to generate a list of "angles" (i.e., a triangle minus one edge). The angle list is then joined with the graph edge list to form a list of triangles. This method, as described by Jonathan Cohen [5], fits naturally into the MapReduce paradigm and has traditionally been implemented with Hadoop. The Hadoop framework allows this algorithm to span multiple compute nodes easily, but it suffers from a high data movement overhead. In total, the algorithm requires several iterations of maps and reduces. For each map and reduce, Hadoop must not only shuffle key-value pairs across the network, but it must also write to disk the output of each map and reduce operation.

We adapt triangle enumeration for heterogeneous hardware by combining these two existing parallel triangle enumeration methods to create a new hybrid method. The new method consists of breaking the graph down into multiple subgraphs, as demonstrated in Figure 2. The triangles in each subgraph are then computed using the first method, and the triangles that span the subgraphs are counted with the second method. This hybrid combination of the existing methods avoids their downsides and maximizes their benefits. The first method, computing A^3, only runs on subgraphs, which means that it will not overflow the memory of the GPUs on the cluster. This allows the GPUs to be utilized on graphs that are larger than can fit on a single node. The second method, joining of the angle and edge lists, no longer has to detect every triangle in the entire graph, but instead only has to detect the triangles that span subgraphs. This reduces the data movement overhead by reducing the number of edges and angles that are emitted and shuffled across the network by the MapReduce framework. In Subsection III-B, we describe how we implement this new hybrid method.

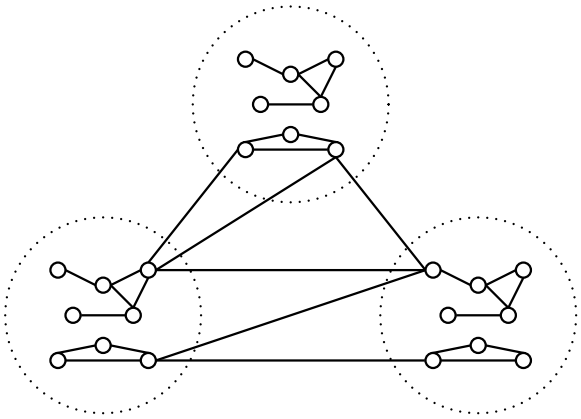


Fig. 2: An example graph that is broken down into subgraphs by our hybrid triangle enumeration method. The triangles in the subgraphs are counted on the CPU or GPU with a BLAS library while the triangles that span the subgraphs are counted with a MapReduce framework.

III. Methodology

In this section, we describe the general strategy behind our portable, high-level graph analytics framework, which enables both applications to run on heterogeneous HPC and cloud-based systems. Generally speaking, our framework uses Apache Spark [10] for automatic data/task distribution and CUDA/OpenCL ("X") for local task computation. Figure 3 shows a high-level depiction of this framework.

More specifically, we use Apache Spark for all inter-node operations (i.e. data movement, task distribution, and global computation). Such operations include Spark's built-in functions for broadcasting, reducing, joining, streaming, etc. For

intra-node operations, our framework utilizes computational libraries (e.g. ScikitCUDA, PyOpenCL, SciPy, and NumPy) in conjunction with user-written functions.

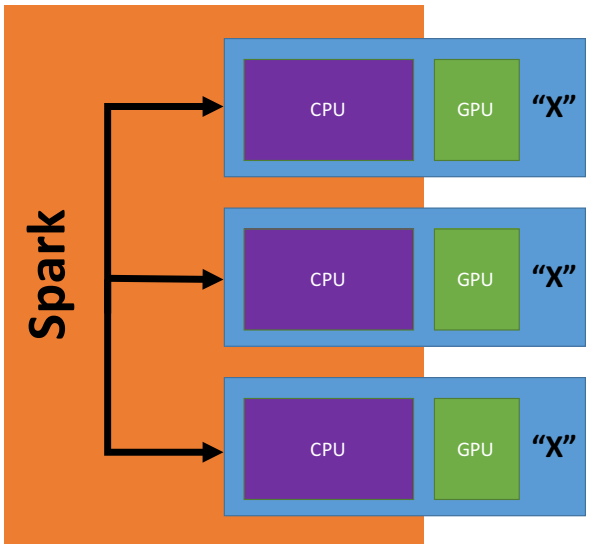


Fig. 3: A high-level depiction of our proposed framework, which uses Spark for data/task distribution in conjunction with a local computational framework (X) on each node.

Figure 4 shows our two applications (FSK and Triangle Enumeration) written using our framework. The orange text represents inter-node communication and computation performed using Spark. The blue text represents the intra-node computation performed by user-written functions that utilize PyOpenCL and ScikitCUDA. Note that each application requires 10 or fewer lines of additional code in order to utilize Spark and run on distributed systems. This small amount of programmer overhead allows for massive gains in parallelism by distributing tasks/data across all the nodes in a distributed system. Once this distribution is complete, our framework can utilize any available accelerators local to each node, via CUDA/OpenCL, to add a level of fine-grained parallelism to our framework, in addition to the coarse-grained parallelism provided by Spark.

