Solving Large Graph Problems in MapReduce-Like Frameworks via Optimized Parameter Configuration

Huanle Xu, Ronghai Yang[1(✉)], Zhibo Yang, and Wing Cheong Lau

The Department of Information Engineering
The Chinese University of Hong Kong, Hong Kong
{xh112,yr013,yz014,wclau}@ie.cuhk.edu.hk

Abstract. In this paper, we propose a scheme to solve large dense graph problems under the MapReduce framework. The graph data is organized in terms of blocks and all blocks are assigned to different map workers for parallel processing. Intermediate results of map workers are combined by one reduce worker for the next round of processing. This procedure is iterative and the graph size can be reduced substantially after each round. In the last round, a small graph is processed on one single map worker to produce the final result. Specifically, we present some basic algorithms like Minimum Spanning Tree, Finding Connected Components and Single-Source Shortest Path which can be implemented efficiently using this scheme. We also offer a mathematical formulation to determine the parameters under our scheme so as to achieve the optimal running-time performance. Note that the proposed scheme can be applied in MapReduce-like platforms such as Spark. We use our own cluster and Amazon EC2 as the testbeds to respectively evaluate the performance of the proposed Minimum Spanning Tree algorithm under the MapReduce and Spark frameworks. The experimental results match well with our theoretical analysis. Using this approach, many parallelizable problems can be solved in MapReduce-like frameworks efficiently.

Keywords: MapReduce; Graph Problems; Parallel Computing

1 Introduction

The amount of data worthwhile to be analyzed has grown at an astonishing rate in recent years. Although the processing power of commercially available servers has also grown at a remarkable pace over past decades, it remains woefully inadequate to deal with such huge amount of data. Malewicz et al. show that the World Wide Web graph can consist of billions of nodes and trillions of edges in [12]. Such graph data contributes a lot to the Big Data space. J. Leskovec el al. demonstrate that 9 different massive graphs from 4 different domains are growing over time and these graphs continue to become denser [10]. Under such a situation, it is beneficial to handle large dense graphs via parallel processing over multiple machines.

In the literature, there has been many works seeking parallelism to solve graph problems. However, most of those algorithms need to build a complicated communication model between parallel processors [13], [3], [15], [4]. Today, the MapReduce framework in Google [2] and the open source project Hadoop[1] provide a very flexible way for developers to write parallel programs. Under this framework, different processors do not need to communicate with each other in a Map or Reduce phase.

In this paper, we present a method to solve a series of graph problems under the MapReduce framework. The primary idea is to encode the original graph into $< key, value >$ pairs where each key represents an edge and each value represents the corresponding metric. The encoded graph is managed by the Distributed File System (DFS) in terms of blocks and each block is assigned to an individual map worker. In this way, each map worker only processes a subgraph and outputs an intermediate result whose size is much smaller than the original input. These intermediate results from each map worker are then combined together to form a new graph. The resultant new graph is then fed as the input of the following MapReduce round. By repeating this procedure, the graph can finally be processed by one single machine.

Following the aforementioned process, it is natural to ask how to choose the two important parameters: the number of MapReduce rounds and the number of map workers in each round. We find these two parameters from the view of running-time optimization for graph algorithms under the MapReduce framework. To be more specific, we formulate an optimization problem to minimize the running time, whose solution determines the optimal values of these two parameters. Our proposed techniques can be applied to tackle various classical graph problems including Finding Minimum Spanning Tree, Connected Components and the Single Source Shortest Path on large graphs. To validate the efficacy of our scheme, we implement the proposed Minimum Spanning Tree (MST) algorithm as a showcase using our own private cluster. Empirical measurement results indicate that our theoretical analysis is valid and can be extended to a variety of parallelizable problems. In summary, the technical contributions of this paper include:

- We propose a method for solving large dense graph problems under the MapReduce framework. This method is also applicable to other frameworks such as Spark [19].
- Under our proposed scheme, we formulate a running time minimization problem whose solution determines the number of MapReduce rounds and the number of map workers in each round.
- To validate our theoretical analysis, we implement the MapRedcue-based MST algorithm in real clusters and conduct extensive experiments.

The rest of this paper is organized as follows. After reviewing the literature work for solving graph problems under parallel processing frameworks in Section 2, we present the preliminaries of large graph problems and the MapReduce framework in Section 3. Section 4 discusses three classical graph problems to show how our

[1] http://hadoop.apache.org

proposed scheme can be used to solve various classical graph problems under MapReduce-like frameworks. We then present a mathematical model treating the running time performance as an optimization problem in Section 5. We implement the MapReduce-based Minimum Spanning Tree algorithm and show the experimental results in Section 6. Finally, Section 7 concludes our work.

2 Related Work

In the literature, designing graph algorithms under parallel processing frameworks has drawn a lot of attentions from both industry and academia.

