Mizan: A System for Dynamic Load Balancing in Large-scale Graph Processing

Zuhair Khayyat‡ Karim Awara‡ Amani Alonazi‡ Hani Jamjoom† Dan Williams†
Panos Kalnis‡

‡King Abdullah University of Science and Technology, Saudi Arabia
†IBM T. J. Watson Research Center, Yorktown Heights, NY

Abstract

Pregel [23] was recently introduced as a scalable graph mining system that can provide significant performance improvements over traditional MapReduce implementations. Existing implementations focus primarily on graph partitioning as a preprocessing step to balance computation across compute nodes. In this paper, we examine the runtime characteristics of a Pregel system. We show that graph partitioning alone is insufficient for minimizing end-to-end computation. Especially where data is very large or the runtime behavior of the algorithm is unknown, an adaptive approach is needed. To this end, we introduce *Mizan*, a Pregel system that achieves efficient load balancing to better adapt to changes in computing needs. Unlike known implementations of Pregel, Mizan does not assume any a priori knowledge of the structure of the graph or behavior of the algorithm. Instead, it monitors the runtime characteristics of the system. Mizan then performs efficient fine-grained vertex migration to balance computation and communication. We have fully implemented Mizan; using extensive evaluation we show that—especially for highly-dynamic workloads—Mizan provides up to 84% improvement over techniques leveraging static graph pre-partitioning.

1. Introduction

With the increasing emphasis on big data, new platforms are being proposed to better exploit the structure of both the data and algorithms (e.g., Pregel [23], HADI [14], PEGASUS [13] and X-RIME [31]). Recently, Pregel [23] was introduced as a message passing-based programming model specifically targeting the mining of large graphs. For a wide array of popular graph mining algorithms (including PageRank, shortest paths problems, bipartite matching, and semi-clustering), Pregel was shown to improve the overall performance by 1-2 orders of magnitude [17] over a traditional MapReduce [6] implementation. Pregel builds on the Bulk Synchronous Parallel (BSP) [30] programming model. It operates on graph data, consisting of vertices and edges. Each vertex runs an algorithm and can send messages—asynchronously—to any other vertex. Computation is divided into a number of supersteps (iterations), each separated by a global synchronization barrier. During each superstep, vertices run in parallel across a distributed infrastructure. Each vertex processes incoming messages from the previous superstep. This process continues until all vertices have no messages to send, thereby becoming inactive.

Not surprisingly, balanced computation and communication is fundamental to the efficiency of a Pregel system. To this end, existing implementations of Pregel (including Giraph [8], GoldenOrb [9], Hama [28], Surfer [4]) primarily focus on efficient partitioning of input data as a preprocessing step. They take one or more of the following five approaches to achieving a balanced workload: (1) provide simple graph partitioning schemes, like hash- or range-based partitioning (e.g., Giraph), (2) allow developers to set their own partitioning scheme or pre-partition the graph data (e.g., Pregel), (3) provide more sophisticated partitioning techniques (e.g., GraphLab, GoldenOrb and Surfer use min-cuts [16]), (4) utilize distributed data stores and graph indexing on vertices and edges (e.g., GoldenOrb and Hama), and (5) perform coarse-grained load balancing (e.g., Pregel).

In this paper, we analyze the efficacy of these workload-balancing approaches when applied to a broad range of graph mining problems. Built into the design of these approaches is the assumption that the structure of the graph is static, the algorithm has predictable behavior, or that the developer has deep knowledge about the runtime characteristics of the algorithm. Using large datasets from the Laboratory for Web Algorithmics (LAW) [2, 20] and a broad set of graph algorithms, we show that—especially when the

Eurosys'13 April 15-17, 2013, Prague, Czech Republic

graph mining algorithm has unpredictable communication needs, frequently changes the structure of the graph, or has a variable computation complexity—a single solution is not enough. We further show that a poor partitioning can result in significant performance degradation or large up-front cost.

More fundamentally, existing workload balancing approaches suffer from poor adaptation to variability (or shifts) in computation needs. Even the dynamic partition assignment that was proposed in Pregel [23] reacts poorly to highly dynamic algorithms. We are, thus, interested in building a system that is (1) adaptive, (2) agnostic to the graph structure, and (3) requires no a priori knowledge of the behavior of the algorithm. At the same time, we are interested in improving the overall performance of the system.

To this end, we introduce *Mizan*[1], a Pregel system that supports fine-grained load balancing across supersteps. Similar to other Pregel implementations, Mizan supports different schemes to pre-partition the input graph. Unlike existing implementations, Mizan monitors runtime characteristics of all vertices (i.e., their execution time, and incoming and outgoing messages). Using these measurements, at the end of every superstep, Mizan constructs a migration plan that minimizes the variations across workers by identifying which vertices to migrate and where to migrate them to. Finally, Mizan performs efficient vertex migration across workers, leveraging a distributed hash table (DHT)-based location service to track the movement of vertices as they migrate.

All components of Mizan support distributed execution, eliminating the need for a centralized controller. We have fully implemented Mizan in C++, with the messaging—BSP—layer implemented using MPI [3]. Using a representative number of datasets and a broad spectrum of algorithms, we compare our implementation against Giraph, a popular Pregel clone, and show that Mizan consistently outperforms Giraph by an average of 202%. Normalizing across platforms, we show that for stable workloads, Mizan's dynamic load balancing matches the performance of the static partitioning without requiring a priori knowledge of the structure of the graph or algorithm runtime characteristics. When the algorithm is highly dynamic, we show that Mizan provides over 87% performance improvement. Even with inefficient input partitioning, Mizan is able to reduce the computation overhead by up to 40% when compared to the static case, incurring only a 10% overhead for performing dynamic vertex migration. We even run our implementation on an IBM Blue Gene/P supercompter, demonstrating linear scalability to 1024 CPUs. To the best of our knowledge, this is the largest scale out test of a Pregel-like system to date.

To summarize, we make the following contributions:

- We analyze different graph algorithm characteristics that can contribute to imbalanced computation of a Pregel system.
- We propose a dynamic vertex migration model based on runtime monitoring of vertices to optimize the end-to-end computation.
- We fully implement Mizan in C++ as an optimized Pregel system that supports dynamic load balancing and efficient vertex migration.
- We deploy Mizan on a local Linux cluster (21 machines) and evaluate its efficacy on a representative number of datasets. We also show the linear scalability of our design by running Mizan on a 1024-CPU IBM Blue Gene/P.

This paper is organized as follows. Section 2 describes factors affecting algorithm behavior and discusses a number of example algorithms. In Section 3, we introduce the design of Mizan. We describe implementation details in Section 4. Section 5 compares the performance of Mizan using a wide array of datasets and algorithms. We then describe related work in Section 6. Section 7 discusses future work and Section 8 concludes.

2. Dynamic Behavior of Algorithms

In the Pregel (BSP) computing model, several factors can affect the runtime performance of the underlying system (shown in Figure 1). During a superstep, each vertex is either in an active or inactive state. In an active state, a vertex may be computing, sending messages, or processing received messages.

Naturally, vertices can experience variable execution times depending on the algorithm they implement or their degree of connectivity. A densely connected vertex, for example, will likely spend more time sending and processing incoming messages than a sparsely connected vertex. The BSP programming model, however, masks this variation by (1) overlapping communication and computation, and (2) running many vertices on the same compute node. Intuitively, as long as the workloads of vertices are roughly balanced across computing nodes, the overall computation time will be minimized.

Counter to intuition, achieving a balanced workload is not straightforward. There are two sources of imbalance: one originating from the graph structure and another from the algorithm behavior. In both cases, vertices on some compute nodes can spend a disproportional amount of time computing, sending or processing messages. In some cases, these vertices can run out of input buffer capacity and start paging, further exacerbating the imbalance.

Existing implementations of Pregel (including Giraph [8], GoldenOrb [9], Hama [28]) focus on providing multiple alternatives to partitioning the graph data. The three common approaches to partitioning the data are hash-based, range-based, or min-cut [16]. Hash- and range-based parti-

[1] Mizan is Arabic for a double-pan scale, commonly used in reference to achieving balance.

