GPS: A Graph Processing System*

Semih Salihoglu and Jennifer Widom

Stanford University

{semih,widom}@cs.stanford.edu

Abstract

GPS (for *Graph Processing System*) is a complete open-source system we developed for scalable, fault-tolerant, and easy-to-program execution of algorithms on extremely large graphs. GPS is similar to Google's proprietary *Pregel* system [MAB+11], with some useful additional functionality described in the paper. In distributed graph processing systems like GPS and Pregel, *graph partitioning* is the problem of deciding which vertices of the graph are assigned to which compute nodes. In addition to presenting the GPS system itself, we describe how we have used GPS to study the effects of different graph partitioning schemes. We present our experiments on the performance of GPS under different *static* partitioning schemes—assigning vertices to workers "intelligently" before the computation starts—and with GPS's *dynamic repartitioning* feature, which reassigns vertices to different compute nodes during the computation by observing their message sending patterns.

1 Introduction

Building systems that process vast amounts of data has been made simpler by the introduction of the *MapReduce* framework [DG04], and its open-source implementation *Hadoop* [HAD]. These systems offer automatic scalability to extreme volumes of data, automatic fault-tolerance, and a simple programming interface based around implementing a set of functions. Unfortunately, for reasons outlined below, these systems are not always suitable when the data being processed is in the form of an extremely large graph.

*This work was supported by the National Science Foundation (IIS-0904497), a KAUST research grant, and a research grant from Amazon Web Services.

At the same time, graph data is becoming more and more prevalent, e.g., the web graph, social networks, blogs, instant-messaging systems, consumer-product relations on e-commerce web sites, and others. Furthermore, the types of algorithms people wish to run over these enormous graphs are becoming more and more complex, e.g., ranking web-sites, identifying important e-commerce products, recommending friendships in social networks. A framework similar to MapReduce—scalable, fault-tolerant, easy to program—but geared specifically towards graph data, would be of immense use. Over the last year we have been implementing the initial version of such a system; we call it *GPS*, for *Graph Processing System*.

The design of GPS has drawn from the earlier *Pregel* [MAB+11] system, a *distributed message-passing system*, in which the vertices of the graph are distributed across machines, and send each other messages to perform the computation. There are three main differences between Pregel and GPS:

- GPS is open-source.
- While only fully vertex-centric algorithms can be efficiently implemented with the Pregel API, the GPS API has an extension that enables efficient implementations of all algorithms composed of one or more vertex-centric computations, combined with global computations. We are currently adding the same extension to *Giraph*, an open-source Apache Incubator project that has also drawn from Pregel. We explain the Pregel API and our extension momentarily.
- Unlike Pregel, GPS can dynamically repartition the graph across machines during the computation to reduce communication and improve performance.

In addition to building the system itself, using GPS we have made progress on the following *graph partitioning* question: When performing large-scale computations on large-scale graphs in a distributed message-passing setting, how do we decide which vertices of the graph reside on which machines? Will some algorithms perform much better if we "intelligently" assign vertices to machines before the computation begins? As an example, consider running the PageRank [BP98] algorithm on the web graph. How fast would the algorithm run and how would the performance change if we partition the web-pages according to their domains, i.e., if all web-pages with the same domain names reside on the same machine, as opposed to partitioning the graph randomly or using the popular METIS [MET] algorithm? Furthermore, is it helpful to move vertices from machine to machine during the computation (*dynamic repartitioning*) by observing communication behavior? Suppose at certain intervals, every vertex moves itself to the machine that would

minimize the number of messages it sends across the network based on communication behavior so far. Is it possible to have an efficient and scalable implementation of this greedy scheme and, if so, would the performance change significantly? We will see that the answers to all of these questions are positive, in certain settings.

Next we explain why the MapReduce framework is not suitable "as is" for running graph algorithms. Then we describe the alternative computational framework used by Pregel and GPS. Then we motivate the API extension in GPS. Finally, we outline our main contributions: the open-source GPS system itself, and our algorithms and experiments related to how graphs are partitioned, and possibly repartitioned, for coaxing maximum efficiency out of the GPS system.

Many graph algorithms, e.g., computing PageRank [BP98] or clustering algorithms based on spectral partitioning, as in [HGO+10], [LC10], are iterative. For example, the common implementation of PageRank starts by assigning initial PageRank values to each vertex in the graph and updates these values in iterations until they converge. MapReduce is a two-phased computational model and as such cannot express iterative computations. One approach to solve this limitation has been to build systems on top of Hadoop, in which the programmer can express a graph algorithm as a series of MapReduce jobs, each one corresponding to one iteration of the algorithm. Pegasus [KTF09], HaLoop [BHBE10], iMapReduce [ZGGW11] and Surfer [CWHY10] are examples of such systems. This approach suffers from 2 inefficiencies resulting from the batch nature of MapReduce: (1) At the end of each MapReduce job the entire graph, which does not change from iteration to iteration, and other intermediate state information (e.g., PageRank values of vertices) that does change must be written to disk to be read by the next MapReduce job. (2) Checking for the convergence criterion may require additional MapReduce jobs.

In contrast to MapReduce, the model introduced by Pregel and used by GPS is inherently iterative and is based on the *Bulk Synchronous Parallel (BSP)* computation model [Val90]. In the beginning of the computation, the vertices of the graph are distributed across machines. Computation consists of iterations called *supersteps*. At the end of each superstep, all machines synchronize before starting the next superstep. In each superstep, a user-specified *vertex.compute()* function is applied to each vertex in parallel. Inside the *vertex.compute()* function, the vertices update their state information, send other vertices messages to be used in the next iteration, and check for the convergence criterion. The iterations stop when all vertices have converged. This model has three benefits compared to the Hadoop-based systems: (1) It is inherently iterative. (2) The graph

and the state data remain in memory throughout the computation. (3) Checking for the convergence criterion happens inside the *vertex.compute()* function.

Implementing a graph algorithm inside *vertex.compute()* is ideal for certain graph algorithms, such as computing PageRank, finding shortest paths, or finding connected components, all of which can be performed in a fully "vertex-centric" and hence parallel fashion. However, some algorithms are a combination of vertex-centric (parallel) and global (sequential) computations.

As an example, consider the following k-means-like graph clustering algorithm that consists of four parts: (a) pick k random vertices as "cluster centers", a computation global to the entire graph, (b) assign each vertex to a cluster center, a vertex-centric computation, (c) assess the goodness of the clusters by counting the number of edges crossing clusters, a vertex-centric computation, (d) decide whether to stop, if the clustering is good enough, or go back to (a), a global computation.

Vertex-centric computations can be easily and efficiently implemented inside *vertex.compute()*. One might want to implement global computations also inside *vertex.compute()* by designating a "master" vertex to run these computations. However, this approach has 2 problems: (1) Local computations may depend on the results of global computations. Therefore during the superstep that the "master" vertex executes the global computation, all other vertices would be idle, wasting resources. (2) The *vertex.compute()* code becomes harder to understand as it contains some sections that are written for all vertices and others that are written for the special vertex.

To incorporate global computations, GPS extends the API of Pregel with an additional function, *master.compute()*, invoked once in the beginning of each superstep, immediately after all the workers synchronize. Using *master.compute()* in addition to *vertex.compute()*, we have been able to implement a large number of algorithms that would be difficult in Pregel, including spectral partitioning using k-means clustering, a recursive spectral partitioning algorithm, and the well known social network analysis algorithm betweenness centrality [MEJ+09].

The specific contributions of this paper are as follows:

1. In Section 2, we present GPS, an open-source Pregel-like distributed message passing system. We present the architecture and the programming API.

2. In Section 3, we study how different static graph partitioning schemes, i.e. those that assign vertices to machines before the computation starts, affect the network and runtime time performance of GPS when executing different algorithms. We repeat some

of our experiments using *Giraph* [GIR] and report the results. We also describe and report experiments using *LALP* (large adjacency list partitioning), an optimization which partitions the adjacency lists of high-degree vertices across all workers.

3. In Section 4, we describe a greedy dynamic partitioning scheme, in which the vertices of the graph move across machines in each iteration by observing their own communication patterns. We repeat several of our experiments from Section 3 using dynamic repartitioning.

