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

Zuhair Khayyat\textsuperscript{1}  Karim Awara\textsuperscript{1}  Amani Alonazi\textsuperscript{1}  
Hani Jamjoom\textsuperscript{2}  Dan Williams\textsuperscript{2}  Panos Kalnis\textsuperscript{1}

\textsuperscript{1}King Abdullah University of Science and Technology
Thuwal, Saudi Arabia

\textsuperscript{2}IBM Watson Research Center
Yorktown Heights, NY
Graphs abstract application-specific algorithms into generic problems represented as interactions using vertices and edges.

- Max flow in road network
- Diameter of WWW
- Ranking in social networks
- Simulating protein interactions

Different applications vary in their computation requirements.
Pregel was introduced by Google as a scalable abstraction for graph processing

- Overcomes the limitations of processing graphs on MapReduce
- Based on vertex-centric computation
- Utilizes bulk synchronous parallel (BSP)

<table>
<thead>
<tr>
<th>System</th>
<th>Programming Abstraction</th>
<th>Data Exchange</th>
<th>Computations</th>
</tr>
</thead>
<tbody>
<tr>
<td>MapReduce</td>
<td>map()</td>
<td>Key-based grouping</td>
<td>Disk-based</td>
</tr>
<tr>
<td></td>
<td>combine()</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>reduce()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pregel</td>
<td>compute()</td>
<td>Message passing</td>
<td>In memory</td>
</tr>
<tr>
<td></td>
<td>combine()</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>aggregate()</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Balanced computation and communication is fundamental to Pregel’s efficiency
Existing work focus on **optimizing for graph structure** (static optimization):

- **Optimize graph partitioning:**
  - Simple graph partitioning schemes (e.g., hash or range): Giraph
  - User-defined partitioning function: Pregel
  - Sophisticated partitioning techniques (e.g., min-cuts): GraphLab, PowerGraph, GoldenOrb and Surfer

- **Optimize graph data access:**
  - Distributed data stores and graph indexing: GoldenOrb, Hama and Trinity
Existing work focus on **optimizing for graph structure** (static optimization):

- **Optimize graph partitioning:**
  - Simple graph partitioning schemes (e.g., hash or range): Giraph
  - User-defined partitioning function: Pregel
  - Sophisticated partitioning techniques (e.g., min-cuts): GraphLab, PowerGraph, GoldenOrb and Surfer

- **Optimize graph data access:**
  - Distributed data stores and graph indexing: GoldenOrb, Hama and Trinity

**What about algorithm behavior?**

- Pregel provides coarse-grained load balancing, is it enough?
Algorithms behave differently, we classify algorithms in two categories depending on their behavior:

<table>
<thead>
<tr>
<th>Algorithm Type</th>
<th>In/Out Messages</th>
<th>Vertices State</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Stationary</strong></td>
<td>Predictable</td>
<td>Fixed</td>
<td>PageRank, Diameter estimation, WCC</td>
</tr>
<tr>
<td><strong>Non-stationary</strong></td>
<td>Variable</td>
<td>Variable</td>
<td>Distributed minimum spanning tree, Belief propagation, Graph queries, Ad propagation</td>
</tr>
</tbody>
</table>
Types of Algorithms – First Superstep

PageRank

DMST
Types of Algorithms – $\kappa$ Superstep

PageRank

DMST
Types of Algorithms – $k + m$ Superstep

PageRank

DMST
What Causes Computation Imbalance in Non-stationary Algorithms?
Why to Optimize for Algorithm Behavior?

- Difference between stationary and non-stationary algorithms

![Graph showing the difference between stationary and non-stationary algorithms](image)

- In Messages (Millions)
- SuperSteps
- PageRank - Total
- PageRank - Max/W
- DMST - Total
- DMST - Max/W
What is Mizan?

- Open source BSP-based graph processing
- Provides runtime load balancing through fine grain vertex migrations
- Follows Pregel model
void compute(messageIterator * messages, userVertexObject * data, 
    messageManager * comm) {

    double currVal = data->getVertexValue().getValue();
    double newVal = 0; double c = 0.85;

    if (data->getCurrentSS() > 1) {
        while (messages->hasNext()) {
            newVal = newVal + messages->getNext().getValue();
        }
        newVal = newVal * c + (1.0 - c) / ((double) vertexTotal);
        data->setVertexValue(mDouble(newVal));
    } else {newVal = currVal;}

    if (data->getCurrentSS() <= maxSuperStep) {
        mDouble outVal(newVal / ((double) data->getOutEdgeCount()));
        for (int i = 0; i < data->getOutEdgeCount(); i++) {
            comm->sendMessage(data->getOutEdgeID(i), outVal);
        }
    } else {data->voteToHalt();}
Migration Planning Objectives

- Decentralized
- Simple
- Transparent
  - No need to change Pregel’s API
  - Does not assume any a priori knowledge to graph structure or algorithm
Mizan’s Migration Barrier

- Mizan performs both planning and migrations after all workers reach the BSP barrier
1. Identify the source of imbalance: By comparing the worker’s execution time against a normal distribution and flagging outliers.
Mizan’s Migration Planning Steps

1. **Identify the source of imbalance:** By comparing the worker’s execution time against a normal distribution and flagging outliers.