Unlike most traditional HPC applications, we decided to forego the use of MPI and instead use Spark for two reasons.

- First, Spark automatically distributes the data/tasks with a single function call (i.e., sc.parallelize). MPI requires manual distribution by the programmer.
- Second, Spark is natively fault-tolerant. If a particular distributed operation fails, Spark automatically detects the failure and recovers. Fault-tolerance for MPI is not natively supported. While it is possible via extensions to the library, these extensions induce additional programmer overhead [11], [12].

Unlike many traditional Big Data applications, we chose to distribute our data/tasks with Spark rather than Hadoop for three reasons.

- First, Spark is a memory-based MapReduce framework. Unlike Hadoop, it is not required to write each intermediate state out to disk.
- Second, in addition to `map` and `reduce`, Spark supports a wide-range of high-level functions such as `join`, `union`, `intersection`, and `cartesian`. This greatly reduces the programming overhead for many common distributed operations.
- Third, Spark supports applications written in Python. Python has excellent support for calling out to C/C++ libraries. This greatly simplifies the use of BLAS and GPU libraries for our local task computations.

The following subsections elaborate on the specific implementation details of both applications.

A. Fast Subtree Kernel - Task Distribution on a Compute-Bound Application

As mentioned in Subsection II-A, the Fast Subtree Kernel (FSK) has two major components: the pairwise comparison of all graphs in a given dataset (coarse-grained parallelism) and the comparison of all feature vectors within a given pair of graphs (fine-grained parallelism). As stated in Section III, Spark is our chosen solution for automatic task distribution. To this end, we utilize Spark to distribute the pairwise graph comparison tasks. Once these tasks have been delegated to nodes, we can run FSK locally on each node and compare the encoded subtrees of each pair of graphs delegated to that particular node. Note that graphs are not loaded from the disk into memory until they are ready to be analyzed. In most real applications, the dataset would be too large for all the graphs to fit in memory. For example, applications in the fields of climate change (sea level rise, carbon footprint), cancer illness, national nuclear security (real–time evaluation of urban nuclear detonation), Nuclear Physics (dark matter), life sciences (biofuels), medical modeling, and design of future drugs for rapidly changing and evolving viruses all deal with massive amounts of data. As the data sets examined by these types of applications grow exponentially, computers today do not offer enough resources to gather, calculate, analyze, compute and process them by themselves. Instead, it becomes necessary to utilize multiple machines simultaneously in order to process such massive amounts of data.

Much like these applications, FSK faces the challenge of performing operations across massive datasets. More specifically, FSK has to perform a pairwise similarity calculation of every pair of graphs in the dataset it is given. Because of this, we simply create a list of comparison tasks to be performed, and we use Spark to distribute these tasks across our nodes. When a node receives a set of tasks, it loads the appropriate graphs for each task from disk, executes FSK on the given pair, and stores the output in a list that is collected by the master Spark process and used to construct the resulting similarity matrix.

It is worth noting that not all tasks in this system are computationally comparable. Executing a set of pairwise comparison tasks involving many large graphs is much more expensive

```
conf = SparkConf()
sc = SparkContext(conf=conf)
comp_rdd = sc.parallelize(graph_comparisons, numSlices=1024)
broadcast_graph_list = sc.broadcast(encoded_graph_list)
comp_sim = comp_rdd.map(MapGraphSimilarity)
collected_sim = comp_sim.collect()
```

(a) FSK Spark code

```
conf = SparkConf()
sc = SparkContext(conf=conf)
G_spark = sc.parallelize(G, numSlices=args.num_partitions)
global_angles = G_spark.mapPartitions(count_triangles)
G_zero = G_spark.flatMap(MakeUpperEdgeListSpark)
global_tris = global_angles.union(G_zero)
global_tris = global_tris.combineByKey(Combiner, MergeValue, MergeCombiners)
global_tris = global_tris.flatMap(EmitTrianglesSpark)
global_count = global_tris.count()
```

(b) Triangle Enumeration Spark code

Fig. 4: Spark code snippets from both applications. Function calls highlighted in orange are Spark function calls. Functions highlighted in blue are user-written functions that are accelerated on the GPU.

than executing a set of tasks involving predominantly small graphs. Fortunately, Spark provides functionality to overcome this obstacle as well. The `parallelize` function used to split our list of comparison tasks, as shown in Figure 4, also has an optional second argument which allows the programmer to specify how many partitions to split the data into. In this case, we are simply partitioning a list of comparison tasks that need to be performed. By increasing the number of partitions, we decrease the number of tasks sent to a node in a given partition. Spark distributes these partitions out to workers one at a time, and it sends additional tasks out to workers as they complete the tasks they have already been assigned. In the case of FSK, this creates a load-balancing dynamic. Since the graphs that workers have to examine vary in size, some comparison tasks will take longer to execute than others. New partitions of the task list are sent to workers as they complete their assigned tasks, so nodes with less demanding tasks are not left idling while other nodes with more computationally intensive tasks are finishing up their comparisons. In the case of a cluster with multi-generational hardware, this load-balancing functionality allows older hardware to contribute to the computation without creating a bottleneck. Overall, this helps to ensure that we get the most out of every node on our HPC system.