4. We finish by discussing several additional optimizations that reduce memory and increase the overall speed of GPS.

2 GPS System

GPS uses the distributed message-passing model of Pregel [MAB+11], which is based on bulk synchronous processing [Val90]. We give an overview of the model here and refer the reader to [MAB+11] for details. Broadly, the input is a directed graph, and each vertex of the graph maintains a user-defined *value* and a *flag* indicating whether or not the vertex is active. Optionally, edges may also have values. The computation proceeds in iterations called *supersteps*, terminating when all vertices are inactive. Within a superstep i, each active vertex u in parallel: (a) looks at the messages that were sent to u in superstep $i-1$; (b) modifies its value; (c) sends messages to other vertices in the graph and optionally becomes inactive. A message sent in superstep i from vertex u to vertex v becomes available for v to use in superstep $i+1$. The behavior of each vertex is encapsulated in a function *vertex.compute()*, which is executed exactly once in each superstep.

2.1 Overall Architecture

The architecture of GPS is shown in Figure 1. As in Pregel, there are two types of *processing elements (PEs)*: one master and k workers, $W_0 ... W_{k-1}$. The master maintains a mapping of PE identifiers to physical compute nodes and workers use a copy of this mapping to communicate with each other and the master. PEs communicate using Apache MINA [MIN], a network application framework built on java.nio, Java's asynchronous network I/O package. GPS is implemented in Java. The compute nodes run *HDFS (Hadoop Distributed File System)* [HDF], which is used to store persistent data such as the input graph and the checkpointing files. We next explain how the input graph is partitioned across workers.

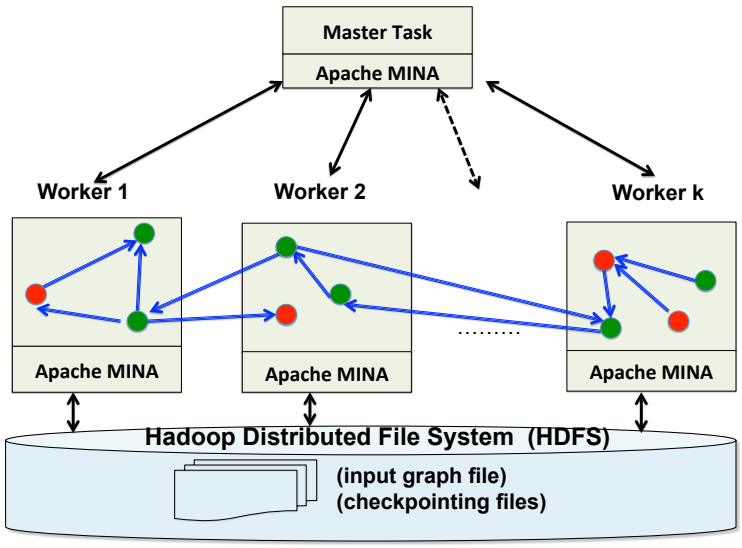


Figure 1: GPS Architecture

The master and worker implementations are described in Section 2.3. Section 2.4 explains the API and provides examples.

2.2 Input Graph Partitioning Across Workers

The input graph G is specified in HDFS files in a simple format: each line starts with the ID of a vertex u, followed by the IDs of u's outgoing edges. The input file may optionally specify values for the vertices and edges. GPS assigns the vertices of G to workers using the same simple round-robin scheme used by Pregel: vertex u is assigned to worker $W_{(u \mod k)}$. When we experiment with more sophisticated partitioning schemes (Section 3), we run a preprocessing step to assign node IDs so that the round-robin distribution reflects our desired partitioning. GPS also supports optionally repartitioning the graph across workers during the computation, described in Section 4.

2.3 Master and Worker Implementation

The master and worker *PEs* are again similar to Pregel [MAB+11]. The master coordinates the computation by instructing workers to: (a) start parsing input files; (b) start a new superstep; (c) terminate computation; and (d) checkpoint their states for fault-tolerance. The master awaits notifications from all workers before instructing workers what to do next, and so serves as the centralized location where workers synchronize between supersteps.

The master also calls a *master.compute()* function at the beginning of each superstep, described in Section 2.4.

Workers store vertex values, active flags, and *message queues* for the current and next supersteps. Each worker consists of three "thread groups", as follows.

1. A *computation thread* loops through the vertices in the worker and executes *vertex.compute()* on each active vertex. It maintains an outgoing *message buffer* for all workers in the cluster, including itself. When a buffer is full it is either given to *MINA threads* for sending over the network, or passed directly to the local *message parser thread*.

2. *MINA threads* send and receive message buffers, as well as simple *coordination messages* between the master and the worker. When a message buffer is received, it is passed to the message parser thread.

3. A *message parser thread* parses incoming message buffers into separate messages and enqueues them into the receiving vertices' message queues for the next superstep.

One advantage of this thread structure is that there are only two lightweight points of synchronization: when the computation thread passes a message buffer directly to the message parser thread, and when a MINA thread passes a message buffer to the message parser thread. Since message buffers are large (the default size is 100KB), these synchronizations happen infrequently.

2.4 API

Similar to Pregel, the programmer of GPS subclasses the *Vertex* class to define the vertex value, message and optionally edge value types. The programmer codes the vertex-centric logic of the computation by implementing the *vertex.compute()* function. Inside *vertex.compute()*, vertices can access their values, their incoming messages, and a map of *global objects*—our implementation of the *aggregators* of Pregel. Global objects are used for coordination, data sharing, and statistics aggregation. At the beginning of each superstep, each worker has the same copy of the map of global objects. During a superstep, vertices can update objects in their worker's local map, which are merged at the master at the end of the superstep, using a user-specified merge function. When ready, a vertex declares itself inactive by calling the *voteToHalt()* function in the API.

```
1  public class HCCVertex extends Vertex<IntWritable, IntWritable> {
2    @Override
3    public void compute(Iterable<IntWritable> messages, int superstepNo) {
4      if (superstepNo == 1) {
5        setValue(new IntWritable(getId()));
6        sendMessages(getNeighborIds(), getValue());
7      } else {
8        int minValue = getValue().value();
9        for (IntWritable message : messages) {
10         if (message.value() < minValue) {
11           minValue = message.value(); }}
12       if (minValue < getValue().getValue()) {
13         setValue(new IntWritable(minValue));
14         sendMessages(getNeighborIds(), getValue());
15       } else {
16         voteToHalt(); }}}}
```

Figure 2: Connected components in GPS.

```
1  Input: undirected G(V, E), k, τ
2   int numEdgesCrossing = INF;
3   while (numEdgesCrossing > τ)
4     int [] clusterCenters = pickKRandomClusterCenters(G)
5     assignEachVertexToClosestClusterCenter(G, clusterCenters)
6     numEdgesCrossing = countNumEdgesCrossingClusters(G)
```

Figure 3: A simple k-means like graph clustering algorithm.

Algorithms whose computation can be expressed in a fully vertex-centric fashion are easily implemented using this API, as in our first example.

Example 2.1 *HCC* [KTF09] is an algorithm to find the weakly connected components of an undirected graph: First, every vertex sets its value to its own ID. Then, in iterations, vertices set their values to the minimum value among their neighbors and their current value. When the vertex values converge, the value of every vertex v is the ID of the vertex that has the smallest ID in the component that v belongs to; these values identify the weakly connected components. HCC can be implemented easily using *vertex.compute()*, as shown in Figure 2. □

A problem with this API (as presented so far) is that it is difficult to implement algorithms that include global as well as vertex-centric computations, as shown in the following example.

Example 2.2 Consider the simple k-means like graph clustering algorithm introduced in Section 1 and outlined in Figure 3. This algorithm has two vertex-centric parts:

```
1  public class ClusteringVertex extends Vertex<TwoIntWritable, TwoIntWritable> {
2    @Override
3    public void compute(Iterable<TwoIntWritable> messages, int superstepNo){
4      if (superstepNo == 1) {
5        Set<Integer> clusterCenters = getGlobalObjects(“cluster−centers”);
6        setValue(clusterCenters.contains(getId()) ?
7          new TwoIntWritable(0, getId()) :
8            new TwoIntWritable(Integer.MAX_VALUE, null));
9        if (clusterCenters.contains(getId())) {
10         sendMessages(getNeighborIds(), getValue());}
11     } else {
12       int minDistance = getValue().value().fst;
13       int minDistanceClusterId = getValue().value().snd;
14       for (TwoIntWritable message : messages) {
15         if (message.value().fst < minValue) {
16           minValue = message.value();
17           minDistanceClusterId = message().value().snd;}}
18       if (minDistance < getValue().value().fst) {
19         setValue(new TwoIntWritable(minDistance, minDistanceClusterId));
20         sendMessages(getNeighborIds(), getValue());
21       } else {
22         voteToHalt(); }}}}
```

Figure 4: Assigning each vertex to the closest cluster with *vertex.compute()*.

