Creating a Portable, High-Level Graph Analytics Paradigm For Compute and Data-Intensive Applications

Robert Searles, Stephen Herbein, Travis Johnston, Michela Taufer, Sunita Chandrasekaran

Department of Computer and Information Sciences,
University of Delaware,
Newark, DE, USA
E-mail: {rsearles,sherbein,travisj,taufer,schandra}@udel.edu

Abstract: HPC offers tremendous potential to process large amount of data commonly referred to as 'Big Data'. Due to the immense computational requirements of Big Data applications, the HPC and Big Data communities are converging. As a result, 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. In this paper, we develop a portable, high-level paradigm that can be used to run Big Data applications on existing and future HPC systems. More specifically, we will target graph analytics applications, since these types of applications are becoming increasingly more popular in the Big Data and Machine Learning communities. Using our paradigm, we accelerate three real-world, compute and data intensive, graph analytics applications: a function call graph similarity application, a triangle enumeration subroutine, and a graph assaying application.

Our paradigm utilizes the popular MapReduce framework, Apache Spark, in conjunction with CUDA in order to simultaneously take advantage of automatic data distribution and specialized hardware present on each node of our HPC systems. We demonstrate scalability with regard to compute intensive portions of the code that are parallelizable, as well as an exploration of the parameter space for each application. 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.

Keywords: Graph Analytics; GPU; Distributed Systems; High Performance Computing; Cloud; Heterogeneous Systems; Cluster; Big Data; Apache Spark; CUDA

1 Introduction

As the Big Data and High Performance Computing communities continue to converge, large-scale computing on single nodes is becoming less and less feasible. Modern HPC systems are comprised of many nodes, each containing heterogeneous computing resources. This poses a challenge for programmers because they need to devise efficient ways to program such systems in order to distribute their computational tasks across these nodes, as well

as utilize all the computational resources each node has to offer. In this paper, we tackle this challenge by proposing a portable, high-level paradigm that can be used to leverage modern HPC systems to solve Big Data problems. Our high-level paradigm uses Big Data's MapReduce framework, Apache Spark (3), to distribute tasks across nodes in our cluster. We combine this with the HPC framework CUDA (23) 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 paradigm enables applications to be both backwards compatible and forwards compatible (i.e., they can efficiently run on any legacy, current, or next-generation hardware).

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 (24). 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 framework CUDA. 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 kernels. We demonstrate this technique on three real-world applications: the Fast Subtree Kernel (FSK), triangle enumeration (7), and graph assaying.

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 (4). 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 (8).

Graph assaying is the process of decomposing a graph G into a deck of k-vertex, induced subgraphs and counting the frequency of each resulting graph. If one views the k-vertex subgraphs as genes then a graph assay can be thought of as a characterization of the *genetic makeup* of the graph G. Triangle enumeration is, effectively, a partial 3-vertex graph assay where we are only concerned with the expression of a single gene (the triangle). While graph assaying can be used anytime a complete *genetic* profile of a graph is needed, it has recently been proposed as an important tool for understanding the efficacy of, and training, Spike Coupled Neural Networks (SCNNs). An SCNN is a time dependant, recurrent neural network which is fundamentally an edge-weighted, directed graph which

models communication in a brain. Currently, the only methods available for training such networks are genetic algorithms. The hope is that by understanding the genetic profiles of successful SCNNs, one could design new, larger SCNNs by seeding these genetic algorithms with networks having a similar genetic profile appropriately normalized.

The major contributions of this paper are as follows:

- We propose a high-level paradigm for combining MapReduce and accelerators to run applications portably and scalably on heterogeneous HPC and cloud-based systems.
- We adapt two compute-bound HPC applications, FSK and graph assaying, and demonstrate scalability with our solution.
- We adapt a data-bound Big Data application, triangle enumeration, and demonstrate performance optimization through parameter tuning.
- We present results on our heterogeneous cluster with a variety of CPUs and GPUs, as well as NVIDIA's PSG cluster, which is homogeneous and utilizes NVIDIA's next-generation P100 GPUs.

The rest of the paper is organized as follows: Section 2 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 3 describes the methodology used to adapt these applications to run on HPC and cloud-based systems using our proposed solution. Section 4 presents our results. We discuss related work in Section 5, and we conclude in Section 6.

2 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 paradigms are designed with one particular application category in mind, our paradigm is designed to support applications in either category as well as applications that share elements from both. To demonstrate the robustness of our high-level paradigm, we select two compute-bound HPC applications 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 three real-world applications used in this paper.

2.1 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 (2), 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:

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

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 3.1, we describe how we implement FSK using our high-level paradigm.

2.2 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).

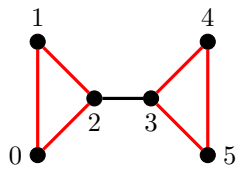


Figure 1: This graph contains 2 triangles (highlighted in red).

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 (15). 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 over counting 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 method 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 (7), 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 3.2, we describe how we implement this new hybrid method.

