Efficient Breadth-First Search on Large Graphs with Skewed Degree Distributions

Haichuan Shang
Institute of Industrial Science
The University of Tokyo, Japan
shang@tkl.iis.u-tokyo.ac.jp

Masaru Kitsuregawa
Institute of Industrial Science
The University of Tokyo, Japan
kitsure@tkl.iis.u-tokyo.ac.jp

ABSTRACT

Many recent large-scale data intensive applications are increasingly demanding efficient graph databases. Distributed graph algorithms, as a core part of practical graph databases, have a wide range of important applications, but have been rarely studied in sufficient detail. These problems are challenging as real graphs are usually extremely large and the intrinsic character of graph data, lacking locality, causes unbalanced computation and communication workloads.

In this paper, we explore distributed breadth-first search algorithms with regards to large-scale applications. We propose DPC (Degree-based Partitioning and Communication), a scalable and efficient distributed BFS algorithm which achieves high scalability and performance through novel balancing techniques between computation and communication. In experimental study, we compare our algorithm with two state-of-the-art algorithms under the Graph500 benchmark with a variety of settings. The result shows our algorithm significantly outperforms the existing algorithms under all the settings.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems

General Terms

Algorithms, Experimentation, Performance

Keywords

graph database, breadth-first search, graph indexing

1. INTRODUCTION

Graph databases are playing an increasingly important role in science and industry. A large number of graph data processing techniques have been proposed in recent years and applied to a wide range of applications including social networks [3, 8, 24, 27], telecommunication [23], chemistry informatics [7, 32], medical informatics [31, 33, 34] and bioinformatics [21, 22]. However, distributed graph algorithms, as a critical component of practical large-scale graph databases, have been scarcely studied.

Breadth-first search (BFS) is a fundamental problem in graph databases. Given a graph $G = (V, E)$ and a particular root vertex r, a BFS algorithm wants to find a BFS tree rooted at r. That is, the path from r to any other vertex v must contain the minimum number of edges among all the paths from r to v in the original graph G. BFS algorithms play a critical role in numerous applications. For example, social network games may search for a certain user's k-hop friends to deploy the game, while online advertising companies may want to generate the BFS tree rooted at a product customer to advertise the product to the customer's k-hop social neighborhoods. Ideally, the running time of a sequential BFS algorithm is in $O(|V| + |E|)$ complexity when G is represented in an adjacency list. Unfortunately, when we consider the large graphs generated from social networks such as Facebook (350 million users), Twitter (75 million users) and LinkedIn (60 million users), it is impossible to store such graphs in the main memory of a single computer. This makes the sequential BFS algorithms [9, 10, 13] and the traditional shared-memory parallel algorithms [12, 18, 19, 25] (i.e. the algorithms based on the PRAM model) impractical for the large-scale applications, because these algorithms take several minutes or more for a single query.

In contrast to sequential/shared-memory parallel BFS algorithms, a distributed BFS algorithm is an algorithm for a communication network to find a BFS tree. Initially, each compute node has a subset of the edges, while it is unaware of the topology of the whole graph. The compute nodes, which have the edges emanating from the root, start the algorithm and send messages to the other compute nodes. The other compute nodes participate in the computation when they receive the messages. When the algorithm stops, each reachable vertex from the root knows its parent in the BFS tree. Note that a vertex does not need to know the shortest path to the root, since the vertex can send a message to its parent and so forth to generate the path.

Most existing approaches use predefined partitioning functions to partition the edges into the compute nodes. However, large real world graphs usually have highly skewed degree distributions. It causes the compute nodes which have the high degree frontiers become the communication bottlenecks at each level of the BFS tree. This problem is challenging since (1) online applications require very fast response time and the graphs in these applications are often extremely huge which make most data mining algorithms

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
EDBT/ICDT '13, March 18 - 22 2013, Genoa, Italy
Copyright 2013 ACM 978-1-4503-1597-5/13/03$15.00.

impractical for this problem; (2) the graph density and the degree distribution vary depending on the type of applications; and (3) the computing power of compute nodes and the communication bandwidth and latency vary depending on the configuration of a communication network.

To address these challenges, we propose DPC (Degree-based Partitioning and Communication), a scalable and efficient distributed BFS algorithm which achieves high scalability and performance through novel and clever techniques to achieve the balance between computation and communication for different degree distributions and hardware configurations.

Our contributions can be summarized as follows:

1. We are the first to introduce a general framework to capture the commonalities of many existing algorithms which are based on various kinds of graph partitioning methods and communication patterns. We extensively explore the partitioning methods and analyze the communication patterns of these methods.

2. We propose an effective graph partitioning method based on the degree of each vertex. Our method avoids the heavy computational and communication workloads on the compute nodes which own high degree frontiers and achieves high scalability and performance by balancing communication-loads.

3. We observe that BFS is neither a simple communication-intensive nor computation-intensive application on modern communication networks. Tuning the communication-to-computation ratio is a critical issue. Our proposed algorithm can always achieve a high performance by tuning this ratio. We propose an effective tuning technique to achieve the balance between computation and communication. Furthermore, buffer length optimization methods are also proposed.

4. We comprehensively study the theoretical effect of these techniques and evaluate their performance. Compared with two state-of-the-art algorithms, our proposed algorithm outperforms the existing algorithms under the Graph500 benchmark with a variety of settings.

The rest of the paper is organized as follows. Section 2 presents the problem definitions and the preliminaries. Section 3 describes the general framework. Section 4 introduces two baseline algorithms under our proposed framework. Section 5 proposes our BFS algorithm. The related tuning methods and technique details are discussed in Section 6. Section 7 reports the experimental results and analyses. The related work and the conclusion are given in Section 8 and Section 9, respectively.

2. PRELIMINARIES

Following the Graph500 benchmark[2], we mainly focus on undirected pseudographs in this paper. A pseudograph is a graph that allows multiple edges and self-loops. Clearly, the techniques discussed in this paper can be straightforwardly applied to multigraphs where self-loops are not allowed, and simple graphs where neither self-loops nor multiple edges are allowed. Furthermore, we assume each undirected edge $e(v_1, v_2)$ is represented as two directed edges $e(v_1, v_2)$ and $e'(v_2, v_1)$, and our techniques can be easily extended to directed graphs. For presentation simplicity, an undirected pseudograph is hereafter abbreviated to a graph.

A graph can be represented by $G = (V, E)$ where V is the set of vertices, and $E \subseteq V \times V$ is the set of edges. We denote the number of vertices and edges by $|V|$ and $|E|$, respectively.

Given a graph $G = (V, E)$, a **path** p of length k from a vertex v to a vertex v' is a sequence $(v_0, v_1, ..., v_k)$ of vertices such that $v = v_0$, $v' = v_k$, $v_i \in V$ for $i = 0, 1, ..., k$, and $(v_i - 1, v_i) \in E$ for $i = 1, 2, ..., k$.

DEFINITION 1. *(Reachablity) Given a graph $G = (V, E)$ and two vertex v and v', we say v' is **reachable** from v if there exists a path from v to v'.*

DEFINITION 2. *(Shortest Path) Given a graph $G = (V, E)$ and two vertex v and v', a shortest path from v to v' is a path with length k satisfying that k is the minimum length among all the paths from v to v'.*

Breadth-First Search. Breadth-first search(BFS) is defined as follows.