1. Assigning each vertex to the closest “cluster center” (line 5 in Figure 3). This process is a simple extension of the algorithm from [MAB+11] to find shortest paths from a single source and is shown in Figure 4.

2. Counting the number of edges crossing clusters (line 6 in Figure 3). This computation requires two supersteps; it is shown in Figure 5.

Now consider lines 2 and 3: checking the result of the latest clustering and terminating if the threshold has been met, or picking new cluster centers. With the API so far, we must put this logic inside *vertex.compute()* and designate a special *“master”* vertex to do it. Therefore, an entire extra superstep is spent at each iteration of the while loop (line 3 in Figure 3) to do this very short computation at one vertex (with others idle). Global objects cannot help us with this computation, since they only store values. □

In GPS, we have addressed the shortcoming illustrated in Example 2.2 by extending the Pregel API to include an additional function, *master.compute()*. The programmer subclasses the *Master* class, and implements the *master.compute()* function, which gets called at the beginning of each superstep. The *Master* class has access to all of the merged

```
1  public class EdgeCountingVertex extends Vertex<IntWritable, IntWritable> {
2    @Override
3    public void compute(Iterable<IntWritable> messages, int superstepNo){
4      if (superstepNo == 1) {
5        sendMessages(getNeighborIds(), getValue().value());
6      } else if (superstepNo == 2) {
7        for (IntWritable message : messages) {
8          if (message.value() != getValue().value()) {
9            minValue = message.value();
10           updateGlobalObject(“num−edges−crossing−clusters”,
11             new IntWritable(1));}}
12       voteToHalt(); }}}
```

Figure 5: Counting the number of edges crossing cluster with *vertex.compute()*.

global objects, and it can store its own global data that is not visible to the vertices. It can update the global objects map before it is broadcast to the workers.

Figure 6 shows an example *master.compute()*, used together with the vertex-centric computations already described (encapsulated in *SimpleClusteringVertex*, not shown) to implement the overall clustering algorithm of Figure 3. Lines 2 and 3 in Figure 3 are implemented in lines 24 and 25 of Figure 6. *SimpleClusteringMaster* maintains a global object named *“comp-stage”* that coordinates the different stages of the algorithm. Using this global object, the master signals the vertices what stage of the algorithm they are currently in. By looking at the value of this object, vertices know what computation to do and what types of messages to send and receive. Thus, we are able to encapsulate vertex-centric computations in *vertex.compute()*, and coordinate them globally with *master.compute()*.

3 Static Graph Partitioning

We next present our experiments on different static partitioning of graphs. In Section 3.2 we show that by partitioning large graphs “intelligently” before computation begins, we can reduce total network I/O by up to 13.6x and run-time by up to 2.5x. The effects of partitioning depend on three factors: (1) the graph algorithm being executed; (2) the graph itself; and (3) the configuration of the worker tasks across compute nodes. We show experiments for a variety of settings demonstrating the importance of these factors. We also explore partitioning the adjacency lists of high-degree vertices across workers. We report on those performance improvements in Section 3.4. Section 3.1 explains our experimental set-up, and Section 3.3 repeats some of our experiments on the *Giraph* open-source graph processing system.

```
1  public class SimpleClusteringMaster extends Master {
2    @Override
3    public void compute(int nextSuperstepNo) {
4      if (nextSuperstepNo == 1) {
5        pickKVerticesAndPutIntoGlobalObjects();
6        getGlobalObjects().put(“comp−stage”,
7          new IntGlobalObject(CompStage.CLUSTER_FINDING_1));
8      } else {
9        int compStage = getGlobalObject(“comp−stage”).value();
10       switch(compStage) {
11       case CompStage.CLUSTER_FINDING_1:
12         getGlobalObjects().put(“comp−stage”,
13           new IntGlobalObject(CompStage.CLUSTER_FINDING_2));
14       break;
15       case CompStage.CLUSTER_FINDING_2:
16         if (numActiveVertices() == 0) {
17           getGlobalObjects().put(“comp−stage”,
18             new IntGlobalObject(CompStage.EDGE_COUNTING_1));}
19       break;
20       case CompStage.EDGE_COUNTING_1:
21         getGlobalObjects().put(“comp−stage”,
22           new IntGlobalObject(CompStage.EDGE_COUNTING_2));
23       break;
24       case CompStage.EDGE_COUNTING_2:
25         int numEdgesCrossing = getGlobalObject(“num−edges−crossing”).value();
26         if (numEdgesCrossing > threshold) {
27           pickKVerticesAndPutIntoGlobalObjects();
28           getGlobalObjects().put(“comp−stage”,
29             new IntGlobalObject(CompStage.CLUSTER_FINDING_1));
30         } else {
31           terminateComputation(); }}}}
```

Figure 6: Clustering algorithm using *master.compute()*.

3.1 Experimental Setup

We describe our computing set-up, the graphs we use, the partitioning algorithms, and the graph algorithms used for our experiments.

We ran all our experiments on the Amazon EC2 cluster using large instances (4 virtual cores and 7.5GB of RAM) running Red Hat Linux OS. We repeated each experiment five times with checkpointing turned off. The numeric results we present are the averages across all runs ignoring the initial data loading stage. Performance across multiple runs varied by only a very small margin.

The graphs we used in our experiments are specified in Table 1.[1] We consider four

[1]These datasets were provided by “The Labaratory for Web Algorithmics” [LAW], using software packages WebGraph [BV04], LLP [BRSV11] and UbiCrawler [BCSV04].

Name	Vertices	Edges	Description
uk-2007-d	106M	3.7B	web graph of the .uk domain from 2007 (directed)
uk-2007-u	106M	6.6B	undirected version of uk-2007-d
sk-2005-d	51M	1.9B	web graph of the .sk domain from 2005 (directed)
sk-2005-u	51M	3.2B	undirected version of sk-2005-d
twitter-d	42M	1.5B	Twitter "who is followed by who" network (directed)
uk-2005-d	39M	750M	web graph of the .uk domain from 2005 (directed)
uk-2005-u	39M	1.5B	undirected version of uk-2005-d

Table 1: Data sets.

different static partitionings of the graphs:

- *Random*: The default "mod" partitioning method described in Section 2, with vertex IDs ensured to be random.
- *METIS-default*: METIS [MET] is publicly-available software that divides a graph into a specified number of partitions, trying to minimize the number of edges crossing the partitions. By default METIS balances the number of vertices in each partition. We set the *ufactor* parameter to 5, resulting in at most 0.5% imbalance in the number of vertices assigned to each partition [MET].
- *METIS-balanced*: Using METIS' multi-constraint partitioning feature [MET], we generate partitions in which the number of vertices, outgoing edges, and incoming edges of partitions are balanced. We again allow 0.5% imbalance in each of these constraints. METIS-balanced takes more time to compute than METIS-default, although partitioning time is not a focus of our study.
- *Domain-based*: In this partitioning scheme for web graphs only, we locate all web pages from the same domain in the same partition, and partition the domains randomly across the workers.

Note that we are assuming an environment in which partitioning occurs once, while graph algorithms may be run many times, therefore we focus our experiments on the effect partitioning has on algorithms, not on the cost of partitioning itself.

Finally, we use four different graph algorithms in our experiments:

- *PageRank (PR)* [BP98]
- Finding shortest paths from a single source *(SSSP)*, as implemented in [MAB+11]

- The *HCC* [KTF09] algorithm to find connected components
- *RW-n*, a pure random-walk simulation algorithm. Each vertex starts with an initial number of n walkers. For each walker i on a vertex u, u randomly picks one of its neighbors, say v, to simulate i's next step. For each neighbor v of u, u sends a message to v indicating the number of walkers that walked from u to v.