1) Interoperating Python and OpenCL: Once the task list has been partitioned and workers have received their tasks, FSK can be run one of two ways: using the CPU or using an accelerator. Another major benefit of Spark is that we can specify how many executors to spawn on each node. This enables our framework to provide parallel execution of tasks on the CPU with no programmer overhead. This is accomplished by spawning one Spark executor per core. However, we also wanted to explore whether the GPU is better suited to handle these pairwise comparisons. Since our Spark code is written using the python interface of Spark (i.e. PySpark) and our GPU code is written in OpenCL, we needed a way to launch OpenCL kernels from Python source code. PyOpenCL provides exactly the interface needed to do this, and it does it in a way that is very intuitive, while requiring very minimal effort on the programmer's end [13]. When PyOpenCL is enabled, our framework spawns one Spark executor per GPU. With the combination of Spark and PyOpenCL, our framework is able to achieve a massive amount of parallelism in both the coarse-grained and fine-grained components of FSK, respectively.

B. Triangle Enumeration - Data Distribution on a Data-Bound Application

As mentioned in Subsection II-B, the hybrid triangle enumeration consists of three steps. First, we use Spark to partition the graph and distribute the subgraphs across the cluster. Second, we use either CPU or GPU BLAS libraries, SciPy [14] and ScikitCUDA [15] respectively, to calculate A^3 in order to count the number of triangles in each subgraph. Third and finally, we again use Spark to count the number of triangles that span subgraphs and sum up the triangles found in each subgraph.

Specifically, the partitioning and distribution of the subgraphs in done automatically using Spark's `parallelize` function. This function provides an option for manually specifying the number of partitions, a parameter we explore further in Subsection IV-C. The local task computations (i.e., calcu-

lating A^3) are performed using either SciPy, a python module that supports sparse matrix multiplication on the CPU, or ScikitCUDA, a python module that wraps the CUBLAS GPU library. For both of these BLAS libraries, we calculate A^3 by performing two consecutive symmetric matrix multiplications (SYMM). One SYMM is used to perform $A * A = A^2$ and another SYMM is used to perform $A^2 * A = A^3$). Determining where to perform the local computation, either on the CPU or GPU, is another parameter that is explored in Subsection IV-C. As shown in Figure 4, the counting of the subgraph-spanning triangles is done using a combination of Spark's `flatMap`, `union`, and `count` methods.

1) Optimizing the Graph Partitioning: It is also important to note that while we consider changing the number of partitions that Spark creates when we call `parallelize`, we do not attempt to control the content of the partitions for two reasons. First, the random ER graphs lends themselves well to uniform, random partitioning. Specifically, when partitioning an ER graph, the only attribute that affects the characteristics of the subgraphs is the size of the subgraphs. Spark's default partitioning behavior is to make each partition the same size, which results in the subgraphs all being approximately the same. This behavior of ER graphs leaves little room for optimization of the content's partitions since the cost of optimizing the partitioning would outweigh the benefits. Second, a real-world use-case of Triangle Enumeration will most likely use it as a subroutine, meaning that the data will already be partitioned based on the requirements of the higher-level use-case. Thus, Triangle Enumeration should avoid reshuffling the already distributed graph. For these two reasons, we avoid the use of expensive partitioning algorithms (which we expect will not improve performance) and allow Spark to partition the graph into equally-sized subgraphs.

IV. Evaluation & Results

This section presents our results for both applications (FSK and triangle enumeration) using our portable, high-level framework described in Section III.

A. Experimental Infrastructure

Both applications were run on a cluster consisting of four nodes, with a total of 40 CPU cores, connected over a commodity gigabit ethernet network. The specifications of each node are detailed in Table I. The nodes in our cluster were specifically chosen in order to demonstrate the portability and generality of our framework. The cluster's hardware spans both multiple vendors and also multiple architectural generations. We utilize 3 different generations of NVIDIA GPUs as well as the latest AMD GPU, which includes the latest memory technology, High-Bandwidth Memory (HBM), instead of the GDDR5 memory technology used in the NVIDIA GPUs [16]. The fact that we used the latest GPU architecture equipped with next-generation memory technology in conjunction with older hardware proves that our framework is both forward and backward compatible.

B. Fast Subtree Kernel

The runtime of FSK is examined in two stages. First, we examine the scalability and portability of running FSK across multiple nodes in a distributed system. Second, we run FSK on a single machine in order to examine its runtime on different components of heterogeneous systems.