DEFINITION 3. *(Breadth-First Search) Given a graph $G = (V, E)$ and a vertex r, breadth-first search is to produce a BFS tree T with root r satisfying that*

1. *T contains all reachable vertices from r in G.*
2. *$\forall v \in T$, the path from r to v corresponds to a shortest path from r to v in G.*

3. FRAMEWORK

Initially, the graph is obtained as a list of edge tuples. Each edge tuple $e(src, tgt)$ is undirected with its two endpoints given in the tuple as source vertex(src) and target vertex(tgt). In the preprocessing phase, the edge list are randomly partitioned into N equal-sized parts where N is the number of compute nodes. In Figure 1, there are four compute nodes, denoted by $CN1$-$CN4$. We do not consider the obtaining edge lists phase in our framework, since the access to data sources can vary depending on different applications.

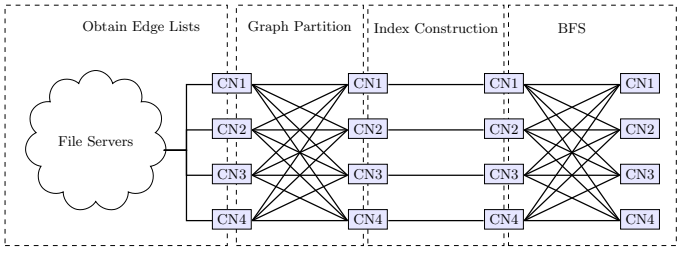


Figure 1: Framework

Assuming a graph has already been obtained as N initial edge lists by the compute nodes, our framework consists of three phases: graph partition, index construction and BFS.

1. **Graph Partition Phase.** In the first phase, the compute nodes exchange the edge tuples to generate local edge sets. Each compute node owns a local edge set. These local edge sets can be either overlapped or non-overlapped. Statistical or meta information such as the number of vertices is computed in this phase.

2. **Index Construction Phase.** Based on the local edge set and the other statistical information, each compute node builds its own index.

3. **BFS Phase.** A root vertex is given in this phase. The compute nodes produce a BFS tree with the given root by both local computing and message passing.

The core part of a BFS algorithm is the partitioning method. We formalize it as a partition function as follows.

DEFINITION 4. *(Partition Function) Given a graph $G(V, E)$ and the number N of compute nodes, a graph partition function is a function $H(e)$ where $e \in E$ and $H(e) \in \{0, 1, ..., N-1\}$.*

We analyze the existing algorithms and our algorithm under this framework by considering its computational complexity and communication cost in all three phases. As sending or receiving an edge is usually much more costly than processing one on a compute node, we analyze the communication cost by considering the total message size rather than the communication complexity.

The performance is also affected by several other issues including the degree distribution, the communication to computation ratio and the workload balancing methods, we will evaluate these issues in a experimental study. As there is no communication in the index construction phase, bandwidth will be wasted if the framework is strictly implemented to follow a sequential order. In our implementation, we always mix the graph partition and index construction phases together whenever the local data is enough to start the index construction.

4. BASELINE ALGORITHMS

In this section, we survey the existing work and introduce two baseline algorithms.

4.1 SourceBFS

SourceBFS [2] is an efficient and effective approach. This algorithm partitions the edges by the source vertex. The partitioning function is shown as follows:

$$H_{src}(e) = e.src \textbf{ mod } N$$

Algorithm 1: **Headfile for all the algorithms**

```
1 size = the number of compute nodes;
2 rank = the rank of the current compute node;
```

Algorithm 1 describes the declarations of two common variables: *size* which is the number of compute nodes in a communication network and *rank* satisfying $rank \in \{0, 1, ..., size-1\}$ which is the rank of the current compute node. For presentation simplicity, we omit these declarations in the rest algorithms. Initially, each compute node owns an initial edge set E_{init} as shown in Algorithm 2. In lines 1-4, the algorithm scans the initial edge set and stores the edges into different sending buffers by the source vertex. In lines 5-8, the algorithm exchanges the edges to ensure that each compute node owns the vertices satisfying v **mod** $size = rank$. Each compute node also owns the emanating edges from the vertices which it owns.

Algorithm 2: **SourceBFS-Partition(E_{init})**

```
   Input  : E_init is the local initial edge list of a graph;
   Output: E is the local edge set;
1  for each e (src, tgt) ∈ E_init do
2  |  q = e.src mod size;
3  |  S_q = S_q ∪ {e};
4  end for
5  for each q in [0, size − 1] except rank do
6  |  Send S_q to CN q;
7  |  Receive R_q from CN q;
8  end for
9  R_rank = S_rank;
10 E = ⋃_{q=0}^{size−1} R_q;
```

The BFS phase is described in Algorithm 3. Initially, the compute node which owns the root r inserts the root into its frontier set F_0 for the first iteration in lines 1-5. At each level i, each compute node has a set of frontier vertices F_i. The edges emanating from F_i are retrieved from each compute node's local edge set, and then the targets of these edges generate a set S which is the neighbors of F_i in lines 6-9. Some of the vertices in S which still belongs to the same compute node such as S_{rank}. Therefore, these vertices are moved to F_{i+1} of the current compute node for next iteration in line 13. For the other vertices, messages are sent to other compute nodes which own the vertices. Each compute node receives the messages R from other compute nodes and merges them to form F_{i+1} which will be the frontier set for next iteration in lines 11-15. When the algorithm terminates, the function $Parent(v)$ stores the parent of the vertices in lines 16-20.

Algorithm 3: **SourceBFS-BFS(E, r)**

```
   Input  : E is the local edge set;
            r is a root;
   Output: Parent(v) stores the parent of each vertex,
            initialized as null for all vertices;
1  if rank = r mod size then
2  |  F_0 = {r}; Parent(r) = r;
3  else
4  |  F_0 = ∅;
5  end if
6  for i = 0 to ∞ do
7  |  if F_i = ∅ for all CNs then TERMINATE.
   |  E_i = {e|e ∈ E ∧ e.src ∈ F_i};
8  |  for each q in [0, size − 1] except rank do
9  |  |  S_q = {e|e ∈ E_i ∧ q = e.tgt mod size };
10 |  |  Send S_q to CN q;
11 |  |  Receive R_q from CN q;
12 |  end for
13 |  R_rank = S_rank;
14 |  R = ⋃_{q=0}^{size−1} R_q;
15 |  F_{i+1} = {e.tgt|e ∈ R};
16 |  for each e in R do
17 |  |  if Parent(e.tgt) = null then
18 |  |  |  Parent(e.tgt) = e.src;
19 |  |  end if
20 |  end for
21 end for
```

Cost Analysis. The communication cost in the graph partition phase is $\frac{N-1}{N^2}|E|$, because each compute node initially has $\frac{|E|}{N}$ edges and $\frac{|E|}{N^2}$ of them still belong to the compute node itself after the graph partition phase. Therefore, each compute node sends $\frac{N-1}{N^2}|E|$ edges. As the function $H_{src}(e)$ takes constant time for sending each edge, the computational complexity is $O(\frac{|E|}{N})$. In the index construction phase, the computational complexity of constructing compressed sparse rows(CSR) on each compute node is $O(\frac{|E|}{N})$.