3.2 Performance Effects of Partitioning

Because of their bulk synchronous nature, the speed of systems like Pregel, GPS, and Giraph is determined by the slowest worker to reach the synchronization points between supersteps. We can break down the workload of a worker into three parts:

1. *Computation*: Looping through vertices and executing *vertex.compute()*
2. *Networking*: Sending and receiving messages between workers
3. *Parsing and enqueuing messages*: In our implementation, where messages are stored as raw bytes, this involves byte array allocations and copying between byte arrays.

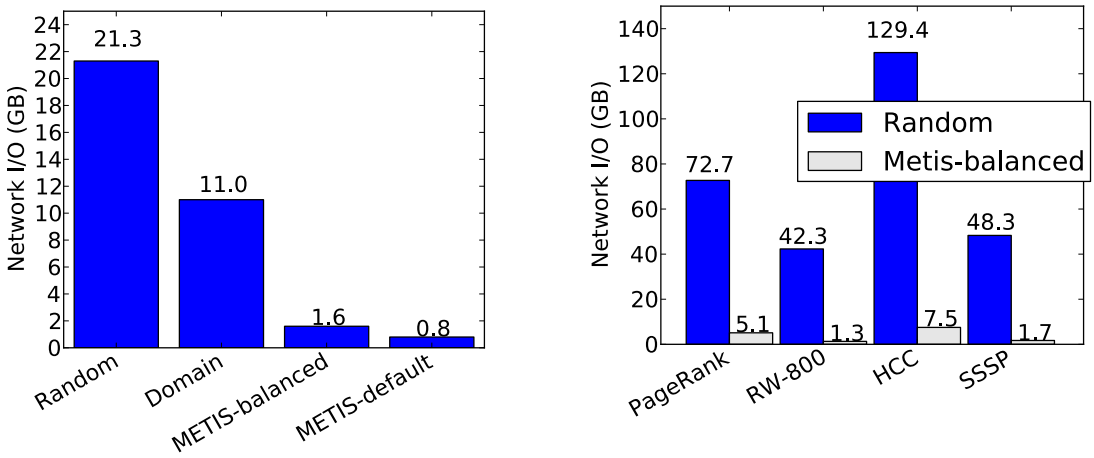
Although random partitioning generates well-balanced workloads across workers, almost all messages are sent across the network. We show that we can both maintain a balanced workload across workers and significantly reduce the network messages and overall run-time by partitioning the graph using our more sophisticated schemes.

With sophisticated partitioning of the graph we can obviously reduce network I/O, since we localize more edges within each worker compared to random partitioning. Our first set of experiments, presented in Section 3.2.1, quantifies the network I/O reduction for a variety of settings.

In Section 3.2.2, we present experiments measuring the run-time reduction due to sophisticated partitioning when running various algorithms in a variety of settings. We observe that partitioning schemes that maintain workload balance among workers perform better than schemes that do not. Finally, in Section 3.2.3, we discuss how to fix the workload imbalance among workers when a partitioning scheme generates imbalanced partitions.

3.2.1 Network I/O

In our first set of experiments, we measure network I/O (network writes in GB across all workers) when running different graph algorithms under different partitioning schemes



(a) *PageRank on sk-2005-d* (b) *Different algorithms on uk-2007-u*

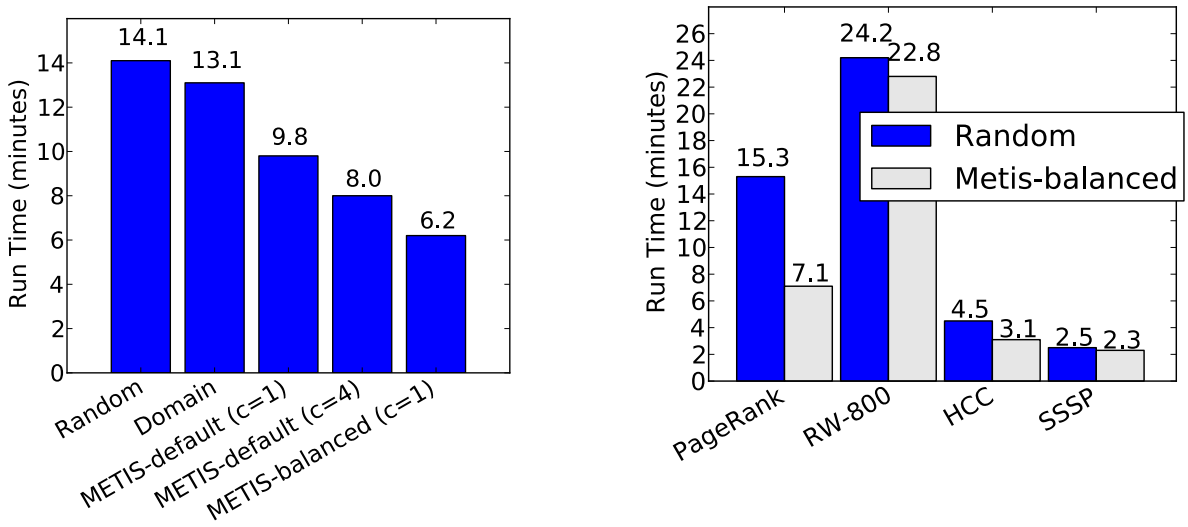
Figure 7: Network I/O performance of different partitioning schemes

in a variety of settings. The network I/O reductions we report next are relative to the performance of random partitioning. Overall, network I/O reductions varied between 1.8x to 2.2x for partitioning by domain, 13.3x and 36.3x for METIS-balanced, and 26.6x and 58.5x for METIS-default. We present two of our experiments in Figure 7. Figure 7a shows network I/O for different partitioning schemes when running PageRank on the *sk-2005-d* graph on 60 workers running on 60 compute nodes. Figure 7b shows network I/O for random and METIS-balanced partitioning when executing different algorithms on the *uk-2007-u* graph also with 60 workers and 60 compute nodes. We present per superstep network I/O for PageRank and RW-800, and total network I/O for HCC and SSSP. We also experimented with different number of workers and compute nodes and found that network I/O reduction percentages were similar. Of course, network I/O is not the only contributor to overall run-time, so the remainder of our experiments consider the effect of partitioning schemes and other parameters on run-time.

3.2.2 Run-time

In this section, we set out to test how much sophisticated partitioning improves overall run-time. We measure the run-time performance of four algorithms on different graphs, partitioning schemes, and worker and compute node configurations. We used between 15 and 60 nodes, and between one and four workers on each node.

Since the main benefit of sophisticated partitioning is reducing the number of messages



(a) *PageRank (50 iterations) on sk-2005-d* (b) *Different algorithms on uk-2007-u*

Figure 8: Run-time performance of different partitioning schemes

sent over the network, we expect partitioning to improve run-time most in algorithms that generate a lot of messages and have low computational workloads. Overall, here is the computation and communication workloads of the graph algorithms we use:

- PageRank: short per-vertex computation, high communication
- HCC: short per-vertex computation, medium communication
- RW-800: long per-vertex computation (due to random number generation), medium communication
- SSSP: short per-vertex computation, low communication

A sample of our experimental results is shown in Figure 8. Figure 8a shows PageRank on the *sk-2005-d* graph on 60 compute nodes with 60 workers. Parameter c indicates the number of partitions assigned to each worker when using METIS-default and METIS-balanced. We will discuss the difference between (c = 1) and (c = 4) in the next section. In this experiment, improvements ranged between 1.1x for domain-based partitioning to 2.3x for METIS-balanced. In other experiments for PageRank, METIS-balanced consistently performed best, reducing run-time between 2.1x to 2.5x over random partitioning. Improvements for METIS-default varied from 1.4x to 2.4x and for domain-based partitioning from 1.1x to 1.7x.