To handle the issue of limited memory in a MapReduce cluster, Karloff et al. present a model of computation of MapReduce called $\mathcal{MRC}$ in [5]. This work describes a basic building block for many algorithms under the MapReduce framework and it focuses fitting a large dense graph on a machine with memory constraint. Following this framework, Lattanzi et al. propose several algorithms whose primary goal is to produce correct results with high probability even with limited memory [9]. In this paper, we adopt the same idea as [9], however, our work manages to optimize the running time of graph algorithms under MapReduce regardless of the memory restriction. In the meanwhile, our solution approach can guarantee the final correctness.Instead of studying the large dense graph, Spangler et al. present algorithms in MapReduce for sparse graph including maximal matching, approximate edge covering over grid graphs [16].

There also exist some graph algorithms which are problem-oriented under the MapReduce framework. For example, Xiang et al. propose several parallel schemes to partition a graph in [17] such that different worker nodes can compute the maximum cliques of different subgraphs independently. Kolda et al. implement a *wedge sampling* approach to efficiently and accurately estimate the clustering coefficient for massive graphs [6]. However, one limitation of these works is that they can only solve very specific problems.

Most of existing iterative algorithms under MapReduce(-like) model manages to minimize the number of rounds. Andoni et.al present algorithms that work in a constant number of rounds for geometric graphs under limited memory with unbounded computation power [1]. Though this model solves some specific graph-related problems including computing MST in a small number of rounds, it lacks of experimental results to show the efficiency of the algorithm. Kumar et al. proposes a sampling technique that aids in parallelization of sequential algorithms under the MapReduce framework [8]. The experimental results demonstrate that their proposed algorithms can reduce the number of rounds substantially comparing to the standard sequential algorithms. Going forward, Qin et al. introduce a scalable graph processing class to solve two graph problems (i.e., Finding Connected Components and Minimum Spanning Forest) with logarithmic number of rounds and linear communication costs per round [14].

Other than MapReduce, Pregel [12] and Graphlab [11] are also very popular programming platforms for solving graph problems. However, under these two platforms, different worker nodes need to communicate for information exchange very frequently. Under the Pregel framework, Yan et al. propose algorithms that

require linear space, communication and computation per round with logarithmic number of rounds [18]. In contrast, our solution approach for solving large graph problems is easy to implement while keeping a low communication cost between different worker nodes.

3 Algorithm Design Framework

Let $G = (V, E)$ be an undirected graph where $|V| = n$ and $|E| = m$. Assume G is a $c - dense$ graph, i.e., $m = n^{1+c}$ where $0 < c \leq 1$. Consider an operation on the graph, h, namely, $h(G)$ is the targeting result. When n is very large and c is non negligible, directly operating h on the original graph with a single machine is time-consuming and may not be feasible due to the memory bottleneck. To tackle these issues, we design a MapReduce-based algorithm to compute $h(G)$.

Before going to the details of the graph algorithm design under the MapReduce framework , we first introduce the basics of the MapReduce programming model.

3.1 MapReduce Programming Model

Under the MapReduce framework, the input data for all applications are stored and organized on DFS (Distributed File Systems) in terms of blocks. Each block consists of multiple records in the form of $< key_1, value_1 >$. A MapReduce operation essentially includes two phases: Map and Reduce. In the map phase, each block is assigned to one map worker and the worker calls a map function to compute and generate an intermediate result. The intermediate results in the form of $< key_2, list(value_2) >$ are distributed to different reduce workers based on $key_2's$ with the guarantee that records with the same key_2 are delivered to a unique reduce worker. During the reduce phase, each reduce worker executes the same reduce function to process these key-value pairs and the computation result is also stored on DFS. The map and reduce functions need to be defined in the application before submitted. One important feature with regard to MapReduce is that all workers in each phase can run in parallel. It's worthy to note that, one application may involve more than one MapReduce rounds where the output from the previous round is treated as input of the next round.

3.2 A General Graph Algorithm Design Framework in MapReduce

We represent the graph data with multiple $< key, value >$ records so that it can fit into the MapReduce programming model. To be more specific, we create a record, i.e., $< (v_i, v_j), w_{i,j} >$, for each edge $(v_i, v_j) \in E$ and associate this record with a metric $w_{i,j}$ which can be the edge weight, capacity, cost, etc. In the first MapReduce round, each block is assigned to one unique map worker which applies another function f to process the input data. In the reduce phase, only one reduce worker combines all intermediate results generated from map workers and applies the reduce function g to produce key value pairs with the form of $< (v_i, v_j), w_{i,j} >$. This procedure is repeated with multiple rounds and the

output size is expected to be much smaller than the input size within each round. In the final round, the input data, which is denoted by G_l, can be processed by one map worker with Operation h. Moreover, to guarantee the correctness of these operations, $h(G_l)$ must be equivalent to $h(G)$.