In the BFS phase, all the reachable vertices will be accessed. Then each edge $e(src,\ tgt)$ will be sent once except the ones whose src and tgt belong to a same compute node. Therefore, the communication cost is $\frac{N-1}{N^2}|E|$. If we assume that both operating an edge on communication buffers and accessing to a vertex are in $O(1)$ time complexity, the computational complexity is $O(\frac{|V|+|E|}{N})$.

4.2 TargetBFS

Intuitively, TargetBFS is alternative algorithm which partitions the edges by the target vertex. The partitioning function, $H_{tgt}(e)$ is described as follows.

$$H_{tgt}(e) \ = \ e.tgt \ \textbf{mod} \ N$$

The graph partition phase of TargetBFS is the same as that of SourceBFS except using $H_{tgt}(e)$ instead of $H_{src}(e)$, while the BFS phase is different. In the beginning of the BFS phase, all the compute nodes insert r into their frontier set F_0. At each level i, each compute node has a set of frontier vertices, F_i. The edges emanating from F_i are retrieved from the local edge set. Clearly, the targets of these edges are non-overlapped among the compute nodes. All these targets are broadcasted to the other compute nodes. The union of them generates the frontier set for next iteration.

Cost Analysis. In the graph partition and index construction phases, the communication cost and the computational complexity of TargetBFS are the same as those of SourceBFS. In the BFS phase, all the reachable vertices will be broadcasted to $N-1$ compute nodes once and each compute node owns $\frac{|V|}{N}$ vertices. Therefore, the communication cost is $\frac{N-1}{N}|V|$ and the computational complexity is $O(|V| + \frac{|E|}{N})$.

5. PARTITIONING, INDEXING AND BFS

5.1 Partitioning by statistical measures

In contrast to the existing methods which use predefined partition functions, our proposed algorithm, DPC (Degree-based Partitioning and Communication), determine its partition function based on the degree of the vertices. Let us consider that we have a graph which has already been partitioned by the partition function $H_{src}(e)$. For a given vertex v and its degree $d(v)$, the compute node sends $\frac{N-1}{N}d(v)$ edges to the other compute nodes when v is in the frontier set. However, if we repartition the emanating edges from v by their target vertices, the compute node broadcasts v to all the other compute nodes. The other compute nodes access the local subset of the emanating edges from v. If we assume that sending an edge and broadcasting a vertex have a same communication cost, the communication cost for broadcasting v is $N-1$.

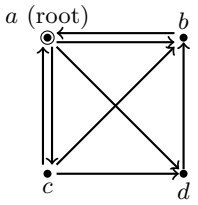
For a given vertex v, the communication cost by using $H_{tgt}(e)$ is less than the communication cost by using $H_{src}(e)$ when $d(v) > N$, and the performance gap is $\frac{N-1}{N}(d(v)-N)$ regarding the communication cost. In contrast, the computational cost of the whole network increases from $C_{vertex} + d(v) \times C_{edge}$ to $N \times C_{vertex} + d(v) \times C_{edge}$ where C_{vertex} is the cost of accessing a vertex and C_{edge} is the cost of operating an edge, respectively. Because all compute nodes have to access v by using $H_{tgt}(e)$, even if the compute node does not own any edge emanating from v. Based on the above observation, we define the partition function of DPC as follows.

DEFINITION 5. *Given an integer partitioning threshold σ, the partition function of DPC algorithm is that*

$$H_{degree}(e) \ = \begin{cases} e.src \ \textbf{mod} \ N \ \textbf{if} \ d(e.src) < \sigma \\ e.tgt \ \textbf{mod} \ N \ \textbf{if} \ d(e.src) \geq \sigma \end{cases}$$

We will discuss how to determine the value of the partitioning threshold σ in Subsection 6.1.

The partition algorithm of DPC consists of two substeps, counting and partitioning. In the counting step, each compute node counts the local degree for each vertex based on the local initial edge list and followed by a reduction operation which produces the global degree for each vertex. In the partitioning step, the algorithm is the same as that of SourceBFS except using $H_{degree}(e)$ instead of $H_{src}(e)$. The counting step takes $O(|V| + |E|)$ computational time and extra $|V|$ communication cost for the reduction.



(a) A sample graph

Owned edges	CN1	$a-\{b,c,d\}$ $b-\{a\}$		
	CN2	$c-\{a,b,d\}$ $d-\{b\}$		
Iteration	CN	Visited	Messages	Frontiers
1	CN1	$\{a\}$	Forward $\{c,d\}$ to CN2	$\{b,c,d\}$
	CN2	$\emptyset$		$\emptyset$
2	CN1	$\{a,b\}$		$\emptyset$
	CN2	$\{c,d\}$	Forward $\{a,b\}$ to CN1	$\emptyset$

(b) The communication schedule of SourceBFS

		By source vertex	By target vertex	
Owned edges	CN1	$b-\{a\}$	$a-\{b\}$ $c-\{a,b\}$	
	CN2	$d-\{b\}$	$a-\{c,d\}$ $c-\{d\}$	
Iteration	CN	Visited	Messages	Frontiers
1	CN1	$\{a\}$	Broadcast: $\{a\}$	$\{b\}$
	CN2	$\{a\}$		$\{c,d\}$
2	CN1	$\{a,b,c\}$		$\emptyset$
	CN2	$\{a,c,d\}$	Broadcast: $\{c\}$ Forward $\{b\}$ to CN1	$\emptyset$

(c) The communication schedule of DPC ($\sigma = 2$)

Figure 2: Communication schedules

For example, the graph is shown in Figure 2(a). There are four vertices and eight edges in this directed graph. The BFS is rooted at vertex a. Figure 2(b) illustrates the partitions

and the communication schedule of SourceBFS. There are two compute nodes CN1 and CN2. CN1 owns vertices a and b while CN2 owns vertices c and d. In the first iteration, CN1 sends the frontiers c and d to $CN2$, and marks its own vertex b as a frontier. In the second iteration, CN1 accesses b's edge list $\{a\}$, but a is not marked as a frontier since it has been accessed. CN2 accesses the edge lists of c and d, and then sends a and b to $CN1$ as frontiers. As a, b and d have been accessed, there is no frontier for the next iteration.

As described in Figure 2(c), the communication schedule of DPC is more complicated than SourceBFS. Assuming the partitioning threshold $\sigma = 2$, the edges emanating from a and c are partitioned by the target vertex, while the edges emanating from b and d are partitioned by the source vertex. As a result, CN1 owns the edges ab, ba, ca and cb, while CN2 owns the edges ac, ad, cd and db. There are two kinds of communication messages, (1) broadcasting the vertices which are partitioned by the target vertex, and (2) forwarding the frontiers of the vertices which are partitioned by the source vertex. In the first iteration, CN1 broadcasts the root a to CN2. Then, CN1 and CN2 parallelly access the partial edge lists of a and consider b, c and d as frontiers. In the second iteration, CN2 broadcasts the vertex c to CN1 since c is partitioned by the target vertex. As all the vertices in c's edge list have been accessed, they are not considered as the frontiers for the next iteration. In the meanwhile, CN2 accesses d's edge list and forwards the frontier b to CN1. As b has been accessed, there is no frontier for the next iteration. It is tricky that the forwarding messages do not have to wait for the broadcasting messages, because the partial edge lists only contain the vertices owned by the same compute node, and these vertices do not need to be forwarded. Therefore, the two kinds of communication messages, broadcasting and forwarding, do not block each other and can share a same message buffer. Compared with SourceBFS and TargetBFS, there is no additional delay for DPC in the communication.