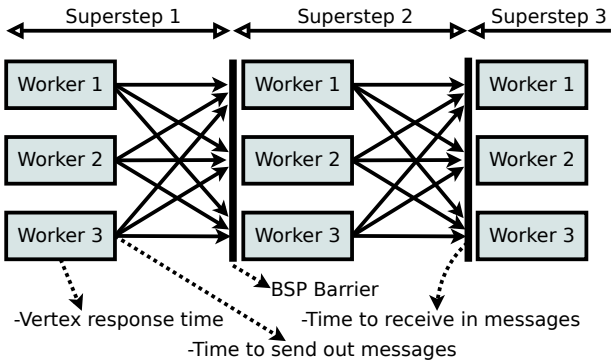

Figure 1. Factors that can affect the runtime in the Pregel framework

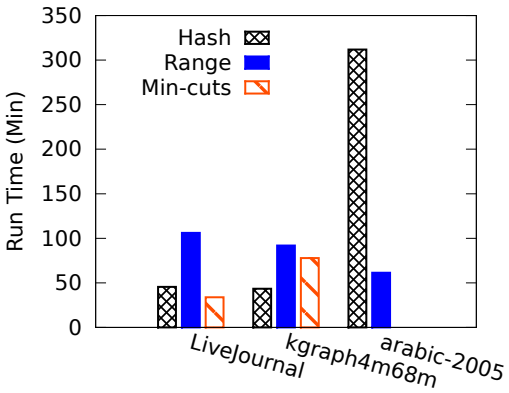

Figure 2. The difference in execution time when processing PageRank on different graphs using hash-based, range-based and min-cuts partitioning. Because of its size, `arabic-2005` cannot be partitioned using min-cuts (ParMETIS) in our local cluster.

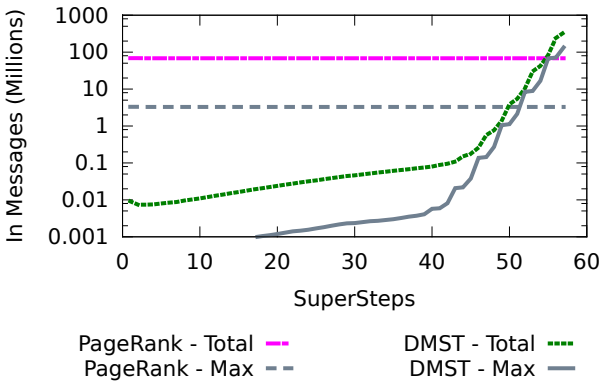

Figure 3. The difference between stationary and non-stationary graph algorithms with respect to the incoming messages. *Total* represents the sum across all workers and *Max* represents the maximum amount (on a single worker) across all workers.

tioning methods divide a dataset based on a simple heuristic: to evenly distribute vertices across compute nodes, irrespective of their edge connectivity. Min-cut based partitioning, on the other hand, considers vertex connectivity and partitions the data such that it places strongly connected vertices close to each other (i.e., on the same cluster). The resulting performance of these partitioning approaches is, however, graph dependent. To demonstrate this variability, we ran a simple—and highly predictable—PageRank algorithm on different datasets (summarized in Table 1) using the three popular partitioning methods. Figure 2 shows that none of the partitioning methods consistently outperforms the rest, noting that ParMETIS [15] partitioning cannot be performed on the `arabic-2005` graph due to memory limitations.

In addition to the graph structure, the running algorithm can also affect the workload balance across compute nodes. Broadly speaking, graph algorithms can be divided into two categories (based on their communication characteristics across supersteps): *stationary* and *non-stationary*.

Stationary Graph Algorithms. An algorithm is stationary if its active vertices send and receive the same distribution of messages across supersteps. At the end of a stationary algorithm, all active vertices become inactive (terminate) during the same superstep. Usually graph algorithms represented by a matrix-vector multiplication[2] are stationary algorithms, including PageRank, diameter estimation and finding weakly connected components.

Non-stationary Graph Algorithms. A graph algorithm is *non-stationary* if the destination or size of its outgoing messages changes across supersteps. Such variations can create workload imbalances across supersteps. Examples of non-stationary algorithms include distributed minimal spanning tree construction (DMST), graph queries, and various simulations on social network graphs (e.g., advertisement propagation).

[2] The matrix represents the graph adjacency matrix and the vector represents the vertices' value.

To illustrate the differences between the two classes of algorithms, we compared the runtime behavior of PageRank (stationary) against DMST (non-stationary) when processing the same dataset (LiveJournal1) on a cluster of 21 machines. The input graph was partitioned using a hash function. Figure 3 shows the variability in the incoming messages per superstep for all workers. In particular, the variability can span over five orders of magnitude for non-stationary algorithms.

The remainder of the section describes four popular graph mining algorithms that we use throughout the paper. They cover both stationary and non-stationary algorithms.

2.1 Example Algorithms

PageRank. PageRank [24] is a stationary algorithm that uses matrix-vector multiplications to calculate the eigenval-

ues of the graph's adjacency matrix at each iteration.[3] The algorithm terminates when the PageRank values of all nodes change by less than a defined $error$ during an iteration. At each superstep, all vertices are active, where every vertex always receives messages from its in-edges and sends messages to all of its out-edges. All messages have fixed size (8 bytes) and the computation complexity on each vertex is linear to the number of messages.

Top-K Ranks in PageRank. It is often desirable to perform graph queries, like using PageRank to find the top k vertex ranks reachable to a vertex after y supersteps. In this case, PageRank runs for x supersteps; at superstep $x + 1$, each vertex sends its rank to its direct out neighbors, receives other ranks from its in neighbors, and stores the highest received k ranks. At supersteps $x + 2$ and $x + y$, each vertex sends the top-K ranks stored locally to its direct out neighbors, and again stores the highest k ranks received from its in neighbors. The message size generated by each vertex can be between $[1..k]$, which depends on the number of ranks stored the vertex. If the highest k values maintained by a vertex did not change at a superstep, that vertex votes to halt and becomes inactive until it receives messages from other neighbors. The size of the message sent between two vertices varies depending on the actual ranks. We classify this algorithm as non-stationary because of the variability in message size and number of messages sent and received.

Distributed Minimal Spanning Tree (DMST). Implementing the *GHS* algorithm of Gallager, Humblet and Spira [7] in the Pregel model requires that, given a weighted graph, every vertex classifies the state of its edges as (1) *branch* (i.e., belongs to the minimal spanning tree (MST)), (2) *rejected* (i.e., does not belong to the MST), or (3) *basic* (i.e., unclassified), in each superstep. The algorithm starts with each vertex as a fragment, then joins fragments until there is only one fragment left (i.e., the MST). Each fragment has an ID and level. Each vertex updates its fragment ID and level by exchanging messages with its neighbors. The vertex has two states: active and inactive. There is also a list of auto-activated vertices defined by the user. At the early stage of the algorithm, active vertices send messages over the minimum weighted edges (i.e., the edge that has the minimum weight among the other in-edges and out-edges). As the algorithm progresses, more messages are sent across many edges according to the number of fragments identified during the MST computations at each vertex. At the last superstep, each vertex knows which of its edges belong to the minimum weighted spanning tree. The computational complexity for each vertex is quadratic to the number of incoming messages. We classify this algorithm as non-stationary because the flow of messages between vertices is unpredictable and depends on the state of the edges at each vertex.

Simulating Advertisements on Social Networks. To simulate advertisements on a social network graph, each vertex represents a profile of a person containing a list of his/her interests. A small set of selected vertices are identified as sources and send different advertisements to their direct neighbors at each superstep for a predefined number of supersteps. When a vertex receives an advertisement, it is either forwarded to the vertex's neighbors (based on the vertex's interest matrix) or ignored. This algorithm has a computation complexity that depends on the count of active vertices at a superstep and whether the active vertices communicate with their neighbors or not. It is, thus, a non-stationary algorithm.

We have also implemented and evaluated a number of other—stationary and non-stationary—algorithms including diameter estimation and finding weakly connected components. Given that their behavior is consistent with the results in the paper, we omit them for space considerations.

3. Mizan

Mizan is a BSP-based graph processing system that is similar to Pregel, but focuses on efficient dynamic load balancing of both computation and communication across all worker (compute) nodes. Like Pregel, Mizan first reads and partitions the graph data across workers. The system then proceeds as a series of supersteps, each separated by a global synchronization barrier. During each superstep, each vertex processes incoming messages from the previous superstep and sends messages to neighboring vertices (which are processed in the following superstep). Unlike Pregel, Mizan balances its workload by moving selected vertices across workers. Vertex migration is performed when all workers reach a superstep synchronization barrier to avoid violating the computation integrity, isolation and correctness of the BSP compute model.

