Optimizations and Analysis of BSP Graph Processing Models on Public Clouds

Mark Redekopp, Yogesh Simmhan, and Viktor K. Prasanna
University of Southern California, Los Angeles CA 90089
{redekopp, simmhan, prasanna}@usc.edu

Abstract— **Large-scale graph analytics is a central tool in many fields, and exemplifies the size and complexity of Big Data applications. Recent distributed graph processing frameworks utilize the venerable *Bulk Synchronous Parallel (BSP)* model and promise scalability for large graph analytics. This has been made popular by Google's Pregel, which offers an architecture design for BSP graph processing. Public clouds offer democratized access to medium-sized compute infrastructure with the promise of rapid provisioning with no capital investment. Evaluating BSP graph frameworks on cloud platforms with their unique constraints is less explored. Here, we present optimizations and analysis for computationally complex graph analysis algorithms such as betweenness-centrality and all-pairs shortest paths on a native BSP framework we have developed for the Microsoft Azure Cloud, modeled on the Pregel graph processing model. We propose novel heuristics for scheduling graph vertex processing in swaths to maximize resource utilization on cloud VMs that lead to a 3.5x performance improvement. We explore the effects of graph partitioning in the context of BSP, and show that even a well partitioned graph may not lead to performance improvement due to BSP's barrier synchronization. We end with a discussion on leveraging cloud elasticity for dynamically scaling the number of BSP workers to achieve a better performance than a static deployment, and at a significantly lower cost.**

Keywords- Graph analytics; Cloud computing; Pregel; MapReduce; Bulk Synchronous Parallel; Betweennness centrality

I. INTRODUCTION

The Big Data paradigm refers not just to the size of data but also its complexity. Graph data structures and algorithms exemplify these challenges. Large-scale graph data sets with billions of vertices are becoming common in the context of social networks and web-scale data sets. Graph analytics attempt to extract useful information such as central vertices or clusters from the structure of the graph and related attributes and is an important tool for epidemiology [1], protein interactions [2], and even ecological connectivity [3]. The majority of these data sets exhibit small-world properties [4] with small average diameter and high clustering. Recognizing the importance of scalable graph analytics in both scientific and commercial domain, many new parallel and distributed graph processing approaches [5] [6] [7] have emerged in the academic and open source communities. These frameworks often provide a simplified programming abstraction (e.g. vertex-centric or MapReduce) to aid programmers in mapping algorithms while providing scalability. Meanwhile, the frameworks themselves partition the graph data across machines, handle synchronization and communication, and provide a simplified user interface to the system.

At the same time public cloud computing services that offer either compute infrastructure-as-a-service (IaaS) or a software platform-as-a-service (PaaS) have gained popularity. The ability to rapidly provision compute resources on-demand promises cost scalability based on utilization and democratizes resource access for the "long tail" of data-driven science. Popular commercial/public cloud service providers include Amazon EC2[1] and Microsoft Azure[2]. However, these benefits come with the overhead associated with *infrastructure* virtualization on commodity hardware, execution in a controlled *platform* environment, and the inability to control exact VM placement and thus communication latency/bandwidth. In addition, multi-tenancy impacts performance consistency. Users choosing a commercial, public cloud service may want to trade dollar cost against performance or even reliability. Attaching a real monetary cost to public cloud resources also limits their typical usage for scientific computing to 10-100's of cores rather than the 1000's of cores, typical of a private/academic HPC center. *Thus public clouds fit the scalability and manageability sweet-spot for scientific applications with resource needs that fall beyond a single large server and below HPC/supercomputing resources.*

One of the key gaps in literature is an evaluation of graph processing frameworks on public cloud platforms. While these frameworks may offer a certain scalability on HPC clusters with fast interconnects [5] [7], their behavior on virtualized commodity hardware that is accessible to a wider population of users is less understood. MapReduce (MR) [8] has been well explored for graph processing [10] despite its origins in processing tuple-based datasets, due to its pervasiveness in large-scale data processing. Of particular novel interest is the Bulk Synchronous Parallel (BSP) [9]model that has seen a revival of-late for *vertex-centric* graph processing, championed by Google's

[1] Amazon Web Services. http://aws.amazon.com
[2] Microsoft Azure. http://www.microsoft.com/windowsazure

Pregel. BSP uses the notion of independent tasks that run concurrently within a superstep, synchronize and exchange messages at the end of a superstep, and then start the next superstep. In mapping this to graph processing, Pregel associates graph partition(s) with each machine, a user logic with every graph vertex that runs as a task in a superstep and allows the user logic to exchange messages with other vertices (usually to neighboring vertices) at superstep boundaries.

Open source BSP frameworks for graph processing, such as Apache Hama [11] and Giraph[3], are still evolving. As a result, the performance characteristics of BSP, for different varieties of graph algorithms and on public clouds, are less known. Furthermore, these open implementations, while offering a BSP programming model, rely on the Hadoop MapReduce framework for their execution rather than use a native execution model suited for BSP. As a result, optimization approaches for graph processing using the BSP model, and an analysis of their performance, are harder to narrow down and evaluate without Hadoop's side-effects.

In this paper, we make three primary contributions. One, we introduce *adaptive and generalizable vertex and superstep scheduling heuristics* to the BSP graph processing model that alleviate the burden of correctly predicting appropriate resource provisioning levels *a priori*. These execution heuristics are incorporated into our own .NET implementation of a BSP distributed graph processing framework, based on Pregel's programming abstractions, which natively runs on the Microsoft Azure Cloud platform. At the same provisioning level, our evaluations show that our heuristics can achieve up to 3.5x speedup. Second, we *analyze the impact of graph partitioning for BSP frameworks* showing that while partitioning can be effective at reducing inter-worker communication it can also be a source of load imbalance that can cancel the positive effects of partitioning on BSP frameworks. Finally, *we present a model for exploiting the elasticity of clouds to adaptively scale the number of VMs based on demand* and show the promise of this technique for BSP graph processing. Using experimental results of statically provisioned systems our extrapolation shows that this technique offers better runtimes than even statically over-provisioned systems and at a reduced monetary cost. In all of our work we use a classical betweenness-centrality (BC) algorithm over several real "small-world" network datasets as a stress case for both the BSP model and cloud platform rather than approximate versions of BC or other workloads (though we include PageRank as point of comparison).

Our work is scoped to evaluate the efficacy of using BSP to scale graph problems beyond a single local server into the cloud to benefit a broad category of researchers rather than offer comparisons with problems that run on HPC-class clusters. Further, we recognize that the real monetary cost of running applications on public clouds will limit their use to medium-scale rather than massive-scale graph problems and constrain our experimental analysis accordingly.

II. Background and Related Work

A. *Large-scale graph processing frameworks*

Many distributed programming frameworks exist to process large datasets. These offer programming abstractions for developers to describe their algorithms and operations, and map them to the underlying distributed compute fabric for scalable and efficient execution.

MapReduce (and its variants such as iterative MapReduce) is arguably the most pervasive large-scale data processing model with its numerous implementations [12], [14], [15], [16]. Algorithms are described in two user kernels: *map* and *reduce*. The concurrent Map tasks transform input data to intermediate key-value tuples that are then grouped by key and aggregated by concurrent Reduce tasks. The framework handles the scheduling of Map and Reduce tasks, data storage and distribution between tasks through a distributed file system, and coordination of compute hosts. While MapReduce provides efficient, scalable operation for a large class of tuple-based data-intensive applications, it is less efficient for graph processing [5] [16]. This is due to the fact that Map and Reduce tasks *do not implicitly share state* between tasks or across iterations. Thus, graph algorithms represented using MapReduce (e.g. [10]) that run for many iterations incur the overhead associated with communicating the graph structure to Map or Reduce tasks at each iteration.