1) Multi-Node Scalability: As shown in Figure 5, we achieve linear scalability for any dataset containing 100 graphs or more. Note that since runtimes were normalized against the slowest node, the speedups appear to be super-linear on the datasets of 100 and 1000 graphs. This is due to the variety of hardware we used, as shown in Table I. The small amount of overhead in distributing tasks across nodes is noticeable in the case of a very small dataset (10 graphs) because the computation is so trivial. A real-world application examines thousands (if not millions) of graphs, leading us to conclude that these results are very promising. Additional nodes can be added for larger datasets in order to process very large datasets in short amounts of time. It is worth noting that our initial experiments on multiple nodes were sub-optimal due to Spark using a small number of partitions. As mentioned in Section III, increasing the number of partitions used to distribute tasks creates an automatic load-balancing system between Spark's executor processes. Since we achieved linear multi-node scalability, we maintain that this technique can be used to efficiently distribute tasks on any HPC or cloud-based system.

2) Single-Node Parallelization: The machine used for these tests is machine 4 in Table I. In addition to achieving the best runtimes on the GPU, as shown in Figure 6, we demonstrate the portability of our framework. Since FSK's compute kernel is written in OpenCL, it can run on a variety of multi-core hardware, including both NVIDIA and AMD GPUs. In conjunction with linear multi-node scalability, this proves that our framework is suitable for any type of HPC or cloud-based system, no matter what type(s) of accelerators they are equipped with.

C. Triangle Enumeration

Previous work has shown that triangle enumeration is a data-bound application that is sensitive to the data it is operating on [9]. Dense graphs, like the community subgraphs found in social networks, are well suited for the GPU while sparse graphs, like computer network graphs, are well suited for the CPU. Therefore, the performance of triangle enumeration is evaluated and optimized in two stages in order to take advantage of this prior knowledge. First, we analyze the impact of changing the number of partitions on the performance of triangle enumeration in an attempt to minimize the overhead incurred by moving data. Second, we analyze the scenarios where it is optimal to execute the local computation on the GPU vs execute the local computation on the CPU. Since our hybrid triangle enumeration method is implemented with ScikitCUDA, these tests are performed using only the machines in the cluster with NVIDIA GPUs (i.e., machines 1-3 in Table I).

Machine #	CPU	GPU
1	Intel Xeon E5520 (Dual socket, 4 cores each)	NVIDIA K20 (5GB GDDR5)
2	Intel Core i7-5930K (6 cores)	NVIDIA GTX 970 (4GB GDDR5)
3	Intel Core i7-950 (4 cores)	NVIDIA GTX 470 (1.2GB GDDR5)
4	Intel Core i7-4771 (4 cores)	AMD Fury X (4GB HBM)

TABLE I: Specifications of the machines in the cluster used for testing. Note that the cluster consists of both server and desktop hardware from different vendors, spanning multiple architectural generations, with differing amounts of memory as well as differing types of memory technology.

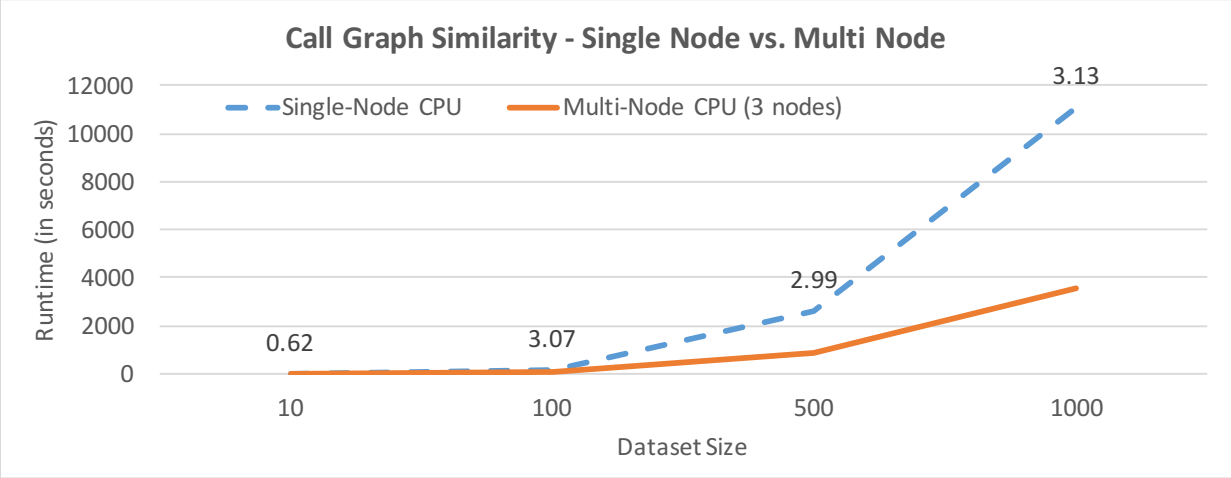


Fig. 5: Multi-node runtime results for FSK on different dataset sizes. Note that the numbers above the "single-node" line at a given dataset size represent the speedup achieved when processing that dataset using our multi-node system as compared to a single node. Additionally, runtimes were normalized against the slowest of the three nodes used.