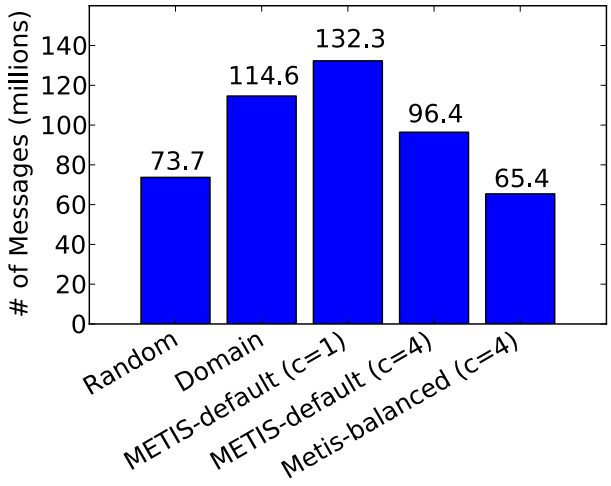


Figure 9: Number of messages processed by the slowest worker.

Not surprisingly, run-time reductions when executing other graph algorithms are less than PageRank. Figure 8b shows four algorithms on the *uk-2007-u* graph using 30 workers running on 30 compute nodes. We compared the performance of random partitioning and METIS-balanced. As shown, METIS-balanced reduces the run-time by 2.2x when executing PageRank, and by 1.47x, 1.08x and 1.06x for HCC, SSSP and RW-800, respectively.

3.2.3 Workload Balance

In all our experiments reported so far, METIS-default performed better than METIS-balanced in network I/O but worse in run-time. The reason for this counterintuitive performance is that when using METIS-default there are specific bottleneck workers that slow down the system. For all the graph algorithms we are considering, messages are sent along the edges. Recall that METIS-default balances only the number of vertices in each partition and not the edges. As a result, when using METIS-default, some workers process a significantly higher number of messages than average, which is a costly operation. Figure 9 shows the number of messages processed by the slowest workers when using different partitioning schemes for the experiment of Figure 8a. Both under random and METIS-balanced partitioning schemes, workers get homogeneous workloads. When (c=1) and we generate 60 partitions with METIS-default, partition 44 has roughly 132M incoming and outgoing edges, more than twice the average load of other partitions. As a result, worker 44, which is assigned partition 44, slows down the entire system.

We next show how to improve workload imbalance, and in turn improve the run-time

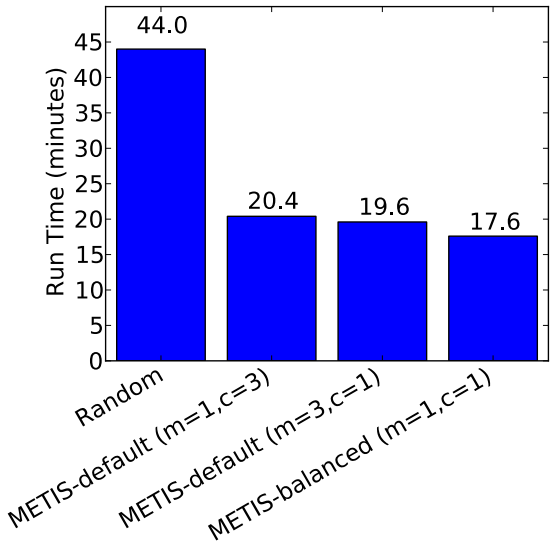


Figure 10: 50 iterations of PageRank running on 20 compute nodes on the *sk-2005-d* graph.

benefits when using a sophisticated partitioning scheme that generates imbalanced partitions. Our approach is to generate more partitions than we have workers, then assign multiple partitions to each worker, thus averaging the workloads from "heavy" and "light" partitions. We tested the effectiveness of this approach by measuring the run-time of PageRank on *sk-2005-d* in two different set-ups. First, we used the same 60 partitions generated by METIS-default and METIS-balanced from before. We used 20 workers running on 20 compute nodes, and assigned three partitions to each worker. The result of this experiment is shown in METIS-default and METIS-balanced bars with (m=1, c=3) parameters in Figure 10. Parameter m refers to the number of workers running on each compute node. METIS-default when (m=1, c=3) improved run-time by 2.2x over random partitioning—a significant improvement compared to the 1.4x improvement when assigning one partition to each worker. Second, we ran 60 workers on 60 compute nodes, but generated 240 partitions with METIS-default, and assigned four partitions to each worker. The result of this experiment is shown in the METIS-default (c = 4) bar in Figure 8a. As shown, assigning four partitions performed better than assigning one partition to each worker.

As a final remark, we note that we observe the same "averaging" effect when we assign one partition to each worker but run multiple workers per compute node. When multiple workers are running on the same compute node, workers start slowing each other down. However, the faster workers are slowed down more than slower workers, averaging the speed

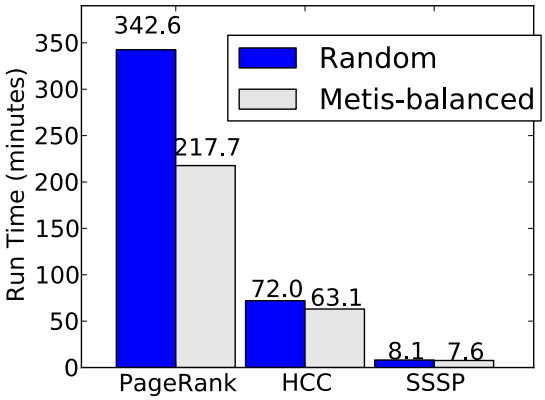


Figure 11: PageRank on Giraph [GIR].

of workers. The METIS-default bar with (m=3, c=1) in Figure 10 shows the performance of using the 60 partitions generated by METIS-default from before, assigning one partition to each worker and running three workers per compute node. As shown, assigning one partition to each worker and three workers to each compute node performs similarly to assigning three partitions to each worker and one worker to each compute node.

3.3 Experiments on Giraph

The network I/O reductions in our experiments are a direct consequence of the number of edges crossing partitions, determined by the partitioning scheme. Therefore, our network reduction results carry over to other distributed message-passing systems, such as Giraph [GIR]. However, the run-time results are implementation-dependent and may vary from system to system. To test whether sophisticated partitioning of graphs can improve run-time in other systems, we repeated some of our experiments in Giraph. Figure 11 summarizes our results. METIS-balanced yields 1.6x run-time improvement over random partitioning for PageRank. Similar to our results in GPS, the improvements are less for SSSP and HCC. We also note that GPS runs ∼12x faster than Giraph on the same experiments. We explain the main implementation differences between Giraph and GPS in Section 5.

3.4 Large Adjacency-List Partitioning

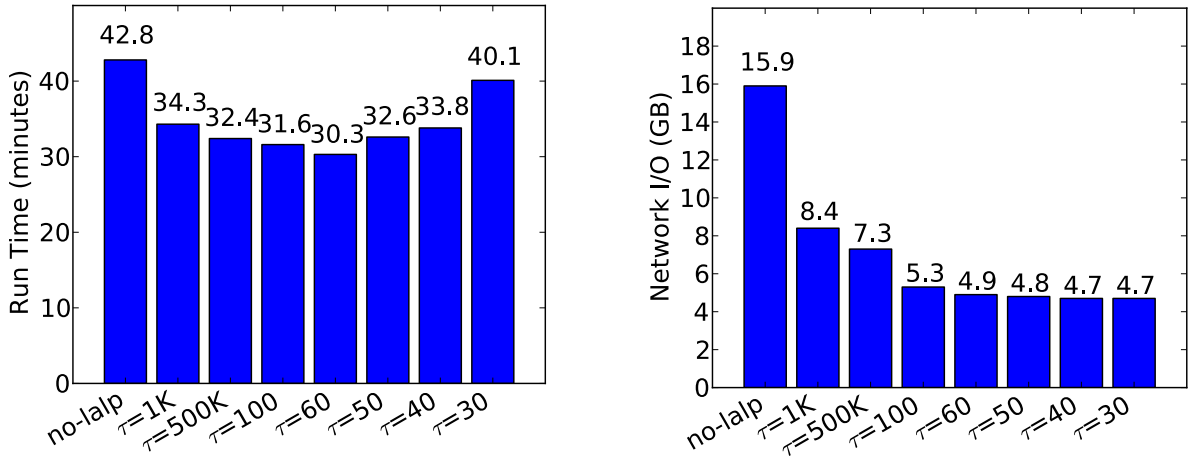
GPS includes an optimization called *LALP* (large adjacency list partitioning), in which adjacency lists of high-degree vertices are not stored in a single worker, but rather are

partitioned across all workers. This optimization can improve performance, but only for algorithms with two properties: (1) Vertices use their adjacency lists (outgoing neighbors) only to send messages and not for computation; (2) If a vertex sends a message, it sends the same message to all of its outgoing neighbors. For example, in PageRank each vertex sends its latest PageRank value to all of its neighbors, and that is the only time vertices access their adjacency lists. On the other hand, RW-n does not satisfy property 2: a message from vertex u to its neighbor v contains the number of walkers that move from u to v and is not necessarily the same as the message u sends to its other neighbors.