Bulk Synchronous Parallel & Pregel. Several graph processing frameworks have been proposed recently to specifically address these shortcomings [5] [7]. Many use a *vertex-centric* programming abstraction where computation is described for a generic vertex that operates independently and communicates with other vertices through (bulk) message-passing at synchronization points called *supersteps* [9]. The framework coordinates data and message distribution, mapping vertex tasks to compute hosts, and coordination. Further, these frameworks implicitly distribute the graph structure across hosts and provide constructs for the user program to access this structure. An example of this BSP approach is shown in the inset of Figure 1.

In this space, Google's Pregel [5] has proposed a tightly-coupled parallel abstraction for graph processing using explicit message-based communication. While the Pregel framework itself is proprietary, early public implementations of this model include Apache Hama [17], Giraph, GoldenOrb[4], and, recently, GPS [18]. The former two actually use Hadoop MapReduce as their

[3] http://incubator.apache.org/giraph

[4] http://goldenorbos.org

execution model; while not specifically designed for clouds, they can possibly run on EC2. Pregel distributes a graph's vertices across the available compute hosts with applications explicitly passing data messages along the edges of the graph. It leverages the BSP approach by aggregating outgoing message traffic and delivering it by the end of a superstep. Incoming messages are available to a vertex only at the beginning of the next superstep. This synchronized message-passing paradigm simplifies reasoning about application logic as well as coordination issues with the framework. User code simply describes how a generic vertex should process any received messages from the previous superstep (usually from its neighboring vertices), update the state of the vertex, and emit new messages to be delivered to a vertex for processing in the next superstep. Computation completes when all vertices either vote-to-halt or have no incoming messages. Graph structure and vertex state are maintained across supersteps making it more attractive than MapReduce.

Pregel's BSP approach offers scalability with the graph size but its tightly-coupled distributed architecture has two side-effects. First, communication between two vertices connected by an edge usually requires network I/O as the vertices may reside on different hosts. Even if the graph is partitioned intelligently to reduce edge-cuts (which is often not so), an all-to-all network communication between hosts is required between supersteps. Second, a superstep completes and the next initiated only when *all* vertices have completed their processing and the outgoing messages delivered. Thus, (costly) barrier synchronization is required at each step and a slow worker causes all to wait idly until it is finished.

In the BSP domain, GPS [18] is the most similar to our work. It extends the Pregel API to allow certain global computation tasks to be specified and run by a master worker. It also explores partitioning effects on BSP performance while introducing certain dynamic re-partitioning approaches. However, it uses PageRank as its most intensive algorithm and suffers from the same lack of evaluation as the Pregel model for more intensive algorithms such as betweenness-centrality.

Other frameworks. In the context of parallel graph processing, frameworks can be categorized based on their primary message buffering approach as *disk-based* or *memory-based*, with different performance and scalability properties. The common implementations of *MapReduce* offer a disk-based communication paradigm across distributed operations – data exchange between Map and Reduce phases is often through a distributed file system. The *Bulk Synchronous Parallel (BSP)* model such as Giraph and Apache Hama also utilize a disk-based approach. However, in many cases, memory-based message buffering systems can offer superior performance [7] due to its higher bandwidth compared to disk I/O. Several frameworks acknowledge this benefit and seek to exploit it by using in-memory architectures. GPS appears to use an in-memory message buffering approach. Trinity [7] goes beyond this and maintains the entire graph in an in-memory, partitioned data storage layer that facilitates either a message passing (similar to BSP) or shared-memory paradigm. While offering novel in-memory data layouts and optimizations akin to Pregel's combiners that reduce communication, it was evaluated on a cluster with high-speed interconnect rather than commodity clouds and was not publicly available at the time of this writing.

At the extreme edge of the spectrum, several frameworks store the entire graph structure in the memory of each worker machine. Using a task parallel or tuple-space approach [19] each host or processor core performs the computation for a subset of vertices in the graph (though that may require read access to the whole graph) using their own in-memory copy of the graph data structure, combining vertex results before computation completes. While using this approach – either using *large, shared memory processors* [20] *or a loosely-coupled execution model* [19] – should yield better performance it limits scalability with the graph size. As graph data sets grow into the billions of vertices and graph algorithms require significant state to be maintained per vertex, this constraint of storing the graph in a single worker becomes too restrictive and leads to virtual memory thrashing.

Alternate frameworks for distributed graph analytics include [6] and [21]. These use alternative programming abstractions and are beyond the scope of this work. Again, our work is not targeted at scaling massive graphs on large HPC systems or private datacenters with 1000's of cores of "free" cycles available, but rather focuses on medium-sized graphs that are practically suited for pay-as-you-go public clouds running on commodity hardware, and offering democratized access to a wider group of scientists.

B. Graph Algorithm and Communication Patterns

Graph analysis algorithms can be categorized based on their complexity. The ubiquitous PageRank algorithm [22] runs for many iterations with every iteration passing a message along each edge (to and from neighboring vertices). However, several interesting graph analysis algorithms exhibit greater complexity. These include betweenness-centrality (BC), all-pairs shortest paths (APSP), and community detection (CD). Often, evaluation of graph frameworks fail to consider these algorithms and instead opt for less complex algorithms such as PageRank. Consequently, frameworks that scale to billions of vertices for PageRank may fall orders of magnitude shorter for BC.

We focus our work on this class of high complexity graph algorithms and choose betweenness-centrality

(BC) as a representative algorithm for our evaluation. BC is commonly used to find key vertices in a graph, defined as those sitting upon a large percentage of shortest paths between two other vertices. Computing BC on an unweighted, undirected graph requires a breadth-first traversal be performed *from each vertex*. This often makes BC intractable for large graphs if not scaled appropriately and also serves as a stress test for a graph processing framework. For our evaluation we use a parallel BC version based on Brandës' algorithm [23], which performs a breadth-first traversal of the entire graph rooted from each vertex maintaining the number of shortest paths that pass through a given vertex and then accumulates these values by walking back up the tree formed by the traversal. Each vertex initiates a traversal and produces a score for all other vertices. These scores are then summed over all the traversals to produce a centrality score for each vertex. We include APSP in several evaluations to substantiate the results of BC and, in addition, include the PageRank algorithm since it forms a *de facto* baseline for graph frameworks.

III. BSP Framework Implementation on Azure

The original Pregel framework is proprietary to Google and the only public BSP implementations available at the time of our evaluation were layered on top of Hadoop, using the MapReduce programming model, which does not allow a fair comparison of the BSP approach. Hence, we developed a native .NET/C# implementation of a BSP graph processing framework, Pregel.NET[5], which uses the Pregel programming model and runs on the *Microsoft Windows Azure* public cloud. The Azure cloud offers a Platform-as-a-Service (PaaS) where .NET applications, called *roles*, can be run on one or more VM instances. As with other infrastructure and platform cloud providers, Azure offers services for basic network communication (virtual TCP sockets), reliable message passing (queues), and persistent storage (blob files and tables), all built for scalability.