5.2 Index structure and BFS algorithm

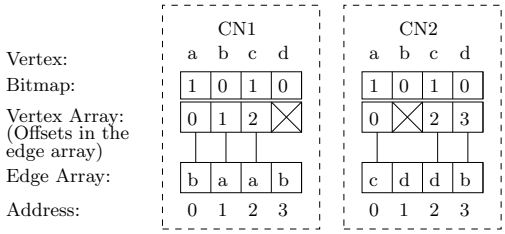


Figure 3: Index structure

The index of DPC consists of two components, a bitmap and a compressed sparse row (CSR). The bitmap indicates whether the edges emanating from the vertex are partitioned by the source vertex or the target vertex. The bitmap uses "1" to indicate the vertex whose degree is equal or greater than the threshold σ, and the edges emanating from this vertex are partitioned by the target vertex. In contrast, the vertex whose degree is less than the threshold σ is marked with "0" in the bitmap, and its emanating edges are partitioned by the source vertex. The CSR stores vertices and edges of the graph in two separate arrays. The edge array is sorted by the source vertices of the edges. It only contains the target vertices for the edges. The vertex array stores the offsets within the edge array which locates the address of the first edge emanating from each vertex. For example, Figure 3 shows the index of the graph in Figure 2(a) and the partition in Figure 2(c). In this partition, the bitmap $\{1, 0, 1, 0\}$ indicates that the edges emanating from the vertices a and c are partitioned by the target vertex, while the edges emanating from the vertices b and d are partitioned by the source vertex. The distributed CSR has much less overhead than many other graph formats such as distributed adjacency list. In the worst case, the space requirement is $O(|V| + \frac{|E|}{N})$.

In the index construction phase, we first generate the bitmap with size $|V|$ in $O(|V_G|)$ time complexity. Then we construct the vertex array and the edge array. The computational complexity of constructing the vertex array and the edge array is $O(|V| + \frac{|E|}{N})$. The overall computational complexity is $O(|V| + \frac{|E|}{N})$ since $|V_G| \leq |V|$ always holds.

The BFS algorithm is illustrated in Algorithm 4. Given a root r, the compute node which owns r inserts it into its frontier set F_0 for the first iteration in lines 1-3. For each iteration i, the algorithm first stores the frontiers with $B[v] = 1$ (i.e. $v \in V_G$) into a set SV. The other frontiers' edge sets are retrieved and stored in E_i in line 10. In line 12, a set RV is generated as the union of SV among all the compute nodes. The neighbors S_q of the frontiers are sent to the other compute nodes which own these vertices in lines 13-17. The neighbors S_{rank} which still belongs to the current compute node are directly moved to R_{rank} in line 18. Each compute node receives the messages R_q from other compute nodes and merges them together with the neighbors of the vertices in RV to generate F_{i+1} which will be the frontier set for the next iteration in lines 19-23. When the algorithm terminates, the function $Parent(v)$ stores the parent of the vertices in lines 24-26 .

	SourceBFS	TargetBFS	DPC
M_1	$\frac{N-1}{N^2}\|E\|$	$\frac{N-1}{N^2}\|E\|$	$\frac{\|V\|}{N} + \frac{N-1}{N^2}\|E\|$
C_1	$O(\frac{\|E\|}{N})$	$O(\frac{\|E\|}{N})$	$O(\frac{\|E\|}{N})$
C_2	$O(\frac{\|V\|+\|E\|}{N})$	$O(\|V\| + \frac{\|E\|}{N})$	$O(\|V\| + \frac{\|E\|}{N})$
M_3	$\frac{N-1}{N^2}\|E\|$	$\frac{N-1}{N}\|V\|$	$\frac{N-1}{N^2}\|E_L\| + \frac{N-1}{N}\|V_G\|$
C_3	$O(\frac{\|V\|+\|E\|}{N})$	$O(\|V\| + \frac{\|E\|}{N})$	$O(\frac{\|V_L\|+\|E_L\|}{N} + \|V_G\| + \frac{\|E_G\|}{N})$

Figure 4: The comparison of BFS algorithms

Cost Analysis. Given a graph with a vertex set V and an edge set E, σ is the predefined partitioning threshold. The vertex set V is divided into two subsets, $V_L = \{v | v \in V \wedge d(v) < \sigma\}$ and $V_G = \{v | v \in V \wedge d(v) \geq \sigma\}$. The edge set E is also divided into two subsets by the source vertex, $E_L = \{e | e.src \in V_L\}$ and $E_G = \{e | e.src \in V_G\}$. Clearly, $E = E_L \cup E_G$ and $E_L \cap E_G = \emptyset$. In the BFS phase, the communication costs of processing the edges in E_L and E_G are $\frac{N-1}{N^2}|E_L|$ and $\frac{N-1}{N}|V_G|$, respectively. The computational complexities of processing the edges in E_L and E_G are $\frac{|V_L|+|E_L|}{N}$ and $|V_G| + \frac{|E_G|}{N}$, respectively. SourceBFS and TargetBFS can be considered as two special cases of DPC when $V_G = \emptyset$ and $V_L = \emptyset$, respectively. The summary of the computational complexities $C_{\{1,2,3\}}$ and the communication costs $M_{\{1,3\}}$ in three phases is illustrated in Figure 4. If we choose $\sigma = N$, the communication costs of SourceBFS and TargetBFS can be considered as two upper bounds of that cost of DPC. These two upper bounds are achieved by $\forall v \in V, \exists d(v) \leq N$ and $\forall v \in V, \exists d(v) \geq N$, respectively.

Algorithm 4: **BFS**(B, E, r)

Input : B is the bitmap which stores V_G and V_L;
E is the local edge set;
r is a root;
Output: $Parent(v)$ stores the parent of each vertex, initialized as *null* for all vertices;

```
1  if rank = r mod size then
2  |  Parent(r) = r; F_0 = {r};
3  else
4  |  F_0 = ∅;
5  end if
6  for i = 0 to ∞ do
7  |  if F_i = ∅ for all CNs then TERMINATE.
8  |  for each v in F_i do
9  |  |  if B[v] = 1 then SV = SV ∪ {v};
10 |  |  else E_i = E_i ∪ {e|e ∈ E ∧ e.src = v};
11 |  end for
12 |  RV = ⋃_{q=0}^{size-1} SV on all CNs;
13 |  for each q in [0, size − 1] except rank do
14 |  |  S_q = {e|e ∈ E_i ∧ q = e.tgt mod size };
15 |  |  Send S_q to CN q;
16 |  |  Receive R_q from CN q;
17 |  end for
18 |  R_rank = S_rank;
19 |  R = ⋃_{q=0}^{size-1} R_q;
20 |  for each v in RV do
21 |  |  R = R ∪ {e|e ∈ E ∧ e.src = v};
22 |  end for
23 |  F_{i+1} = {e.tgt|e ∈ R};
24 |  for each e in R and e.tgt mod size = rank do
25 |  |  if Parent(e.tgt) = null then
   |  |  Parent(e.tgt) = e.src;
26 |  end for
27 end for
```

6. OPTIMIZATIONS

We further discuss the technique details and optimizations in this section. This includes the tuning technique of the parameter σ, the optimization for the buffer length and the discussion on network topologies.

