BSP IMPLEMENTATION OF BORŮVKA’S MINIMUM SPANNING TREE ALGORITHM

A thesis submitted

to Kent State University in

partial fulfillment of the requirements

for the degree of Master of Science

by

Ding Chu

May 2015
Thesis written by

Ding Chu

B.S., University of Electronic Science and Technology of China, 2005
M.S., University of Electronic Science and Technology of China, 2008
M.S., Kent State University, 2015

Approved by

Dr. Ye Zhao, Advisor
Dr. Javed Khan, Chair, Department of Computer Science
Dr. James L. Blank, Dean, College of Arts and Sciences
# TABLE OF CONTENTS

LIST OF FIGURES ................................................................. v

LIST OF TABLES ................................................................. vii

Acknowledgements .............................................................. viii

Dedication ................................................................. ix

1 Introduction ................................................................. 1

2 Related Work ................................................................. 4
   2.1 Spanning Tree on Symmetric Multiprocessors ......................... 4
   2.2 Spanning Tree on GPUs ................................................. 5

3 Minimum Spanning Tree Algorithms .................................. 8
   3.1 Kruskal’s Algorithm .................................................. 9
   3.2 Prim’s Algorithm ...................................................... 9
   3.3 Boruvka’s Algorithm ................................................ 10

4 Graph Computation Model ............................................... 12
   4.1 Pregel BSP Model .................................................... 13
   4.2 Hadoop Map-Reduce Model .......................................... 16
   4.3 GraphSQL Computation Model .................................... 19
LIST OF FIGURES

1 The CUDA Programming Model ........................................... 6
2 The BSP model ............................................................... 14
3 The Pregel Maximum Value Example .................................... 15
4 The MapReduce Model ....................................................... 17
5 The GraphSQL Model ....................................................... 19
6 Vertex Centric Example of Borůvka’s Algorithm ....................... 22
7 Algorithm Flowchart ....................................................... 24
8 Original Graph ............................................................... 30
9 Iteration 1 - Edge Selection Mode ...................................... 31
10 Iteration 2 - Coordinator Recolor Mode .............................. 32
11 Iteration 3 - Coordinator Recolor Mode .............................. 33
12 Iteration 4 - Coordinator Recolor Mode .............................. 33
13 Iteration 5 - Color Broadcast Mode .................................... 34
14 Iteration 6 - Test Convergence Mode .................................. 35
15 Iteration 7 - Edge Selection Mode .................................... 36
16 Iteration 8 - Coordinator Recolor Mode .............................. 36
17 Iteration 9 - Coordinator Recolor Mode .............................. 37
18 Iteration 10 - Coordinator Recolor Mode ............................ 37
19 Iteration 11 - Color Broadcast Mode .................................. 38
20 Iteration 12 - Test Convergence Mode .............................. 39
<table>
<thead>
<tr>
<th>Page</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>Final MST Result</td>
<td>39</td>
</tr>
<tr>
<td>22</td>
<td>Testing Results in Line Chart</td>
<td>44</td>
</tr>
</tbody>
</table>
## LIST OF TABLES

<table>
<thead>
<tr>
<th></th>
<th>Table Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Testing Graphs Statistics</td>
<td>41</td>
</tr>
<tr>
<td>2</td>
<td>Experimental Results</td>
<td>43</td>
</tr>
<tr>
<td>3</td>
<td>Experimental Results with <em>Kruskal</em> Algorithm</td>
<td>45</td>
</tr>
</tbody>
</table>
Acknowledgements

First, I would like to express my sincere gratitude to my advisor Prof. Ye Zhao for his continuous support of my study and research work. His patience, enthusiasm and wide range of knowledge inspired me on both academic and ordinary life. He guided me through many research challenges. I cannot forget about the intensive discussion we had for various research topics. Without his help, I could not have the degree and thesis finished. Besides my advisor, I would like to thank Prof. Ruoming Jin for offering me the internship opportunity and leading me working on various interesting projects. I would also like to thank Prof. Cheng-Chang Lu for the encouragement and constructive suggestions for my study. Next, I would like to give a special thanks to our graduate secretary Marcy Curtiss for her kind help of my entire study. My sincere thanks also goes to all the members of Visualization and Graphics Lab. Last but not the least, I would like to thank my beloved mother and the best father in the world. I have made it this far because of you, but I will go even further in the future.
Dedicated to my beloved parents.
CHAPTER 1

Introduction

Graph theory has a long history dating back to 300 years ago when Leonhard Euler first proposed the famous walking along seven bridges problem. After that, the graph theory, study the relationship of connectivities of objects, becomes one important and interesting research field. Graph, by definition, is a symbolic representation of a network and its connectivities and it could be denoted as \( G = (V, E) \). The \( V \) and \( E \) are the two most fundamental terms in the kingdom of graph: node and edge. The node is one abstract entity represents the object in the actual graph. It could be one city in the road network, or it could be one person in the friendship network. When it comes to the edge, it is the relationship between any two linked nodes. The edge in the road network means two cities are directly reachable. And the link between any two connected human entities indicates their friendship. Graph theory is a branch of mathematics concerned about how network can be encoded and having their properties measured. During the last decade, with the booming influence from the social and complex networks, the graph research has been greatly extended and enriched.

Nowadays, the graph data is ubiquitous and everything could be represented in a proper graph way. As the growing influence of graph study in social network, webpage network, logistic network, graph has been put on the center stage of both academic and industrial than ever before. Facebook use the social network to organize the friendship of almost the whole world. LinkedIn is another classic example who specialized the graph
technology in the professional fields. The PageRank [1] algorithm, the core of Google’s page ranking criteria, is also based on the graph computation. Logistical company uses the graph theory to optimize the daily delivery routes. Another popular term comes along with the booming application of graph theory is the NoSQL technology which has been pioneered by the leading internet companies - including Google, Facebook, Amazon and LinkedIn in an effort to overcome the limitation of forty-years-old relational database technology. Today, enterprises are adopting NoSQL for growing number of user data management. The graph database is one outstanding example of NoSQL technology in dealing the overwhelming amount of Internet user data.