1) Optimizing the Data Movement: Figure 7 shows the runtime of triangle enumeration when running with 36, 72, and 144 partitions (which are all multiples of the number of cores on the cluster) for graphs of a fixed size (i.e., 5000 nodes) with two different edge densities (i.e., .001 and .05). For a graph density of .001, as shown in Figure 7a, the fewer the number of partitions, the faster the runtime. In this case of a sparse graph, the local computation will only find triangles if the partitions are large enough. If the partitions are too small, the local computations will find no triangles, and the entirety of the counting will be performed by the Spark computation at the end of the application. This will result in a high amount of inter-node data movement. For a graph density of .05, as shown in Figure 7c, the greater the number of partitions, the faster the runtime. In this case of a dense graph, each individual partition contains a significant number of triangles and a 4 to 1 mapping of tasks to GPUs means that multiple kernels can be queued to run on the same GPU. This allows data transfers to and from the GPU to overlap with computation on the GPU which reduces the intra-node data movement costs.

2) Optimizing the Local Computation: Figure 8 shows the runtime of the local computation of triangle enumeration on the CPU and on the GPU w.r.t the properties of the graph (i.e., graph size and density). Figure 8a shows the runtime of the local computation when running on the CPU using the sparse matrix multiplication methods provided by SciPy. In this figure, runtimes above 4 seconds are clipped to 4 seconds. As expected with sparse matrix multiplication, this method only performs well when the graph is sparse (i.e., density ¡ .005). Figure 8b shows the runtime of the local computation when running on the GPU using dense matrix multiplication provided by ScikitCUDA, which ultimately uses NVIDIA's CUBLAS library. As expected with dense matrix multiplication, the performance of this method is not affected by the density of the graph and is only affected by the graph size. Also expected is that the runtime increases polynomially w.r.t the graph size, since the size of the adjacency matrix is N^2 where N is the size of the graph. It is important to note that for very sparse graphs (density ¡ .003), the CPU implementation actually performs better than the GPU implementation. For these sparse graphs, especially large ones, it is not worth the cost of transferring the adjacency matrix to the GPU.

D. Reproducibility

For anyone interested in reproducing our work, our framework and the two applications are open-source and can be found at https://github.com/rsearles35/WACCPD-2016. Instructions for setting up a Spark cluster, running the code, and ultimately reproducing our results can also be found in the GitHub repository.

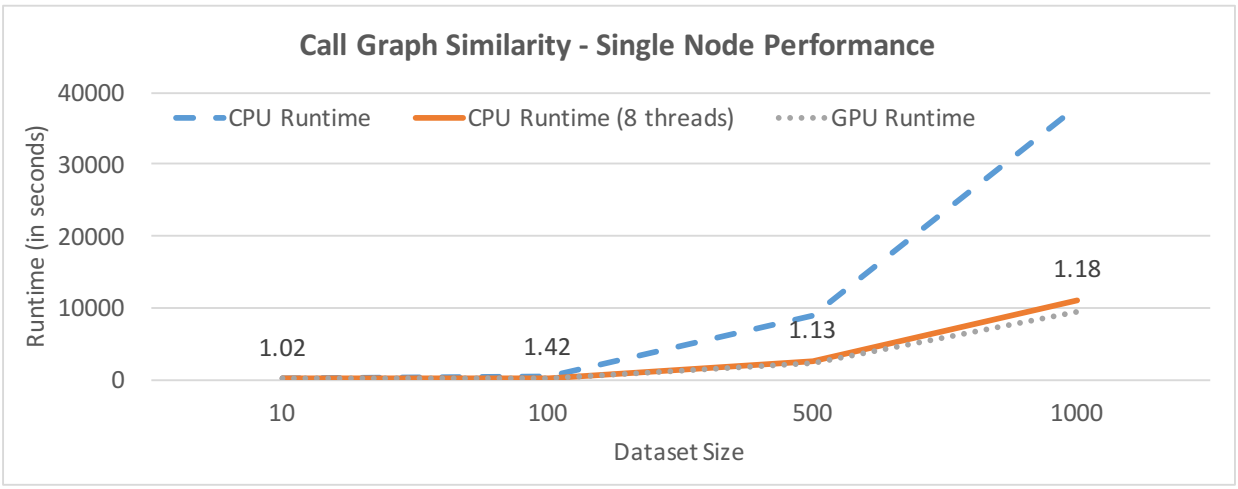


Fig. 6: Single-node runtime results for FSK on different dataset sizes. Runtimes for single thread CPU, 8 thread CPU, and GPU are shown. Note that GPU runtimes include data transfer to and from the accelerator. The numbers above the multi-core CPU line at a given dataset size represent the speedup achieved when processing that dataset using the GPU as compared to multi-core CPU.