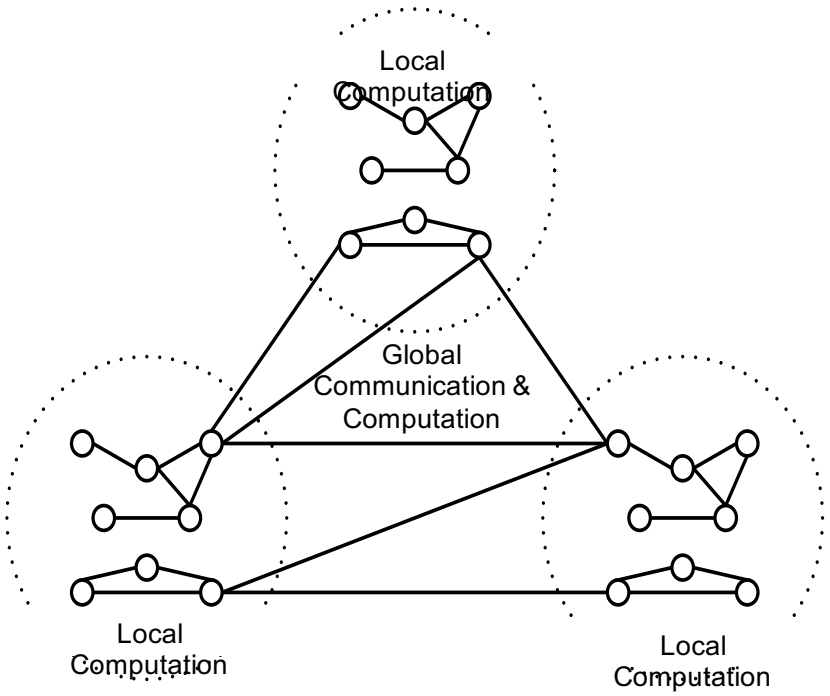


Figure 2: An example graph that is broken down into subgraphs by our hybrid triangle enumeration method. The triangles in the subgraphs are counted on the CPU or GPU with a BLAS library while the triangles that span the subgraphs are counted with a MapReduce framework.

2.3 Graph Assaying

Assaying a graph G is a general term for characterizing the frequency of each non-isomorphic subgraph of G of a specified size k, called a k-assay. A k-assay of G consists of the counts of every induced subgraph of G on k vertices. Figure 3 shows the simplest case of a graph assay, a 2-assay. For simple, undirected graphs there are only two non-isomorphic graphs on 2 vertices: the empty graph (denoted G_0 in Figure 3) which has no edge, and

the complete graph (denoted G_1 in Figure 3) consisting of a single edge. A 2-assay of G is therefore a count of the edges (and non-edges) of G. In the more general case when G is directed (allowing (a, b) and (b, a) to be edges), or when multiple undirected edges are allowed, then a 2-assay is no longer a simple edge count. Figure 4 shows a 3-assay of the

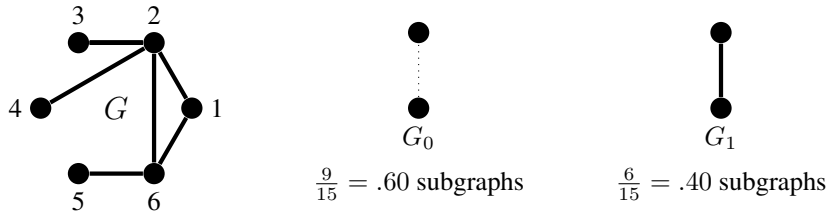


Figure 3: An example of a 2-assay of an undirected graph on 6 vertices. A 2-assay is a count of the edges and non-edges of a graph.

same graph. There are 4 non-isomorphic (simple, undirected) graphs on 3 vertices. Again, each of those 4 non-isomorphic graphs is characterized by the number of edges it has: G_i having i edges.

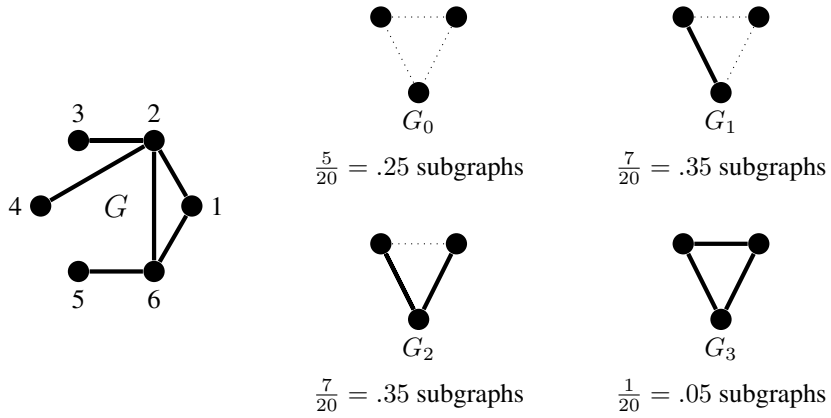


Figure 4: An example of a 3-assay of an undirected graph on 6 vertices. There are 3 non-isomorphic, simple (undirected) graphs on 3 vertices. A 3-assay counts the frequency of each of these.