Our Pregel.NET architecture (Figure 1) consists of three Azure roles: a *Web role* implementing a web form for job submission and status, a *job manager* role that runs on a single VM instance and coordinates the supersteps of the BSP, and a *partition worker* role, several VM instances of which hold the distributed graph partitions and perform vertex-centric tasks.

Job Manager. Requests from the user are sent from the web role to the job manager for processing through Azure queues. The manager parses the request, which specifies the desired graph application, graph file location on cloud blob (file) storage, and number of partition workers (also called "workers"). The manager replicates the request based on the desired number of workers and places them in a queue that any available worker can accept. Workers that accept the request then read the graph file from blob storage and load the vertices that belong to their partition based on different partition schemes (e.g. hash, METIS). Workers report back to the manager with their logical ID so the manager can build a mapping of these workers and broadcast the topology so that the workers can establish peer-to-peer communication channels. Computation begins with the manager placing superstep tokens into a "step" queue that workers monitor. The manager then waits for each worker to check in at the end of a superstep. If no vertex has an incoming message and all are inactive, the manager stops computation, collects results, and notifies the user via the Web UI.

Partition Workers. Partition workers (or "workers") perform compute tasks on their partition of graph vertices in each BSP superstep. Workers wait on the step queue for the beginning of a superstep, and then call a user-defined *compute()* method on each vertex in parallel using .NET's task parallel library that leverages multiple CPU cores. The *compute()* method contains the application (e.g. BC) logic and is also templatized for user-defined object types to be associated with Vertex, Edge, and Message. The framework determines if a message emitted by *compute()* is destined for a vertex on a remote worker and, if so, places it in a memory queue for network transmission. If the message

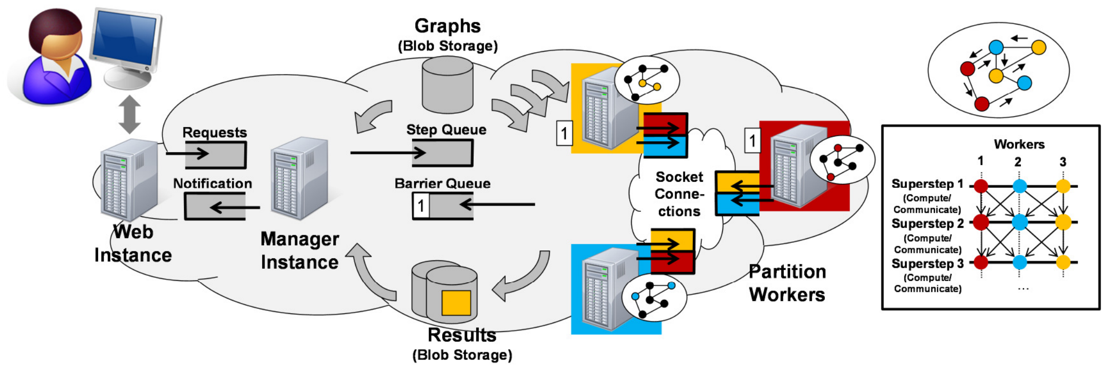

Figure 1. Pregel.NET BSP architecture on the Azure Cloud. The Pregel/BSP model partitions vertices in the graph across workers. Workers in each superstep call *compute()* for vertex and communicate emitted messages along edges, either as "bulk" data messages to remote workers or through in-memory buffers until the next superstep. Inset on right shows the BSP dataflow.

is for a local vertex, it is delivered in-memory.

Communication. There are two types of communication that take place as part of the BSP execution: control and data. *Control messages* are used to synchronize the start and end of a superstep across all worker instances. Vertices within a superstep pass *data messages* between each other and these carry user-defined message payload.

In the BSP model, a synchronization barrier at the end of a superstep is reached when two conditions are satisfied: all vertices have completed their computation and all messages generated by a source vertex have been delivered to the sink vertex. The Job Manager coordinates this barrier operation, waiting for each worker to respond with a message in a "barrier" queue. Each message indicates how many vertices are active in a worker's partition and allows the manager to determine when to halt computation. Since these messages are small and are generated only once per worker per superstep, we use Azure queues as a convenient and reliable transport.

The number of data messages generated in a superstep can be high and requires low-latency, high-bandwidth worker-to-worker communication. We implement this using Azure's TCP Endpoint socket communication between every pair of workers. This data communication stack is shared by all vertices in a particular worker (i.e. the number of sockets are a function of the number of workers). Since jobs can be long running, these connections are reestablished per-superstep to avoid socket timeouts. Message transfers can be demanding, but the BSP model does not impose any constraints on message order. Independent background threads running on multi-cores perform data communication. To better utilize network bandwidth we buffer serialized messages bound for a particular remote worker to perform "bulk" transfers. A receive thread at the remote partition worker, deserializes the messages and routes them to the target vertex's incoming queue for processing on the next superstep.

Our design inherits only the core BSP execution pattern and the Pregel graph programming model. While our work can easily be extended to support more advanced Pregel features such as combiners, aggregators and fault recovery, these are not the focus of our evaluation or optimizations and thus are omitted. The goal is to allow a fair evaluation of the Pregel BSP model on public clouds. In addition, the impact of these advanced features is algorithm dependent with some algorithms unable to exploit them fully.

IV. Vertex-Scheduling Heuristics for Optimizing BSP Graph Execution

Graph applications composed using the Pregel/BSP model encounter performance and scalability bottlenecks for applications that exhibit a log-normal distribution in messages exchanged during the supersteps when operating on small world graphs. In other words, applications like BC and all-pairs shortest path, when mapped to a BSP model, sharply ramp up the cumulative messages exchanged between supersteps as their traversals reach supernodes in the small world graph, and drain down with a long tail of supersteps whose total number typically corresponds to the graph's diameter. Figure 3 shows an example of messages exchanges per superstep for BC and APSP. These applications can be distinguished from those like PageRank that exhibit a uniform distribution in messages across supersteps. Issues with scalability for applications with variable message exchange patterns arise due to the corresponding changes in resources required over different supersteps that can lead to costly (money) over-provisioning or costly (time) under-provisioning of partition workers. Here, we introduce heuristics to optimize the costs associated with using the BSP model on public clouds for the class of graph applications that have non-uniform message patterns over supersteps, and use the BC application as an exemplar to discuss the benefits.

The number of data messages exchanged between workers across supersteps can be extremely large O(|V||E|) for BC and is a function of the graph and application. On BSP frameworks messages must be buffered since they must be delivered by the end of a superstep but not drained (processed) until the next. This can be done either using (costly) disk access or by retaining them in memory, leading to memory pressure. We abjure disk-based buffering since it uniformly adds a multiplicative overhead that is comparable to the disk-based communication of Hadoop. Hence we would like to maximize memory utilization for buffering messages *without spilling over to virtual memory on disk*, which may be even worse than disk-based buffering due to virtual memories random access (vs. sequential for disk-based buffering) I/O patterns.

Making the problem more complex is the variability in the number of messages per superstep. In graph traversal algorithms such as BC, messages are passed to the collection of newly discovered vertices, or frontier, at a particular superstep. This frontier starts as the neighbors of the source vertex but quickly grows as neighbors, in turn, emit messages to their neighbors. For graphs exhibiting small-world or scale-free structure, the vast majority of vertices will be discovered in a few steps, causing a near-exponential ramp-up in messages exchanged. Since Pregel/BSP allows a message to traverse only a single edge per superstep, we expect a peak number of bi-sectional messages to be exchanged at the superstep that roughly equals the average shortest path length. For a small-world graph with a million edges, this peak can be more than 70% of a graph's edges for a breadth first traversal. Because BC additionally performs a backward traversal of the BFS tree, this peak will then taper off as data is passed back up the tree.