3.3 Graph Algorithm Design under Spark

Spark [19] is a fast and general engine for large-scale data processing with in-memory computing supported. To make it more efficient for iterative and interactive jobs, Spark introduces an abstraction called resilient distributed datasets (RDD). Readers can refer to [19] for the detailed description. It's important to note that, the aforementioned operations can be readily implemented under Spark efficiently. To demonstrate this, we adopt the same data structure as the MapReduce framework. The algorithm is iterative, at the beginning of each iteration, one task is created for a RDD object and it applies Operation f on this RDD, which then transforms the original RDD into a new one. At the end of this iteration, these newly generated RDDs are merged together and further repartitioned into several small RDD objects, which are fed as the input of the subsequent iteration. Finally, at the last iteration, only one RDD is processed by a single task using Operation h.

4 Design Examples of Graph Algorithms

We adopt the MapReduce framework, which partitions the graph based on edges, to solve some basic graph problems such as Finding MST, Connected Components and Single-Source Shortest Path.

4.1 Finding the Minimum Spanning Tree

To compute MST on a single map worker, we adopt the Kruskal algorithm [7] whose major steps are illustrated as shown below.

1. Create a forest F, where each vertex in the graph is a separate tree;
2. Create a set E which contains all edges sorted in a non-decreasing order;
3. While E is nonempty and F is not yet connected:
   - Remove an edge with minimum weight from E;
   - If the removed edge connects two different trees, then adds it to the forest F.
4. Return F.

As shown in Algorithm 1, Kruskal algorithm can be correctly and efficiently implemented in a parallel and distributed manner. We adopt the framework proposed in Section 3.2 to partition a large dense graph into multiple small subgraphs such that each of them can be processed by a map worker. We then apply the Kruskal algorithm to find a MST for each subgraph. Since an MST contains at most $n-1$ edges, the graph size is therefore reduced substantially. In what follows, we merge the intermediate MSTs into a single graph. When the

resultant graph is small enough to fit into the memory of a single machine, we compute the MST of this graph on one map worker and output the final result.

Since each mapper throws away the heaviest edge that satisfies the cycle property[2] at each iteration, these edges should not to be a part of any MST. As such, the aforementioned processes can generate the correct MST, as illustrated in the following lemma:

Algorithm 1 Finding Minimum Spanning Tree of Graph G(V, E)

1: **for** $|E| > \eta$ **do**
2: **Partition**: partition a large graph into multiple sub-graphs.
3: **Map Phase**: Each mapper i runs Kruskal algorithm on sub-graphs $G(V, E_i)$;
4: **Reduce Phase**: Merge the intermediate MST into a smaller graph: $G(V, \bigcup_i S_i)$.
5: **Re-Configure**: Configure the number of map workers m such that $\frac{|E|}{m} < \eta$.
6: **end for**
7: One mapper runs Kruskal algorithm on the resulting graph and return the result.

Proposition 1 *Algorithm 1 can output a correct Minimum Spanning Tree for any graph.*

Proof. For any cycle C in the graph, the edge with the highest weight in it cannot belong to an MST. In each round, the computation of any map worker is to delete the heaviest edges that are in some particular cycles. As such, these deleted edges do not belong to the final MST. Therefore, the remaining edges after each round are sufficient to produce the correct Spanning Tree. This completes the proof.

In the literature, there exists many work for parallelizing the MST algorithm in a multi-processors cluster. Most of them can speed up the processing but suffer from a high communication cost between processors [3,13]. As a comparison, the communication cost in our approach is low while the speedup is significant as characterized in the following sections.

4.2 Finding Connected Components

In recent years, there exist several works which manage to find the connected components in a large-scale graph under the MapReduce framework. For instance, the algorithm proposed in [15] can find connected components in logarithmic rounds. However, the requirement of running on $O(n)$ machines makes it difficult to be implemented in practice.

Since computing MST does not destroy the connectivity of the whole graph, hence, Algorithm 1 can be readily applied to finding connected components in a graph. The only difference is that the result would be a forest containing multiple Spanning Trees instead of only one MST.

4.3 Finding the Single-Source Shortest Path

Finding the single-source shortest path (SSSP) between a pair of nodes on a weighted graph is a classical problem in Graph Theory. Here, we compute the

[2] http://en.wikipedia.org/wiki/Spanning_tree

SSSP by adopting the same method as discussed in Section 3.2. The graph data is partitioned into several blocks and each block is assigned to one map worker. To guarantee the correctness of the final result, every map worker deletes the edges whose wight are the heaviest in some particular cycles. Towards the end, a single node computes the shortest path on the resultant graph whose size is small enough.

5 Performance Optimization for Graph Problems under the MapReduce Framework

The correctness of all graph algorithms presented in Section 4 are guaranteed regardless of the memory capacity of each machine in a cluster. People may naturally ask how these algorithms improve the performance comparing to the single instance case. When we implement the MapReduce-based MST algorithm in a real cluster, we observe that the performance in terms of running time is a function of the number of map workers as well as the number of MapReduce rounds. Therefore, there exist optimal solutions for these two parameters to minimize the running time. In this section, we propose a general framework to optimize the running time performance of parallel algorithms similar to fining MSTs under MapReduce-like frameworks.