Graph assays have been either used, or proposed, for a number of applications. One of the most popular questions related to graph assays is whether an $(n-1)$-assay of a graph G on n vertices uniquely determines the graph G. The conjecture that $(n-1)$-assays uniquely determine graphs is more commonly referred to as the reconstruction conjecture (see (21) for a good survey of the problem). While the full conjecture remains open, it has been shown that a number of graph properties can be reconstructed from assays; for example, for any $k \geq 2$ a k-assay of G can be used to determine the number of edges of G. Additionally, a number of infinite families of graphs are uniquely determined by their $(n-1)$-assays; these include trees, disconnected graphs, unit interval graphs, maximal planar graphs, outerplanar

graphs, and others. Though we may not (yet) be able to assert that any two graphs having the same k-assay are isomorphic, we can be certain that graphs with different k-assays are non-isomorphic. Graph assays (especially hypergraph assays) are also a fundamental part of using Flag Algebras (26). Flag Algebras have become an increasing popular and powerful tool in extremal combinatorics where they are used to understand the asymptotic behavior of constrained graphs. More recently, graph assays have been proposed, and are currently being used, to design and train spike coupled neural networks. The assays characterize the *genetic* traits of neural networks and are subsequently used to seed evolutionary algorithms which design newer, larger networks. The software currently used to design and train a spike coupled neural network produces, through generations of evolution and adaptation, thousands of diverse networks. The expectation is that a complete assay of these (many) networks will reveal characteristics (genes) that lead to the best performing networks. These genes can then be engineered into future generations of neural networks.

3 Methodology

In this section, we describe the general strategy behind our portable, high-level graph analytics paradigm, which enables our applications to run on heterogeneous HPC and cloud-based systems. Generally speaking, our paradigm uses Apache Spark (3) for automatic data/task distribution and CUDA ("X") for local task computation. Figure 5 shows a high-level depiction of this paradigm.

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 paradigm utilizes computational libraries (e.g. PyCUDA, ScikitCUDA, SciPy, and NumPy) in conjunction with user-written functions.

Figure 6 shows our three applications (FSK, Triangle Enumeration, and Graph Assaying) written using our paradigm. 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 PyCUDA 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 paradigm can utilize any available accelerators local to each node, via CUDA, to add a level of fine-grained parallelism to our paradigm, 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 (5; 9).

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

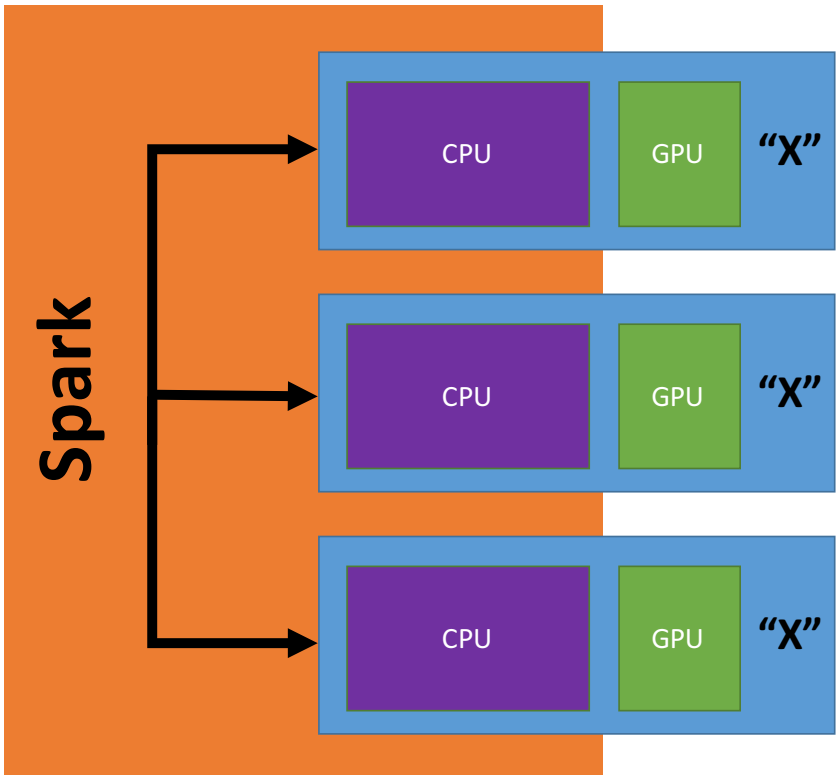


Figure 5: A high-level depiction of our proposed paradigm, which uses Spark for data/task distribution in conjunction with a local computational framework (X) on each node.

- 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 our applications.

3.1 Fast Subtree Kernel - Task Distribution on a Compute-Bound Application

As mentioned in Subsection 2.1, 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 3, Spark is our chosen solution for automatic task distribution. To this end, we utilize Spark to distribute the pairwise graph comparison tasks. Once these

```
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