This section describes two aspects of the design of Mizan related to vertex migration: the distributed runtime monitoring of workload characteristics and a distributed migration planner that decides which vertices to migrate. Other design aspects that are not related to vertex migration (e.g., fault tolerance) follow similar approaches to Pregel [23], Giraph [8], GraphLab [22] and GPS [27]; they are omitted for space considerations. Implementation details are described in Section 4.

3.1 Monitoring

Mizan monitors three key metrics for each vertex and maintains a high level summary these metrics for each worker node; summaries are broadcast to other workers at the end of every superstep. As shown in Figure 4, the key metrics for every vertex are (1) the number of outgoing messages to other (remote) workers, (2) total incoming messages, and (3)

[3] Formally, each iteration calculates: $v^{(k+1)} = cA^t v^k + (1-c)/|N|$, where v^k is the eigenvector of iteration k, c is the damping factor used for normalization, and A is a row normalized adjacency matrix.

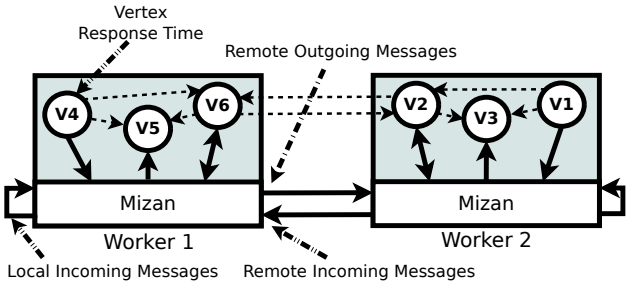

Figure 4. The statistics monitored by Mizan

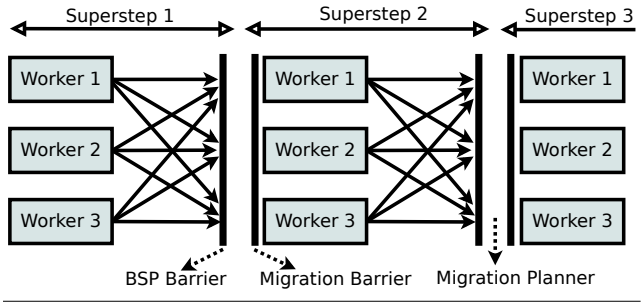

Figure 5. Mizan's BSP flow with migration planning

the response time (execution time) during the current superstep:

Outgoing Messages. Only outgoing messages to other vertices in remote workers are counted since the local outgoing messages are rerouted to the same worker and do not incur any actual network cost.

Incoming Messages. All incoming messages are monitored, those that are received from remote vertices and those locally generated. This is because queue size can affect the performance of the vertex (i.e., when its buffer capacity is exhausted, paging to disk is required).

Response Time. The response time for each vertex is measured. It is the time when a vertex starts processing its incoming messages until it finishes.

3.2 Migration Planning

High values for any of the three metrics above may indicate poor vertex placement, which leads to workload imbalance. As with many constrained optimization problems, optimizing against three objectives is non-trivial. To reduce the optimization search space, Mizan's migration planner finds the strongest cause of workload imbalance among the three metrics and plans the vertex migration accordingly.

By design, all workers create and execute the migration plan in parallel, without requiring any centralized coordination. The migration planner starts on every worker at the end of each superstep (i.e., when all workers reach the synchronization barrier), after it receives summary statistics (as described in Section 3.1) from all other workers. Additionally, the execution of the migration planner is sandwiched between a second synchronization barrier as shown in Figure 5, which is necessary to ensure correctness of the BSP model. Mizan's migration planner executes the following five steps, summarized in Figure 6:

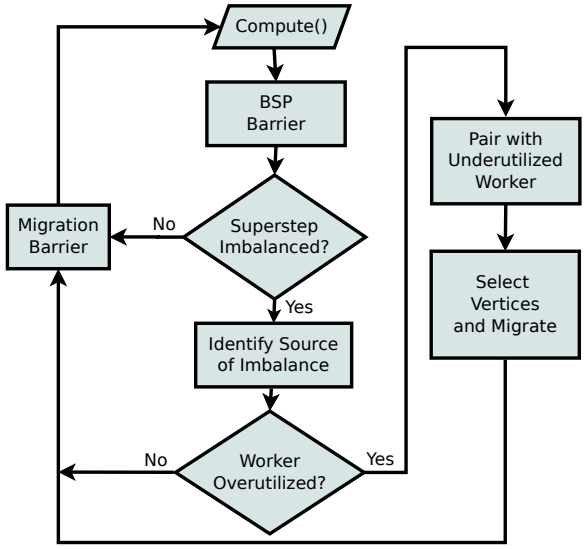

Figure 6. Summary of Mizan's Migration Planner

1. Identify the source of imbalance.
2. Select the migration objective (i.e., optimize for outgoing messages, incoming messages, or response time).
3. Pair over-utilized workers with under-utilized ones.
4. Select vertices to migrate.
5. Migrate vertices.

STEP 1: Identify the source of imbalance. Mizan detects imbalances across supersteps by comparing the summary statistics of all workers against a normal random distribution and flagging outliers. Specifically, at the end of a superstep, Mizan computes the z-score[4] for all workers. If any worker has z-score greater than z_{def}, Mizan's migration planner flags the superstep as imbalanced. We found that $z_{def} = 1.96$, the commonly recommend value [26], allows for natural workload fluctuations across workers. We have experimented with different values of z_{def} and validated the robustness of our choice.

STEP 2: Select the migration objective. Each worker—identically—uses the summary statistics to compute the correlation between outgoing messages and response time, and also the correlation between incoming messages and response time. The correlation scores are used to select the objective to optimize for: to balance outgoing messages, balance incoming messages, or balance computation time. The default objective is to balance the response time. If the

[4] The z-score $Wz_i = \frac{|x_i - x_{max}|}{standard\ deviation}$, where x_i is the run time of worker i

outgoing or incoming messages are highly correlated with the response time, then Mizan chooses the objective with highest correlation score. The computation of the correlation score is described in Appendix A.

STEP 3: Pair over-utilized workers with under-utilized ones. Each overutilized worker that needs to migrate vertices out is paired with a single underutilized worker. While complex pairings are possible, we choose a design that is efficient to execute, especially since the exact number of vertices that different workers plan to migrate is not globally known. Similar to the previous two steps in the migration plan, this step is executed by each worker without explicit global synchronization. Using the summary statistics for the chosen migration objective (in Step 2), each worker creates an ordered list of all workers. For example, if the objective is to balance outgoing messages, then the list will order workers from highest to lowest outgoing messages. The resulting list, thus, places overutilized workers at the top and least utilized workers at the bottom. The pairing function then successively matches workers from opposite ends of the ordered list. As depicted in Figure 7, if the list contains n elements (one for each worker), then the worker at position i is paired with the worker at position $n - i$. In cases where a worker does not have memory to receive any vertices, the worker is marked unavailable in the list.

STEP 4: Select vertices to migrate. The number of vertices to be selected from an overutilized worker depends on the difference of the selected migration objective statistics with its paired worker. Assume that w_x is a worker that needs to migrate out a number of vertices, and is paired with the receiver, w_y. The load that should be migrated to the underutilized worker is defined as Δ_{xy}, which equals to half the difference in statistics of the migration objective between the two workers. The selection criteria of the vertices depends on the distribution of the statistics of the migration objective, where the statistics of each vertex is compared against a normal distribution. A vertex is selected if it is an outlier (i.e., if its Vz^i_{stat}[5]). For example, if the migrating objective is to balance the number of remote outgoing messages, vertices with large remote outgoing messages are selected to migrate to the underutilized worker. The sum of the statistics of the selected vertices is denoted by $\sum V_{stat}$ which should minimize $|\Delta_{xy} - \sum V_{stat}|$ to ensure the balance between w_x and w_y in the next superstep. If there not enough outlier vertices are found, a random set of vertices are selected to minimize $|\Delta_{xy} - \sum V_{stat}|$.