We consider a parallelizable algorithm h whose time complexity is $g(x)$, where x is the input size. Each map worker produces an intermediate result with the size of $p(x)$ for an input, whose size is x. We assume the the MapRedcue-based algorithm runs for k rounds. Therefore, the total running time of this algorithm can be formulated in the following equation:

$$f^k(l_1, l_2, \cdots, l_k) = g\left(\frac{x}{l_1}\right) + \sum_{i=2}^{k} g\left(\frac{p\left(\frac{x_{i-1}}{l_{i-1}}\right) l_{i-1}}{l_i}\right) \tag{1}$$

where l_i $(1 \leq i \leq k)$ and x_i are the number of map workers and the input size in the ith round respectively. Following the computing framework in Section 4, l_k is equal to one. Due to the size of an output in each round is smaller than that in the previous round, it holds that $l_i < l_{i-1}$ for $2 \leq i \leq k$.

Our objective is to find appropriate values for k and $l_1, l_2, \cdots, l_k$ such that Equation (1) is minimized, which yields the following optimization problem (P1):

$$\begin{aligned} \min_{l_1, l_2, \cdots, l_k} \quad & f^k(l_1, l_2, \cdots, l_k) \\ s.t. \quad & l_i < l_{i-1}; \quad \forall\ 2 \leq i \leq k \\ & l_k = 1 \end{aligned} \tag{2}$$

We will solve this optimization problem in the next section using the MapReduce-based MST algorithm as a representative example. Although we only solve one specific problem, our solution approach can be applicable to a variety of parallelizable algorithms.

6 Performance Evaluation

In this section, we implement our proposed MST algorithm under MapReduce and Spark respectively. We begin by introducing the environment setup and then evaluate the running time with regard to the number of workers and MapReduce rounds. The solutions to Optimization Problem P1 are compared with the experimental results.

6.1 Environment Setup with MapReduce Framework

The graph data is distributed across our private cluster which has two master nodes and 4 slave nodes with 16 GB memory and 8-cores CPU each. Under our implementation, the weighted graph is randomly generated with $n = 10^7$ vertexes and $m = 7 \times 10^8$ edges. The size of the graph is 14 GB, which is difficult to be processed on one single machine. We generate the weight for each edge uniformly at random between [1,20000]. Although we only process a graph with such an order of magnitude, our algorithm is scalable to apply to graphs with much larger size.

6.2 Environment Setup with Spark Framework

Since our private cluster only deploys the MapReduce framework, we turn to Amazon EC2 for running the MST algorithm under Spark. This cloud-based cluster consists of 1 master node and 12 slave nodes. Each node is configured with the m1.large instance, i.e., 7.5 GB memory and 2-core CPU. In this experiment, the weighted graph is randomly generated with $n = 2.9 \times 10^6$ vertexes and $m = 1.5 \times 10^8$ edges. The weight of each edge is produced uniformly at random between [1,20000]. This graph keeps the same dense (i.e., c) as the graph in the previous experiment.

6.3 The Optimal Number of Workers in Two MapReduce rounds

We first analyze the performance with respect to the number of workers when the large graph is processed within two rounds. In the following theoretical analysis, we only use MapReduce as the programming platform. Nevertheless, this analysis is also applicable to Spark. For ease of presentation, we will use the notations of map worker and worker node interchangeably in the rest of this paper.

Analytical Results. We denote by ι the number of machines used to run MST in the first round. In the second round, there is only one worker node to process the graph data and finds a correct MST.

In the first round, the size of the graph data allocated to each worker node is $\frac{m}{\iota}$. Therefore, the running time of a map worker in the first round can be characterized by:

$$f_1(\iota) = \frac{m}{\iota} \log \frac{m}{\iota} \tag{3}$$

After the first MapReduce round completes, each worker node finds a spanning tree whose size is at most $n - 1$. Thus, the size of the resultant graph

generated by the first round is at most $(n-1)\cdot\iota$. Hence, the running time in the second round is bounded by:

$$f_2(\iota) = (n-1)\iota \cdot \log\left((n-1)\iota\right) \tag{4}$$

And the total running time for the whole algorithm is $f(\iota) = f_1(\iota) + f_2(\iota)$. By setting the derivative of $f(\iota)$ to zero, we obtain the optimal value for ι which minimizes the total running time in the below:

$$\iota^* = \sqrt{m/(n-1)} \approx \sqrt{m/n} \tag{5}$$

We illustrate the picture of $f(\iota)$ in Fig. 1 (a) where $m = 7\times10^8$ and $n = 10^7$. It shows that, when ι is set to 8, the overall running time achieves its minimum. We further let $\iota = \sqrt{m/n}$, f^2 can be upper bounded by:

$$f^2 = 2\sqrt{mn}\log\sqrt{mn} \tag{6}$$

When the memory of the worker node is of size no less than the graph size. We can find an MST by using only one map worker with one MapReduce round. In this case, the running time is

$$r^1 = m\log m \tag{7}$$

Combine Equation (6) and (7), we have:

$$\frac{f^2}{r^1} \leq \frac{2\sqrt{mn}\log m}{m\log m} = 2\sqrt{\frac{n}{m}} \tag{8}$$