V. Related Work

Many applications take advantage of heterogeneous hardware using an approach known as MPI+X that leverages MPI for communication and an accelerator language (e.g., CUDA and OpenCL) or directive-based language (e.g., OpenMP and OpenACC) for computation. Codes that utilize MPI+OpenACC include: the electromagnetics code NekCEM [17], the Community Atmosphere Model - Spectral Element (CAM-SE) [18], and the combustion code S3D [19]. Codes that utilize MPI+OpenMP include computational fluid dynamics MFIX [20], Second- order Mller-Plesset perturbation theory (MP2) [21], and Molecular Dynamics [22].

Several prototype MapReduce frameworks have been specifically designed to take advantage of multi-core CPUs and GPUs: Mars [23], MapCG [24], and MATE-CG [25]. Unfortunately, they all have limitations which reduce their portability and incur a much higher programming overhead than our solution. All three prototypes are restricted to a single node or GPU, which greatly limits the size of problems that they can handle. In addition, all three prototypes use CUDA as their backend GPU language, which limits the supported hardware to only NVIDIA GPUs. Mars stores all of the intermediate results in GPU memory, which requires the user to specify beforehand how much data will be emitted during the `Map` phase. This step requires additional effort from the programmer and is highly error-prone. MapCG uses a C-like language for its `Map` and `Reduce` functions which is then converted to OpenMP and CUDA code for parallelism. This restricts the capabilities of the application to their C-like language, which doesn't support many of the advanced feature of CUDA. MATE-CG does not support a `Map` operation and limits the user to using only `Reduce` and `Combine` operations, which makes porting existing MapReduce applications much harder.

Shirahata et al. [26] present a method for scheduling `Map` tasks on either the CPU or GPU depending on a dynamic profile of the task. Chen et al. [27] create a MapReduce framework that is optimized specifically for AMD's Fusion APUs. With the Fusion APU, the GPU shares the same memory space as the CPU, which enables their framework to do both pipelining and scheduling of MapReduce tasks across the CPU and GPU.

VI. Conclusion & Future Work

This paper presents a portable, high-level framework for combining the MapReduce paradigm with accelerators in order to run on heterogeneous HPC and cloud-based systems. We show linear scalability for a compute-bound HPC application, and we demonstrate effective techniques for optimizing a data-bound Big Data application. We also demonstrate the portability of this solution by collecting results on cluster composed of multiple generations of differing hardware, including multi-core Intel CPUs, NVIDIA GPUs, and AMD GPUs.

In the future, we plan to further optimize both applications' local performance by building a runtime scheduler that would determine whether to run a given task on the CPU or GPU based on that task's characteristics. We will also further examine the scalability of our solution by running both applications on larger distributed systems with much larger datasets.

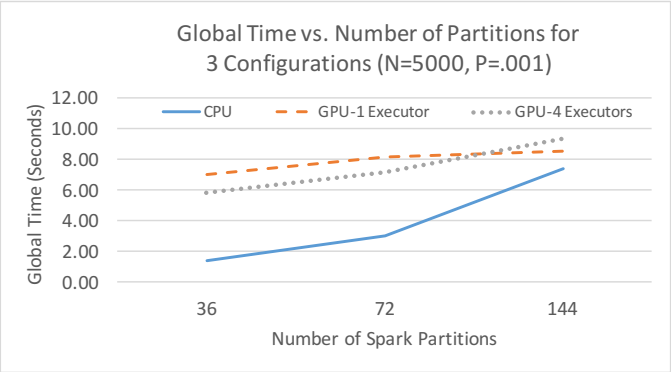
VII. Acknowledgments

The authors thank Jeremy Travis Johnston for his help in brainstorming on the hybrid triangle enumeration method used in this paper and his Erdős-Rényi random graph generator.

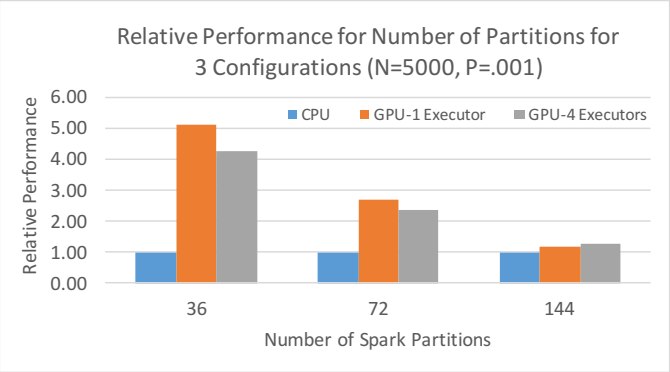
References

[1] "Spark - lightning-fast cluster computing," Retrieved Sept 7, 2016 from http://spark.apache.org/.

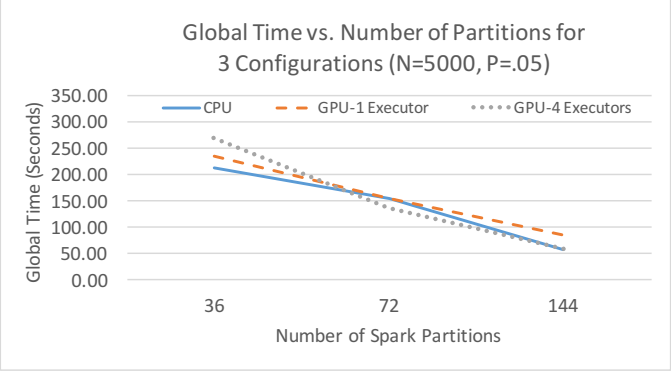
[2] NVIDIA, "NVIDIA accelerated computing - cuda," Retrieved Sept 7, 2016 from https://developer.nvidia.com/cuda-zone.