With the development of graph technology and huge amount of network data, there is an evolution process of the graph algorithm design. At the most beginning, graph source is relatively tiny, most of the graph problem could be solved in a sequential algorithms. Later, when the size of the network data grows, the sequential one cannot solve the problem in a neat way. Research scientist use the parallel algorithm to handle this issue, like the PRAM (Parallel Random-Access Model) based parallel algorithms and GPU (Graphics Processing Units) based graph algorithms. Most recently, the BSP (Bulk Synchronous Parallel) model for graph processing becomes very popular than ever before. The BSP-like processing system uses either vertex centric or edge centric method to compute graph data in a sequential consecutive steps. It is totally different from the legend sequential graph counterparts. It makes the parallel graph algorithm to be able to implement on regular machine with satisfied performance. Google’s Pregel [2] system is the pioneer of vertex oriented programming model and system for graph processing with built-in features for iterative processing on graphs. Then follows the Giraph, GraphLab,
GraphChi graph platforms.

MST (Minimum Spanning Tree) is one classic graph problem since the graph world begins. Given a connected, undirected graph, a spanning tree of a graph is a subgraph that is a tree and connects all the vertices together. Minimum spanning tree is the spanning three with the minimum weights among all other ones. In another word, given a graph G(V, W, E), MST is minimum set of E connects all V and has the least total W. MST has many applications in the real life. For instance, the IC network design, route finding, approximation solution to Traveling Salesman Problem and approximation solution to Steiner Tree Problem. With rapid development of recent graph processing platform, the MST becomes an interesting topic again.

In this thesis, we provide a parallelized MST implementation on a BSP-like massive graph processing system. Before the details of algorithm implementation, some parallel MST related work on modern parallel computation device will be shortly described. Besides, two other traditional sequential MST computation algorithms will be introduced. Based on the distributed algorithm, the thesis then will explain the BSP processing system, Hadoop Map-Reduce and the GraphSQL’s patented processing platform. In the following chapter, we will go through the implementation details of this parallel MST computation algorithm. At last, some real-life graphs’ MST computation will be studied among this work.
CHAPTER 2

Related Work

Finding the Minimum Spanning Tree or Spanning Tree on graph has been an research
topics for a long time. In the early days, there are several classic sequential algorithms
to extract the MST in an efficient way. For instance, the Kruskal’s algorithm and the
Prim’s algorithm. With noticeable increasing of the source data size and parallel com-
putation power, the research community agrees that it is worthwhile to design parallel
Spanning Tree algorithms to make the best use of hardware power and gain better perform-
ance. Among all those parallel models, the work on SMPs (Symmetric Multiprocessors)
Section 2.1 and GPUs (Graphic Processing Unit) Section 2.2 deserves a special mention.

2.1 Spanning Tree on Symmetric Multiprocessors

SMP (Symmetric Multiprocessing) involves one symmetric multiprocessor hardware
and software architecture that connects multiple identical processors to a single, shared
main memory, and they all have the full access to all I/O devices. The operating system
running on the architecture will treat each processor equally. Regarding the multi-core
processors, it will treat each core as one individual processors. PRAM (Parallel Random
Access Machine) is a shared-memory abstract machine. The PRAM was designed as the
parallel-computing analogy to the RAM(Random Access Machine) and it is most used
by the parallel-algorithm designer to measure the parallel algorithmic performance.

Under the SMP model, the research community has produced a rich collection of
theoretic deterministic and parallel spanning tree or minimum spanning tree algorithms. Those implementations have achieved significant parallel speedup on sparse or dense, regular or irregular graphs when compared against to their best sequential counterparts.

The best time complexity for a serial solution of the MST problem, proposed by Bernard Chazelle [3], is $O(E\alpha(E,V))$, where $\alpha$ is the functional inverse of Ackermann’s function.

In the paper, a comparison of data-parallel algorithms for connected components [4], Greiner implemented several connected components algorithms (Shiloach-Vishkin (SV), Awerbuch-Shiloach, random matching and one hybrid algorithm) using NESL on the Cray Y-MP/C90 and TMC CM-2. Greiner reports a maximum speedup of 3.5 using the hybrid algorithm. Hsu [5] et al. implemented several algorithms with fine-tuning techniques for finding connected graph components in a parallel way. They reported that their code runs slower compare to Greiner’s implementation. But Greiner’s work didn’t include the compressed data structure for graph with improved concurrent write operations. Krishnamurthy [6] implemented the Shiloach-Vishkin algorithm on a partition of graph distributed over the processors. They obtain speedup on the order of 20 on a 32-processor CM-5 and 238 on a 512-processor CM-5. Chung [7] provided a parallel implementation of MST based on the sequential algorithm of Borůvka. Their work on sparse graphs with 64,000 vertices on Thinking Machine’s CM-5 achieve a speedup factor of about 4 on 16 processors.

2.2 Spanning Tree on GPUs

Originally, a graphics processing unit (GPU) is a specialized electronic circuit designed to rapidly manipulate and alter memory to accelerate the creation of images in a
frame buffer for output to a display. Modern GPUs highly parallel structure provides high
computation power at low cost and have been treated as desk supercomputers. General-
purpose computing on graphics processing units (GPGPU) is the utilization of a GPU
to perform computation in applications traditionally handled by the CPU. The GPUs
expose a general data-parallel programming model in the form of CUDA. The Compute
Unified Device Architecture (CUDA) from Nvidia present a heterogeneous programming
model allows the parallel hardware to be used with the CPU. CUDA expose the GPU
as a massively threaded parallel architecture allowing up to millions of threads to run in
parallel over its processors with the access to its private and global memory Figure 1.
This massively multithreaded mode is also suitable for the algorithms of irregular data
structures like graph and this make it possible to accelerate the graph algorithms in a
massive parallel way.

Because the high computation power and low cost of GPUs, many research work has
been conducted to explore the graph algorithms on GPUs. Vibhav [8] et al. implemented
a MST algorithm on Nvidia GPUs with CUDA, as a recursive formulation of Borůvka’s
approach. They claimed that it obtained 20 to 50 times speedup over the CPU implementation on most graphs. It constructed the MST on a 5 million node and 30 million edge graph in under 1 second on one quarter of the Tesla S1070 GPU. Fu [9] presented a high performance parallel graph processing framework called MapGraph that can deliver up to 3 billion Traversed Edges Per Second (TEPS) on a GPU. This framework provides a high-level abstraction that makes it easy to write graph programs and obtain good parallel speedup on GPUs. Zhong [10] proposed a programming framework called Medusa to simplify the graph processing on GPUs by writing sequential C/C++ code.
CHAPTER 3

Minimum Spanning Tree Algorithms

Electronic circuit designers often need to make the pins of several components electronically equivalent by wiring them together. To interconnect a set of $n$ pins, we can use an arrangement of $n - 1$ wires, each connecting two pins. Among all such arrangements, the one with the least amount of wire is the desired one. The wiring problem could be formalized in the graph way. Given a connected, undirected graph $G = (V, E)$, where the $V$ is the set of pins, and for each edge $(u,v) \in E$, there is one weight $\omega(u, v)$ indicating the cost to connect $u$ and $v$. Then we wish to find a subgraph of edges that connect all the vertices with the least total weight. The process of determining such a subgraph is called the minimum-spanning-tree problem.