Following Equation (8), we conclude that we can achieve a speedup of at least $\sqrt{\frac{m}{4n}}$ comparing to the single instance case by using $\sqrt{\frac{m}{n}}$ map workers in two MapReduce rounds. The analytical result is illustrated in Fig. 1 (a) and the running time is normalized.

Simulation Results. As illustrated in Fig. 1(b), the MapRedcue-based MST algorithm enjoys its optimal performance with 9 worker nodes in the first round. In contrast, Fig. 1(a) shows that the theoretical result attains its optima with 8 worker nodes. Given other factors that we do not take into consideration, the nuance (11s) between theoretical results and practical performance is reasonable.

Intuitively, there exists a trade-off between the first and second round. When the number of map workers in the first round becomes larger, each worker spends less time for computation on the graph data of a smaller size. Hence, it leads to less edges being deleted and a larger intermediate graph generated in the first round. As a result, it takes a longer time to find the final MST for the worker node in the second round. This intuition matches well with the observations from Fig. 1 that increasing the number of map workers in the first round results in a corresponding rise of the time spent in the second round.

The experimental result under the Spark framework is depicted in Fig. 2. It indicates that the optimal number of workers is 7 in the first iteration and the simulation result matches quite well with the theoretical result.

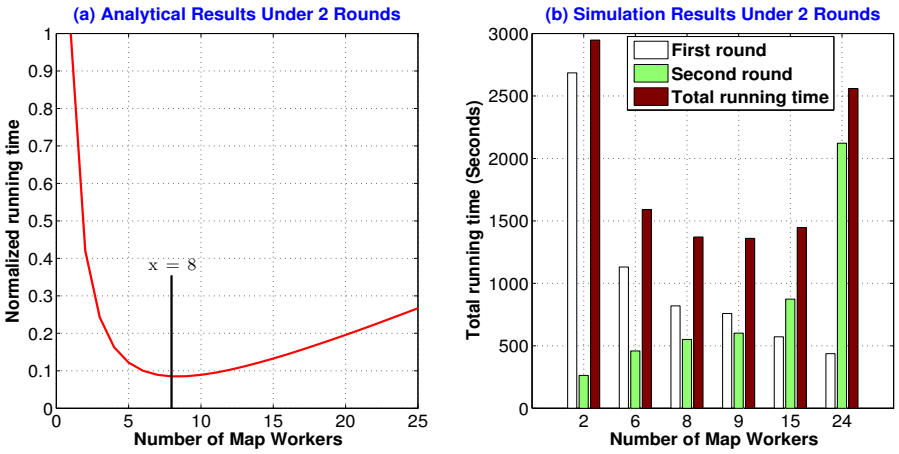

Fig. 1. The analytical and simulation results for the running time of MST algorithm in two MapReduce rounds where $m = 7 \times 10^8$, $n = 10^7$.

6.4 Deriving the Optimal Number of MapReduce Rounds

The number of MapReduce rounds is another important parameter which has a heavy impact on the running-time performance. When this number is too small, then in each round, all worker nodes need to process a large subgraph and thus take a long time. On the other hand, when the number of MapReduce rounds is very large, the overall running time is also significant though each node can work out the result within a short time.

With the above observations in mind, we aim to explore the relationship among the running time, the number of MapReduce rounds and map workers. In the following sections, we first present the analytical results and then conduct extensive simulations to evaluate the performance.

Analytical Results. The time complexity of Kruskal algorithm is $O(mlogm)$, where m is the number of edges. Substitute $g(m) = m \log m$ into Equation (1), we derive the total running time of the MapReduce-based MST algorithm, which can be formulated as follows:

$$f^k(l_1, l_2, \cdots, l_k) = \frac{m}{l_1} \log \frac{m}{l_1} + \sum_{i=2}^{k} \frac{(n-1)l_{i-1}}{l_i} \log \frac{(n-1)l_{i-1}}{l_i} \tag{9}$$

Further, we substitute Equation (9) into P1 and derive an upper bound for the optimal value, which is illustrated in the following theorem :

Theorem 1 *For all* $k \geq 2$,

$$f^k \leq g^k \triangleq \frac{m}{l_1} \log \frac{m}{l_1} + (k-1)(n-1)l_1^{1/(k-1)} \log (n-1)l_1^{1/(k-1)} \tag{10}$$