STEP 5: Migrate vertices. After the vertex selection process, the migrating workers start sending the selected vertices while other workers wait at the migration barrier. A migrating worker starts sending the selected set of vertices to its unique target worker, where each vertex is encoded into a stream that includes the vertex ID, state, edge information and the received messages it will process. Once a vertex stream is successfully sent, the sending worker deletes the sent vertices so that it does not run them in the next superstep. The receiving worker, on the other hand, receives vertices (together with their messages) and prepares to run them in the next superstep. The next superstep is started once all workers finish migrating vertices and reach the migration barrier. The complexity of the migration process is directly related to the size of vertices being migrated.

[5] The z-score $Vz^i_{stat} = \frac{|x_i - x_{avg}|}{standard\ deviation}$, where x_i is the statistics of the migration objective of vertex i is greater than the z_{def}

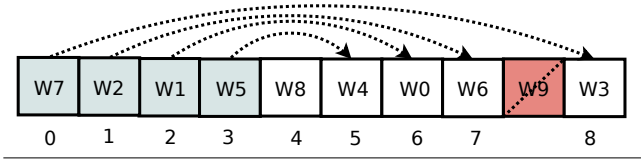

Figure 7. Matching senders (workers with vertices to migrate) with receivers using their summary statistics

4. Implementation

Mizan consists of four modules, shown in Figure 8: the *BSP Processor*, *Storage Manager*, *Communicator*, and *Migration Planner*. The BSP Processor implements the Pregel APIs, consisting primarily of the `Compute` class, and the `SendMessageTo`, `GetOutEdgeIterator` and `getValue` methods. The BSP Processor operates on the data structures of the graph and executes the user's algorithm. It also performs barrier synchronization with other workers at the end of each superstep. The Storage Manager module maintains access atomicity and correctness of the graph's data, and maintains the data structures for message queues. Graph data can be read and written to either HDFS or local disks, depending on how Mizan is deployed. The Communicator module uses MPI to enable communication between workers; it also maintains distributed vertex ownership information. Finally, the Migration Planner operates transparently across superstep barriers to maintain the dynamic workload balance.

Mizan allows the user's code to manipulate the graph connectivity by adding and removing vertices and edges at any superstep. It also guarantees that all graph mutation commands issued at $superstep_x$ are executed at the end of the same superstep and before the BSP barrier, which is illustrated in Figure 5. Therefore, vertex migrations performed by Mizan do not conflict with the user's graph mutations and Mizan always considers the most recent graph structure for migration planning.

When implementing Mizan, we wanted to avoid having a centralized controller. Overall, the BSP (Pregel) model naturally lends itself to a decentralized implementation. There were, however, three key challenges in implementing a distributed control plane that supports fine-grained vertex migration. The first challenge was in maintaining vertex ownership so that vertices can be freely migrated across work-

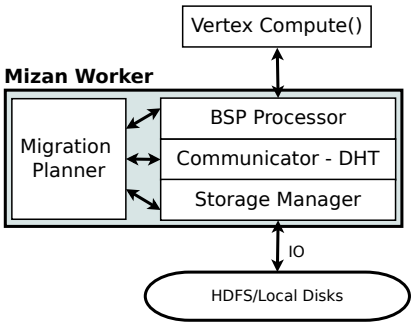

Figure 8. Architecture of Mizan

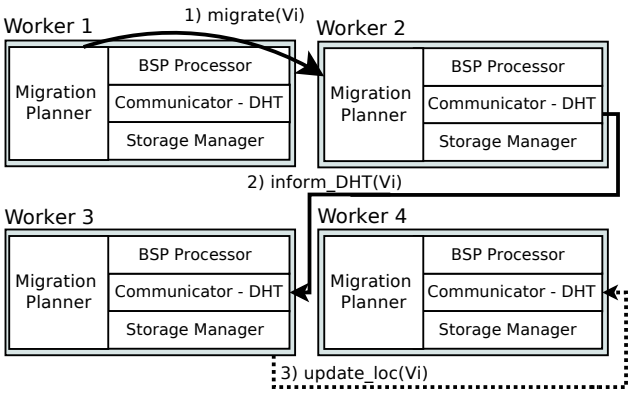

Figure 9. Migrating vertex v_i from Worker 1 to Worker 2, while updating its DHT home worker (Worker 3)

ers. This is different from existing approaches (e.g., Prege Giraph, and GoldenOrb), which operate on a much coarse granularity (clusters of vertices) to enable scalability. Th second challenge was in allowing fast updates to the verte ownership information as vertices get migrated. The thir challenge was in minimizing the cost of migrating vertices with large data structures. In this section, we discuss the implementation details around these three challenges, which allow Mizan to achieve its effectiveness and scalability.

4.1 Vertex Ownership

With huge datasets, Mizan workers cannot maintain the management information for all vertices in the graph. Management information includes the collected statistics for each vertex (described in Section 3.1) and the location (ownership) of each vertex. While per-vertex monitoring statistics are only used locally by the worker, vertex ownership information is needed by all workers. When vertices send messages, workers need to know the destination address for each message. With frequent vertex migration, updating the location of the vertices across all workers can easily create a communication bottleneck.

To overcome this challenge, we use a distributed hash table (DHT) [1] to implement a distributed lookup service. The DHT implementation allows Mizan to distribute the overhead of looking up and updating vertex location across all workers. The DHT stores a set of *(key,value)* pairs, where the key represents a vertex ID and the value represents its current physical location. Each vertex is assigned a *home* worker. The role of the home worker is to maintain the current location of the vertex. A vertex can physically exist in any worker, including its home worker. The DHT uses a globally defined hash function that maps the keys to their associated home workers, such that `home_worker = location_hash(key)`.

During a superstep, when a (source) vertex sends a message to another (target) vertex, the message is passed to the Communicator. If the target vertex is located on the same worker, it is rerouted back to the appropriate queue. Otherwise, the source worker uses the `location_hash` function to locate and query the home worker for the target vertex. The home worker responds back with the actual physical location of target vertex. The source worker finally sends the queued message to the current worker for the target vertex. It also caches the physical location to minimize future lookups.

4.2 DHT Updates After Vertex Migration

Figure 9 depicts the vertex migration process. When a vertex v migrates between two workers, the receiving worker sends the new location of v to the home worker of v. The home worker, in turn, sends an update message to all workers that have previously asked for—and, thus, potentially cached—the location of v. Since Mizan migrates vertices in the barrier between two supersteps, all workers that have cached the location of the migrating vertex will receive the updated physical location from the home worker before the start of the new superstep.

If for any reason a worker did not receive the updated location, the messages will be sent to the last known physical location of v. The receiving worker, which no longer owns the vertex, will simply buffer the incorrectly routed messages, ask for the new location for v, and reroute the messages to the correct worker.

4.3 Migrating Vertices with Large Message Size

Migrating a vertex to another worker requires moving its queued messages and its entire state (which includes its ID, value, and neighbors). Especially when processing large graphs, a vertex can have a significant number of queued messages, which are costly to migrate. To minimize the cost, Mizan migrates large vertices using a *delayed migration* process that spreads the migration over two supersteps. Instead of physically moving the vertex with its large message queue, delayed migration only moves the vertex's information and the ownership to the new worker. Assuming w_{old} is the old owner and w_{new} is the new one, w_{old} continues to process the migrating vertex v in the next superstep, SS_{t+1}. w_{new} receives the messages for v, which will be processed at the following superstep, SS_{t+2}. At the end of SS_{t+1}, w_{old} sends the new value of v, calculated at SS_{t+1}, to w_{new} and completes the delayed migration. Note that migration plan-

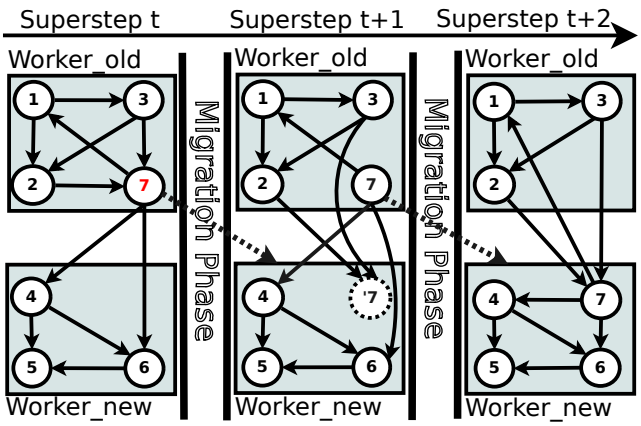

Figure 10. Delayed vertex migration

$G(N, E)$	$\|N\|$	$\|E\|$
kg1	1,048,576	5,360,368
kg4m68m	4,194,304	68,671,566
web-Google	875,713	5,105,039
LiveJournal1	4,847,571	68,993,773
hollywood-2011	2,180,759	228,985,632
arabic-2005	22,744,080	639,999,458

Table 1. Datasets—N, E denote nodes and edges, respectively. Graphs with prefix kg are synthetic.

ning is disabled for superstep SS_{t+1} after applying delayed migration to ensure the consistency of the migration plans.