The number of messages exchanged per superstep impacts network, memory and CPU resource needs. Network is obvious due to message delivery to remote vertices. The incoming messages are also buffered in memory till the next superstep starts and the user logic drains the input message queue. Further, the CPU utilization for delivering messages by our framework is comparable to the user's vertex compute logic. All of this leads to oscillations in resource utilization across supersteps. This is exacerbated by the Pregel model where all vertices logically start at the same time, leading to |V| traversals starting at the same time for BC, amplifying the spike in utilization. Consequently, the number of buffered messages can easily overwhelm the physical memory and punitively spill over to virtual memory on disk. Provisioning enough workers (aggregate memory capacity) to store the frontier of |V| traversals all at once may require a prohibitive number of *costly* cloud VM's (i.e. 1000s) that are liable to be idle during non-peak supersteps. Alternatively, this offers an opportunity to shape the resource demand by flattening and shifting this peak to allow for more uniform resource usage with better cost/performance tradeoffs.

We propose a *vertex-scheduling model* for Pregel/BSP, where computation for only a subset, or *swath*, of vertices is initiated at a time, though that computation may require action from all vertices in the graph as a traversal progresses. For BC this would mean starting traversals from the first swath of *k* vertices, allowing the computation to proceed, and then starting the next swath of vertices in succession until all |V| vertices have been initiated and completed. This approach allows resource utilization to be spread over time and shaped based on the swath size and the initiation interval between swaths. Clearly the swath size should be set as large as possible to provide good utilization but small enough to allow messages during peak supersteps to fully reside in physical memory. To this end, we propose several heuristics for choosing the swath size and deciding when to initiate the next swath.

Swath Size Heuristics. We first attempt to use intrinsic graph properties such as average edge degree or degree distribution to predict how many traversals may be performed concurrently without overflowing memory. However, this is difficult given the diversity of graphs and no consistent prediction model could be extracted. We then investigate two approaches based on runtime decisions. One is to run a few small swaths of vertices as a **sampling** run, monitoring its peak memory usage, and then extrapolate the results based on available VM memory to a static swath size for use during the rest of the computation. The alternative heuristic we propose is an **adaptive** system that allows the size to vary from one swath to the next based on the maximum memory usage between one swath initiation and the next. While many possible control strategies could be used, we select a simple linear interpolation that bases the next swath size on the peak memory usage of the previous.

Swath Initiation Heuristics. In the Pregel/BSP model, computation concludes when all vertices vote to halt and have no incoming messages to process. Thus, the baseline implementation of executing swaths would be for the entire swath of vertices to complete before starting the next swath. However, this causes resources to be underutilized for all but the peak supersteps. In addition, the more the supersteps required, the greater the synchronization overheads. Since utilization is low for the tail of a swath, we can initiate the next swath and overlap their execution. The trigger for initiating another swath can be a **static** number of supersteps or determined dynamically by monitoring the application. Initiating another swath too soon before the peak utilization of the previous swath may exacerbate the memory utilization; too late and resources are underutilized. Ideally, we want to keep memory usage near the physical memory threshold without exceeding it. Thus, setting the static initiation interval to the number of supersteps required for a swath to reach its peak utilization is desirable. An accurate prediction of this interval is not unreasonable since the average shortest path length is nearly constant over number of vertices for small world networks (think 6-degrees from Kevin Bacon) and is usually on the order of 5-10 for small-world networks. Alternatively, we propose a **dynamic** initiation heuristic that can monitor the message traffic, memory utilization, or even number of active vertices (those that have not voted to halt) to determine when the peak utilization has passed. For BC, since the number of messages will begin to decrease once a traversal reaches the bottom of its BFS tree and begins to reverse, our dynamic initiation heuristic monitors the statistics of sent messages from one superstep to the next until a superstep shows an increase followed by a decrease in message traffic i.e. a phase change in the messages transferred across supersteps.

V. Experimental Setup

We implement and run multiple graph algorithms, including BC, PageRank and All-Pairs shortest path, using our Pregel.NET BSP framework on the Microsoft Azure public cloud platform. We use four real datasets [24] of varying sizes that exhibit small-world proper-

Graph	***Vertices***	***Edges***	***90% eff. diameter***
SlashDot0922 (SD)	82,168	948,464	4.7
web-Google (WG)	875,713	5,105,039	8.1
cit-Patents (CP)	3,774,768	16,518,948	9.4
LiveJournal (LJ)	4,847,571	68,993,773	6.5

Table 1. Evaluation data sets and their properties [24]

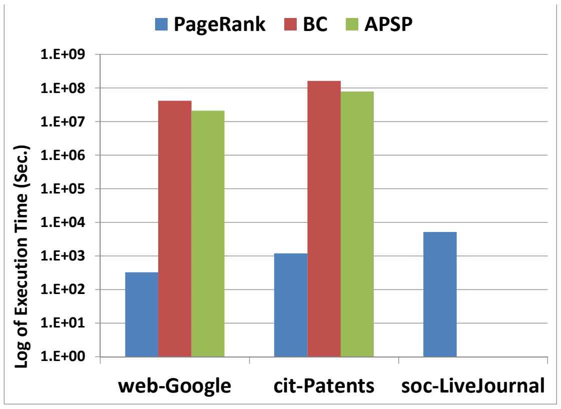

Figure 2. Total time taken (log scale) to perform PageRank, BC and All-Pairs Shortest Path (APSP) for the WG and CP graphs on 8 workers. LJ is shown for PageRank.

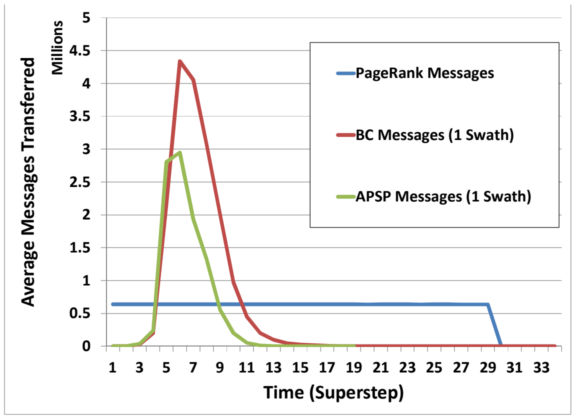

Figure 3. The average number of messages transferred per worker across supersteps for the WG graph for one static swath (BC and APSP), and for the entire graph (PageRank)

ties: SlashDot (SD) and LiveJournal (LJ) social networks, a web graph from Google (WG), and a patent citation network (CP), whose statistics are shown in Table 1. Our implementation does not perform any aggressive memory management or compression of .NET object representation of graphs; hence the sizes of graphs selected were conservative to fit available VM memory. However, these do not detract from the key benefits of the heuristics and the analysis of Pregel/BSP that we present empirically.

We use large Azure VM instances for all partition worker roles and small instances for web UI and manager roles since the latter just perform coordination. Large VMs have 4-cores rated at 1.6GHz, 7GB RAM and 400Mbps network and cost $0.48/VM-Hour. Small ones are exactly a fourth of these specifications. These extents were chosen based on the Azure resources made available as part of a research grant, but the scalability trends can be extrapolated to larger numbers of VMs. Given the time complexity of the BC algorithm, our large graph datasets can run for days or even weeks. Thus, we perform a 4-hour run for each of our experiments and extrapolate these results to the entire graph. ***Since BC traverses the entire graph rooted at each vertex, extrapolating from a subset of vertices is reasonable and was empirically verified***.