**Input Description**: A undirected graph $G = (V, E)$ with weighted edges.

**Output Description**: The least weight edges $E' \subset E$ connect all $V$ in $G$.

Minimum spanning tree problem is of important for several reasons. It provides a way to quickly identify the clusters in a set of vertices by deleting the longest edges in the graph. It is also the stepping stone to solve other challenge graph problems, such as the Steiner Tree Problem and Traveling Salesman Problem. Last, the algorithm is one great evidence that how the greedy algorithm work. There are several classic algorithms for MST problem. Kruskal’s algorithm Section 3.1 and Prim’s algorithm Section 3.2 are greedy and sequential algorithms. The Borůvka’s algorithm is a greedy and parallel algorithm even though it is less well known than the previous two. Despite of this, the
Borůvka’s algorithm is a better fit for the parallel framework and easy to implement.

The work of this thesis is based on the Borůvka’s algorithm

3.1 Kruskal’s Algorithm

Kruskal’s algorithm looks for a safe edge to add in for each step. It constructs the MST in this way: it starts with all the vertices represent as one individual tree. At each iteration, one unselected edge \( e(u, v) \) with the minimum weight will be chosen. If the edge \( e \) will make two distinct components or trees connected, then, the edge \( e \) will be added into the final MST edge set. Repeat this process until there is only one big tree exist.

Algorithm 1 MST-Kruskal(G, \( \omega \))

1: Initialize \( A \leftarrow \emptyset \);
2: for all each vertex \( v \in G. V \) do
3: \( \text{MAKE-SET}(v) \)
4: end for
5: sort the edges of \( G.E \) into nondecreasing order by weight \( \omega \)
6: for all each edge \( (u, v) \in G.E, \) taken in the sorted order do
7: if \( \text{FIND-SET}(u) \neq \text{FIND-SET}(v) \) then
8: \( A = A \cup (u, v) \)
9: \( \text{UNION}(u, v) \)
10: end if
11: end for
12: return A

3.2 Prim’s Algorithm

Prim’s algorithm also looks for one edge at a time. The difference, however, is that it starts with a single node and intent to grow the discovered tree by adding one new vertex who is the most closest node to the tree. At any stage of the computation, all the vertices will be categorized as three different kinds of nodes. The one that has been discovered,
the one on the fringe about to be included and the unseen vertices. The *Prim’s algorithm* is similar with *Dijkstra’s algorithm* for finding the shortest path in graph. It works in this way: it starts the tree from one arbitrary root vertex $v$. At each step, add one light edge that connects one unseen node to the tree. Repeat this process until all the vertices become visible.

**Algorithm 2** MST-Prim(G, $\omega$, r)

1: for all each $u \in G.V$ do
2: $u$.Key = $\infty$
3: $u$.parent = NIL
4: end for
5: r.key $\leftarrow$ 0
6: $Q \leftarrow G.V$
7: while ($Q \neq \emptyset$) do
8: $u = \text{Extract-Min}(Q)$
9: for all each $v \in G.\text{Adj}[u]$ do
10: if ($v \in Q$) and $\omega(u, v) < v$.key then
11: $v$.parent = $u$
12: $v$.key = $\omega(u, v)$
13: end if
14: end for
15: end while

3.3 Borůvka’s Algorithm

Based on the observation that the lowest weighted edge of each vertex must be reside in the final minimum spanning tree. Then if treat all the vertices as individual trees at first and use the light weight edge of each vertex to union all the trees, this will produce at most $n/2$ trees. After that, if each tree is treated as one vertex, this will repeat the previous scenario of forest with at most half the size. Continue this process until there is only one giant tree left. The most appealing advantages of *Borůvka’s algorithm* is that it could be easily deployed into a parallel computation way. Each constructed tree at
any stage only concerns the edge that incident to the vertex of that tree. This makes the
collection of tree component could be speed up greatly in a parallel environment. The
number of trees will be at least halved in each round, this guarantee it is a $O(m\log n)$
algorithm without using any fancy data structure. It is also the feature that this thesis
decide to adopt this algorithm into a *Bulk Synchronous Parallel* (BSP) one.

<table>
<thead>
<tr>
<th>Algorithm 3 MST-Borůvka(G)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: Initialize a forest $T$ to be a set of one-vertex tree, one for each vertex</td>
</tr>
<tr>
<td>2: while number($T$) &gt; 0 do</td>
</tr>
<tr>
<td>3: for all each component $C$ of $T$ do</td>
</tr>
<tr>
<td>4: Empty set of edges $S \leftarrow \emptyset$</td>
</tr>
<tr>
<td>5: for all each vertex $v \in C$ do</td>
</tr>
<tr>
<td>6: find the least weight edge $e$ from $v$ to a vertex $u \supset C$</td>
</tr>
<tr>
<td>7: add $e$ into $S$</td>
</tr>
<tr>
<td>8: end for</td>
</tr>
<tr>
<td>9: add the lightest edge $e \in S$ to $T$</td>
</tr>
<tr>
<td>10: end for</td>
</tr>
<tr>
<td>11: end while</td>
</tr>
<tr>
<td>12: $T$ is the expected MST of $G$</td>
</tr>
</tbody>
</table>
Graph theory is one of long history and interesting research field since its invention. There are many research directions within the kingdom of graph theory. For instance, shortest path, minimum spanning tree, steiner tree, etc. Among all these challenges, there is one essential question need to be answered before any further computation. That is how to represent the graph in a computer way. In the graph theory, there are two popular ways to store the graph data. They are adjacent list and adjacent matrix.

**Adjacent List**: In this representation, for a given graph $G = (V, E)$, it will consist $|V|$ adjacent list for each vertex in V. For each $u \in V$, the adjacent list of $u$ will contain all the vertices $v$ that there is an edge $(u, v) \in E$. In another word, the adjacent list of $u$ has all the vertices adjacent to $u$ in G. Adjacent list is one compact way to store the sparse graph and it requires $|E|$ to store the whole graph data.