An example is shown in Figure 10, where Vertex 7 is migrated using delayed migration. As Vertex 7 migrates, the ownership of Vertex 7 is moved to `Worker_new` and messages sent to Vertex 7 at SS_{t+1} are addressed to `Worker_new`. At the barrier of SS_{t+1}, `Worker_old` sends the edge information and state of v to `Worker_new` and starts SS_{t+2} with v fully migrated to `Worker_new`.

With delayed migration, the consistency of computation is maintained without introducing additional network cost for vertices with large message queues. Since delayed migration uses two supersteps to complete, Mizan may need more steps before it converges on a balanced state.

5. Evaluation

We implemented Mizan using C++ and MPI and compared it against Giraph [8], a Pregel clone implemented in Java on top of Hadoop. We ran our experiments on a local cluster of 21 machines equipped with a mix of i5 and i7 processors with 16GB RAM on each machine. We also used an IBM Blue Gene/P supercomputer with 1024 PowerPC-450 CPUs, each with 4 cores at 850MHz and 4GB RAM. We downloaded publicly available datasets from the Stanford Network Analysis Project[6] and from The Laboratory for

[6] `http://snap.stanford.edu`

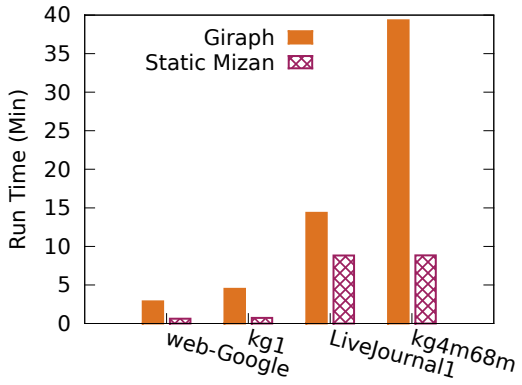

Figure 11. Comparing Static Mizan vs. Giraph using PageRank on social network and random graphs

Web Algorithmics (LAW) [2, 20]. We also generated synthetic datasets using the Kronecker [21] generator that models the structure of real life networks. The details are shown in Table 1.

To better isolate the effects of dynamic migration on system performance, we implemented three variations of Mizan: Static, Work Stealing (WS), and Mizan. Static Mizan disables any dynamic migration and uses either hash-based, range-based or min-cuts graph pre-partitioning (rendering it similar to Giraph). Work Stealing (WS) Mizan is our attempt to emulate Pregel's coarse-grained dynamic load balancing behavior (described in [23]). Finally, Mizan is our framework that supports dynamic migration as described in Sections 3 and 4.

We evaluated Mizan across three dimensions: partitioning scheme, graph structure, and algorithm behavior. We partition the datasets using three schemes: hash-based, range-based, and METIS partitioning. We run our experiments against both social and random graphs. Finally, we experimented with the algorithms described in Section 2.1.

5.1 Giraph vs. Mizan

We picked Giraph for its popularity as an open source Pregel framework with broad adoption. We only compared Giraph to Static Mizan, which allows us to evaluate our base (non-dynamic) implementation. Mizan's dynamic migration is evaluated later in this section. To further equalize Giraph and the Static Mizan, we disabled the fault tolerance feature of Giraph to eliminate any additional overhead resulting from frequent snapshots during runtime. In this section, we report the results using PageRank (a stationary and well balanced algorithm) using different graph structures.

In Figure 11, we ran both frameworks on social network and random graphs. As shown, Static Mizan outperforms Giraph by a large margin, up to four times faster than Giraph in kg4m68, which contains around 70M edges. We also compared both systems when increasing the number of nodes in random structure graphs. As shown in Figure 12, Static

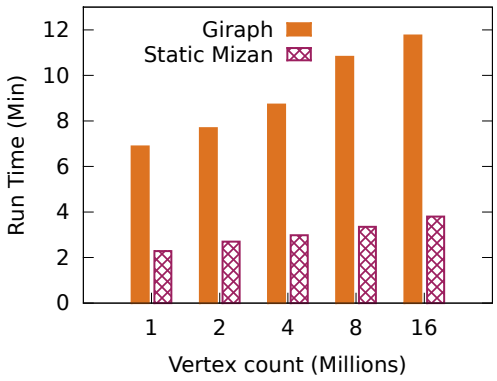

Figure 12. Comparing Mizan vs. Giraph using PageRank on regular random graphs, the graphs are uniformly distributed with each has around 17M edge

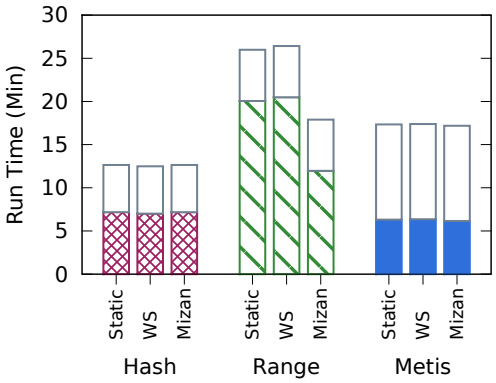

Figure 13. Comparing Static Mizan and Work Stealing (Pregel clone) vs. Mizan using PageRank on a social graph (`LiveJournal1`). The shaded part of each column represents the algorithm runtime while unshaded parts represents the initial partitioning cost of the input graph.

Mizan consistently outperforms Giraph in all datasets and reaches up to three times faster with 16 million vertexes. While the execution time of both frameworks increases linearly with graph size, the rate of increase—slope of the graph—for Giraph (0.318) is steeper than Mizan (0.09), indicating that Mizan also achieves better scalability.

The experiments in Figures 12 and 11 show that Giraph's implementation is inefficient. It is a non-trivial task to discover the source of inefficiency in Giraph since it is tightly coupled with Hadoop. We suspect that part of the inefficiency is due to the initialization cost of the Hadoop jobs and the high overhead of communication. Other factors, like internal data structure choice and memory footprint, might also play a role in this inefficiency.

5.2 Effectiveness of Dynamic Vertex Migration

Given the large performance difference between Static Mizan and Giraph, we exclude Giraph from further experiments and focus on isolating the effects of dynamic migration on the overall performance of the system.

Figure 13 shows the results for running PageRank on a social graph with various partitioning schemes. We notice that both hash-based and METIS partitioning achieve a balanced workload for PageRank such that dynamic migration did not improve the results. In comparison, range-based partitioning resulted in poor graph partitioning. In this case, we observe Mizan (with dynamic partitioning) was able to reduce execution time by approximately 40% when compared to the static version. We have also evaluated the diameter estimation algorithm, which behaves the same as PageRank, but exchanges larger size messages. Mizan exhibited similar behavior with diameter estimation; the results are omitted for space considerations.

Figure 14 shows how Mizan's dynamic migration was able to optimize running PageRank starting with range-based partitioning. The figure shows that Mizan's migration reduced both the variance in workers' runtime and the superstep's runtime. The same figure also shows that Mizan's migration alternated the optimization objective between the number of outgoing messages and vertex response time, illustrated as points on the superstep's runtime trend.

By looking at both Figures 14 and 15, we observe that the convergence of Mizan's dynamic migration is correlated with the algorithm's runtime reduction. For the PageRank algorithm with range-based partitioning, Mizan requires 13 supersteps to reach an acceptable balanced workload. Since PageRank is a stationary algorithm, Mizan's migration converged quickly; we expect that it would require more supersteps to converge on other algorithms and datasets. In general, Mizan requires multiple supersteps before it balances the workload. This also explains why running Mizan with range-based partitioning is less efficient than running Mizan with METIS or hash-based partitioning.