   - Mizan monitors for each vertex:
     - Remote *outgoing messages*
     - All *incoming messages*
     - *Response time*

   - High level summaries are broadcast to each worker.
Mizan’s Migration Planning Steps

1 Identify the source of imbalance
2 Select the migration objective:
   - Mizan finds the strongest cause of workload imbalance by comparing statistics for outgoing messages and incoming messages of all workers with the worker’s execution time
Mizan’s Migration Planning Steps

1. Identify the source of imbalance

2. Select the migration objective:
   - Mizan finds the strongest cause of workload imbalance by comparing statistics for outgoing messages and incoming messages of all workers with the worker’s execution time
   - The migration objective is either:
     - Optimize for outgoing messages, or
     - Optimize for incoming messages, or
     - Optimize for response time
Mizan’s Migration Planning Steps

1. **Identify the source of imbalance**
2. **Select the migration objective**
3. **Pair over-utilized workers with under-utilized ones:**
   - All workers create and execute the migration plan in parallel without centralized coordination
   - Each worker is paired with one other worker at most
Mizan’s Migration Planning Steps

1. Identify the source of imbalance
2. Select the migration objective
3. Pair over-utilized workers with under-utilized ones
4. Select vertices to migrate: Depending on the migration objective
Mizan’s Migration Planning Steps

1. Identify the source of imbalance
2. Select the migration objective
3. Pair over-utilized workers with under-utilized ones
4. Select vertices to migrate
5. Migrate vertices:
   - How to migrate vertices freely across workers while maintaining vertex ownership and fast updates?
   - How to minimize migration costs for large vertices?
Mizan uses distributed hash table (DHT) to implement a distributed lookup service:

- $V$’s home worker $ID = (\text{hash}(ID) \mod N)$
- $V$ can execute at any worker
- Workers ask the home worker of $V$ for its current location
- The home worker is notified with the new location as $V$ migrates
Migrating Vertices with Large Message Size

- Introduce **delayed migration** for very large vertices:
  - At SS $t$: only ownership of vertex $v$ is moved to $\textit{worker}_{new}$

---

**Diagram:**

- Superstep $t$
- Worker$_{old}$
- Worker$_{new}$
- Migration phase
- Introduce **delayed migration** for very large vertices:
  - At SS $t$: only ownership of vertex $v$ is moved to $worker_{new}$
  - At SS $t + 1$:
    - $worker_{new}$ receives vertex 7 messages
    - $worker_{old}$ process vertex 7
Introduce **delayed migration** for very large vertices:

- At SS $t$: only ownership of vertex $v$ is moved to $worker_{new}$
- At SS $t + 1$:
  - $worker_{new}$ receives vertex 7 messages
  - $worker_{old}$ process vertex 7
- After SS $t + 1$: $worker_{old}$ moves state of vertex 7 to $worker_{new}$
Experimental Setup

- Mizan is implemented on C++ with MPI with three variations:
  - **Static Mizan**: Emulates Giraph, disables dynamic migration
  - **Work stealing (WS)**: Emulates Pregel’s coarse-grained dynamic load balancing
  - **Mizan**: Activates dynamic migration

- Local cluster of 21 machines:
  - Mix of i5 and i7 processors with 16GB RAM Each

- IBM Blue Gene/P supercomputer with 1024 compute nodes:
  - Each is a 4 core PowerPC450 CPU at 850MHz with 4GB RAM
Datasets

- Experiments on public datasets:
  - Stanford Network Analysis Project (SNAP)
  - The Laboratory for Web Algorithmics (LAW)
  - Kronecker generator

| name                     | |nodes| | |edges| |
|--------------------------|------------------|------------------|------------------|
| kg1 (synthetic)          | 1,048,576        | 5,360,368        |
| kg4m68m (synthetic)      | 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      |
Giraph vs. Static Mizan

Figure: PageRank on social network and random graphs

Figure: PageRank on regular random graphs, each has around 17M edge
Effectiveness of Dynamic Vertex Migration

- PageRank on a social graph (LiveJournal1)
- The shaded columns: algorithm runtime
- The unshaded columns: initial partitioning cost
Effectiveness of Dynamic Vertex Migration

- Comparing the performance on highly variable messaging pattern algorithms, which are DMST and ad propagation simulation, on a metis partitioned social graph (LiveJournal1)
Mizan’s Overhead with Scaling

Figure: Speedup on Linux Cluster

Figure: Speedup on IBM Blue Gene/P supercomputer
Future Work

- Work with skewed graphs
- Improving Pregel’s fault tolerance
Mizan is a Pregel system that uses fine-grained vertex migration to load balance computation and communication across supersteps.

Mizan is an open source project developed within InfoCloud group in KAUST in collaboration with IBM, programmed in C++ with MPI.

Mizan scales up to thousands of workers.

Mizan improves the overall computation cost between 40% up to two order of magnitudes with less than 10% migration overhead.

Download it at: [http://cloud.kaust.edu.sa](http://cloud.kaust.edu.sa)

Try it on EC2 (ami-52ed743b)
Extra: Migration Details and Costs

![Graph showing migration details and costs](image-url)
Extra: Migrated Vertices per Superstep

![Graph showing migrated vertices and migration cost per superstep.](image)

- Migrated Vertices (x1000)
- Migration Cost (Minutes)
- Supersteps
- Total Migrated Vertices
- Max Vertices Migrated by Single Worker
Extra: Mizan’s Migration Planning

Compute()

No

Yes

Superstep

Imbalanced?

Migration

Barrier

BSP

Barrier

Worker

Overutilized?

Pair with

Underutilized

Worker

Select 
Vertices
and Migrate

Identify Source
of Imbalance

No

Yes

No

No

Yes
Extra: Architecture of Mizan

Mizan Worker

- Migration Planner
- Communicator - DHT
- BSP Processor
- Storage Manager

Vertex Compute()

IO

HDFS/Local Disks