```
conf = SparkConf()
sc = SparkContext(conf=conf)
subgraphs_rdd = sc.parallelize(subgraphs, numSlices=1024)
broadcast_G = sc.broadcast(G)
subgraph_frequencies = comp_rdd.map(MapSubGraphCompare.countByKey())
```

(c) Graph Assaying Spark Code

Figure 6: Spark code snippets from our applications. Function calls highlighted in orange are Spark function calls. Functions highlighted in blue are user-written functions that are accelerated on the GPU.

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 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 6, 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.

3.1.1 Interoperating Python and CUDA

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 paradigm 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 CUDA, we needed a way to launch CUDA kernels from Python source code. PyCUDA 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 (18). When PyCUDA is enabled, our paradigm spawns one Spark executor per GPU. With the combination of Spark and PyCUDA, our paradigm is able to achieve a massive amount of parallelism in both the coarse-grained and fine-grained components of FSK, respectively.

3.2 Triangle Enumeration - Data Distribution on a Data-Bound Application

As mentioned in Subsection 2.2, 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 (16) and ScikitCUDA (11) 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 4.3. The local task computations (i.e., calculating 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 4.3. As shown in Figure 6, the counting of the subgraph-spanning triangles is done using a combination of Spark's `flatMap`, `union`, and `count` methods.

3.2.1 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.

3.3 Graph Assaying - Task and Data Distribution on a Compute-Bound Application

Graph assaying is the process of characterizing the frequency of each non-isomorphic subgraph of a graph G of a specified size k, as mentioned in Section 2.3. This is called a k-assay. We represent a graph as an NxN adjacency matrix, where N is equal to the number of nodes in the graph. The number of subgraphs can be calculated by computing $\binom{N}{k}$. Since, each subgraph characterization is not dependent on any external information, these characterizations can be done in parallel. Once each subgraph is characterized, a reduction is performed in order to count the frequency of each type of subgraph found in G.

We can use Spark's `map` and `countByKey` functions to perform both of these operations. We use `map` to send subgraph characterization tasks to each node in our cluster. We then call `countByKey` on the results to calculate the frequency of each characterized subgraph, and we report these results. As discussed in Section 3.1, our implementation of graph assaying benefits from Spark's lazy evaluation of task assignments. While the computational requirements of each subgraph characterization do not vary greatly, we could experience a difference in runtime on non-homogeneous clusters. If a node with older/slower

hardware is taking a long time to complete its given task, Spark does not have to wait in order to issue new tasks to the faster nodes in the cluster. This creates a load-balancing type of behavior that can help us harness every ounce of performance our cluster has to offer. In addition, Spark's `countByKey` function provides a well-optimized reduction of these characterizations. Each node will reduce its local results, and then Spark will merge the results from each node in a final global reduction stage.

3.3.1 Challenges when using the GPU

Since our graph assaying algorithm only has one layer of exploitable parallelism, we had to adjust our strategy when adapting the code to run on the GPU. Instead of using Spark to distribute each subgraph comparison as a task, we partitioned ranges of subgraphs to be sent as tasks to each node using Spark. For example, if we have a total of 100 subsets in a graph and 4 nodes in a cluster, we have Spark send a chunk of 25 subgraphs to each node. Once the tasks have been assigned to each node, we perform all subgraph analyes on the GPU using CUDA. Since each subgraph analysis is independent of all other subgraph analyses, this approach is well-suited for the GPU. It is worth noting that if the number of subgraphs being examined is small, the computation will not saturate the GPU, and thus, it will not perform well. However, the GPU will greatly outperform the CPU in almost all other cases. The reduction of these results is performed in a similar manner, but since we are using Spark to distribute tasks instead of mapping across all subgraph comparisons individually, we must perform the local and global reductions ourselves, instead of simply calling `countByKey` like we do in the CPU implementation.