6.1 Partitioning threshold

We say that σ is optimal when the BFS algorithm achieves the best overall performance. The optimal value of σ is affected by the latency and bandwidth of the communication network and the computing power of the compute nodes. If the communication cost dominates the overall cost, the optimal value of σ is equal to N which provides a minimum communication cost.

Unfortunately, the computational cost cannot be omitted in modern communication networks and therefore the optimal value of σ is usually greater than N. The optimal value of σ cannot be straightforwardly calculated by the latency and bandwidth of the network and the computing power of the compute nodes because many issues can affect the utilization of the network such as barriers, the cost of starting listeners and etc. Tuning the performance becomes a very challenging task. We propose a heuristic algorithm to solve this problem by tuning communication-to-computation(C/C) ratio. If we define the computational cost as the time spent by a compute node in computing and the communication cost as the time spent by a compute node in sending and receiving messages, the C/C ratio is the communication cost divided by computational cost. As we adopt the non-blocking communication, the compute nodes achieve optimal resource utilization when the computation and communication is balanced. When σ is greater than N, the communication cost can be reduced as a trade-off of additional computation by decreasing σ and vice verse. Assuming that we have known the optimal σ for a given graph size, we prove that σ is optimal for a different graph size if the random variables of degrees follow a same distribution.

The optimal σ is constant to graph size. We say an algorithm satisfying linear C/C ratio to graph size, if the speedup (or slowdown) of computation is linear to the speedup (or slowdown) of communication when increasing (or decreasing) the graph size. We prove the BFS phase of DPC satisfying linear C/C ratio for constant σ if the random variables of degrees follow a same distribution.

PROOF. Given a graph with a vertex set V and an edge set E, σ is the predefined partitioning threshold. The vertex set V is divided into two subsets, $V_L = \{v|v \in V \wedge d(v) < \sigma\}$ and $V_G = \{v|v \in V \wedge d(v) \geq \sigma\}$. The edge set E is also divided into two subsets by the source vertex, $E_L = \{e|e.src \in V_L\}$ and $E_G = \{e|e.src \in V_G\}$.

Suppose the number of compute nodes is N, then communication cost of each computing node is $Comm(N) = \frac{(N-1)}{N} \times \frac{|E_L|}{N} + (N-1)\frac{|V_G|}{N}$, since each compute node owns $\frac{|E_L|}{N}$ edges of E_L and $\frac{(N-1)}{N}$ of them are sent out, and owns $\frac{|V_G|}{N}$ vertices of V_G and each vertex is sent to $N-1$ computing nodes. The computational cost of each compute node is $Cmpt(N) = \frac{|V_L|+|E_L|}{N} + |V_G| + \frac{|E_G|}{N}$, since the computational cost of operating the vertices in V_L and the edges in E_L is as same as that of SourceBFS while the computational cost of operating the vertices in V_G and the edges in E_G is as same as that of TargetBFS. Because the random variables of degrees follow a same distribution, the expected values of $|V_L|$, $|E_L|$, $|V_G|$ and $|E_G|$ are linear to the graph size. The theorem immediate follows. □

Based on the above observations, the tuning method is straightforward. We first consider a small graph (e.g. a simpling graph or a low-scale problem) as the tuning graph. The random variable of the vertex degree in the tunning graph follows the same distribution as the original graph. We set the initial value of σ as N and increase σ step by step. The overall performance increase at the beginning and drops at a certain value. We select the value which achieves the peak performance as the value of σ.

6.2 Buffer length

In contrast to the existing approachs which use the message buffers with fixed length which is the upper bound on the message size in each iteration, we propose to use message buffers of flexible length with a maximum limit which is usually smaller than the message size in each iteration.

As the algorithm uses non-blocking point-to-point communication, our optimization is based on two observations: (1) scheduling the compute nodes to communicate in different time has better bandwidth utilization than scheduling the compute nodes to communicate at a same time; and (2) the message size cannot reach its upper bound in most itera-

tions and flexible short buffer length introduces low latency in such iterations.

Considering the edges in E_L as an example, the number of edges which are sent from a compute node to another in each iteration is at most $\frac{|V|}{N}$ if we do not send the edges with duplicated target vertex. When the buffer length is larger than this upper bound, the communication cannot start until the compute node iterate the all edges emanating from the current frontier set. The bandwidth in this period is wasted. Our solution is that we use a relative small message buffer length limit. Whenever the message size reaches this limit, we immediately send the message to its target compute node via non-blocking communication. As the graph is usually skewed, some of the compute nodes achieve this limit earlier than the others. The compute nodes start to utilize the bandwidth when first one achieve this limit, and the bandwidth will be utilized in the whole computing period. As we reuse the buffer in a same iteration, we use a bitmap to avoid sending edges with duplicated target vertex. The upper bound $\frac{|V|}{N}$ of total message size still applies to our algorithm. When the message size is still less the limit and the iteration is done for the frontier set, we send the message out as its actual size to achieve low latency.

6.3 Network topologies

We discuss the underlying network topologies. A poor choice of topology will reduce performance due to the bottlenecks or the long distances among compute nodes. In network topologies, the average path length is defined as the average distance between two compute nodes in the network over all pairs of distinct compute nodes. Short average path lengths decrease overall network utilization and reduce message latency. The diameter of a network is the longest path in the network between two compute nodes. The diameter is found by recording the shortest paths among all pairs of distinct compute nodes and taking the maximum of them. The diameter usually affects the performance of bottlenecks, since it considers distance between the two farthest compute nodes. The average path length and the diameter are typically related in general network topologies, since a large diameter generally implies a large average path lengths and vice versa.

The algorithms discussed in this paper, as general solutions, are based on the assumption that path lengths between any two distinct compute nodes are same. It implies that the diameter of a network is equal to the average path length. Ideally, this property can be achieved via a fully connected network. However, this topology is impractical for large-scale clusters, since it requires the large number of links, $\frac{n(n-1)}{2}$, as well as the large number of ports of each compute node, $n-1$. In practical, fat-tree and hypercube topologies can almost achieve this property and are widely used in data centers and supercomputing. We choose a fat-tree [26] topology in our configuration with considering scalability evaluation on a subset of compute nodes. Briefly, fat-tree is a tree-structured network topology where the compute nodes are located at the leaves. The internal nodes are switches. The capacities of the links increase as we go up the tree. Our configuration interconnects the compute nodes as a fat-tree with two layers of switch fabric($L1$ and $L2$). Each $L1$ switch is connected to 18 compute nodes via a 10 Gigabit Ethernet(10 GigE) cable and is connected to $L2$ via four 10 GigE cables. It is clear that the capacity between $L1$ and $L2$ is four times larger than that between $L1$ and compute nodes. With a fat-tree topology, the performance is nearly identical no matter which subset of the compute nodes the implementation used, while the performance varies depends on which compute nodes the implementation used for a particular run for a hypercube topology. The details of our cluster will be given in Section 7. In Figure 5 and Figure 6, we use $L1$ to indicate the communication between two nodes under a same $L1$ switch and $L2$ to indicate the communication between two compute nodes under two different $L1$ switches. The results, tested by using OMB [1], show that the latency gap between $L1$ and $L2$ communication is about 12.5% and its affect to uni-directional and bi-directional bandwidth is less than 5%.