VI. Evaluation of Baseline and Heuristics

A. *Standard Graph Applications using Pregel.NET*

Our Pregel.NET BSP framework can be used to implement and run a variety of graph applications on the Azure cloud. As a basic comparison, we implement PageRank, BC, and All-Pairs Shortest Path (APSP), and evaluated their performance with two graphs, WG and CP, using 8 worker VMs. The LJ graph would not fit within the available physical memory of the workers for BC and APSP due to the large numbers of messages that they generate during their traversal. LJ was however run for PageRank since it is a less demanding application.

Figure 2 shows the time taken (log scale) to run these applications for the graph sizes. As noted before, these times are extrapolated from running smaller sampling runs over a subset of the vertices. PageRank however was run to completion over 30 iterations. As can be seen, both BC and APSP take 4 orders of magnitude longer to complete than PageRank for the same graph sizes. BC and APSP are inherently superlinear in computational complexity compared to PageRank as they perform graph traversals rooted at every vertex, while PageRank performs pairwise edge traversals from every vertex to its neighbors.

Further, if we consider the communication complexity of these applications using the BSP model, we notice that PageRank has a constant number of messages exchanged between workers across different supersteps. Figure 3 shows a straight line at ~637,000 average messages exchanged by each of eight workers for the 30 supersteps required to complete PageRank for the WG graph, leading to a uniform performance profile and predictable resource usage. However, the messages transferred for BC and APSP over different supersteps has a triangle waveform pattern, which spikes to a peak of 4.7M and 3M messages respectively – just for a single swath of seven vertices and not for all vertices in the graph. The ramp up and down of messages over supersteps, and their repetitive nature as subsequent swaths of vertices are executed, leads to non-uniform resource usage, in particular, memory usage for buffering these messages between supersteps. The communication cost of message transfer also causes BC and APSP them to take much longer to complete than PageRank.

B. *Swath Size Heuristics*

The original Pregel/BSP approach of running all vertices in parallel can be detrimental to performance because of the memory demand for buffering messages across supersteps. In fact, in a cloud setting, spilling to virtual memory can lead workers to seem unresponsive

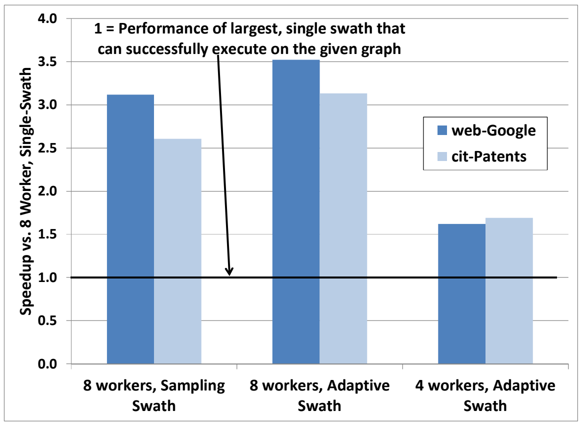

Figure 4. Speedup of Swath Size Heuristics vs. Baseline largest successful single swath size running on 8 workers for the BC application. Taller is better.

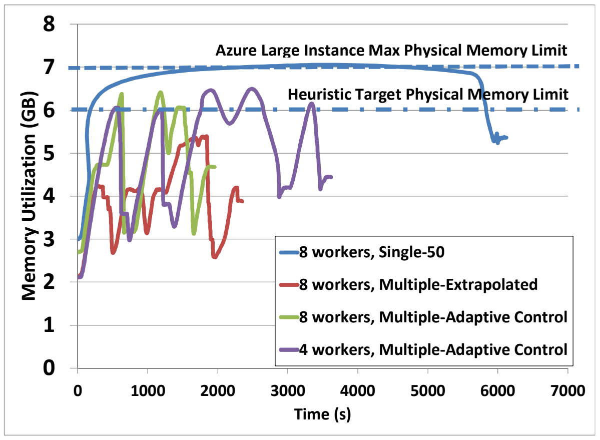

Figure 5. Memory usage over time for running BC on WG graph. Heuristics use a 6GB memory threshold. VMs have 7GB physical memory. Curves close to 6GB imply good memory utilization. Those near 7GB hit virtual memory.

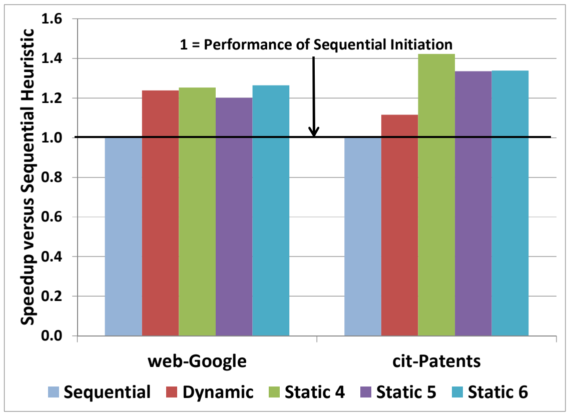

Figure 6. Speedup of Swath Initiation Heuristics vs. Baseline Sequential Initiation. All run the BC application on 8 workers. Taller is better.

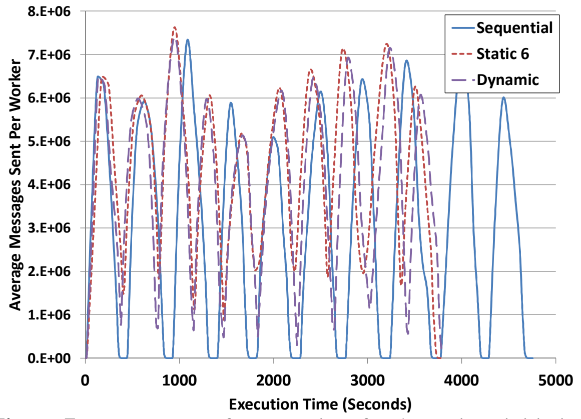

Figure 7. Message transfers over time for the various initiation heuristics for running BC on WG graph. Flatter is better.

and the cloud fabric to restart the VM, as we have observed. This is particularly a concern for applications like BC and APSP that show a non-uniform resource usage pattern. For e.g., in Fig. 15 (top), BC and APSP for the WG graph had to be run in small swaths of around 10 vertices as otherwise they would well surpass available memory during their peaking supersteps. Our swath size and initiation heuristics attempt to break vertex computations into a smaller number of swaths that are run iteratively to better control memory demand. The alternative would require scaling up to 1000's of VM's to accommodate the memory demand.

As a ***baseline***, we manually found the largest swath size we could successfully complete the BC application using 8 workers (7GB memory each) for our WG and CP graphs while allowing them to spill to virtual memory (40 and 25 swathe sizes, respectively). Next, we used our proposed swath size heuristics (***sampling heuristic*** and ***adaptive heuristic***) to pick smaller swath sizes automatically and iteratively execute each swath for the same total number of vertices as the baseline, single swath (40 and 25). In both these heuristics the target maximum VM memory utilization threshold is conservatively set to 6GB since over-estimation can be punitive. We ran these experiments using both 8 and 4 workers.