4 Evaluation & Results

This section presents our results for all three applications (FSK, triangle enumeration, and graph assaying) using our portable, high-level paradigm described in Section 3.

4.1 Experimental Infrastructure and Software Used

Machine #	CPU	GPU
1	Intel Xeon E5520 (2x, 4 cores each)	NVIDIA K20 (5GB GDDR5)
2	Intel Xeon E5-2603 (2x, 4 cores each)	NVIDIA GTX TITAN X (12GB GDDR5)
3	AMD Opteron 6320 (2x, 8 cores each)	NVIDIA K40 (12GB GDDR5)

Table 1 Specifications of the machines in our local hybrid cluster used for testing. Note that the cluster consists of CPUs from different vendors, as well as NVIDIA GPUs spanning multiple architectural generations, with differing amounts of memory.

All three applications were run on both a local cluster, with non-uniform hardware, and NVIDIA's PSG cluster. Our local cluster consists of three nodes, with a total of 32 CPU cores, connected over a commodity gigabit ethernet network. The specifications of each node are detailed in Table 1. The nodes in our cluster were specifically chosen in order to demonstrate the portability and generality of our paradigm. The cluster's hardware spans both multiple vendors and also multiple architectural generations. We also ran our applications on NVIDIA's Professional Solutions Group (PSG) cluster, which granted us

Machine #	CPU	GPU
1-4	Intel Xeon E5-2698 v3 (2x, 16 cores each)	NVIDIA P100 (16GB HBM2)

Table 2 Specifications of the nodes in NVIDIA's PSG cluster used for testing. Note that the cluster consists of uniform hardware in each node, as well as NVIDIA's state-of-the-art P100 GPUs, which utilize next-generation HBM2 memory technology.

access to nodes equipped with NVIDIA's P100 GPU, which includes the latest memory technology, High-Bandwidth Memory (HBM2), instead of the GDDR5 memory technology used in the GPUs in our local cluster (1). HBM is a new memory technology that features 3D die-stacked, on-chip memory. This offers 3.5x bandwidth per watt compared to GDDR5. The fact that we used the latest GPU architecture equipped with next-generation memory technology in conjunction with older hardware proves that our paradigm is both forward and backward compatible.

Non-uniform hardware was not the only challenge we faced in this work. The machines in our local cluster also featured non-uniform hardware. For example, two of the machines are running Ubuntu 14.04, while the third is running CentOS 6.6. This presented a challenge because CentOS 6.6 relies on Python 2.6. Since our other machines have Python 2.7 as their default version, we had to install 2.7 on the CentOS machine in order to be able to run Spark across all three. We overcame this obstacle by creating a Python virtual environment (virtualenv) on each of our machines with the necessary versions of Python and all of our required libraries (PyCUDA, Scikit-CUDA, SciPy, etc). In addition to creating uniform Python virtual environments, we made sure to upgrade all three machines to the latest version of CUDA (8.0) for optimal compatibility. Once this was complete, we were able to successfully run our experiments across all three machines, despite the varying operating systems.

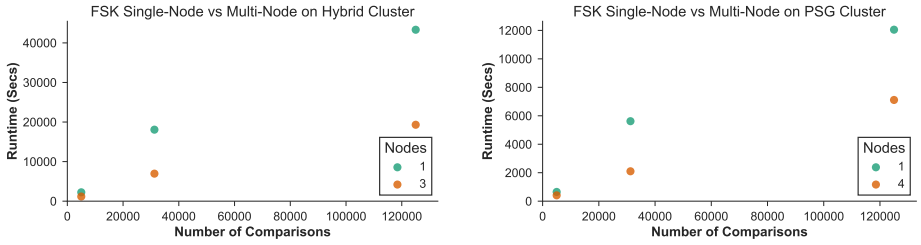
4.2 Fast Subtree Kernel

The runtime of FSK is examined on both clusters 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 the GPU in order to examine its runtime on different components of heterogeneous systems.

4.2.1 Multi-Node Scalability

As shown in Figure 7a, we achieve good scalability for both of the larger dataset sizes examined. The small amount of overhead in distributing tasks across nodes is noticeable in the case of a small dataset because the computation is so trivial. As a result, the achieved speed up is minimal. 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 3, increasing the number of partitions used to distribute tasks creates an automatic load-balancing system between Spark's executor processes. Since we achieved favorable multi-node scalability, we maintain that this technique can be used to efficiently distribute tasks on any HPC or cloud-based system.

Figure 7b's results look slightly less scalable compared to Figure 7a at first glance. Notice that the medium-sized dataset boasts a 3x speedup, while the largest dataset shows less than a 2x speedup. While running FSK on the largest dataset on the cluster, we experienced intermittent failures with one of the nodes in the system. This node was later taken down for maintenance for an extended period, and thus, it did not affect any of our other experiments. We decided to include these results as they are as a testiment to Spark's native fault tolerance. This job completed without any checkpointing or restarting. Despite the overhead caused by these failures, we still achieved almost a 2x speedup.