As described in Section 2.1, Top-K PageRank adds variability in the communication among the graph nodes. As shown in Figure 16, such variability in the messages exchanged leads to minor variation in both hash-based and METIS execution times. Similarly, in range partitioning, Mizan had better performance than the static version. The slight performance improvement arises from the fact that the base algorithm (PageRank) dominates the execution time. If a large number of queries are performed, the improvements will be more significant.

To study the effect of algorithms with highly variable messaging patterns, we evaluated Mizan using two algorithms: DMST and advertisement propagation simulation. In both cases, we used METIS to pre-partition the graph data. METIS partitioning groups the strongly connected subgraphs into clusters, thus minimizing the global communication among each cluster.

In DMST, as discussed in Section 2.1, computation complexity increases with vertex connectivity degree. Because

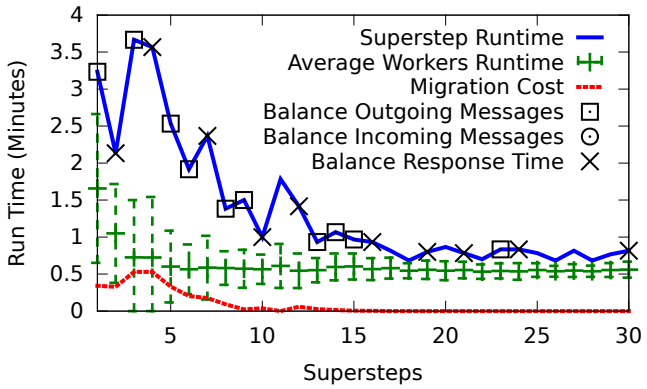

Figure 14. Comparing Mizan's migration cost with the superstep's (or the algorithm) runtime at each supertstep using PageRank on a social graph (`LiveJournal1`) starting with range based partitioning. The points on the superstep's runtime trend represents Mizan's migration objective at that specific superstep.

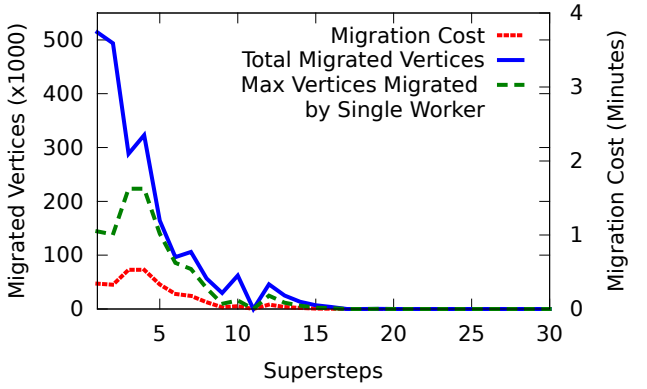

Figure 15. Comparing both the total migrated vertices and the maximum migrated vertices by a single worker for PageRank on `LiveJournal1` starting with range-based partitioning. The migration cost at each superstep is also shown.

of the quadratic complexity of computation as a function of connectivity degree, some workers will suffer from extensive workload while others will have light workload. Such imbalance in the workload leads to the results shown in Figure 17. Mizan was able to reduce the imbalance in the workload, resulting in a large drop in execution time (two orders of magnitude improvement). Even when using our version of Pregel's load balancing approach (called work stealing), Mizan is roughly eight times faster.

Similar to DMST, the behavior of the advertising propagation simulation algorithm varies across supersteps. In this algorithm, a dynamic set of graph vertices communicate heavily with each other, while others have little or no communication. In every superstep, the communication behavior differs depending on the state of the vertex. Therefore, it creates an imbalance across the workers for every super-

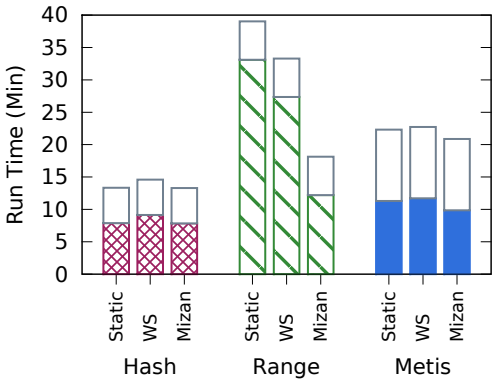

Figure 16. Comparing static Mizan and Work Stealing (Pregel clone) vs. Mizan using Top-K PageRanks on a social graph (`LiveJournal1`). The shaded part of each column represents the algorithm runtime while unshaded parts represents the initial partitioning cost of the input graph.

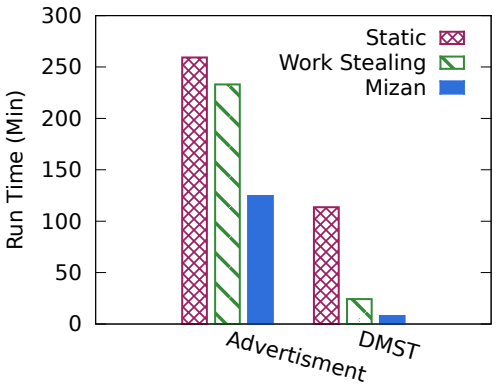

Figure 17. Comparing Work stealing (Pregel clone) vs. Mizan using DMST and Propagation Simulation on a metis partitioned social graph (`LiveJournal1`)

Total runtime (s)	707
Data read time to memory (s)	72
Migrated vertices	1,062,559
Total migration time (s)	75
Average migrate time per vertex (μs)	70.5

Table 2. Overhead of Mizan's migration process when compared to the total runtime using range partitioning on `LiveJournal1` graph

step. Even METIS partitioning in such case is ill-suited since workers' load dynamically changes at runtime. As shown in Figure 17, similar to DMST, Mizan is able to reduce such an imbalance, resulting in approximately 200% speed up when compared to the work stealing and static versions.

Linux Cluster		Blue Gene/P	
hollywood-2011		arabic-2005	
Processors	Runtime (m)	Processors	Runtime (m)
2	154	64	144.7
4	79.1	128	74.6
8	40.4	256	37.9
16	21.5	512	21.5
		1024	17.5

Table 3. Scalability of Mizan on a Linux Cluster of 16 machines (`hollywood-2011` dataset), and an IBM Blue Gene/P supercomputer (`arabic-2005` dataset).

5.3 Overhead of Vertex Migration

To analyze migration cost, we measured the time for various performance metrics of Mizan. We used the PageRank algorithm with a range-based partitioning of the `Live-Journal1` dataset on 21 workers. We chose range-based partitioning as it provides the worst data distribution according to previous experiments and therefore will trigger frequent vertex migrations to balance the workload.

Table 2 reports the average cost of migrating as 70.5 μs per vertex. In the `LiveJournal1` dataset, Mizan paid a 9% penalty of the total runtime to balance the workload, transferring over 1M vertices. As shown earlier in Figure 13, this resulted in a 40% saving in computation time when compared to Static Mizan. Moreover, Figures 14 and 15 compare the algorithm runtime and the migration cost at each superstep, the migration cost is at most 13% (at superstep 2) and on average 6% for all supersteps that included a migration phase.

5.4 Scalability of Mizan

We tested the scalability of Mizan on the Linux cluster as shown in Table 3. We used two compute nodes as our base reference as a single node was too small when running the dataset (`hollywood-2011`), causing significant paging activities. As Figure 18 shows, Mizan scales linearly with the number of workers.

We were interested in performing large scale-out experiments, well beyond what can be achieved on public clouds. Since Mizan's was written in C++ and uses MPI for message passing, it was easily ported to IBM's Blue Gene/P supercomputer. Once ported, we natively ran Mizan on 1024 Blue Gene/P compute nodes. The results are shown in Table 3. We ran the PageRank algorithm using a huge graph (`arabic-2005`) that contains 639M edges. As shown in Figure 19, Mizan scales linearly from 64 to 512 compute nodes then starts to flatten out as we increase to 1024 compute nodes. The flattening was expected since with an increased number of cores, compute nodes will spend more time communicating than computing. We expect that as we continue to increase the number of CPUs, most of the time will be spent communicating (which effectively breaks the BSP model of overlapping communication and computation).