**Adjacent Matrix**: In this representation, for a given graph $G = (V, E)$, it will have one $|V| \times |V|$ matrix. And all the vertices in G will be denoted as 1, 2, ..., $|V|$. For each element $a_{ij}$ in this matrix, $a_{ij} = 1$, if the edge $(i, j) \in E$. Compare to the Adjacent List representation, it uses more space to store the graph data but has better performance to access one specific edge. It only requires the index of source and destination vertex to locate the edge. However, in Adjacent List, it has to iterate the list to search the expected edge.
The graph algorithm are mainly built on these two graph representations. The Kruskal’s algorithm Section 3.1 and Prim’s algorithm Section 3.2 are implemented upon these standard ways. These sequential algorithm works very well in the old days while the graph data is small. When it come to the information age, with the rapid development of Internet, it becomes a different story.

In old days, the graph with size 100K could be treated as big. With the booming of Internet, nowadays, the graph data could be easily grow up to billion class, this brings up the challenge how to carry out the graph computation among these Big Datas. There are many efforts have been put on this field. One natural way is to parallelize the graph algorithms to make the best use of high-performance hardware. Another way is to make the computation jobs distributed across multiple units. Pregel System Section 4.1 is trying to speed up the graph computation in a different perspective called vertex centric. Hadoop Map-Reduce Model Section 4.2 provides an alternative to solve the big graph challenge in a distributed way. Section 4.3 describes the graph computation model used in this thesis work.

4.1 Pregel BSP Model

The Bulk Synchronous Parallel (BSP) abstract computer is a bridging model for designing parallel algorithms. This model was developed by Leslie Valiant of Harvard University during the 1980s. In BSP, a parallel program runs across a set of virtual processors and executes as a sequence of supersteps separated by barrier synchronizations.

A BSP algorithm consists of a sequence of supersteps Figure 2. One superstep may include many small computation tasks. During the execution of each processor, they
may emit many temporary results that need to share between different jobs. The BSP model allow each single processor to share the knowledge in the format of messages with the assistance of communication layer. After each processor finish their single task, there is a hardware facility that assure the synchronization of all computation components. If the termination condition meets, the BSP algorithm will stop. Otherwise, it will launch another superstep.

Inspired by BSP, the pregel [2] is a system for large-scale graph processing. Compare to the virtual processors, the vertices are the computation components of pregel which is called the vertex centric method. In another word, the vertex is the host of individual computation component and it will share knowledges with others by sending out messages. Each vertex will have two states: active or inactive. Within each superstep, only
will the active vertices do computation jobs. The vertex can go inactive when it vote for halt or becomes active when it receives a message from others. The maximum example Figure 3 is one Hello World example. The domo graph is of four vertices with different initial values. At each superstep, each active vertex will send out its value along the edge direction to a neighbour. If any vertex receives a value bigger than its current one, then it will update their own value and keep active for the next superstep. Otherwise, the vertex will go to inactive status in grayed. The system will stop is all vertices are inactive.

Writing a Pregel program involves implementing the predefined Vertex class. Below is the list of supported API by the vertex class.
• **Compute()**: the main computation route carried out by the vertex.

• **Superstep()**: indicate the current iteration of supersteps.

• **GetValue()**: get the value of the current vertex.

• **MutableValue()**: update the value of the current vertex.

• **GetOutEdgeIterator()**: get out edge related information.

• **SendMessageTo()**: send one message to another vertex.

• **VoteToHalt()**: make current vertex as inactive.

The famous *PageRank* algorithm would be rewritten with *Pregel* in the following way:

**Algorithm 4**

```
Algorithm 4 PageRank-Pregel Compute Interface
1: if superstep() ≥ 1 then
2:   sum = 0
3:   for all each msg in Messages do
4:     sum += msg→Value()
5:   end for
6:   *MutableValue() = 0.15 / NumVertices() + 0.85 * sum
7: end if
8: if superstep < 30 then
9:   n = GetOutEdgeIterator().size()
10:  SendMessageToAllNeighbours(GetValue() / n)
11: else
12:  VoteToHalt()
13: end if
```

4.2 Hadoop Map-Reduce Model

*MapReduce* [11] is a programming model and an associated implementation for processing and generating large data sets with a parallel, distributed algorithm on a cluster.
As the name reveals, a MapReduce program consists of two main routes: Map() and Reduce(). A Map process will process one chunk of gigantic input file to generate a set of intermediate key/value pairs. And the Reduce procedure will aggregate all the intermediate values that mapped to the same key. If one iteration of MapReduce is not enough to close the task, then another round of MapReduce will be executed. Finally, the results will be written into disk as files. The whole execution flow is described in Figure 4.

Figure 4: The MapReduce Model

The advantage of this framework is that it will parallelize the map/reduce tasks among a large cluster of commodity machines automatically. The system takes care of the details of partitioning the input source data, organizing the parallel jobs, dealing with task failures and coordinating the communications between different sub-tasks. The users are freed from the messed distributed environment and only need to focus on two main interfaces.

Hadoop MapReduce is one open source MapReduce like programming model for large
data computation of reliable and scalable. The *Hadoop MapReduce* job is running on their distributed file system which is called *HDFS*. Below is a list of the key signatures of Hadoop MapReduce supported *API*.

- **Map()**: map is the function that transfers the input data into intermediate key/value files.

- **Reduce()**: reduce is the function that aggregates a set of values with the same key to a smaller set of value.

To best demonstrate the simplification of *MapReduce* programming model, the following Algorithm 5 is one simple *word count* example in *Hadoop MapReduce*. In the *map* function, it will break each line into multiple words. And then generate a key value pair of (key, 1). The system will send the intermediate results to a appropriate reducer. In the *reduce* phase, it finally sums the total count of one specific word.

<table>
<thead>
<tr>
<th>Algorithm 5 Word Count - Hadoop MapReduce</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: System handle the input source partition details</td>
</tr>
<tr>
<td>2: <strong>Function</strong> map(key, value, context)</td>
</tr>
<tr>
<td>3: <strong>for all</strong> each word ω in one line of source do</td>
</tr>
<tr>
<td>4: context→write(ω, 1)</td>
</tr>
<tr>
<td>5: end for</td>
</tr>
<tr>
<td>6: EndFunction</td>
</tr>
<tr>
<td>7: <strong>Function</strong> reduce(key, values, context)</td>
</tr>
<tr>
<td>8: <strong>for all</strong> each value v in values do</td>
</tr>
<tr>
<td>9: sum = sum + 1</td>
</tr>
<tr>
<td>10: end for</td>
</tr>
<tr>
<td>11: context→write(key, sum)</td>
</tr>
<tr>
<td>12: EndFunction</td>
</tr>
</tbody>
</table>
4.3 GraphSQL Computation Model

The patented GraphSQL’s graph computation framework is also a BSP parallel big graph data processing system. Similar to Pregel, this graph processing system is a combination of vertex centric and edge centric framework. And, like Hadoop MapReduce, it provides several graph processing APIs. The users of the system only need to focus on a small set of interface implementation and the system will take care the details of others like task scheduling, communication and fault handling. This system could be easily parallelized to a large scale of cluster in order to face a huge amount of input graph source. But this thesis work will focus on a single machine case.