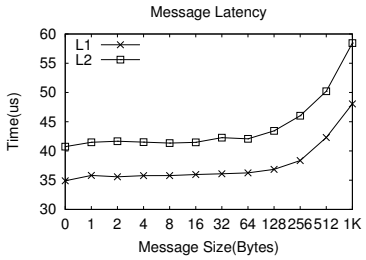


Figure 5: L1/L2 switch fabric latency

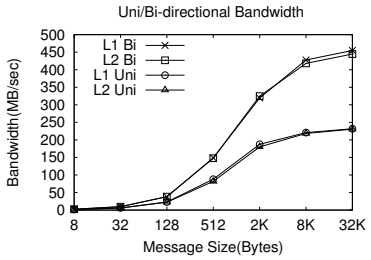


Figure 6: L1/L2 switch fabric bandwidth

7. EXPERIMENTS

In this section, we report a summary of the techniques developed and implemented for a comprehensive performance study.

7.1 Experimental settings

Node Hardware. The cluster is built by interconnecting 128 Dell PowerEdge R610 compute nodes. Each node is configured with two Intel Xeon E5530 2.4GHz CPUs and 24GB memory.

Interconnect. The compute nodes are interconnected as a fat-tree network with two layers of switch fabric(L1 and L2) by Summit X650-24x switches. Each L1 switch is connected to 18 compute nodes via a 10 Gigabit Ethernet(10 GigE) cable and is connected to L2 via four 10 GigE cables. The latency and bandwidth of our cluster has already been reported in Figure 5 and Figure 6 in Section 2.

Software. The nodes are running Debian GNU/Linux 5.0. All algorithms are implemented in C and parallelized using

the MPI library. The compiler and the library are GCC-4.3.2 and Open MPI 1.2.7rc2, respectively. All the algorithms are implemented following the specification of the Graph500 benchmark [2]. This benchmark includes a data generator and two kernels which are timed for performance metric. The data generator produces edge tuples containing the start vertex and the end vertex for each edge. The first kernel constructs an undirected graph in a format usable by the second kernel. The second kernel performs a BFS on the graph. We implemented the graph partition and index construction phases as the first kernel and the BFS phase as the second kernel.

Data Generator. The graph generator is a Kronecker generator similar to [6]. Two parameters are required to generate a graph. The parameter graph scale is used to determine the number of vertices, $|V| = 2^{\text{graph scale}}$. The other parameter edge factor(ef) is the number of edges, $|E| = ef \times |V|$. Therefore, the average degree is $2 \times ef$.

TEPS. The performance rate of Graph500 benchmark is defined as traversed edges per second (TEPS). We measure TEPS through the measures of the BFS phase as follows. Let *time* be the measured execution time for a run of the BFS phase. Let m be the number of the edges within the component traversed by the BFS. We define the normalized performance rate TEPS as:

$$TEPS = \frac{m}{time}$$

Execution Time. We define execution time as the total time of the BFS phase.

Message Size. Message size is defined as the number of bytes sent by each compute node for a BFS run.

Construction Time. We define construction time as the total time of the graph partition and index construction phases, since we always mix the graph partition and index construction phases together for communication efficiency as mentioned in Section 3.

For each index, we randomly sample 16 unique roots. The results always report the median TEPS, the median Execution Time and the median Message Size measured by 16 runs with unique roots.

7.2 Experimental results

We study the scalability and the efficiency of DPC by comparing it against SourceBFS and TargetBFS under the Graph500 benchmark with a variety of settings. As SourceBFS has already been implemented as the reference algorithm by the benchmark, we implement DPC and TargetBFS using the same framework as SourceBFS. The datasets are generated by the data generator with the edge factor $ef = 10$ except the graph density experiments. The buffer length limit in the BFS phase is 4KB for all the algorithms by default. The statistical information of the datasets is shown Figure 7. Top 1% degree refers to the degree of the top $1\%|V|$th vertex when we sort all the vertices in descending order of degree.

7.2.1 Tuning parameter σ

In Figure 8, we discuss the methods of tuning σ to the number of compute nodes(np) and graph size. We use graph scale 19 and 22 as the tuning datasets. Figure 8(a)(d) shows

SCALE	19	22	25
$\|V\|$	5.242×10^5	4.194×10^6	3.355×10^7
$\|E\|$	5.242×10^6	4.194×10^7	3.355×10^8
Average Degree	19.999	19.999	19.999
Max Degree	5.610×10^5	1.983×10^6	7.003×10^6
Top 1% Degree	212	220	235

Figure 7: The statistics of the graph datasets

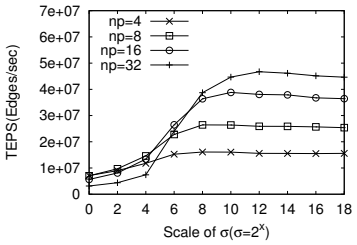
that the highest TEPS for $np = 4$, $np = 8$, $np = 16$ and $np = 32$ are achieved by $\sigma = 256$, $\sigma = 256$, $\sigma = 1024$ and $\sigma = 4096$ receptively, while Figure 8(b)(e) illustrates that the minimal message size is achieved by $\sigma = 64$, $\sigma = 64$, $\sigma = 64$ and $\sigma = 256$ receptively. The result confirms that the communication cost does not dominate the total cost. Both the communication cost and the computation cost affect the overall performance. The traditional optimization methods which simply considering BFS as a communication-intensive problem is not suitable for modern communication networks.

The result also shows that the number of compute nodes is approximately linear to the optimal σ for the best TEPS. The relation is $\sigma = 64 \times np$ in our communication network. By comparing Figure 8(a) and Figure 8(d), we verifies that the optimal σ for the best TEPS does not change when varying the graph size. Figure 8(c)(f) illustrates the percentage of vertices in $|V_G|$. When the highest TEPS is achieved, only $0.1 - 1\%$ high degree vertices belong to V_G which indicates the graph is skewed.

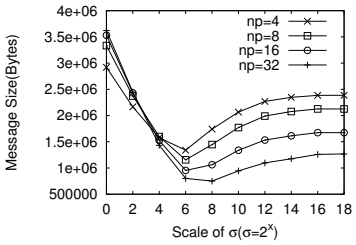
7.2.2 Performance comparison

As depicted in Figure 9(a)(d)(g), DPC is more efficient than SourceBFS and TargetBFS on graph scale 19, 22 and 25. Note that both the performance of DPC and SourceBFS increase with the number of compute nodes. On the contrary, the performance of TargetBFS even slightly decreases with the number of compute nodes since (1) the graph is sparse where $d(v) < np$ holds for most vertices; (2) these low degree frontiers of each compute node need to be broadcasted to all the other nodes; and (3) the more the compute nodes involved in the broadcasting the lower the performance will be. The execution time as shown in Figure 9(b)(e)(h) indicates the same results as that of TEPS. Both DPC and SourceBFS are scalable to the number of compute nodes and DPC always has lower execution time than SourceBFS.