Suppose a vertex u is located in worker W_i and let $N_j(u)$ be the outgoing neighbors of u located in worker W_j. Suppose $|N_j(u)| = 10000$. During the execution of PageRank, W_i sends 10000 copies of the same message to W_j in each superstep, one for each vertex in $N_j(u)$. Instead, if W_j stores $N_j(u)$, W_i need send only a single message to W_j for node u, and W_j replicates this message 10000 times to the message queues of each vertex in $N_j(u)$.

Many real-world graphs are known to have a skewed degree distribution, in which a small number of vertices' adjacency lists contain a significant fraction of all the edges in the graph. For these graphs, *LALP* can improve network traffic and run-time significantly. GPS programmers specify a parameter τ when using this optimization. If a vertex u has more than τ outgoing neighbors, GPS partitions *u's* adjacency list into $N_1(u), N_2(u), \ldots, N_k(u)$, and sends $N_j(u)$ to worker W_j during the initial partitioning of the graph across workers. During execution, when u sends a message to all its neighbors, GPS intercepts the message and sends a single a message to each worker W_j, with W_j delivering the message to all vertices in $N_j(u)$.

To verify that LALP improves performance, we ran experiments on the *twitter-d* graph, with different values of τ. As we reduce τ, GPS partitions more adjacency lists across all workers, and we expect this to reduce network I/O. On the other hand, the map of $< u, N_j(u) >$ pairs, which each worker W_j maintains, grows, incurring some memory and computation overhead during message parsing. We expect there to be an optimal τ which achieves the best run-time performance. As shown in Figure 12, when running 32 workers on 16 compute nodes, the optimal τ is around 60, and achieves 1.41x run-time improvement over running without LALP.



(a) Total run-time

(b) Network I/O

Figure 12: Performance of LALP.

4 Dynamic Repartitioning

To reduce the number of messages sent over the network, it might be helpful to reassign certain vertices to other workers dynamically during algorithm computation. There are three questions any dynamic repartitioning scheme has to answer: (1) which vertices to reassign; (2) how and when to move the reassigned vertices to their new workers; (3) how to locate the reassigned vertices. Below, we explain our answers to these questions in GPS and discuss possible other options. We also present our experiments measuring the network I/O and run-time performance of GPS, when the graph is initially partitioned by one of our partitioning schemes from Section 3, then dynamically repartitioned by our scheme.

4.1 Picking vertices to reassign

One option is to reassign vertex u at worker W_j to a new worker W_i if u sends and receives the most messages to/from W_i than any other worker, and that number of messages is over some threshold. There are two issues with this approach. First, in order to observe incoming messages, we need to include the *source* worker in each message, which can increase the memory requirement significantly when the size of the actual messages are small. To avoid this memory requirement, GPS bases reassignment on sent messages only.

Second, using this basic reassignment technique, we observed that over multiple iterations, more and more vertices are reassigned to only a few workers, creating significant

imbalance. Despite the network benefits, the "dense" workers significantly slow down the system. To maintain balance, GPS exchanges vertices between workers. Each worker W_i constructs a set S_{ij} of vertices that potentially will be reassigned to W_j, for each W_j. Similarly W_j constructs a set S_{ji}. Then W_i and W_j communicate the sizes of their sets and exchange exactly $min(S_{ij}, S_{ji})$ vertices, guaranteeing that the number of vertices in each worker does not change through dynamic repartitioning.

4.2 Moving the reassigned vertices to their new workers

Once a dynamic partitioning scheme decides to reassign a vertex u from W_i to W_j in superstep x, three pieces of data associated with u must be sent to W_j: (a) u's latest value; (b) u's adjacency list; and (c) u's messages for superstep $(x+1)$. One option is to insert a "vertex moving" stage between the end of superstep x and beginning of superstep $x+1$, during which all vertex data is moved. GPS uses another option that combines vertex moving within the supersteps themselves: At the end of superstep x, workers exchange their set sizes described in the previous subsection. Then, between the end of superstep x and beginning of superstep $(x+1)$, the exact vertices to be exchanged are determined and the adjacency lists are relabeled, described in the next subsection. Relabeling of the adjacency lists ensures that all messages that will be sent to u in superstep $x+1$ are sent to W_j. However, u is not sent to W_j at this point. During the computation of superstep $(x+1)$, W_i first calls *u.compute()* and then sends only u's adjacency list and latest value to W_j. Thus, u's messages for superstep $(x+1)$ are not sent to W_j, reducing the network overhead of dynamic repartitioning.

4.3 Locating reassigned vertices

When a vertex u gets reassigned to a new worker, every worker in the cluster has to obtain and store this information in order to deliver future messages to u. An obvious option is for each worker to store an in-memory map consisting of <*vertex-id, new-worker-id*> pairs. Of course, over time, this map can potentially contain as many pairs as there are vertices in the original graph, causing a significant memory and computation bottleneck. In our experiments, up to 90% of vertices can eventually get reassigned. Thus, GPS instead uses an approach based on relabeling the IDs of reassigned vertices. Suppose u has been reassigned to W_j. We give u a new ID u', such that $(u' \mod k) = j$. Since every pair W_i and W_j exchange the same number of vertices, vertex IDs can effectively be exchanged

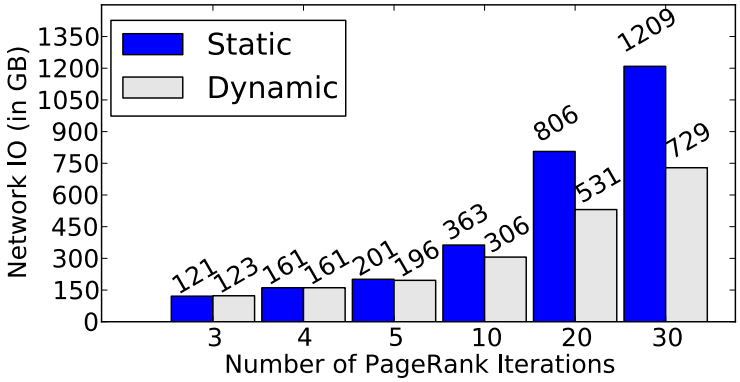
as well. In addition, each worker must go through all adjacency lists in its partition and change each occurrence of u to u'.

There are two considerations in this approach:

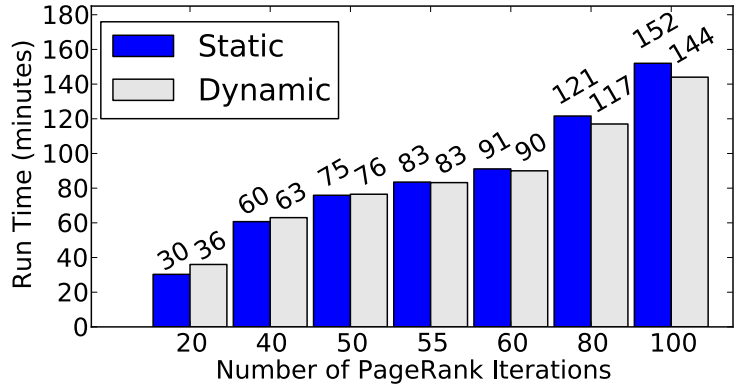
- If the application requires the original node IDs to be output at the end of the computation, this information must be retained with nodes whose IDs are modified, incurring some additional storage.
- When a node u is relabeled with a new ID, we modify its ID in the adjacency lists of all nodes with an edge to u. If the graph algorithm being executed involves messages not following edges (that is, messages from a node u_1 to a node u_2 where there is no edge from u_1 to u_2), then our relabeling scheme cannot be used. In most graph algorithms suitable for GPS, messages do follow edges.