Figure 5 describes the workflow of GraphSQL’s graph processing system. First, the input graph source will be divided into multiple chunks. Each worker thread of the system will handle several chunks in the way either VertexMap() or EdgeMap(). During
the process, there will be some intermediate values generated on active vertices. In the Reduce phase, the values of each vertex will be aggregated to a smaller result set. If the termination condition doesn’t meet after one iteration, another iteration of task will be executed. In this graph framework, vertex has two states: active and inactive. Vertex will remain active as long as one message arrives.

The following list the key interfaces of GraphSQL’s graph processing framework. This thesis work is built upon these primitives.

- **Initialize()**: initialize the graph computation environment.

- **EdgeMap()**: computation on one edge.

- **VertexMap()**: computation on one vertex.

- **Reduce()**: aggregate operation on one vertex.

- **BeforeIteration()**: operation before one iteration start.

- **AfterIteration()**: operation after one iteration.

- **Write()**: output the final results into one file.
CHAPTER 5

Algorithm Implementation

The Minimum Spanning Tree (mst) algorithms mentioned in Chapter 3 are all sequential algorithms. These algorithms are classic and can handle the graph data of small scale easily. However, as the increasing computation power of computer and considerable amount of graph input source, it is a necessary to parallelize the mst algorithm to make the best use of the modern computational hardware and obtain the best algorithm performance. As matter of fact, the Borůvka’s MST algorithm has the build-in features which make it the best candidate to be parallelized. The work of this thesis is to build a parallel Borůvka’s MST algorithm in a BSP environment. Section 5.1 explains why this algorithm is the best fit. Section 5.2 describes the details of implementation in pseudocode. Finally, Section 5.3 goes through a running example to demonstrate how this parallelized algorithm works.

5.1 Parallel Borůvka’s Algorithm

Different from the traditional way, the BSP computation model enables each vertex the most independency. The vertices are the actual host and responsible for one user defined function. Based on this vertex centric model, it is worthwhile to go over the Borůvka’s algorithm again in another perspective. If one vertex is viewed as one independent entity, the sequential algorithm in Section 3.3 could be condensed into two major operations for each vertex as the following list. In this way, the Borůvka’s algorithm could
be built in a $BSP$ environment quickly.

- **search** each vertex of tree $T$ searches the least weighted edge from $T$ to $G - T$

- **merge** all the vertices which are connected by the selected edges will be merged as one new tree

The Figure 6 demonstrate how the *Borůvka’s algorithm* works in a vertex centric way. At first, in Figure 6a, the graph is initialized as one forest in which all edges of Graph are invisible and tagged as unselected. In another word, each vertex in the original graph
represents one independent tree. Next, in Figure 6b, each vertex starts to search for one edge which has the least weight and can reach to another different tree or component. Once those edges are determined, they will be marked as selected. And vertices connected by selected edges will be merged as one new tree or component in Figure 6c. After this, each tree will be treated as one new abstract vertex, the process will repeat among the newly constructed virtual vertices until there is only one tree left in Figure 6d. That one big tree is the desired final MST result.

5.2 Implementation Details

Section 5.1 introduces the basic idea of how to parallelize the Borůvka’s algorithm. Based on the GraphSQL graph processing model in Section 4.3, this parallelized algorithm could be divided into four different phases: edge selection mode, coordinator recolour mode, color broadcast mode and test convergence mode. The Figure 7 describes the flow of algorithm execution.

Before go to the design details of each phase, some terminologies need to be introduced in order to help understanding the later design.

- **real edge**: the original weighted edge in the input graph.
- **virtual edge**: compare to the real edge, the virtual edge is introduced between any newly connected components. It connects the two coordinators of the two chained components and also will be used to synchronize the color information among them.
- **regular vertex**: the original vertex of the graph source.
- **coordinator**: coordinator is the selected leader of one connected components. It
is responsible for selecting the least weighted edge from current component to other components. It also helps synchronization of color between coordinators and the vertices within the same component.

- **vertex color**: the vertex color is used to indicate whether two vertices are belong to the same component. More specifically, the id of coordinator will be used as the vertex color. In another word, the color of each vertex will be initialized with their own vertex id as all the vertices are disconnected with each other at the most beginning.

The vertices will communicate with each other via messages in a BSP environment. In this parallelized Borůvka’s algorithm implementation, there are three main kinds of information need to be shared: regular edge information, vertex color and virtual edge. The
current implementation uses one structure to represents all the messages in Algorithm 6 and Algorithm 7.

**Algorithm 6** structure of MstEdge information

```plaintext
1: Structure MstEdge
2: {
3:   startVertex: VertexID
4:   endVertex: VertexID
5:   weight: Int
6:   // (startColor, endColor) virtual edge information
7:   startColor: VertexID
8:   endColor: VertexID
9: }
```

**Algorithm 7** structure of edge message

```plaintext
1: Structure EdgeMessage
2: {
3:   valueType: enum {VERTEX_COLOR, EDGE_INFO, VIRTUAL_EDGE}
4:   vertexColor: VertexID {vertex color/the sender’s color}
5:   edgeInfo: MstEdge
6: }
```

In order to simplify the messages transmission and IO operation, the emit and store primitives are used to help describing the algorithm.

- **emit** emit(vid, msg) will send one message msg to vertex of vid.

- **store** store(vid, val) will writer the value val to vertex of vid.

Back to the algorithm execution flow in Figure 7, the edge selection mode is the first phase to run. During this phrase, each vertex within one connected component starts to look for the cross component edge which connects two vertices with different color. Then those edges will be sent to the coordinator of current component who then will decide the least weight edge for the next iteration. The cross component edge searching process
happens in `EdgeMap()` and the least weight edge selection process occurs in `Reduce()` as in Algorithm 8. Besides, the *virtual edge* information will be sent out along with the *cross component* edge.