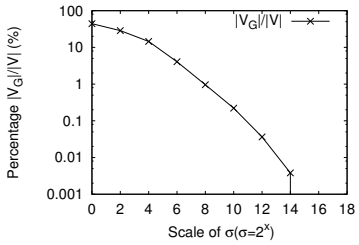
Figure 9(c)(f)(i) describes the construction time of the BFS algorithms. The construction time of DPC is nearly as same as TargetBFS. The result verifies the partition and index construction algorithms of DPC have very little cost overhead. To our surprise, the construction time of SourceBFS is about three times slower than the other two algorithms. We discover the reason by checking the underlying details of the Graph500 benchmark. Although the graph is stored in the distributed CSR, the benchmark uses linked edge pages to store the temporary emanating edges from each vertex when generating the distributed CSR. The linked edge pages become a very long list for SourceBFS, since all the emanating edges from a high degree vertex are stored on a same compute node. But DPC and TargetBFS partition the emanating edges from high degree vertices into different compute nodes and therefore avoid this problem. This construction cost can be reduced by optimizing the temporary data structure. However, we did not change the underlying temporary



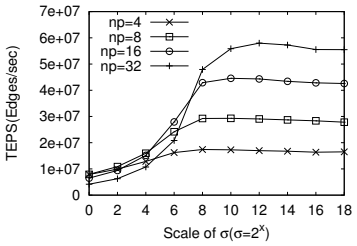
(a) TEPS, Graph Scale=19



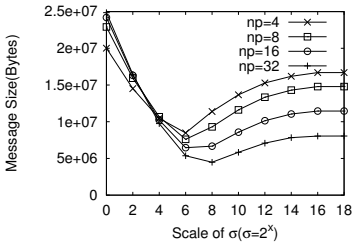
(b) Message Size, Graph Scale=19



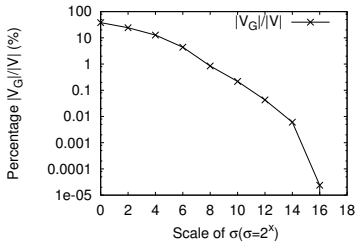
(c) Percentage $|V_G|/|V|$, Graph Scale=19



(d) TEPS, Graph Scale=22

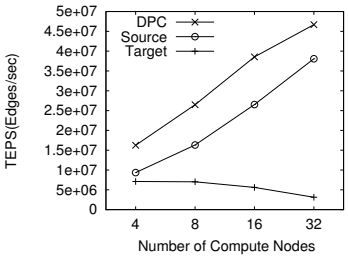


(e) Message Size, Graph Scale=22

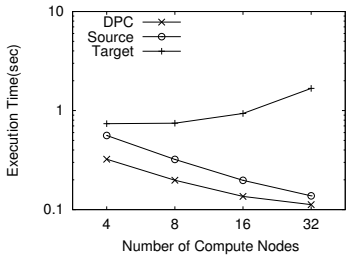


(f) Percentage $|V_G|/|V|$, Graph Scale=22

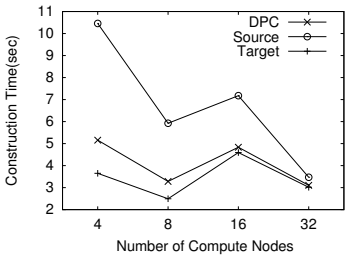
Figure 8: Tuning σ to the number of compute nodes and the graph size



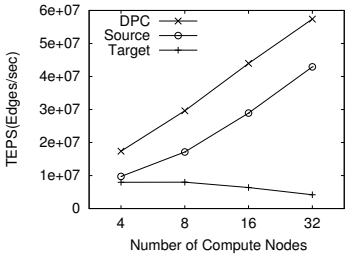
(a) TEPS, Graph Scale=19



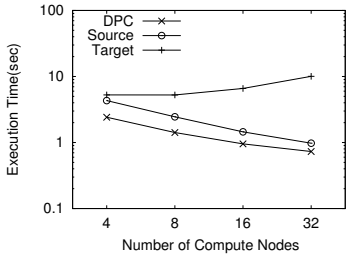
(b) Execution Time, Graph Scale=19



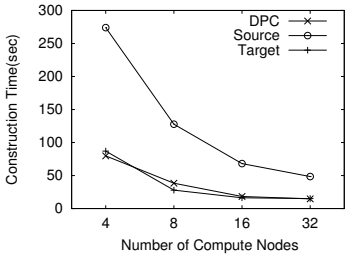
(c) Construction Time, Graph Scale=19



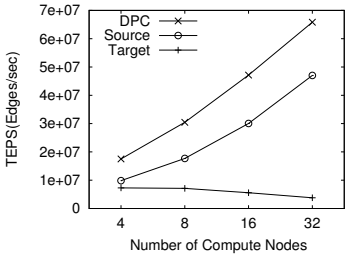
(d) TEPS, Graph Scale=22



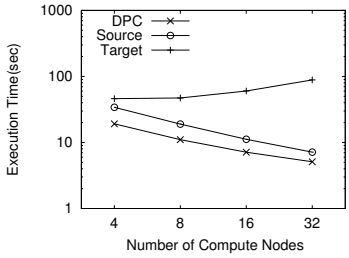
(e) Execution Time, Graph Scale=22



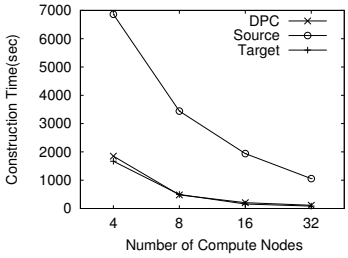
(f) Construction Time, Graph Scale=22



(g) TEPS, Graph Scale=25



(h) Execution Time, Graph Scale=25



(i) Construction Time, Graph Scale=25

Figure 9: The comparison of BFS algorithms by varying the number of compute nodes and the graph size

data structure with the consideration of keeping the originality of the benchmark and the comparison of the construction time between DPC and TargetBFS has already provided sufficient evidence for the low construction cost overhead of DPC. In Figure 9(c), we observe that the construction cost of graph scale 19 does not decrease linearly to the number of compute nodes. It is different from the results of graph scale 22 and 25 as depicted in Figure 9(f)(i). The main reason is that the initialization and controlling communication affects the construction time when the problem scale is too small. The similar observation also can be found in Figure 10(c) when the graph scale is less than 20.

7.2.3 Scalability

As depicted in Figure 10(a)(b), all three algorithms are scalable to graph size. Note that the execution time is linear to graph size, when the TEPS holds a constant value to graph size. Clearly, DPC and SourceBFS achieve the peak performance when the graph size is larger than 6.711×10^7 with $np = 32$. As shown in Figure 10(c), all three algorithms are scalable to graph size in the graph partition and index construction phases.

7.2.4 The effect of graph density

In order to study the effect of graph density, we generate five datasets with graph scale 22 by varying the edge factor ef between 8 and 128. The number of edges is equal to $3.355 \times 10^7 \times ef$ for each dataset. As depicted in Figure 11, we discover the optimal σ for different graph densities.

Figure 12(a) shows DPC outperforms the other two algorithms with all the settings. DPC is 116% and 271% times faster than SourceBFS for $ef = 8$ and $ef = 128$ respectively, while DPC is 1628% and 766% times faster than TargetBFS for $ef = 8$ and $ef = 128$ respectively. The result indicates that SourceBFS is suitable for sparse graphs while TargetBFS fits dense graph, but DPC always outperforms the existing algorithms and achieves more stable performance than the existing algorithms.