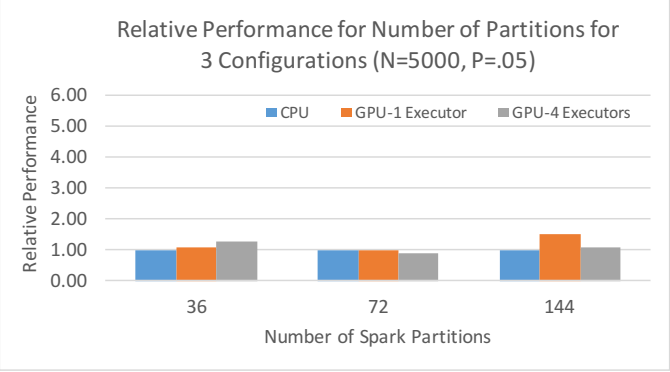
(a) Edge density (P) of .001 for a graph with 5000 nodes (N)



(b) Edge density (P) of .001 for a graph with 5000 nodes (N)



(c) Edge density (P) of .05 for a graph with 5000 nodes (N)



(d) Edge density (P) of .05 for a graph with 5000 nodes (N)

Fig. 7: Performance of Triangle Enumeration with a variable number of partitions for ER graphs with different densities. Subfigures 7a and 7c present the absolute runtimes versus the number of partitions. Subfigures 7b and 7d present the performance relative to the CPU version.

[3] Khronos Group, "The open standard for parallel programming of heterogeneous systems," Retrieved Sept 7, 2016 from https://www.khronos.org/opencl/.

[4] Open MPI, "A high performance message passing library," Retrieved Sept 7, 2016 from https://www.open-mpi.org/.

[5] J. Cohen, "Graph twiddling in a mapreduce world," *Computing in Science Engineering*, vol. 11, no. 4, pp. 29–41, July 2009.

[6] "Support vector machines," Retrieved Sept 7, 2016 from http://svms.org.

[7] P. Erdős and A. Rényi, "On random graphs, i," *Publicationes Mathematicae (Debrecen)*, pp. 290 – 297, 1959.

[8] "Radare2," Retrieved Sept 7, 2016 from http://rada.re/r/.

[9] T. Johnston, S. Herbein, and M. Taufer, "Exploring scalable implementations of triangle enumeration in graphs of diverse densities: Apache-spark vs. gpu," 2016, gPU Technology Conference (GTC).

[10] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. San Jose, CA: USENIX, 2012, pp. 15–28. [Online]. Available: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia

[11] W. Bland, A. Bouteiller, T. Herault, J. Hursey, G. Bosilca, and J. J. Dongarra, *Recent Advances in the Message Passing Interface: 19th European MPI Users' Group Meeting, EuroMPI 2012, Vienna, Austria, September 23-26, 2012. Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, ch. An Evaluation of User-Level Failure Mitigation Support in MPI, pp. 193–203. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-33518-1_24

[12] M. Gamell, D. S. Katz, H. Kolla, J. Chen, S. Klasky, and M. Parashar, "Exploring automatic, online failure recovery for scientific applications at extreme scales," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 895–906. [Online]. Available: http://dx.doi.org/10.1109/SC.2014.78

[13] A. Klckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih, "Pycuda and pyopencl: A scripting-based approach to {GPU} run-time code generation," *Parallel Computing*, vol. 38, no. 3, pp. 157 – 174, 2012. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819111001281

[14] E. Jones, T. Oliphant, P. Peterson *et al.*, "SciPy: Open source scientific tools for Python," 2001–, retrieved May 24, 2016 from http://www.scipy.org/.