**Algorithm 8 edge_selection_mode implementation**

1. **EDGE_MAP:**
   2. for all edges do
   3.     vid = Neighbor’s Color (Coordinator’s VertexID)
   4.     if `CurrentVertexColor` != vid then
   5.         //send the edge & virtual edge information to the coordinator
   6.         `EMIT`(vid, `EdgeMessage`(EDGE_INFO))
   7.     end if
   8. end for

9. **REDUCE:**
10. find the edge with the lowest weight e (u, v).
11. for the lowest weight Edge e (u, v) & virtual Edge f (a, b) do
12.    `STORE`(u, e)
13.    `STORE`(v, e)
14.    `STORE`(a, f)
15.    `STORE`(b, f)
16. end for

After the *edge selection mode*, previous disconnected components may merge as one connected component because of the newly selected *cross component* edge. To reflect the change of the graph, the system need to synchronize the color information of newly connected components and the process is called *coordinator recolor mode* and *color broadcast mode*. First, the color will be updated between the coordinators of the just merged components with the assistance of *virtual edge*. Be aware, the *virtual edges* are temporary edges which only exist between the coordinators of merged components. By using a algorithm similar to the one described in Figure 3, the *lowest color* of the coordinators will be chosen as their new color. This process is outlined in Algorithm 9. Naturally, this process needs multiple iterations to finish.
Algorithm 9 coordinator_recolor_mode implementation

1: Make sure the coordinator who are connected by the virtual edges share the same color
2: 
3: VERTEX_MAP:
4: \text{EMIT}(\text{virtualNeighborVertexID}, \text{EdgeMessage}(\text{VERTEX_COLOR}))
5: 
6: REDUCE:
7: pick the received Lowest\_Color as the new color of current coordinator vertex.
8: if \text{Lowest\_Color} == \text{CurrentVertexColor} then
9: \hspace{1em} Set current vertex as Deactive.
10: else
11: \hspace{1em} Update the current vertex color as Lowest\_Color
12: \hspace{1em} Set current vertex as Active.
13: end if

Then, all the vertices of the merged components must have their color updated. This is accomplished in \textit{color broadcast mode} with a simple broadcast method as in Algorithm 10.

In \textit{EdgeMap}, the vertex will send its \textit{id} to the old coordinator asking for the new color information. Meanwhile, in \textit{Reduce}, the outdated coordinator will propagate the updated color to its subscribers.

Algorithm 10 color_broadcast_mode implementation

1: make sure the vertices have the same color as their coordinator vertex
2: 
3: VERTEX_MAP:
4: \text{EMIT}(\text{coordinatorVertexID}, \text{EdgeMessage}(\text{currentVertexID}))
5: 
6: REDUCE:
7: for all the received member vertex ID \textit{id} do
8: \hspace{1em} \text{STORE}(\textit{id}, \text{newGroupColor})
9: end for

The total components will be at least halved after each merge process. If there is only one component left, that one would be exactly the expected final \textit{mst} result. Otherwise, the whole process starts over again. The process of determining whether the \textit{mst} is found or not is called \textit{test convergence mode}. The implementation of this process in
Algorithm 11 is very simple and natural in a parallel environment. At EdgeMap, each vertex will send out its color to all the neighbours. If any vertex receives a different color at Reduce, then the merge process needs to continue. Otherwise, the mst result is generated and the execution stops.

Algorithm 11 test_converge_mode implementation
1: Test whether all the trees have been merged as one single MST
2: EDGE_MAP:
3: EMIT(neighborVertexID, EdgeMessage(VERTEX_COLOR))
4: REDUCE:
5: if receive a different color from neighbor then
6: needAnotherMSTAround = TRUE
7: end if
8: As there are multiple modes exist, the execution flow needs switch to a different mode at some point. This mode change process is orchestrated by the BeforeIteration() and AfterIteration() interfaces as in Algorithm 12 and Algorithm 13.

Algorithm 12 BeforeIteration() implementation
BeforeIteration() {
    Switch(Mode):
    case EDGE_SELECTION_MODE:
        if needAnotherMSTRound then
            reset all the vertices as active
        else
            reset all the vertices as inactive
        end if
    case COORDINATOR_RECOLOR_MODE:
        if firstRecolroing then
            set all the vertices as active
        end if
    case COLOR_BROADCAST_MODE:
        reset all the vertices as active
    case TEST_CONVERGE_MODE:
        reset all the vertices as active
}
Algorithm 13 AfterIteration() implementation

```plaintext
AfterIteration() {
    Switch(Mode):
    case EDGE_SELECTION_MODE:
        MODE = COORDINATOR_RECOLOR_MODE
        firstRecoloring = True
        existActiveVertex = False
    case COORDINATOR_RECOLOR_MODE:
        if existActiveVertex
            MODE = COORDINATOR_RECOLOR_MODE
            firstRecoloring = False
            existActiveVertex = False
        else
            MODE = COLOR_BROADCAST_MODE
    case COLOR_BROADCAST_MODE:
        MODE = TEST_CONVERGE_MODE
        needAnotherMSTRound = False
    case TEST_CONVERGE_MODE:
        MODE = EDGE_SELECTION_MODE
}
```

5.3 A Running Example

Section 5.2 explains the detail of algorithm implementation in pseudocode. This section provides a concrete example of going through all the different four phases.

Figure 8 is the original graph used in the example. Each vertex is identified by one number. There are also two attributes associated with each vertex: color and edge list. As explained in Section 5.2, the color indicates the component it belongs to and also the coordinator of the current component. The edge list is the selected edges of the current vertex which will be included into the final MST result. As the figure shows, each vertex starts with one empty edge list and one color of its own id. This implies that the graph is initialized as one forest of each vertex is one individual component at the beginning.

As the algorithm moves forward, a status table in Figure 9 is introduced to demonstrate the status of each individual vertex which in turn will help to understand the phase
switch of the whole process. At the top of the status table, it briefly describes the main operations during map() and reduce() stages. The vertex color, current active and vertex message queue shows the current status of one vertex. And the vertex message queue represents the messages received during current iteration. The vertex new color and node active status implies the node's status for next iteration.

In Figure 9, it is the first edge selection mode after the initialization. At map() stage, each vertex looks for cross component edges and then forwards the information to its coordinator for further process. Take the vertex 1 for example, it has two cross component edges: (1, 2) and (1, 8). It sends out the information to itself as the color indicates. At the reduce() stage, vertex 1 figures out the edge [1; (1, 2), 4, (1,2)] has the least weight as indicated in one red box. The first (1,2) indicates the real edge and the latter means the virtual one. Then this real edge will be included into vertex 1’s edge list and marked as selected. The virtual edge will also be saved for the next iteration.