(a) Hybrid Cluster

(b) PSG Cluster

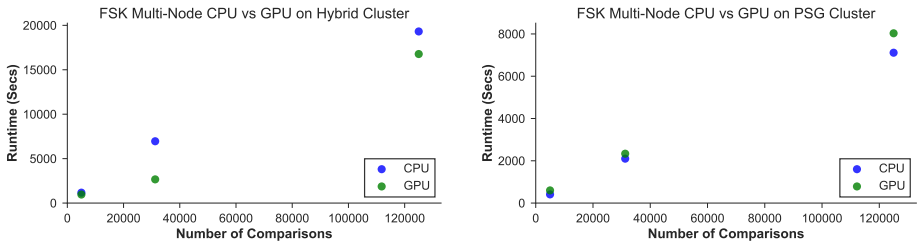
Figure 7: Single-node and multi-node runtime results for FSK run on each cluster. Note that even for a very large dataset, we still achieve a significant speedup.

4.2.2 CPU vs. GPU

The machines used for these tests are the machines described in Tables 1 and 2. In addition to achieving the best runtimes using GPUs, as shown in Figure 8a, we demonstrate the forward and backward compatibility of our paradigm. We were able to successfully run our code across three nodes, each equipped with a different generation of NVIDIA GPU, as shown in Table 1. This demonstrates the portability of our paradigm. As shown in Figure 8b we also successfully ran FSK across nodes in NVIDIA's PSG cluster, which utilize next-generation P100 GPUs that feature state-of-the-art HBM2 memory technology. This supports our claim that our paradigm is portable, since the memory technology used in these GPUs differs from the traditional GDDR5 memory found in all three GPUs on our local cluster. This also allows us to show that our paradigm is forward compatible with next-generation GPU hardware. It is worth noting that each of these nodes were equipped with two extremely powerful state-of-the-art CPUs, totaling 128 physical CPU cores across four nodes. We only utilized one P100 GPU on each of the nodes, and the runtimes were comparable. Since there were a total of 8 CPUs and only 4 GPUs, we consider the comparable runtime to be a success and a testament to the computational ability of the P100 GPU.

4.3 Triangle Enumeration

Previous work has shown that triangle enumeration is a data-bound application that is sensitive to the data it is operating on (15). 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



(a) Hybrid Cluster (b) PSG Cluster

Figure 8: GPU runtime results for FSK running on both clusters. Runtimes for multi-node CPU and multi-node GPU are shown. Note that GPU runtimes include data transfer to and from the accelerator.

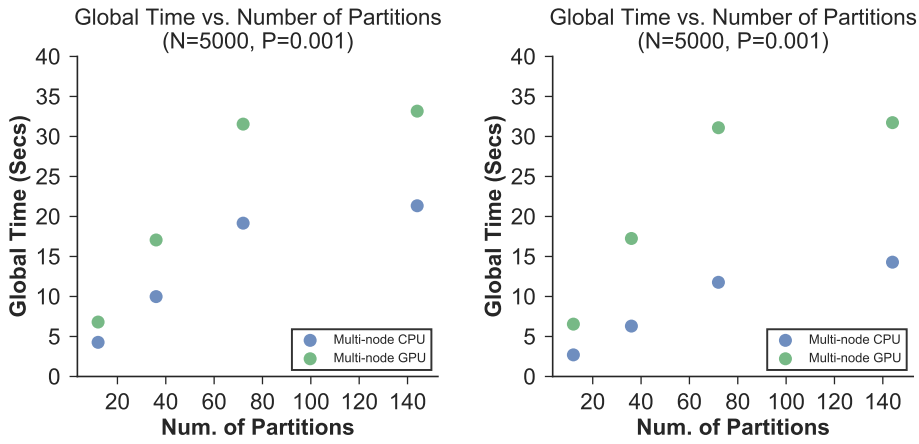
enumeration is evaluated and optimized in two stages in order to take advantage of this prior knowledge. First, we analyze the impact of changing the number of partitions on the performance of triangle enumeration in an attempt to minimize the overhead incurred by moving data. Second, we analyze the scenarios where it is optimal to execute the local computation on the GPU vs execute the local computation on the CPU.

4.3.1 Optimizing the Data Movement

Figures 9 and 10 show the runtime of triangle enumeration when running with 12, 36, 72, and 144 partitions (which are all multiples of the number of nodes on both clusters) 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 9, 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 10, 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.

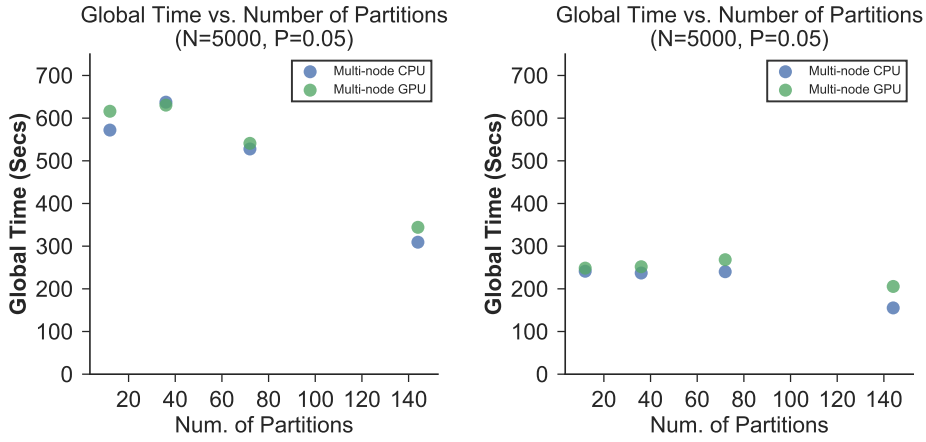
4.3.2 Optimizing the Local Computation

Figure 11 shows the runtime of the local computation of triangle enumeration on the CPU and on the P100 GPU of a single PSG cluster node w.r.t the properties of the graph (i.e., graph size and density). Figure 11a 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 6 seconds are clipped to 6 seconds. As expected with sparse matrix multiplication, this method only performs well when the graph is sparse (i.e., density $\leq .005$) or small (i.e., size ≤ 1000). Figure 11b 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,



(a) Hybrid Cluster (b) PSG Cluster

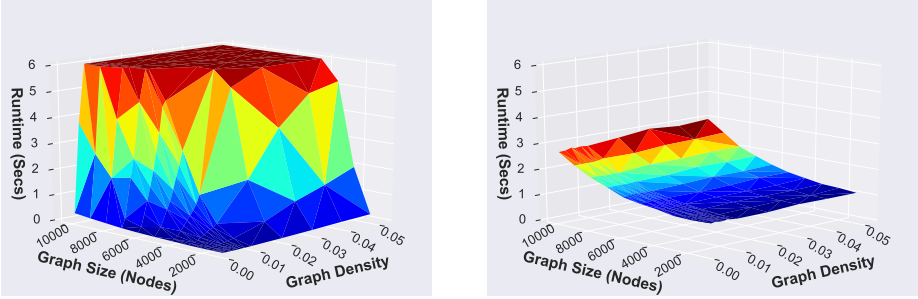
Figure 9: Performance of Triangle Enumeration running on both clusters with a variable number of partitions for ER graphs with 5000 nodes (N) and an edge density (P) of .001.



(a) Hybrid Cluster (b) PSG Cluster

Figure 10: Performance of Triangle Enumeration running on both clusters with a variable number of partitions for ER graphs with 5000 nodes (N) and an edge density (P) of .05.

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 (density $\leq .003$) or very small (size ≤ 1000) graphs, the CPU implementation actually performs better than the GPU implementation. For both small and sparse graphs, it is not worth the cost of transferring the adjacency matrix to the GPU.



(a) CPU (b) GPU

Figure 11: Runtime for computing A^3 on the CPU with SciPy and on the GPU with ScikitCUDA for graphs of varying size and density

4.4 Graph Assaying

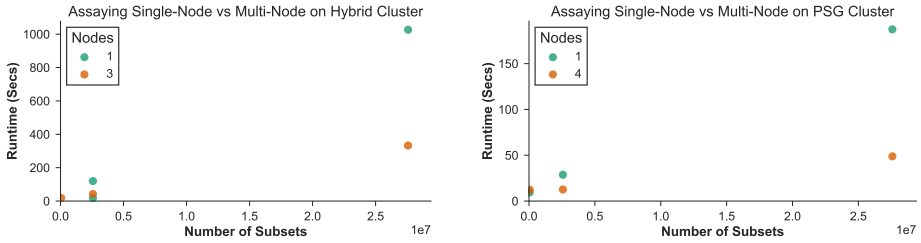
The runtime of our graph assaying application is examined in three stages. Much like we did with FSK in Section 4.2, we will examine the scalability of this application, as well as comparing the runtime of the GPU with that of the CPU. In addition to these metrics, our third metric will involve changing the number of Spark partitions in conjunction with the use of the GPU to see how it affects performance.