Figure 4 summarizes the relative performance gain of our swath size heuristics for performing BC on WG and CP graphs compared to the baseline approach that uses 8 workers. The sampling heuristic yields a speedup of nearly 2.5-3 versus the single large swath baseline while the dynamic heuristic yields a speedup of up to 3.5. Figure 5 illustrates the corresponding physical memory usage during these executions. The baseline spills beyond available physical memory (flat at 7GB), the dynamic heuristic stays close to the target memory threshold of 6GB while the static heuristic stays close to it, but less often so. Intuitively, *the more memory that is utilized (while staying within physical memory limits), the faster the completion time.* This allows the adaptive heuristic to execute BC on just 4 workers in roughly two-thirds the time as the baseline using 8 workers, providing users with cost-performance tradeoffs in a pay-as-you-go cloud environment. The automation offered by the adaptive heuristic to the end user also eliminates the guesswork of picking a static baseline or any potential non-uniformity in sampling using the sampling heuristic.

C. *Swath Initiation Heuristics.*

Given the need to run computation as a series of smaller (optimally sized) swaths, is important to decide when we initiate the next swath. Our initiation heuristics attempt to overlap execution of multiple swaths to flatten the resource (memory, network, CPU) usage variations causes by different supersteps within a single swath. In BC and APSP, we observe a triangle waveform with a central peak; this heuristic is not relevant for applications like PageRank with uniform resource usage. Besides improving resource utilization, overlapping consecutive swath iterations also reduce the cumulative supersteps required and thus reduces the total overhead spent on synchronization between supersteps.

Figure 6 compares the relative performance of our initiation heuristics for the BC application normalized to a baseline approach that runs strictly ***sequentially*** non-overlapping iterations. These run on 8 workers. Figure 7 shows the corresponding messages transferred between supersteps over time, spanning swath iterations. The ***Static-N heuristic*** initiates a new swath every '*N*' supersteps while the ***Dynamic heuristic*** performs initiation when it detects a peak in the number of messages exchanged. Static-N's performance depends on the graph and the value of '*N*' that is chosen. If the average shortest path is greater than '*N*', we will be initiating new heuristics before the previous swath has hit its peak, thereby exacerbating the resource demand. If the average shortest path length is well distributed or is (just) shorter than *N*, it leads to better performance. So *N*=4 for the larger CP graph actually works best. Our dynamic heuristic eliminates this guesswork as it picks the initiation point at runtime without user input or graph preprocessing. Using this dynamic initiation heuristic we achieve up to 24% speedup vs. sequential initiation for the WG graph. The message transfer plot in Figure 7 corroborates this. While sequential shows the message transfers peak and fall to zero (thus showing more variability and poorer utilization), Static-6 (which is optimal but hand-selected) maintains a higher message rate while dynamic is a bit more conservative, but automated.

VII. Evaluating Impact of Graph Partitioning on Pregel.NET

Our Pregel.NET framework is agnostic to how the graphs are partitioned and assigned to workers. The default mode performs a simple hash over the vertex ID to determine the target worker partition. Several works have shown that intelligent graph partitioning can improve the performance of distributed graph algorithms [18] [25], and it is relevant to examine if these benefits carry over to the Pregel/BSP model also. METIS is a commonly used strategy that provides good quality in-place partitioning that minimizes edge-cuts across partitions [26]. Recent work on approximate partitioning using a single graph scan offers an alternative for partitioning online as the graph is read from storage [25]. PageRank is often used in literature to validate the effectiveness of these partitioning strategies. However, as we have seen, PageRank implemented using Pregel/BSP has a uniform message profile while BC and APSP have a triangle waveform message profile. We analyze the consequence of this on the performance gains from intelligent graph partitioning.

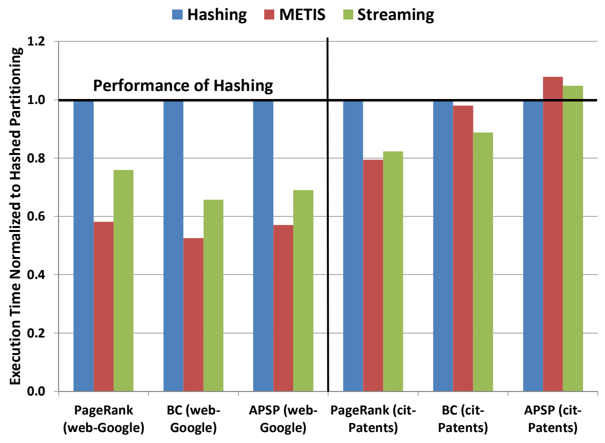

Figure 8. Relative time taken by PageRank, APSP and BC to run on WG and CP graphs partitioned using METIS and Streaming, normalized to Hashing approach. Smaller is better.

Clearly, the benefit of partitioning comes in reduced communication time since messages to remote vertices incur additional delay due to serialization and network I/O when compared to in-memory messages sent to local vertices. Since many distributed graph algorithms are dominated by communication rather than computation, partitioning can improve overall performance. However, the barrier synchronization model in Pregel/BSP means that the total time spent in a superstep is determined by the slowest worker in the superstep. Hence, the balance of work amongst workers in a superstep is as important as the cumulative number of remote messages generated in a superstep. Since vertices communicate with their neighbors along edges in the Pregel/BSP model and partitioning seeks to collocate a majority vertex neighbors in the same partition, there may arise "local maximas" in specific partitions where more vertices are active during the course of execution of a graph application. This difference in workload can cause underutilization of workers that wait for overutilized workers at the superstep barrier.

We evaluate the impact of graph partitioning using the best-in-class *in-place* METIS partitioner as well as the best heuristic (linear-weighted deterministic, greedy approach) *streaming* partitioner from [25] and compare them against a baseline that uses simple *hashing* of vertices by their IDs. We run PageRank, BC and APSP over the WG and CP graphs on 8 workers for this evaluation. Hash, METIS, and Streaming produce 8 partitions whose percentage of remote edges are 87%, 18% and 35% for the WG graph and 86%, 17% and 65% for the CP graph; smaller this number, lower the edge cuts across partitions, and METIS proves a low edge cut for both graphs. Given the large sizes of the graphs, we run these experiments on the same set of vertices as our other experiments (50 vertices for CP and 75 vertices for WG). We report these results when using Pregel.NET without our swath heuristics; however, the trends we observe are consistent even with heuristics turned on, though the absolute performance is uniformly better.

Figure 8 shows the relative time taken when using the METIS and streaming partitioning normalized to hashing for PageRank, BC and APSP running on WG and CP. We see that the WG graph sees a relative improvement of nearly 42-50% for METIS for the three applications, while this improvement drops to 24-35% for the streaming partitioning. When running Pregel.NET with heuristics turned on, we see a best case improvement of 5x in relative time taken by METIS for BC on WG compared to hashing (graph not shown). These are consistent with results reported in [18].