In Figure 10, it is the coordinator recolor mode. The selected real edges and virtual edges of previous iteration are marked as red arrow and blue dash arrow respectively. In
the coordinator recolor mode, it is the virtual edge that will be used to switch the color information between coordinators. At map() stage, each vertex simply sends out its color information via the virtual edges. And at reduce() stage, the minimum color will be used as the new color. Take the vertex 4 for example, it receives the color 3 from vertex 3 and color 5 from vertex 5. It is obvious that color 3 is the minimum one and will be used as the new color of vertex 4. But this doesn’t mean the end of the coordinator recolor mode as the updated color may influence the neighbours of current vertex, for instance vertex 4’s new color 3 will finally replace vertex 5’s new color 4 as vertex 4 and 5 are connected by another virtual edge. As a result, the vertices with new updated color of current iteration will remain active marked in red shape and then it will move the new color further away at the next iteration. The system will regain balance until
all the vertices become *deactive*. Figure 11 shows another round of *coordinator recolor*. It finally comes to a stable stage when all the *coordinators* have their color information updated in Figure 12.

After all the *coordinators* gain their new color information, it is the time to notify their *members* of each component to update their color in Figure 13. Since all the vertices are initialized as *coordinator*, this iteration does nothing useful.

Following the *color broadcast mode*, here it comes the *test convergence mode* in Figure 14. This phase determines whether the final result has been obtained or not. At `map()`, each vertex send out the color to its neighbours. While at `reduce()`, each vertex will exam all the *color* received. If any vertex has more than *one different* color information, then it means that there are at least *two* disconnected components exist and the
Figure 11: Iteration 3 - Coordinator Recolor Mode

Figure 12: Iteration 4 - Coordinator Recolor Mode
procedure needs to continue.

In Figure 15, the second edge selection mode continues as the MST is not generated yet. Similar to Figure 9, each vertex looks for the cross component edge. The number of possible edges has been greatly reduced as the number of trees have been at least halved since the first merge process. Unlike to Iteration 1, only the coordinators of component will remain active during the reduce(). At current iteration, only vertex 1, 3, and 6 will receive the candidate edges. If there is a tie of the edge weight, the edge from a smaller vertex ID will be chosen. For instance, the vertex 1 receives two edges with same weight 8. The edge [1; (1,8), 8, (1,6)] comes from a smaller vertex source, it is therefore selected as the cross component edge of current iteration. The generated cross component edges and virtual edges of Iteration 7 are indicated as bigger red arrow and blue dash arrow.
respectively in Figure 16.

The same procedure repeats again. First, the color information is synchronized between the coordinators of components as in Figure 17 and Figure 18. In this process, coordinator vertices 1, 3 and 6 are synchronized with the same lowest color 1. Next, in Figure 19, the coordinator broadcasts its new color information to all its subscribers. Finally, all the vertices have their color information updated and are ready for the test convergence mode.

At Iteration 12, in Figure 20, each vertex sends out the color information to test whether there is only one tree left. It turns out that all the vertices have been sharing one same color as no vertex reports of receiving a different color. This also means that the final MST result is obtained as indicated in Figure 21. Thus the whole algorithm
Figure 15: Iteration 7 - Edge Selection Mode

Figure 16: Iteration 8 - Coordinator Recolor Mode
Figure 17: Iteration 9 - Coordinator Recolor Mode

Figure 18: Iteration 10 - Coordinator Recolor Mode
will stop and return the result back to the user.

In this chapter, the first two sections describe a parallelized Borůvka’s algorithm for generating MST and the pseudocode implementation in a BSP-like graph computation model - the GraphSQL computation platform. Finally, one comprehensive example which addresses the four key stages of the algorithm is provided.
Figure 20: Iteration 12 - Test Convergence Mode

Figure 21: Final MST Result
6.1 Experiment Setup

The algorithm is implemented in C++ combined with the Standard Template Library STL. All the experiments are executed on a Linux server with 2.0GHz Intel Xeon processors and 128GB RAM. The analysis of the experimental report focuses on the total execution time and time of each algorithm phase.

The testing graphs are from the Social Computing Data Repository [12] and Stanford Large Network Dataset Collection [13]. Table 1 shows the basic information of the testing graphs. The following is a short description of each testing graph.

- **Douban**: Douban.com, launched on March 6, 2005, is a Chinese Web 2.0 website providing user review and recommendation services for movies, books, and music.

- **Amazon Product Purchase**: Network was collected by crawling Amazon website. It is based on Customers Who Bought This Item Also Bought feature of the Amazon website. If a product $i$ is frequently co-purchased with product $j$, the graph contains a directed edge from $i$ to $j$. 
<table>
<thead>
<tr>
<th>Graph Name</th>
<th>Total Vertices</th>
<th>Total Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>Douban</td>
<td>154,908</td>
<td>654,324</td>
</tr>
<tr>
<td>Amazon Product Purchase</td>
<td>262,111</td>
<td>1,799,584</td>
</tr>
<tr>
<td>BlogCatalog</td>
<td>88,784</td>
<td>4,186,390</td>
</tr>
<tr>
<td>LiveMocha</td>
<td>104,103</td>
<td>4,386,166</td>
</tr>
<tr>
<td>Buzznet</td>
<td>101,163</td>
<td>5,326,132</td>
</tr>
<tr>
<td>CA RoadNet</td>
<td>1,965,206</td>
<td>5,533,214</td>
</tr>
<tr>
<td>Hyves</td>
<td>1,402,673</td>
<td>5,554,838</td>
</tr>
<tr>
<td>Flickr</td>
<td>80,513</td>
<td>11,799,764</td>
</tr>
<tr>
<td>Digg</td>
<td>771,231</td>
<td>11,814,826</td>
</tr>
<tr>
<td>LiveJournal</td>
<td>2,238,731</td>
<td>25,632,368</td>
</tr>
<tr>
<td>Taxi Trajectory Data</td>
<td>18,182</td>
<td>51,925</td>
</tr>
</tbody>
</table>

Table 1: Testing Graphs Statistics

- **BlogCatalog**: BlogCatalog is the social blog directory which manages the bloggers and their blogs.

- **LiveMocha**: Livemocha is the world’s largest online language learning community, offering free and paid online language courses in 35 languages to more than 6 million members from over 200 countries around the world.