4.4 Experiments

Dynamic repartitioning is intended to improve network I/O and run-time of the graph algorithm by reducing the number of messages sent over the network. On the other hand, dynamic repartitioning also incurs network I/O overhead by sending vertex data between workers and run-time overhead by doing the extra computation of deciding which vertices to send and relabeling adjacency lists. It would not be surprising if, in the first few supersteps of an algorithm using dynamic repartitioning, network I/O and run-time overhead exceeds the benefits, and eventually, run-time and network I/O reductions overcome the overhead. We expect that there is a crossover superstep s, such that dynamic repartitioning performs better than static partitioning only if the running graph algorithm runs more than s supersteps. Obviously, s could be different for network I/O and run-time performances and would depend on the running graph algorithm and the initial partitioning of the graph. We set out to answer the following question for PageRank: What is the crossover superstep when the graph is initially partitioned randomly? To answer this question, we ran PageRank in the *uk-2007-d* graph for different number of iterations between three and 100, with and without dynamic repartitioning. We used 30 workers running on 30 compute nodes and the graph was initially partitioned randomly. In GPS, the master task turns dynamic repartitioning off when the number of vertices being exchanged is below a threshold, which is by default 0.1% of the total number of vertices in the graph. When executing PageRank long enough, this corresponded to a superstep between 15 and 20 in our experiments.



(a) Network I/O



(b) Run-time

Figure 13: Performance of running PageRank for different number of iterations with and without dynamic repartitioning.

Our results are shown in Figures 13a and 13b. The crossover superstep in this experiment is five iterations for network I/O and around 55 iterations for run-time. When running PageRank for long enough, dynamic repartitioning performs ∼2.0x better than static partitioning in terms of network I/O and ∼1.13x in terms of run-time.

We next ask the same question for the setting in which the graph is initially partitioned by a sophisticated partitioning scheme. To answer this question, we repeat our experiment from above, this time partitioning the graph initially by METIS-balanced and domain. When the initial partitioning is METIS-balanced, we do not get noticeable network I/O or run-time benefits from dynamic repartitioning. That is, no matter how long we run PageRank dynamic repartitioning always performs worse than static partitioning. Surprisingly, when we start with domain-based partitioning, the crossover superstep is four for

network I/O, and 36 for run-time. When running PageRank for long enough, dynamic repartitioning performs ∼2.2x better than static partitioning in terms of network I/O and ∼1.2x in terms of run-time.

In our setting, the run-time benefits of dynamic repartitioning are very modest. However, in settings were networking is slower, we believe the benefits from network I/O would yield significant run-time improvements as well.

5 Other System Optimizations

We describe several optimizations in our system that reduce memory usage and increase the overall speed of GPS.

- **Combining messages at the receiver worker**: In all of the graph algorithms we have been experimenting with, vertices combine their incoming message values using commutative and associative operations. In PageRank and RW-n the message values are summed, and in HCC and SSSP, vertices compute the minimum value. We can use this information to significantly reduce the memory required to store messages in each worker. When a message m is received, if the message list for the receiving vertex is empty, we add m to the message list. Otherwise, instead of appending m to the list, we immediately combine its value with the current one. This optimization reduces the total memory required to store messages among vertices for a particular superstep from $|E|$ to $|V|$—a significant reduction in most graphs where the number of edges is significantly higher than the number of vertices.

  In a related vein, in the MapReduce framework *Combiners* were introduced to reduce the number of intermediate values sent from *Mappers* to *Reducers*. Pregel [MAB+11] and Giraph [GIR] take a similar approach and combine messages at the sender worker, which reduces the network I/O. In earlier versions of GPS, we implemented this optimization. In order to combine messages at a sender worker W_i, W_i needs to store an outgoing message list for each vertex w that receives a message from W_i, which increases memory usage. Also, messages are buffered twice, once in the outgoing messages lists, and then in the message buffers for each worker, which slows down the rate at which buffers fill and are flushed. Overall, we did not observe significant performance improvements by combining at the sender side.

- **Single *Vertex* and *Message* objects**: GPS reduces the memory cost of allocating

many Java objects by storing *canonical* objects. First, instead of storing the value and the adjacency list of each vertex v inside a separate *Vertex* object, and calling the *vertex.compute()* on each object as in Giraph, GPS workers use a single canonical *Vertex* object, with vertex values and adjacency lists stored in separate data structures. For each vertex v in worker W_i, W_i is configured so the canonical *Vertex* object has access to v's value and adjacency list. W_i then calls *vertex.compute()* on the canonical object. Similarly, GPS workers store a single canonical *Message* object. Incoming messages are stored as raw bytes in the message queues, and a message is deserialized into the canonical *Message* object only when the canonical *Vertex* object iterates over it.

- **Controlling the speed of message generation**: Recall from Section 2 that the computation thread generates and buffers outgoing messages by calling *vertex.compute()* on active vertices. When an outgoing message buffer is full, the computation thread gives it to networking threads to send over the network. We have observed that we can significantly improve the overall speed of GPS by controlling the speed of message generation and limiting the number of buffers that are being sent concurrently over the network. GPS controls these factors in two ways. First, at certain intervals, the computation thread checks the number of buffers it has given to the networking layer. If the number is above a threshold (two, by default), the computation thread waits. Second, the networking threads send only a fixed number of the outgoing buffers concurrently over the network (again two, by default). These controls save memory and improve network speed, leading to improved overall performance.

As we showed in Section 3, GPS is ∼12 times faster than Giraph. In addition to our optimizations above, we believe there are two more implementation differences between GPS and Giraph that explain this differential:

1. **Message Buffers**: Giraph organizes its message buffers per vertex rather than per worker. Both Giraph and GPS wait until buffers are full to send them or until the end of the superstep when buffers are flushed. Per-vertex buffers take much longer to fill, delaying overall network speed.

2. **Synchronization**: As explained in Section 2, GPS threads synchronize only for each sent and received message buffer. In Giraph, the RPC calls from different workers to

a particular worker W_i synchronize with each other, as they access the same message queues in W_i. Therefore, there is possible synchronization between threads for each message received by a worker, which is much more frequent than synchronizing per message buffer.

The system optimizations described in this section have been key to the high performance of GPS; we hope to integrate some of them into Giraph.

6 Related Work

GPS is a bulk synchronous, distributed message-passing system for large-scale graph computations. There are several classes of systems designed to do large-scale graph computations:

- **Bulk synchronous message passing systems**: Pregel [MAB+11] introduced the first bulk synchronous distributed message-passing system, which GPS has drawn from. Several other systems are based on Pregel, including Giraph [GIR], GoldenOrb [GOL], Phoebus [PHO], Hama [HAM], JPregel [JPR] and Bagel [BAG] . Giraph is the most popular and advanced of these systems. Giraph jobs run as Hadoop jobs without the reducing phase. Giraph leverages the task scheduling component of Hadoop clusters, by running workers as special *Mappers*, that communicate with each other to deliver messages between vertices and synchronize in between supersteps.

- **Hadoop-based systems**: Hadoop-based systems extend or build libraries on top of Hadoop to support iterative computations, where each iteration of the algorithm corresponds to a Hadoop job. Pegasus [KTF09], HaLoop [BHBE10], iMapReduce [ZGGW11], Surfer [CWHY10] and Twister [ELZ+10] are examples of these systems. These systems suffer from, and offer optimizations for the inefficiencies described in Section 1, which systems like Pregel and GPS avoid.

- **Asynchronous systems**: GPS supports only bulk synchronous graph processing. GraphLab [LGK+10] and Signal-Collect [SBC10] support asynchronous vertex-centric graph processing. An advantage of asynchronous computation over bulk synchronous computation is that fast workers do not have to wait for slow workers to do computation. However, programming in the asynchronous model can be harder than

synchronous models, as programmers have to reason about the non-deterministic order of vertex-centric function calls. Signal-Collect also supports bulk synchronous processing, but its not a distributed system. Its parallelism is due to multithreading.

- **Message Passing Interface (MPI)**: MPI is a standard defining an interface for building a broad range of message passing programs. There are several implementations of MPI [OPE, MPI, PYM, OCA], which can be used to implement parallel message-passing graph algorithms in various programming languages. MPI consists of very low-level communication primitives that do not provide any consistency or fault tolerance. Programmers must build another level of abstraction (e.g. bulk synchronous consistency) on their own, which makes programming harder than bulk synchronous message-passing systems.