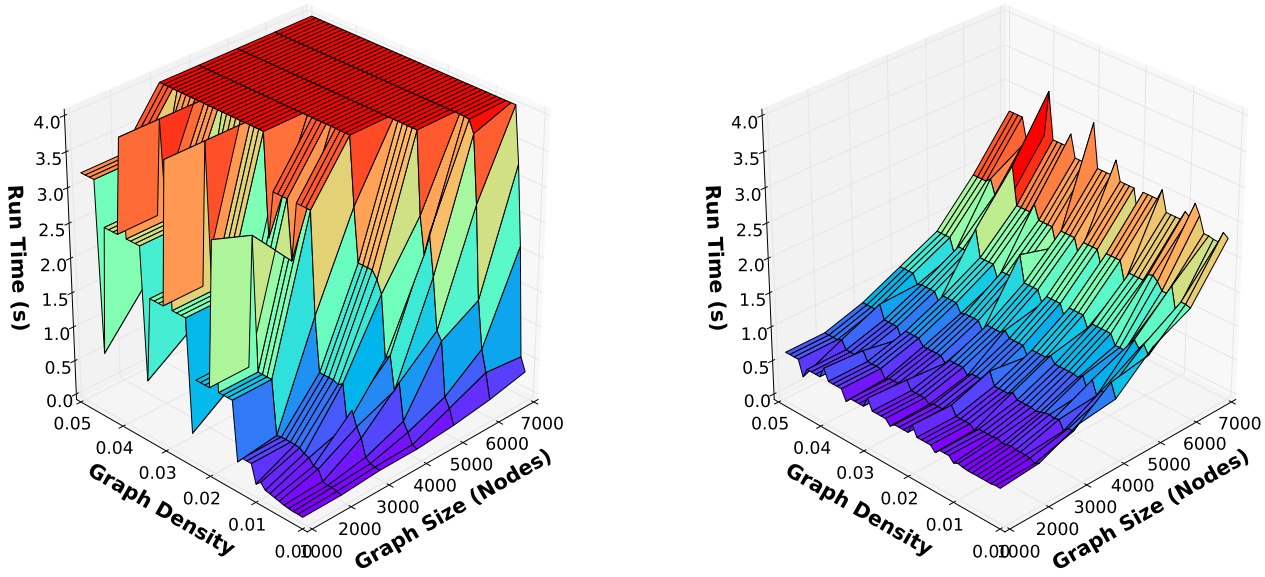
[15] L. E. Givon, T. Unterthiner, N. B. Erichson, D. W. Chiang, E. Larson, L. Pfister, S. Dieleman, G. R. Lee, S. van der Walt, T. M. Moldovan, F. Bastien, X. Shi, J. Schlüter, B. Thomas, C. Capdevila, A. Rubinsteyn, M. M. Forbes, J. Frelinger, T. Klein, B. Merry, L. Pastewka, S. Taylor, F. Wang, and Y. Zhou, "scikit-cuda 0.5.1: a Python interface to GPU-powered libraries," December 2015.

[16] "High bandwidth memory, reinventing memory technology," available at http://www.amd.com/en-us/innovations/software-technologies/hbm.

[17] M. Otten, J. Gong, and A. Mametjanov, "An mpi/openacc implementation of a high–order electromagnetics solver with gpudirect communication," in *International Journal of High Performance Computing Applications*, 2016.

[18] M. Norman, J. Larkin, A. Vose, and K. Evans, "A case study of cuda fortran and openacc for an atmospheric climate kernel," in *Journal of computational science*, 2015.

[19] J. M. Levesque, R. Sankaran, and R. Grout, "Hybridizing s3d into an exascale application using openacc: an approach for moving to multi–petaflops and beyond," in *Proceedings of the International conference on high performance computing, networking, storage and analysis*. IEEE Computer Society Press, 2012, p. 15.



(a) CPU

(b) GPU

Fig. 8: Runtime for computing A^3 on the CPU with SciPy and on the GPU with ScikitCUDA for graphs of varying size and density

[20] A. Gel, C. Guenther, and S. Pannala, "Accelerating clean and efficient coal gasifier designs with high performance computing," in *Proceedings of the 21st International Conference on Parallel CFD*, Moffett Field, CA, 2009.

[21] M. Katouda and T. Nakajima, "Mpi/openmp hybrid parallel algorithm of resolution of identity second–order moller–plesset perturbation calculation for massively parallel multicore supercomputers," in *Journal of chemical theory and computation*, 2013, pp. 5373–5380.

[22] M. Kunaseth, D. F. Richards, and J. N. Glosli, "Analysis of scalable data–privatization threading algorithms for hybrid mpi/openmp parallelization of molecular dynamics," in *The Journal of Supercomputing*, vol. 66. Springer, 2013, pp. 406–430.

[23] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: A mapreduce framework on graphics processors," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '08. New York, NY, USA: ACM, 2008, pp. 260–269. [Online]. Available: http://doi.acm.org/10.1145/1454115.1454152

[24] C. Hong, D. Chen, W. Chen, W. Zheng, and H. Lin, "Mapcg: Writing parallel program portable between cpu and gpu," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '10. New York, NY, USA: ACM, 2010, pp. 217–226. [Online]. Available: http://doi.acm.org/10.1145/1854273.1854303

[25] W. Jiang and G. Agrawal, "Mate-cg: A map reduce-like framework for accelerating data-intensive computations on heterogeneous clusters," in *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, May 2012, pp. 644–655.

[26] K. Shirahata, H. Sato, and S. Matsuoka, "Hybrid map task scheduling for gpu-based heterogeneous clusters," in *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, Nov 2010, pp. 733–740.

[27] L. Chen, X. Huo, and G. Agrawal, "Accelerating mapreduce on a coupled cpu-gpu architecture," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 25:1–25:11. [Online]. Available: http://dl.acm.org/citation.cfm?id=2388996.2389030