However, we also see that the CP graph does not show such a marked improvement in performance due to better partitioning, despite its edge cut ratios from different partitioning being similar to WG. In fact, hashing is faster than METIS and Streaming for APSP on this graph. It is worthwhile to investigate this consistent lack of improvement for the CP graph as opposed to WG. Figure 9 shows the runtime for BC broken into compute+I/O time and the synchronization barrier wait time components for the WG graph and Figure 12 does the same for CP. The plots also show the VM utilization %, calculated as the time spent in compute and I/O communication against the total time (including barrier wait time) on the secondary Y-axis. We see that the VM utilization % for hashing is higher though the total time taken is also higher, for both WG and CP. METIS shows the inverse property, having lower utilization but also lower total time. This is explained by looking at the number of messages emitted by workers in a superstep for both hashing and METIS, shown in Figures 10 and 11 for WG, and in Figures 13 and 14 for CP. We expect that a hashed assignment of vertices to a partition would spread communication roughly evenly over all workers, while also increasing the number of remote communications required. The latter contributes to the increased total time while the former leads to a uniform number of messages seen for all workers in a superstep (Figures 10 and 13). When looking at the messages sent by workers in a superstep for METIS, we see that there are message load imbalances within workers in a superstep, caused due to concentration of vertices being traversed in that superstep in certain partitions. This variability is much more pronounced in CP as compared to WG (Figures

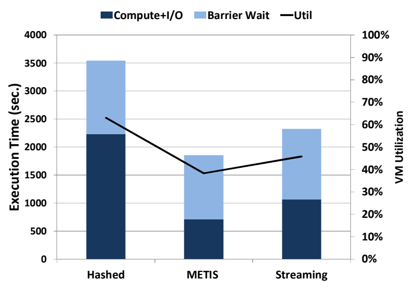

Figure 9. *Total time* taken for BC on a subset of *WG graph* with *different partitioning*. *Utilization* shows the ratio of Compute+I/O time to total time.

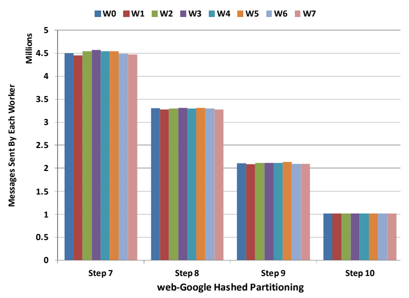

Figure 10. *Number of messages* transferred by each worker in the peak supersteps of BC performed over *WG graph* using *Hash partitioning*.

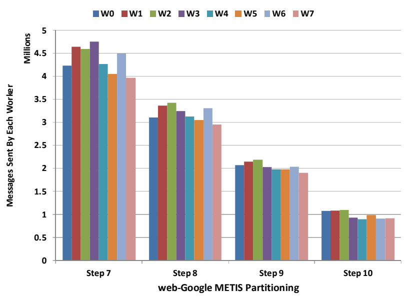

Figure 11. *Number of messages* transferred by each worker in the peak supersteps of BC performed over *WG graph using METIS partitioning*.

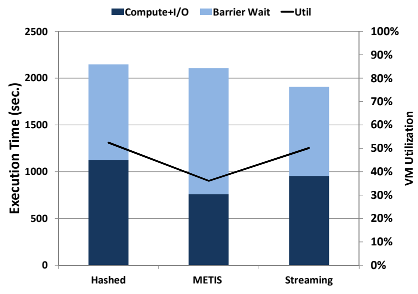

Figure 12. *Total time* taken for BC on a subset of *CP graph* with *different partitioning*. *Utilization* shows the ratio of Compute+I/O time to total time.

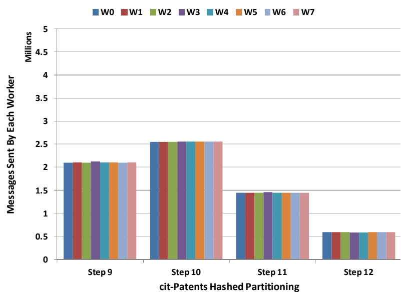

Figure 13. *Number of messages* transferred by each worker in the peak supersteps of BC performed over *CP graph* using *Hash partitioning*.

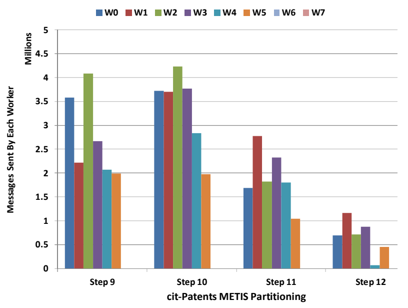

Figure 14. *Number of messages* transferred by each worker in the peak supersteps of BC performed over *CP graph using METIS partitioning*.

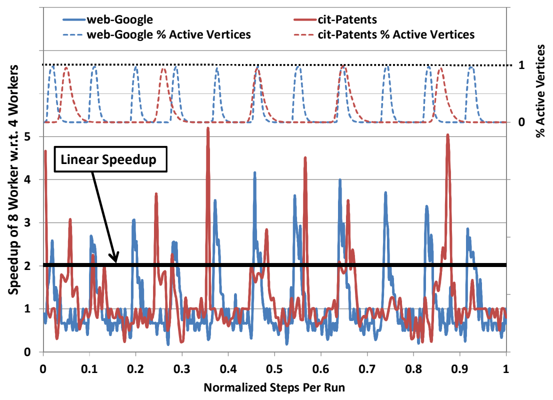

Figure 15. ***(Bottom)*** plot shows speedup of 8 workers relative to 4 workers, for each superstep, when running BC on WG and CP graphs. ***(Top)*** plot shows the number of vertices active in that superstep.

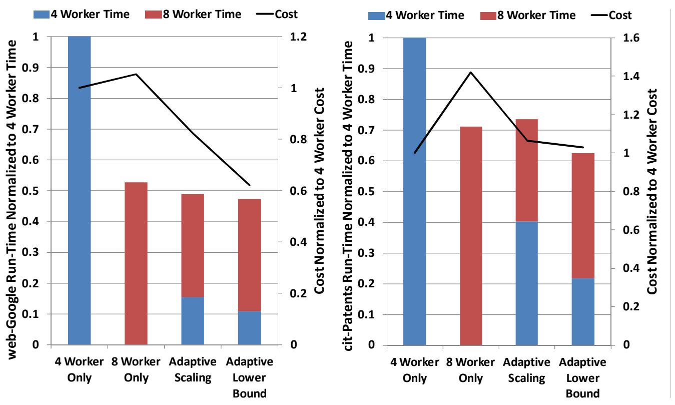

Figure 16. Estimated time for BC using elastic scaling, normalized to time taken for 4 workers. Normalized cost is shown on secondary Y axis. ***(A)*** WG graph shown on left, ***(B)*** CP graph shown on right. Smaller is better.

11 and 14). E.g. in superstep 9 for CP, twice as many messages (4M) are generated by a worker compared to another (2M). For Pregel/BSP, the time taken in a superstep is determined by the slowest worker in that superstep. Hence increased variability in CP causes even "good" partitioning strategies to cause an increase in total execution time when using the Pregel/BSP model.

VIII. Analysis of Elastic Cloud Scaling

Cloud environments offer elasticity – the ability to scale-out or scale-in VMs on-demand and only pay for what one uses [27]. On the flip side, one ends up paying for VMs that are acquired even if they are underutilized. We have already shown the high variation in compute/memory resources used by algorithms like BC and APSP across different super-steps. While our earlier swath initiation heuristics attempt to flatten these out by overlapping swath executions, one can consider leveraging the cloud's elasticity to, instead, scale up and down the concurrent workers (and graph partitions) allocated in each superstep.