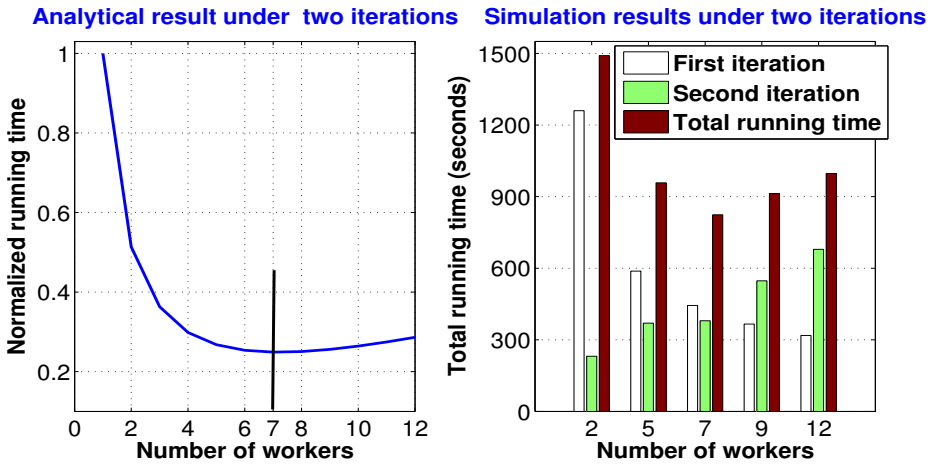

Fig. 2. The analytical and simulation results for the running time of MST algorithm under Spark with two iterations where $m = 1.5 \times 10^8$, $n = 2.9 \times 10^6$

Proof. We prove this theorem by mathematical induction. For $k = 2$, this inequality can be satisfied as demonstrated in Subsection 6.3. When $k = 3$, we have:

$$f^3 = \frac{m}{l_1} \log \frac{m}{l_1} + \frac{(n-1)l_1}{l_2} \log \frac{(n-1)l_1}{l_2} + (n-1)l_2 \log (n-1)l_2$$

Similar to Equation (6), it holds that

$$f^3 \leq \frac{m}{l_1} \log \frac{m}{l_1} + 2(n-1)\sqrt{l_1} \log (n-1)\sqrt{l_1}$$

Hence, f^3 satisfy Inequality (10). Further, we assume that when $k = p$, all (f^i)'s (for $2 \leq i \leq p$) satisfy Equation (10). Then, for $k = p + 1$, we get:

$$\begin{aligned} f^{p+1} &= \frac{m}{l_1} \log \frac{m}{l_1} + \frac{(n-1)l_1}{l_2} \log \frac{(n-1)l_1}{l_2} + \sum_{i=3}^{p+1} \frac{(n-1)l_{i-1}}{l_i} \log \frac{(n-1)l_{i-1}}{l_i} \\ &\leq \frac{m}{l_1} \log \frac{m}{l_1} + \frac{(n-1)l_1}{l_2} \log \frac{(n-1)l_1}{l_2} + (p-1)(n-1)l_2^{1/(p-1)} \log (n-1)l_2^{1/(p-1)} \end{aligned}$$

Further, we define h^{p+1}, which is a function of l_2, as follows:

$$h^{p+1}(l_2) = \frac{(n-1)l_1}{l_2} \log \frac{(n-1)l_1}{l_2} + (p-1)(n-1)l_2^{1/(p-1)} \log (n-1)l_2^{1/(p-1)} \quad (11)$$

By setting the derivative of $h^{p+1}(l_2)$ to zero, we derive the optimal solution to Equation (11), which yields $l_2 = l_1^{\frac{p-1}{p}}$. Therefore, the optimal value of $h^{p+1}(l_2)$ can be characterized as $(h^{p+1}(l_2))^* = h^{p+1}(l_1^{\frac{p-1}{p}}) = p(n-1)l_1^{\frac{1}{p}} \log (n-1)l_1^{\frac{1}{p}}$. It follows that:

$$f^{p+1} = \frac{m}{l_1} \log \frac{m}{l_1} + h^{p+1}(l_2) \leq \frac{m}{l_1} \log \frac{m}{l_1} + p(n-1)l_1^{\frac{1}{p}} \log (n-1)l_1^{\frac{1}{p}} \quad (12)$$