- **Buzznet**: Buzznet is a photo, journal, and video-sharing social media network.

- **CA Road Network**: A road network of California. Intersections and endpoints are represented by nodes and the roads connecting these intersections or road endpoints are represented by undirected edges.

- **Hyves**: Hyves is the most popular social networking site in the Netherlands with mainly Dutch visitors and members and competes in this country with sites such as Facebook and MySpace.

- **Flickr**: A network dataset crawled from Flickr. Both the contact network and selected group membership information are included.
• **Digg**: Digg is a social news website. Prior to Digg v4, its cornerstone function consisted of letting people vote stories up or down, called digging and burying, respectively.

• **LiveJournal**: LiveJournal is a free online blogging community where users declare friendship each other. LiveJournal also allows users form a group which other members can then join.

• **Taxi Trajectory Network**: The taxi trajectory data contains one day trajectories of taxis in a major Chinese City. A street network graph is built upon these trajectories.

### 6.2 Performance Study

Applying the MST algorithm with all the testing graphs, the experimental result is summarized in Table 2. For each test, the total execution time and running iterations have been recorded along with the detailed time information of different algorithm phrases. Take the Douban for example, it takes 4.772 seconds of total 71 iterations to generate the final Minimum Spanning Tree result. Among all the 71 iterations, *edge selection mode*, *color broadcast mode* and *test convergence mode* take 4 iterations separately. And the *coordinator recolor mode* contributes the other 59 iterations. By providing the detailed execution time information, the algorithm implementation could be better analyzed. In Table 2, the graphs which takes more than 1 minute to finish have both high volume of vertices and edges. Another observation is that the execution time of *edge selection mode* and *test convergence mode* increases as the edge number of testing graph grows. It is reasonable as the two phases mainly reply on the *EdgeMap()* operation. In
the same way, the time spent in *color broadcast mode* is highly related with the vertex number of testing graph. While the *coordinator recolor mode* is a little bit complicate, the iterations of each *coordinator recolor* varies as it depends on the *maximum diameter* of the newly connected coordinators. In essence, the iterations of *edge selection mode*, *color broadcast mode* and *test convergence mode* are always the same while the *coordinator recolor mode* is unpredicted.

To better study the algorithm performance, the experimental results are displayed in the format of line chart in Figure 22. In Figure 22a, it is evident that *test convergence mode* takes most of the execution time for all the testing cases. In order to take a close look at the execution of each testing graph, the results are redraw in a log scale in Figure 22b. From this figure, even though most of the testing cases spent most their time on *edge selection* and *test convergence*, there are cases, such as the *CA RoadNet*, are *coordinator recolor mode* expensive. Based on these observations, the *edge selection mode*, *coordinator recolor mode* and *test convergence mode* are the most time consuming phases of the current implementation.

As the *edge selection*, *coordinator recolor* and *test convergence* are the most expensive procedures, possible optimizations could be applied on those phases to improve the performance. As matter of fact, there is little room to improve the *edge selection mode*, there are options to improve the other two phases. One possible optimization is to use a
contraction method instead of broadcast to synchronize the color between coordinators. In this way, the iterations of coordinator recolor mode could be guaranteed no more than half of the maximum diameter of the newly connected coordinators. Another optimization is to reduce the time of test convergence mode, the reason why it is expensive is that messages are sent out along every edge to test whether there is only one tree left or not. To reduce the cost of this process, a global counter could be used to indicate how many trees exist of current moment. This will greatly reduce the time in sending out all the comparison messages, but it will hurt the distributed behavior of this parallel algorithm.

In short, those optimization techniques could be adopted in the future version of this algorithm.

Besides, we rebuild the Minimum Spanning Tree of all the testing graphs with the Kruskal’s algorithm. The experimental result is summarized in Table 3. The total time includes the main computation steps of each algorithm. For the BSP implementation, it is the total execution time of all the four different phases. It comes as no surprise that the Kruskal’s implementation has a better performance than the BSP one. It is reasonable, as all the testing graphs are in memory, the BSP version needs more time in
<table>
<thead>
<tr>
<th>Graph Name</th>
<th>Kruskal - total time (ms)</th>
<th>BSP - total time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Douban</td>
<td>379</td>
<td>3984</td>
</tr>
<tr>
<td>Amazon Product Purchase</td>
<td>995</td>
<td>9576</td>
</tr>
<tr>
<td>BlogCatalog</td>
<td>2298</td>
<td>29764</td>
</tr>
<tr>
<td>LiveMocha</td>
<td>2418</td>
<td>24656</td>
</tr>
<tr>
<td>Buzznet</td>
<td>3014</td>
<td>463646</td>
</tr>
<tr>
<td>CA RoadNet</td>
<td>2612</td>
<td>62850</td>
</tr>
<tr>
<td>Hyves</td>
<td>3771</td>
<td>524899</td>
</tr>
<tr>
<td>Flickr</td>
<td>6515</td>
<td>36523</td>
</tr>
<tr>
<td>Digg</td>
<td>7338</td>
<td>133962</td>
</tr>
<tr>
<td>LiveJournal</td>
<td>17702</td>
<td>269798</td>
</tr>
<tr>
<td>Taxi Trajectory</td>
<td>49</td>
<td>878</td>
</tr>
</tbody>
</table>

Table 3: Experimental Results with *Kruskal* Algorithm

sharing messages, status updates and color synchronization. However, when it comes to the kingdom of *big data*, the *BSP* version will definitely have a better performance than the sequential ones.
CHAPTER 7

Conclusion

In this thesis, we provide a parallelized Borůvka’s MST implementation on a BSP-like massive graph processing platform. At first, the thesis goes over the MST on other high-performance computation platforms, such as the GPUs and SMPs. Then, several classic MST computation algorithms are introduced including the original Borůvka’s algorithm. Based on the distributed algorithm, we explain the Google’s most famous Pregel graph processing model, Hadoop Map-Reduce model, and GraphSQL’s patented model. Finally, it comes to the detailed implementation of Borůvka’s algorithm and some test samples study. With the assistance of BSP-like graph processing platform, our work succeeded in computing MST of big graph data on a single machine. This work also explores several fundamentals of distributed algorithm, for instance, the leader selection problem. In the future, this work could be extended in various ways. One possible way is to optimize the leadership selection procedure for better performance. Another way is to speed up the processing speed by using distributed graph processing model. What is more, some other graph algorithms could be built upon the MST implementation, such as the Steiner Tree Approximation.
BIBLIOGRAPHY