The peak and trough nature of resource utilization combined with Pregel/BSP's synchronous barrier between supersteps offers a window for dynamic scale-out and –in at superstep boundaries. Peak supersteps can greatly benefit from additional workers, while those same workers will contribute to added synchronization overhead for trough supersteps. *Our hypothesis is that an intelligent adaptive scaling of workers can achieve a similar performance as a large, fixed number of workers, but with reduced cost.* We offer an analysis of the potential benefits of elastic scaling by extrapolating from observed results for running BC on WG and CP graphs, using four and eight workers. To provide a fair and focused comparison, we turned off swath heuristics in favor of fixed swath sizes and initiation intervals.

Figure 15 (Bottom) plots the speedup of BC running on eight workers when normalized to BC running on four workers, at corresponding supersteps. The number of workers does not impact the number of supersteps. We also plot the number of active vertices (i.e. vertices still computing for a given swath) at these supersteps, which is a measure of how much work is required (Fig. 15 (Top)). We find that we occasionally get superlinear speedup spikes (i.e. >2x) that shows a strong correlation with the peaks of active messages, for both WG and CP graphs. At other times, the speedup is sublinear or even a speed-down (i.e. <1), corresponding to inactive vertices. The superlinear speedup is attributable to the lower contention and reduced memory pressure for 8 workers when the active vertices peak (similar to what we observed for the swath initiation heuristics). Similarly, the below par speedup during periods of low activity is contributed by the increased overhead of barrier synchronization across 8 workers. Intuitively, by dynamically scaling up the number of workers for supersteps with peaking active vertices and scaling them down otherwise, we can leverage the superlinear speedup and get more value per worker.

Using a threshold of 50% active vertices as the threshold condition for ***dynamically scaling*** between 4 and 8 workers in a superstep, we extrapolate the time per superstep and compared this to the fixed 4 and 8 worker runtimes. We also compute the best-case run time using an "oracle" approach to ***ideal scaling*** i.e. for each superstep, we pick the minimum of the 4 or 8 worker's time. Note that these projections do not yet consider the overheads of scaling, but are rather used to estimate the potential upside if we had an ideal or an automated heuristic for scaling. The total time estimates for running BC on WG and CP graphs, normalized to observed time taken using 4 workers, are plotted in Figures 16(A) and 16(B).

We see that our dynamic scaling heuristic using the percentage of active vertices achieves nearly the same (CP) or better (WG) performance as a fixed 8 worker approach. Clearly there is benefit of using fewer workers for low utilization supersteps to eliminate the barrier synchronization overhead. Also, the dynamic scaling heuristic performs almost as well as the ideal scaling. Finally, when we consider the monetary cost of the proposed approaches, assuming a pro-rata normalized cost per VM-second plotted on the secondary Y axis, we see that dynamic scaling is comparable (CP) or cheaper (WG) than a 4 worker scenario while offering the performance of an 8 worker deployment.

IX. Conclusion

In conclusion, we introduce optimization and heuristics for controlling memory utilization and show they are critical to performance. By breaking computation into swaths of vertices and using our sizing heuristics, we achieve up to 3.5x speedup over the maximum swath size that does not cause the a failure. In addition, overlapping swath executions can provide a 24% gain with automated heuristics and even greater speedup when *a priori* knowledge of the network characteristics is applied.

This evaluation offers eScience users with problems that map to massive graphs, such as in social sciences and biology, to help make framework selection and cost-performance-scalability trade-offs – the manual overhead of rewriting an application for a framework can be high while access to public clouds costs real money. Our heuristics are generalizable and can be leveraged by other BSP and distributed graph frameworks, and for graph applications beyond BC.

X. Bibliography

[1] F. Liljeros, C. Edling, L. Amaral, H. Stanley, and Y. Åberg, "The web of human sexual contacts," *Nature*, vol. 411, pp. 907-908, 2001.

[2] H. Jeong, S. Mason, A.-L. Barabási, and Z. Oltvai, "Lethality and centrality in protein networks," *Nature*, vol. 411, pp. 41-42, 2001.

[3] O. Bodin and E. Estrada, "Using network centrality measures to manage landscape connectivity," *Ecological Applications*, vol. 18, no. 7, pp. 1810-1825, October 2008.

[4] D. Watss and S. Strogatz, "Collective dynamics of 'small-world' networks," *Nature*, vol. 393, no. 6684, pp. 440–442, June 1998.

[5] G. Malewicz, M. Austern, A. Bik, J. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *ACM International Conference on Management of Data (SIGMOD)*, 2010.

[6] D. Gregor and A. Lumsdaine, "The parallel BGL: A generic library for distributed graph computations," in *In Parallel Object-Oriented Scientic Computing (POOSC).*, 2005.

[7] B. Shao, H. Wang, and Y. Li, "The Trinity Graph Engine," Microsoft Research, Technical Report MSR-TR-2012-30, March 2012.

[8] S. Ghemawat and J. Dean, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 3, pp. 107-113, 2008.

[9] L. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103-111, 1990.

[10] J. Lin and M. Schatz, "Design patterns for efficient graph algorithms in MapReduce," in *ACM Workshop on Mining and Learning with Graphs*, 2010.

[11] Apache Hama. [Online]. http://hama.apache.org/

[12] Apache Hadoop. [Online]. http://hadoop.apache.org/

[13] M. Zaharia, M. Chowdhury, M. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster Computing with Working Sets," in *HotCloud*, 2010.

[14] J. Ekanayake, et. al., "Twister: A runtime for iterative MapReduce," in *Proceedings of the 19th ACM International Symposium on High PErformance Distributed Computing (HPDC)*, Chicago, 2010, pp. 810-818.

[15] U. Kang, C. Tsourakakis, and C. Faloutsos, "Pegasus: Mining Peta-scale Graphs," in *Knowledge and Information Systems (KAIS)*, 2010.

[16] M. Pace, "BSP vs. MapReduce," in *International Conference on Computational Science*, vol. 103.2081, 2012.

[17] S. Seo, E. Yoon, J. Kim, S. Jin, J-S. Kim, and S. Maeng, "HAMA: An Efficient matrix computation with the MapReduce framework," in *IEEE International Conference on Cloud Computing Technology and Science*, 2010, pp. 721-726.

[18] S. Salihoglu and J. Widom, "GPS: A Graph Processing System," Stanford University, Technical Report 2011.

[19] R. Lichtenwalter and N. Chawla, "DisNet: A framework for distributed graph computation," in *ACM/IEEE Conference on Advances in Social Network Analysis and Mining (ASONAM),* 2011.

[20] K. Madduri, D. Ediger, K. Jiang, D. Bader, and D. Chavarria-Miranda, "A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2009.

[21] E. Krepska, T. Kielmann, W. Fokkink, H. Bal, "A high-level framework for distributed processing of large-scale graphs," in *International Conference on Distributed Computing and Networking*, 2011, pp. 155-166.

[22] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank citation ranking: Bringing order to the web," Stanford InfoLab, Technical Report 1999-66, 1999.

[23] U. Brandës, "A faster algorithm for betweenness centrality," *Journal of Mathematical Sociology*, vol. 25, no. 2, pp. 163-177, 2001.

[24] Stanford Network Analysis Project. [Online]. http://snap.stanford.edu/

[25] I. Stanton and G. Kliot, "Streaming Graph Partitioning for Large Distributed Graphs," Microsoft Corp., Technical Report MSR-TR-2011-121, 2011.

[26] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," in *International Conference on Parallel Processing*, 1995, pp. 113-122.

[27] M. Armbrust, et al., "A view of cloud computing," *Communications of the ACM*, vol. 53, no. 0001-0782, pp. 50-58, April 2010.