4.4.1 Multi-Node Scalability

Figure 12a demonstrates a significant speedup for large dataset sizes when running on multiple nodes using Spark, since the computational cost greatly outweighs the overhead of distributing tasks and data across nodes. These results are very consistent with the results from our other compute-bound application, FSK, as discussed in Section 4.2, leading us to conclude that these results are promising. Using a very similar technique, we have ported two compute-bound applications to our paradigm using Spark and CUDA, and we achieved a favorable speedup in both cases. Again, Figure 12b looks a little less impressive at first glance due to the overhead of distributing tasks with their associated data across nodes using Spark. Despite this overhead, our results prove that this technique scales well to very large, distributed systems.

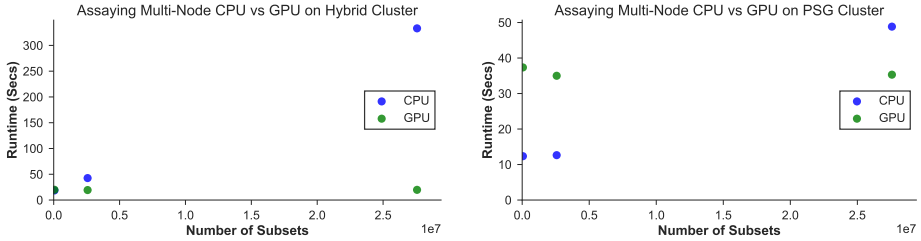
4.4.2 CPU vs. GPU

Figures 13a and 13b demonstrate the scalability of our system utilizing the accelerators present on each of the nodes in our HPC systems. We see that our graph assaying application performs even more efficiently using the GPU than our other compute-bound application, FSK. The GPU outperforms the CPU when using large graphs on both our hybrid cluster consisting of a range of NVIDIA GPUs, as well as NVIDIA's PSG cluster consisting of nodes equipped with their next-generation P100 GPUs. It is worth noting that the graph being examined only needs to be copied to each GPU once. Each subset can then be examined using the GPU, and the results are copied back to the host Spark worker, where they are reduced globally by Spark. Therefore, for a large number of examined subsets, the GPU greatly outperforms the multi-core CPU.



(a) Hybrid Cluster (b) PSG Cluster

Figure 12: Single-node and multi-node runtime results for Graph Assaying run on each cluster. Note that even for a very large dataset, we still achieve a significant speedup.



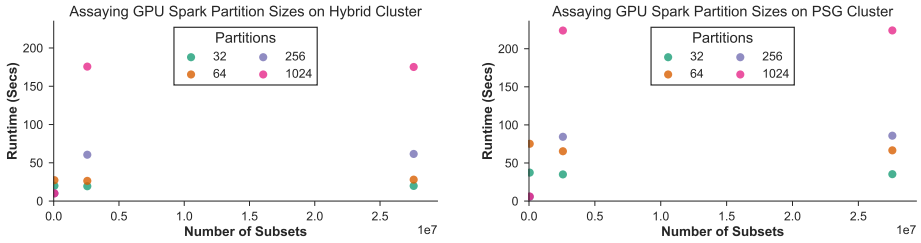
(a) Hybrid Cluster (b) PSG Cluster

Figure 13: GPU runtime results for Graph Assaying running on both clusters. Runtimes for multi-node CPU and multi-node GPU are shown. Note that GPU runtimes include data transfer to and from the accelerator.

4.4.3 Optimizing GPU Performance

We decided to perform an additional experiment for our graph assaying application involving performance tuning of our GPU implementation by varying the number of Spark partitions used. As shown in Figure 14, the number of Spark partitions used can greatly affect the performance of an application depending on how it is structured by the programmer. In our case, the graph assaying algorithm can operate on many subsets, while only copying the graph onto the device once. Therefore, splitting up the computational tasks into more partitions reduces the load on the GPU, leading to performance degradation, since the GPU will become less and less saturated.

In order to achieve the best results on our graph assaying application, the number of partitions used needed to be minimized in order to saturate the GPU, while still maintaining enough Spark partitions to guarantee that any given task's size does not result in GPU memory being overflowed by the results of the computation. Once we achieved this balance, we gained superior performance utilizing our GPUs in our hybrid cluster, as well as the P100 GPUs on NVIDIA's PSG cluster. We expect that this type of tuning will translate well to any GPU application that operates on a large amount of data.



(a) Hybrid Cluster (b) PSG Cluster

Figure 14: GPU runtime results for Graph Assaying running on both clusters with varying numbers of Spark partitions. Note increasing the number of partitions results in less saturation of the GPU.

4.5 Reproducibility

For anyone interested in reproducing our work, our paradigm 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.

5 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 (25), the Community Atmosphere Model - Spectral Element (CAM-SE) (22), and the combustion code S3D (20). Codes that utilize MPI+OpenMP include computational fluid dynamics MFIX (10), Second-order Mller-Plesset perturbation theory (MP2) (17), and Molecular Dynamics (19).

Several prototype MapReduce frameworks have been specifically designed to take advantage of multi-core CPUs and GPUs: Mars (12), MapCG (13), and MATE-CG (14). 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 (12) stores all 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. (27) present a method for scheduling `Map` tasks on either the CPU or GPU depending on a dynamic profile of the task. Chen et al. (6) 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.