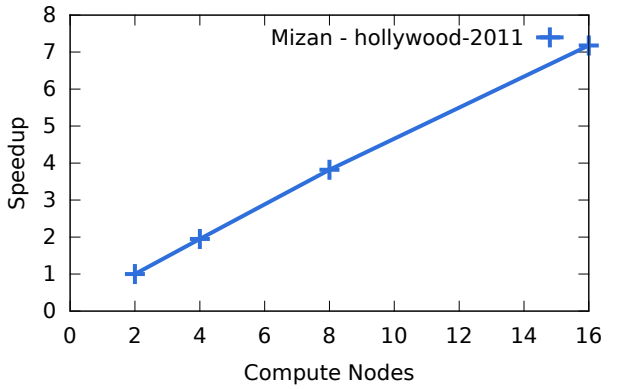

Figure 18. Speedup on Linux Cluster of 16 machines using PageRank on the `hollywood-2011` dataset.

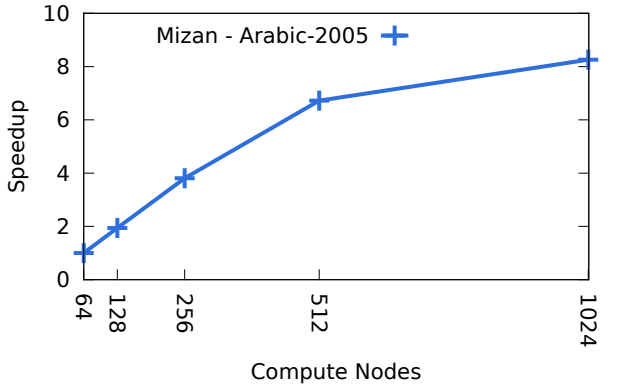

Figure 19. Speedup on Shaheen IBM Blue Gene/P supercomputer using PageRank on the `arabic-2005` dataset.

6. Related Work

In the past few years, a number of large scale graph processing systems have been proposed. A common thread across the majority of existing work is how to partition graph data and how to balance computation across compute nodes. This problem is known as applying dynamic load balancing to the distributed graph data by utilizing on the fly graph partitioning, which Bruce et al. [11] assert to be either too expensive or hard to parallelize. Mizan follows the Pregel model, but focuses on dynamic load balancing of vertexes. In this section, we examine the design aspects of existing work for achieving a balanced workload.

Pregel and its Clones. The default partitioning scheme for the input graph used by Pregel [23] is hash based. In every superstep, Pregel assigns more than one subgraph (partition) to a worker. While this load balancing approach helps in balancing computation in any superstep, it is coarse-grained

and reacts slowly to large imbalances in the initial partitioning or large behavior changes. Giraph, an open source Pregel clone, supports both hash-based and range-based partitioning schemes; the user can also implement a different graph partitioning scheme. Giraph balances the partitions across the workers based on the number of edges or vertices. GoldenOrb [9] uses HDFS as its storage system and employs hash-based partitioning. An optimization that was introduced in [12] uses METIS partitioning, instead of GoldenOrb's hash-based partitioning, to maintain a balanced workload among the partitions. Additionally, its distributed data storage supports duplication of the frequently accessed vertices. Similarly, Surfer [4] utilizes min-cut graph partitioning to improve the performance of distributed vertex centric graph processing. However, Surfer focuses more on providing a bandwidth-aware variant of the multilevel graph partitioning algorithm (integrated into a Pregel implementation) to improve the performance of graph partitioning on the cloud. Unlike Mizan, these systems neither optimize for dynamic algorithm behavior nor message traffic.

Power-law Optimized Graph Processing Systems. Stanford GPS (GPS) [27] has three additional features over Pregel. First, GPS extends Pregel's API to handle graph algorithms that perform both vertex-centric and global computations. Second, GPS supports dynamic repartitioning based on the outgoing communication. Third, GPS provides an optimization scheme called *large adjacency list partitioning*, where the adjacency lists of high degree vertices are partitioned across workers. GPS only monitors outgoing messages, while Mizan dynamically selects the most appropriate metric based on changes in runtime characteristics. PowerGraph [10], on the other hand, is a distributed graph system that overcomes the challenges of processing natural graphs with extreme power-law distributions. PowerGraph introduces vertex-based partitioning with replication to distribute the graph's data based on its edges. It also provides a programming abstraction (based on gather, apply and scatter operators) to distribute the user's algorithm workload on their graph processing system. The authors in [10] have focused on evaluating stationary algorithms on power-law graphs. It is, thus, unclear how their system performs across the broader class of algorithms (e.g., non-stationary) and graphs (e.g., non-power-law). In designing Mizan, we did not make any assumptions on either the type of algorithm or shape of input graph. Nonetheless, PowerGraph's replication approach is complementary to Mizan's dynamic load balancing techniques. They can be combined to improve the robustness of the underlying graph processing system.

Shared Memory Graph Processing Systems. GraphLab [22] is a parallel framework (that uses distributed shared memory) for efficient machine learning algorithms. The authors extended their original multi-core GraphLab implementations to a cloud-based distributed GraphLab, where they apply a two-phase partitioning scheme that minimizes the edges among graph partitions and allows for a fast graph repartitioning. Unlike BSP frameworks, computation in GraphLab is executed on the most recently available data. GraphLab ensures that computations are sequentially consistent through a derived consistency model to guarantee the end result data will be consistent as well. HipG [18] is designed to operate on distributed memory machines while providing transparent memory sharing to support hierarchical parallel graph algorithms. HipG also support processing graph algorithms following the BSP model. But unlike the global synchronous barriers of Pregel, HipG applies more fine-grained barriers, called synchronizers, during algorithm execution. The user has to be involved in writing the synchronizers; thus, it requires additional complexity in the users' code. GraphChi [19] is a disk based graph processing system based on GraphLab and designed to run on a single machine with limited memory. It requires a small number of non-sequential disk accesses to process the subgraphs from disk, while allowing for asynchronous iterative computations. Unlike, Mizan these systems do not support any dynamic load balancing techniques.

Specialized Graph Systems. Kineograph [5] is a distributed system that targets the processing of fast changing graphs. It captures the relationships in the input data, reflects them on the graph structure, and creates regular snapshots of the data. Data consistency is ensured in Kineograph by separating graph processing and graph update routines. This system is mainly developed to support algorithms that mine the incremental changes in the graph structure over a series of graph snapshots. The user can also run an offline iterative analysis over a copy of the graph, similar to Pregel. The Little Engine(s) [25] is also a distributed system designed to scale with online social networks. This system utilizes graph repartitioning with one-hop replication to rebalanced any changes to the graph structure, localize graph data, and reduce the network cost associated with query processing. The Little Engine(s) repartitioning algorithm aims to find min-cuts in a graph under the condition of minimizing the replication overhead at each worker. This system works as a middle-box between an application and its database systems; it is not intended for offline analytics or batch data processing like Pregel or MapReduce. Trinity [29] is a distributed in-memory key-value storage based graph processing system that supports both offline analytics and online query answering. Trinity provides a similar—but more restrictive—computing model to Pregel for offline graph analytics. The key advantage of Trinity is its low latency graph storage design, where the data of each vertex is stored in a binary format to eliminate decoding and encoding costs.While these systems address specific design challenges in processing large graphs, their focus is orthogonal to the design of Mizan.

7. Future Work

We are planning on making Mizan an open source project. We will continue to work on number of open problems as outlined in this section. Our goal is to provide a robust and adaptive platform that frees developers from worrying about the structure of the data or runtime variability of their algorithms.

Although Mizan is a graph-agnostic framework, its performance can degrade when processing extremely skewed graphs due to frequent migration of highly connected vertices. We believe that vertex replication, as proposed by PowerGraph [10], offers a good solution, but mandates the implementation of custom combiners. We plan to further reduce frequent migrations in Mizan by leveraging vertex replication, however, without requiring any custom combiners.

So far, dynamic migration in Mizan optimizes for workloads generated by a single application or algorithm. We are interested in investigating how to efficiently run multiple algorithms on the same graph. This would better position Mizan to be offered as a service for very large datasets.

8. Conclusion

In this paper, we presented *Mizan*, a Pregel system that uses fine-grained vertex migration to load balance computation and communication across supersteps. Using distributed measurements of the performance characteristics of all vertices, Mizan identifies the cause of workload imbalance and constructs a vertex migration plan without requiring centralized coordination. Using a representative number of datasets and algorithms, we have showed both the efficacy and robustness of our design against varying workload conditions. We have also showed the linear scalability of Mizan, scaling to 1024 CPUs.