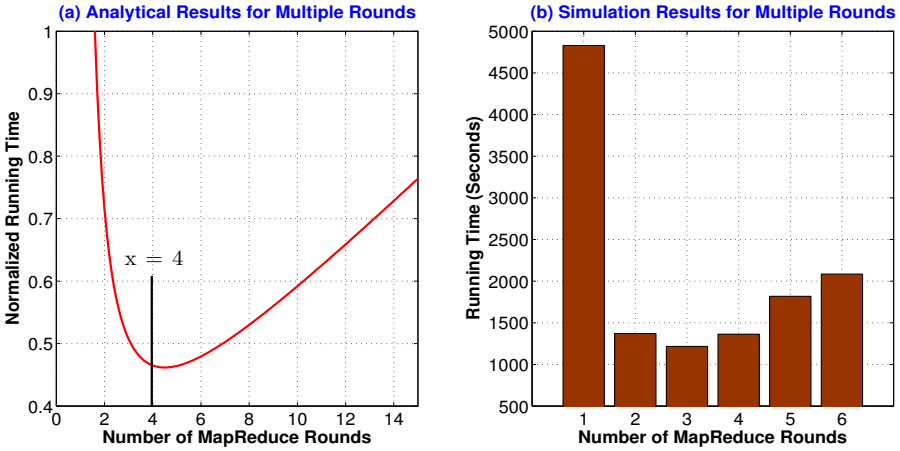

Fig. 3. The analytical and simulation results for the running time of MST algorithm related to the number of MapReduce rounds. $n = 10^7$ and $c = 0.2636$.

Hence, f^{p+1} satisfies Equation (10). This completes the proof.

Based on the proof, it can be readily shown that the optimal solution satisfies the following condition:

$$l_{i+1} = l_i^{\frac{k-i-1}{k-i}} \quad \forall\ 1 \leq i \leq k-1 \tag{13}$$

Further, we set the derivative of g^k to zero and get $l_1 = (\frac{m}{n-1})^{\frac{k-1}{k}}$. In the meanwhile, optimal value to P1 is upper bounded by:

$$(f^k)^* \leq km^{\frac{1}{k}}(n-1)^{\frac{k-1}{k}} \log\left(m^{\frac{1}{k}}(n-1)^{\frac{k-1}{k}}\right) \tag{14}$$

Since $m = n^{1+c}$, we have:

$$(f^k)^* \approx (k+c)n^{1+\frac{c}{k}} \log n \tag{15}$$

Define $\xi(k) = (k+c)n^{1+\frac{c}{k}}$ and set the derivative of $\xi(k)$ to zero, we get $k \approx c \ln n$. When k is below $c \ln n$, $\xi(k)$ is an increasing function of k. On the other hand, when k is above $c \ln n$, $\xi(k)$ is a decreasing function. As such, $\xi(k)$ attains the minimum value when $k = c \ln n$. Moreover, l_{k-1} is above one due to the first constraint in P1, therefore, we have $(\frac{m}{n-1})^{\frac{1}{k}} \geq 2$, which is equivalent to $k \leq c \log n$. Hence, we conclude that the optimal number of MapReduce rounds is $c \ln n$ and the corresponding number of worker nodes in each round is determined by Equation (13).

We illustrate the running time as a function of the number of MapReduce rounds in Fig. 3 under $n = 10^7$ and $c = 0.2636$. In this case, the optimal number of MapReduce rounds is 4.

Simulation Results. The experiments result under the MapReduce framework are illustrated in Fig. 3(b). It shows that the minimum running-time is achieved

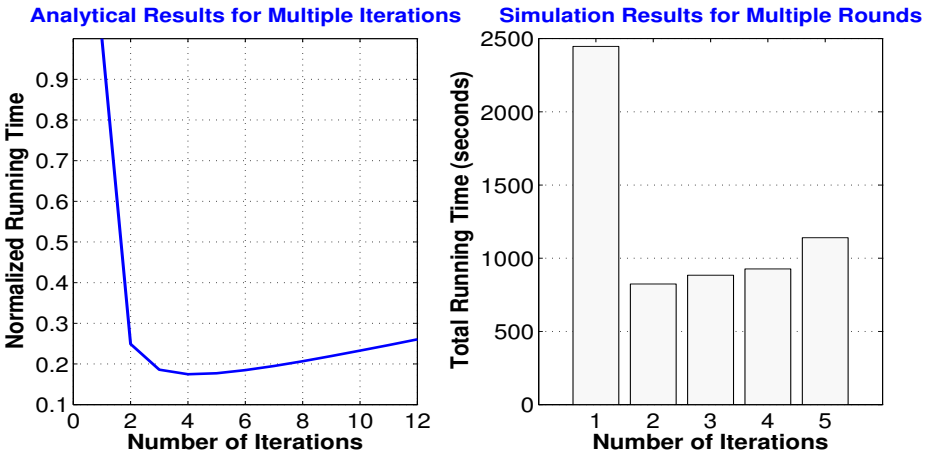

Fig. 4. The analytical and simulation results for the running time of MST algorithm where $n = 2.9 \times 10^6$ and $c = 0.2636$.

under 3 MapReduce rounds other than 4 rounds. Considering other factors such as the overhead of job scheduling, memory, network delay in shuffle phase and I/O operation, this deviation is reasonable. Although the theoretical analysis does not quite fit well with the practical results, it still offers an useful guideline to enjoy a suboptimal performance. Moreover, Fig. 3 (b) shows the computation under one round takes 3.52 times longer than that under two rounds, which fits well with our theoretical ratio (i.e., 4.18) in Equation (8). This result exactly reflects the advantage by solving graph problems in a parallel manner.