6 Conclusion & Future Work

This paper presents a portable, high-level paradigm for combining the MapReduce paradigm with accelerators in order to run on heterogeneous HPC and cloud-based systems. We demonstrate scalability using our FSK and graph assaying applications, and we demonstrate effective techniques for optimizing a data-bound Big Data application using triangle enumeration. 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 and NVIDIA GPUs, as well as NVIDIA's PSG cluster containing nodes fitted with their next-generation P100 GPUs.

In the future, we plan to further optimize each 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 all three applications with much larger datasets across many more nodes.

References

[1] High bandwidth memory, reinventing memory technology. Available at `http://www.amd.com/en-us/innovations/software-technologies/hbm`.

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

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

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

[5] Wesley Bland, Aurelien Bouteiller, Thomas Herault, Joshua Hursey, George Bosilca, and Jack J. Dongarra. *Recent Advances in the Message Passing Interface: 19th European MPI Users' Group Meeting, EuroMPI 2012, Vienna, Austria, September 23-26, 2012. Proceedings*, chapter An Evaluation of User-Level Failure Mitigation Support in MPI, pages 193–203. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

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

[7] J. Cohen. Graph twiddling in a mapreduce world. *Computing in Science Engineering*, 11(4):29–41, July 2009.

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

[9] Marc Gamell, Daniel S. Katz, Hemanth Kolla, Jacqueline Chen, Scott Klasky, and Manish 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*, SC '14, pages 895–906, Piscataway, NJ, USA, 2014. IEEE Press.

[10] 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.

[11] Lev E. Givon, Thomas Unterthiner, N. Benjamin Erichson, David Wei Chiang, Eric Larson, Luke Pfister, Sander Dieleman, Gregory R. Lee, Stefan van der Walt, Teodor Mihai Moldovan, Frédéric Bastien, Xing Shi, Jan Schlüter, Brian Thomas, Chris Capdevila, Alex Rubinsteyn, Michael M. Forbes, Jacob Frelinger, Tim Klein, Bruce Merry, Lars Pastewka, Steve Taylor, Feng Wang, and Yiyin Zhou. scikit-cuda 0.5.1: a Python interface to GPU-powered libraries, December 2015.

[12] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K. Govindaraju, and Tuyong Wang. Mars: A mapreduce framework on graphics processors. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 260–269, New York, NY, USA, 2008. ACM.

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

[14] 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*, pages 644–655, May 2012.

[15] Travis Johnston, Stephen Herbein, and Michela Taufer. Exploring scalable implementations of triangle enumeration in graphs of diverse densities: Apache-spark vs. gpu. GPU Technology Conference (GTC), 2016.

[16] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001–. Retrieved May 24, 2016 from `http://www.scipy.org/`.

[17] Michio Katouda and Takahito 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*, pages 5373–5380, 2013.

[18] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, Bryan Catanzaro, Paul Ivanov, and Ahmed Fasih. Pycuda and pyopencl: A scripting-based approach to {GPU} run-time code generation. *Parallel Computing*, 38(3):157 – 174, 2012.

[19] Manaschai Kunaseth, David F Richards, and James N Glosli. Analysis of scalable data–privatization threading algorithms for hybrid mpi/openmp parallelization of molecular

dynamics. In *The Journal of Supercomputing*, volume 66, pages 406–430. Springer, 2013.

[20] John M Levesque, Ramanan Sankaran, and Ray 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*, page 15. IEEE Computer Society Press, 2012.

[21] C. St. J.A. Nash-Williams. *Selected Topics in Graph Theory*, chapter The Reconstruction Problem, pages 205–236. Academic Press, London, 1978.

[22] Matthew Norman, Jeffrey Larkin, Aaron Vose, and Katherine Evans. A case study of cuda fortran and openacc for an atmospheric climate kernel. In *Journal of computational science*, 2015.

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

[24] MPI. The Message Passing Interface (MPI) standard Retrieved May 3, 2017 from `http://www.mcs.anl.gov/research/projects/mpi/`.

[25] Matthew Otten, Jing Gong, and Azamat Mametjanov. An mpi/openacc implementation of a high–order electromagnetics solver with gpudirect communication. In *International Journal of High Performance Computing Applications*, 2016.

[26] Alexander A. Razborov. Flag algebras. *J. Symb. Log.*, 72(4):1239–1282, 2007.

[27] 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*, pages 733–740, Nov 2010.