A. Computing the Correlation Score

In Mizan, the correlation scores between the workers' response time and their summary statistics are calculated using two tests: *clustering* and *sorting*. In the clustering test CT, elements from the set of worker response times S_{Time} and the statistics set S_{Stat} (either incoming or outgoing message statistics) are divided into two categories based on their z-score [7]: positive and negative clusters. Elements from a set S are categorized as positive if their z-scores are greater than some z_{def}. Elements categorized as negative have a z-score less than $-z_{def}$. Elements in the range $(-z_{def}, z_{def})$ are ignored. The score of the cluster test is calculated based on the total count of elements that belongs to the same worker in the positive and negative sets, which is shown by the following equation:

$$CT_{score} = \frac{count(S^{+}_{Time} \cap S^{+}_{Stat}) + count(S^{-}_{Time} \cap S^{-}_{Stat})}{count(S_{Time}) + count(S_{Stat})}$$

The sorting test ST consists of sorting both sets, S_{Time} and S_{Stat}, and calculating the z-scores for all elements. Elements with a z-score in the range $(-z_{def}, z_{def})$ are excluded from both sorted sets. For the two sorted sets S_{Time} and S_{Stat} of size n and $i \in [0..n-1]$, a mismatch is defined if the i-th elements from both sets are related to different workers. That is, S^{i}_{Time} is the i-th element from S_{Time}, S^{i}_{Stat} is the i-th element from S_{Stat}, and $S^{i}_{Time} \in Worker_x$ while $S^{i}_{Stat} \notin Worker_x$. A match, on the other hand, is defined if $S^{i}_{Time} \in Worker_x$ and $S^{i}_{Stat} \in Worker_x$. A score of 1 is given by the sorting test if:

$$ST_{score} = \frac{count(matches)}{count(matches) + count(mismatches)}$$

The correlation of sets S_{Time} and S_{Stat} are reported as strongly correlated if $(CT+ST)/2 \geq 1$, reported as weakly correlated if $1 > (CT + ST)/2 \geq 0.5$ and reported as uncorrelated otherwise.

[7] The z-score $S^{i}_{Time}\&S^{i}_{Stat} = \frac{|x_i - x_{avg}|}{standard\ deviation}$, where x_i is the run time of worker i or its statistics

References

[1] H. Balakrishnan, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Looking Up Data in P2P Systems. *Communications of the ACM*, 46(2):43–48, 2003.

[2] P. Boldi, B. Codenotti, M. Santini, and S. Vigna. UbiCrawler: A Scalable Fully Distributed Web Crawler. *Software: Practice & Experience*, 34(8):711–726, 2004.

[3] D. Buntinas. Designing a Common Communication Subsystem. In *Proceedings of the 12th European Parallel Virtual Machine and Message Passing Interface Conference (Euro PVM MPI)*, pages 156–166, Sorrento, Italy, 2005.

[4] R. Chen, M. Yang, X. Weng, B. Choi, B. He, and X. Li. Improving Large Graph Processing on Partitioned Graphs in the Cloud. In *Proceedings of the 3rd ACM Symposium on Cloud Computing (SOCC)*, pages 3:1–3:13, San Jose, California, USA, 2012.

[5] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, and E. Chen. Kineograph: Taking the Pulse of a Fast-Changing and Connected World. In *Proceedings of the 7th ACM european conference on Computer Systems (EuroSys)*, pages 85–98, Bern, Switzerland, 2012.

[6] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th USENIX Symposium on Opearting Systems Design and Implementation (OSDI)*, pages 137–150, San Francisco, California, USA, 2004.

[7] R. G. Gallager, P. A. Humblet, and P. M. Spira. A Distributed Algorithm for Minimum-Weight Spanning Trees. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5(1):66–77, 1983.

[8] Giraph. Apache Incubator Giraph. `http://incubator.apache.org/giraph`, 2012.

[9] GoldenOrb. A Cloud-based Open Source Project for Massive-Scale Graph Analysis. `http://goldenorbos.org/`, 2012.

[10] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 17–30, Hollywood, California, USA, 2012.

[11] B. Hendrickson and K. Devine. Dynamic Load Balancing in Computational Mechanics. *Computer Methods in Applied Mechanics and Engineering*, 184(2–4):485–500, 2000.

[12] L.-Y. Ho, J.-J. Wu, and P. Liu. Distributed Graph Database for Large-Scale Social Computing. In *Proceedings of the IEEE 5th International Conference in Cloud Computing (CLOUD)*, pages 455–462, Honolulu, Hawaii, USA, 2012.

[13] U. Kang, C. E. Tsourakakis, and C. Faloutsos. PEGASUS: A Peta-Scale Graph Mining System Implementation and Observations. In *Proceedings of the IEEE 9th International Conference on Data Mining (ICDM)*, pages 229–238, Miami, Florida, USA, 2009.

[14] U. Kang, C. Tsourakakis, A. P. Appel, C. Faloutsos, and J. Leskovec. HADI: Fast Diameter Estimation and Mining in Massive Graphs with Hadoop. *ACM Trasactions on Knowledge Discovery from Data (TKDD)*, 5(2):8:1–8:24, 2011.

[15] G. Karypis and V. Kumar. Parallel Multilevel K-Way Partitioning Scheme for Irregular Graphs. In *Proceedings of the ACM/IEEE Conference on Supercomputing (CDROM)*, pages 278–300, Pittsburgh, Pennsylvania, USA.

[16] G. Karypis and V. Kumar. Multilevel k-way Partitioning Scheme for Irregular Graphs. *Journal of Parallel and Distributed Computing*, 48(1):96–129, 1998.

[17] Z. Khayyat, K. Awara, H. Jamjoom, and P. Kalnis. Mizan: Optimizing Graph Mining in Large Parallel Systems. Technical report, King Abdullah University of Science and Technology, 2012.

[18] E. Krepska, T. Kielmann, W. Fokkink, and H. Bal. HipG: Parallel Processing of Large-Scale Graphs. *ACM SIGOPS Operating Systems Review*, 45(2):3–13, 2011.

[19] A. Kyrola, G. Blelloch, and C. Guestrin. GraphChi: Large-Scale Graph computation on Just a PC. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 31–46, Hollywood, California, USA, 2012.

[20] LAW. The Laboratory for Web Algorithmics. `http://law.di.unimi.it/index.php`, 2012.

[21] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, and Z. Ghahramani. Kronecker Graphs: An Approach to Modeling Networks. *Journal of Machine Learning Research*, 11: 985–1042, 2010.

[22] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. Hellerstein. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *Proceedings of the VLDB Endowment (PVLDB)*, 5(8):716–727, 2012.

[23] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A System for Large-Scale Graph Processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, pages 135–146, Indianapolis, Indiana, USA.

[24] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank Citation Ranking: Bringing Order to the Web. `http://ilpubs.stanford.edu:8090/422/`, 2001.

[25] J. M. Pujol, V. Erramilli, G. Siganos, X. Yang, N. Laoutaris, P. Chhabra, and P. Rodriguez. The Little Engine(s) That Could: Scaling Online Social Networks. *IEEE/ACM Transactions on Networking*, 20(4):1162–1175, 2012.

[26] D. Rees. *Foundations of Statistics*. Chapman and Hall, London New York, 1987. ISBN 0412285606.

[27] S. Salihoglu and J. Widom. GPS: A Graph Processing System. Technical report, Stanford University, 2012.

[28] S. Seo, E. Yoon, J. Kim, S. Jin, J.-S. Kim, and S. Maeng. HAMA: An Efficient Matrix Computation with the MapReduce Framework. In *Proceedings of the 2010 IEEE Second International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 721–726, Athens, Greece.

[29] B. Shao, H. Wang, and Y. Li. Trinity: A Distributed Graph Engine on a Memory Cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, New York, New York, USA.

[30] L. G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33:103–111, 1990.

[31] W. Xue, J. Shi, and B. Yang. X-RIME: Cloud-Based Large Scale Social Network Analysis. In *Proceedings of the 2010 IEEE International Conference on Services Computing (SCC)*, pages 506–513, Shanghai, China.