We illustrate the experimental results of the MST algorithm under the Spark framework in Fig. 4. The findings under MapReduce and Spark are similar.

7 Conclusions

In this paper, we propose a parallel algorithm design scheme to solve large graph problems under the MapReduce framework. Our main idea is to exploit the flexibility of the MapReduce Programming model to partition a large graph into multiple small subgraphs and each map worker processes a subgraph independently. We demonstrate that some classical graph problems such as Finding Minimum Spanning Tree, Connected Components and Single-Source Shortest Path can also be solved efficiently under this scheme. Since this scheme involves multiple rounds, it fits well with the Spark framework as well.

To enjoy the optimal running-time performance, we formulate a running-time minimization problem and develop a mathematical tool to derive appropriate values for key parameters. More importantly, we implement the MapRedcue-base Minimum Spanning Tree algorithm as a representative example to evaluate the running-time performance. Both the experimental and the theoretical results demonstrate that the running-time is a function of the number of worker nodes as well as the number of MapReduce rounds. Our optimization framework is also applicable to a variety of parallelizable algorithms.

Acknowledgements

Huanle Xu and Ronghai Yang have contributed equally to this work. This project was partially supported by the Mobile Technologies Centre (MobiTeC), The Chinese University of Hong Kong.

References

1. Andoni, A., Nikolov, A., Onak, K., Yaroslavtsev, G.: Parallel algorithms for geometric graph problems. In: Proceeding of STOC. pp. 574–583 (2014)
2. Dean, J., Ghemawat, S.: MapReduce: Simplified data processing on large clusters. In: Proceedings of OSDI. pp. 137–150 (December 2004)
3. Dehne, F., Gotz, S.: Practical parallel algorithms for minimum spanning trees. In: Proceedings of Seventeenth IEEE Symposium on Reliable Distributed Systems. pp. 366–371 (October 1998)
4. Israel, A., Itai, A.: A fast and simple randomized parallel algorithm for maximal matching. In: Information Processing Letters (abs/12035387, 2012)
5. Karloff, H., Suri, S., Vassilvitskii, S.: A model of computation for MapReduce. In: Proceedings of SODA. pp. 938–948 (2010)
6. Kolda, T.G., Pinar, A., Plantenga, T., Seshadhri, C., Task, C.: Counting triangles in massive graphs with MapReduce. In: arXiv:1301.5887v3 (2013)
7. Kruskal, J.B.: On the shortest spanning subtree of a graph and the traveling salesman problem. Proceedings of the American Mathematical society (1956)
8. Kumar, R., Moseley, B., Vassilvitskii, S., Vattani, A.: Fast greedy algorithms in MapReduce and streaming. In: Proceeding of SPAA. pp. 1–10 (2013)
9. Lattanzi, S., Moseley, B., Suri, S.: Fitering: A method for solving graph problems in MapReduce. In: SPAA (June 2010)
10. Leskovec, J., Kleinberg, J., Faloutsos, C.: Graphs over time: Densification laws, shrinking diameters and possible explanations. In: KDD (2005)
11. Low, Y., Bickson, D., Gonzalez, J., Guestrin, C., Kyrola, A., Hellerstein, J.M.: Distributed graphlab: a framework for machine learning and data mining in the cloud. In: Proceedings of the VLDB Endowment. pp. 716–727 (2012)
12. Malewicz, G., Austern, M.H., Bik, A.J., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: A system for large-scale graph processing. In: SIGMOD (2010)
13. Moussa, M.I.: A new parallel algorithm for computing MINIMUM SPANNING TREE. International Journal of Soft Computing, Mathematics and Control. (2013)
14. Qin, L., Yu, J.X., Chang, L., Cheng, H., Zhang, C., Lin, X.: Scalable big graph processing in MapReduce. In: Proceeding of SIGMOD. pp. 827–838 (June 2014)
15. Rastogi, V., A., M., Chitnis, L., Sarma, A.: Finding connected components in Map-Reduce in logarithmic rounds. In: Computing Research Repository (CoRR) (22(2):77C80, 1986)
16. Spangler, T.: Algorithms for grid graph in MapReduce model. Thesis of master (December, 2013)
17. Xiang, J., Guo, C., Aboulnaga, A.: Scalable maximum clique computation using MapReduce. In: International conference on data engineering (April 2013)
18. Yan, D., Cheng, J., Lu, Y., Ng., W.: Practical Pregel algorithms for massive graphs. In: Technical Report (September 2013)
19. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M.J., Shenker, S., Stoica, I.: Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In: NSDI (2012)