- **Other systems**: HipG [KKFB10] is a system, in which each vertex is a Java object and the computation is done sequentially starting from a particular vertex. The code is expressed as if the graph is in a single machine and the reads and writes to vertices residing in other machines are translated as RPC calls. HipG incurs a significant overhead from RPC calls when executing algorithms, such as PageRank, that compute a value for each vertex in the graph. Mahout [MAH] and Spark [ZCF+10] are general cluster computing systems, whose APIs are designed to express generic iterative computations. As a result, programming graph algorithms on Mahout and Spark require significant more coding effort than GPS.

In addition to presenting GPS, we studied the effects of different graph partitioning schemes on the performance of GPS when running different graph algorithms. We also studied the effects of GPS's dynamic repartitioning scheme on performance. There are previous studies on the performance effects of different partitionings of graphs on other systems. [HAR11] shows that by partitioning *Resource Description Framework* [RDF04] (RDF) data with METIS and then intelligent replication of certain tuples, SPARQL [SPA06] query run-times can be improved significantly over random partitioning. We study the effects of partitioning under batch algorithms, whereas SPARQL queries consist of short path-finding workloads. [SK11] develops a heuristic to partition the graph across machines during the initial loading phase. They study the reduction in the number of edges crossing machines and run-time improvements on Spark when running PageRank. They do not study the effects of other sophisticated partitioning schemes.

7 Conclusions and Future Work

We presented GPS, an open source distributed message-passing system for large-scale graph computations. Like Pregel [MAB+11] and Giraph [GIR], GPS is designed to be scalable, fault-tolerant, and easy to program through simple user-provided functions. Using GPS, we studied the network and run-time effects of different graph partitioning schemes in a variety of settings. We also described GPS's dynamic repartitioning feature, and presented several other system optimizations that increase the performance of GPS.

As future work, we are interested in exploring which graph algorithms are suitable for running on systems like GPS and Pregel. On the theoretical side, we want to understand exactly which graph algorithms can be implemented efficiently using bulk synchronous parallelism and by passing messages between vertices. For example, although there are bulk synchronous message-passing algorithms to find the weakly connected components of undirected graphs, we do not know of any such algorithm for finding strongly connected components.

On the practical side, we want to make implementing bulk synchronous message-passing algorithms easier for GPS programmers by adopting a higher-level language as a programming interface. We believe for many programmers, it is more intuitive to express graph algorithms sequentially, assuming the entire graph is in the RAM of single machine, than expressing them as parallel bulk synchronous message-passing computations. We want programmers to express their GPS algorithms in a high-level domain specific language sequentially and a compiler to automatically generate an equivalent code in the current GPS API. As a first step in this direction, we are working on compiling Green-Marl [HCSO12], a domain-specific language for graph algorithms, into GPS.

8 Acknowledgements

We are grateful to Sungpack Hong, Jure Leskovec and Hyunjung Park for numerous useful discussions.

References

[BAG] Bagel Programming Guide. `https://github.com/mesos/spark/wiki/Bagel-Programming-Guide/`.

[BCSV04] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. UbiCrawler: A Scalable Fully Distributed Web Crawler. *Software: Practice & Experience*, 34(8):711–726, 2004.

[BHBE10] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D. Ernst. HaLoop: Efficient iterative data processing on large clusters. *Proceedings of the International Conference on Very Large Databases*, pages 285–296, 2010.

[BP98] Sergey Brin and Lawrence Page. The Anatomy of Large-Scale Hypertextual Web Search Engine. In *Proceedings of the International Conference on The World Wide Web*, pages 107–117, 1998.

[BRSV11] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks. In *Proceedings of the 20th international conference on World Wide Web*. ACM Press, 2011.

[BV04] Paolo Boldi and Sebastiano Vigna. The WebGraph framework I: Compression techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*, pages 595–601, Manhattan, USA, 2004. ACM Press.

[CWHY10] R. Chen, X. Weng, B. He, and M. Yang. Large graph processing in the cloud. In *Proceedings of the International Conference on Management of Data*, pages 1123–1126, 2010.

[DG04] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of the Symposium on Operating System Design and Implementation*, pages 137–150, 2004.

[ELZ+10] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. Twister: a runtime for iterative mapreduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 810–818, New York, NY, USA, 2010. ACM.

[GIR] Apache Incubator Giraph. `http://incubator.apache.org/giraph//`.

[GOL] GoldenOrb. `http://www.raveldata.com/goldenorb/`.

[HAD] Apache Hadoop. http://hadoop.apache.org/.

[HAM] Apache Hama. http://incubator.apache.org/hama/.

[HAR11] Jiewen Huang, Daniel J. Abadi, and Kun Ren. Scalable SPARQL Querying of Large RDF Graphs. PVLDB, 4(21), August 2011.

[HCSO12] Sungpack Hong, Hassan Chafi, Eric Sedlar, and Kunle Olukotun. Green-marl: a dsl for easy and efficient graph analysis. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 349–362. ACM, 2012.

[HDF] Hadoop Distributed File System. http://hadoop.apache.org/hdfs/.

[HGO+10] K. Heath, N. Gelfand, M. Ovsjanikov, M. Aanjaneya, and L.J. Guibas. Image Webs: Computing and Exploiting Connectivity in Image Collections. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3432 – 3439, 2010.

[JPR] JPregel. http://kowshik.github.com/JPregel/.

[KKFB10] E. Krepska, T. Kielmann, W. Fokkink, and H. Bal. A high-level framework for distributed processing of large-scale graphs. In *Proceedings of the International Conference on Distributed Computing and Networking*, pages 1123–1126, 2010.

[KMF11] U. Kang, B. Meeder, and C. Faloutsos. Spectral analysis for billion-scale graphs: Discoveries and implementation. In *Proceedings of Pacific Asia Conference on Knowledge Discovery and Data Mining*, pages 13–25, 2011.

[KTF09] U. Kang, C. E. Tsourakakis, and C. Faloutsos. PEGASUS: A peta-scale graph mining system – Implementation and observations. In *In Proceedings of the IEEE International Conference on Data Mining*, pages 229–238, 2009.

[LAW] The Laboratory for Web Algorithmics. http://law.dsi.unimi.it/datasets.php.

[LC10] Frank Lin and William W. Cohen. Power iteration clustering. In Johannes Fürnkranz and Thorsten Joachims, editors, *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 655–662, Haifa, Israel, June 2010. Omnipress.

[LGK+10] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. GraphLab: A New Framework for Parallel Machine Learning. *CoRR*, abs/1006.4990, 2010.

[MAB+11] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A System for Large-Scale Graph Processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 155–166, 2011.

[MAH] Apache mahout. `http://mahout.apache.org/`.

[MEJ+09] K. Madduri, D. Ediger, K. Jiang, D.A. Bader, and D. Chavarria-Miranda. A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1 –8, may 2009.

[MET] METIS Graph Partition Library. `http://exoplanet.eu/catalog.php`.

[MIN] Apache MINA. `http://mina.apache.org/`.

[MPI] MPICH2. `http://www.mcs.anl.gov/research/projects/mpich2/`.

[OCA] OCaml MPI. `http://forge.ocamlcore.org/projects/ocamlmpi/`.

[OPE] Open MPI. `http://www.open-mpi.org/`.

[PHO] Phoebus. `https://github.com/xslogic/phoebus`.

[PYM] pyMPI. `http://pympi.sourceforge.net/`.

[RDF04] RDF Primer. W3C Recommendation. `http://www.w3.org/TR/rdf-primer`, 2004.

[SBC10] P Stutz, A Bernstein, and W W Cohen. Signal/Collect: graph algorithms for the (Semantic) Web. In *ISWC 2010*, 2010.

[SK11] Isabelle Stanton and Gabriel Kliot. Streaming Graph Partitioning for Large Distributed Graphs. Technical report, Microsoft Research Lab, November 2011.

[SPA06] SPARQL Query Language for RDF. W3C Working Draft 4. `http://www.w3.org/TR/rdf-sparql-query/`, October 2006.

[Val90] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.

[ZCF+10] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[ZGGW11] Y. Zhang, Q. Gao, L. Gao, and C. Wang. iMapreduce: A distributed computing framework for iterative computation. *DataCloud*, 2011.