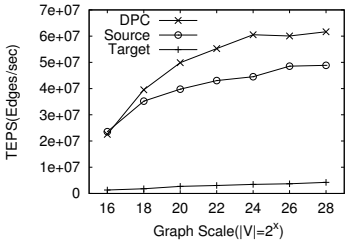
Figure 12(b) illustrates that TargetBFS shows nearly constant execution time to graph density. This result clarifies that (1)the communication cost of TargetBFS is influenced by $|V|$ and np while the graph density only affects its computation cost; (2) the communication cost dominate its overall cost; and (3) TargetBFS is unbalanced between computation and communication on sparse graphs. It is possible that TargetBFS may achieve high performance on extremely dense graphs. However, we cannot find any real graph in such extremely high density and the BFS on highly dense graphs is not in the scope of this paper. Figure 12(c) illustrates that all three algorithms are scalable to graph density since they are scalable to graph size and the graph density is linear to the number of edges if we fix the number of vertices in the graph.

7.2.5 Buffer length optimization

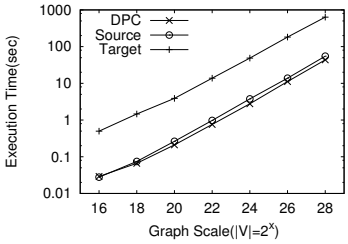
We evaluate the performance with different buffer length limits in Figure 13. As the protocol is changed from Eager to Rendezvous when the message size reaches a limit, we modify this limit to ensure all the experiments are conducted under Eager protocol. Each edge occupies 16 bytes in a message. In Figure 13(a), we use graph scale 16 as the dateset where the graph has 65536 vertices. Each compute node owns 4096 and 9192 vertices for $np = 16$ and $np = 8$, respectively. When the buffer length limit reaches 16KB(i.e. 1024 edges) and 32KB(i.e. 2048 edges), the performance of $np = 16$ and $np = 8$ drastically falls since the buffer length limit is larger than the number of edges sending in each iteration and the communication has to wait until the computation is done. $np = 32$ avoids this problem because we use flexible message size and the latency is much lower for small messages. $np = 4$ achieves stable performance since the size of most messages is above the buffer length limit. In Figure 13(b), we observe similar performance decreasing for $np = 16$ and $np = 32$ since the buffer length limit is too high and the compute nodes have to wait and communicate at a same time when the computation is done. In Figure 13(c), this performance fall only happens to $np = 32$ since the message size in each iteration increases with the graph scale and most of the messages are larger than the buffer length limit for np = 4-16. The results show that the small buffer length limit always achieves high and stable performance and we set the buffer length limit as 4KB for three algorithms in the other experiments.

8. RELATED WORK

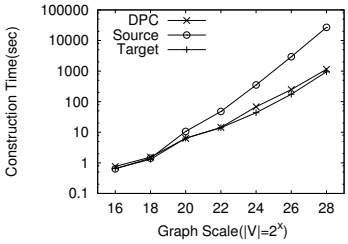
The most related work [2] has already been introduced as the SourceBFS in the baseline algorithms. In constrast to general algorithms, the authors of [35] focus on investigating algorithms specific to the torus network on BlueGene/L. In a recent study, Pregel [29] is developed as a scalable and fault-tolerant platform for large graph processing. Although our proposed techniques can be implemented on Pregel, the Graph500 platform [2] adopted by this paper is more suitable for the performance evaluation than Pregel platform. BFS algorithms on weighted graphs have also been well studied under the name single source shortest path(SSSP). Dijkstra's algorithm [13] is a well known approach to SSSP problem. There is a great effort from different areas to exploit pre-computed indexes to speed up the running time of SSSP discovery. The 2-hop [9, 10] indexes assign each vertex with two vertex label set and discover the shortest path by intersecting the two sets, while the combination of A* algorithm and landmark index is also studied by Goldberg [17]. In a recent work, Gao [16] studies to utilize could computing to efficiently compute SSSP on large graphs with neighborhood-privacy protected. In addition, approximate algorithms with different error bounds have also been exploited in [30]. The classical parallel solution to SSSP problem is a distributed form [20] of the Bellman-Ford algorithm [11]. This algorithm operates same as a simple BFS algorithm, except that it updates a vertex's distance to the root when a subsequent message of the vertex arrives with a shorter path, and then the algorithm resends the messages with the reduced new distance. Another obvious solution is called coordinated BFS algorithm. The subsequent work on this solution are focus on reducing the cost of synronization. Gallager [15] and Frederickson [14] processes a group of levels at a time, synchronizing back to the root only once for each group. In [14], the Bellman-Ford is used within a group and the coordinated algorithm is used between groups. This leads to a result of $O(|V|\sqrt{|E|})$ in both time and communication complexity. Awerbuch [4] perform the synchronization based on constructing a subgraph, which leads to a communication complexity of $O(|V|^2)$ and a time complexity of $O(|V| \log |V|)$. The communication complexity is subsequently reduced to $O(|V|^{1.6} + |E|)$ in [5].



(a) TEPS, np=32

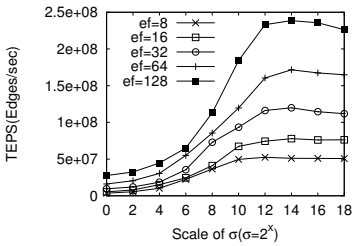


(b) Execution Time, np=32

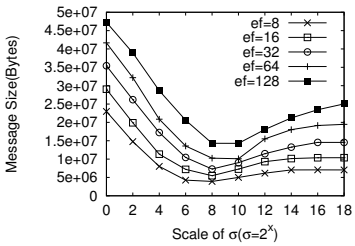


(c) Construction Time, np=32

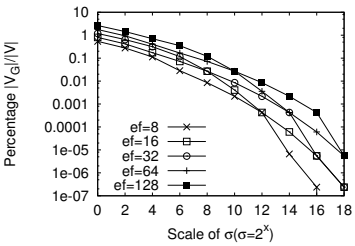
Figure 10: Scalability to the graph size



(a) TEPS, np=32, Graph Scale=22

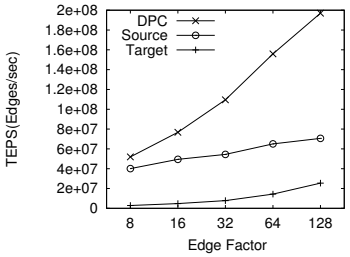


(b) Message Size, np=32, Graph Scale=22

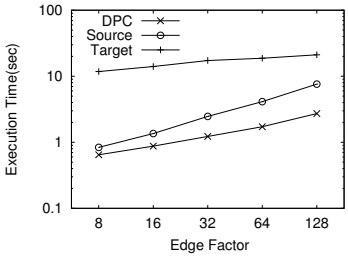


(c) Percentage $|V_G|/|V|$, np=32, Graph Scale=22

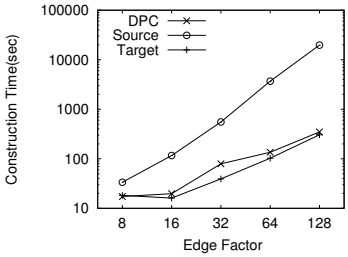
Figure 11: Tuning σ to the graph density



(a) TEPS, np=32, Graph Scale=22

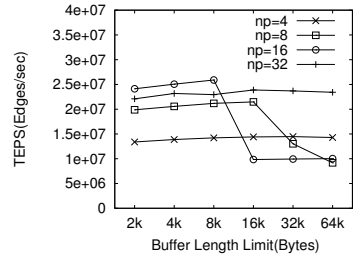


(b) Execution Time, np=32, Graph Scale=22

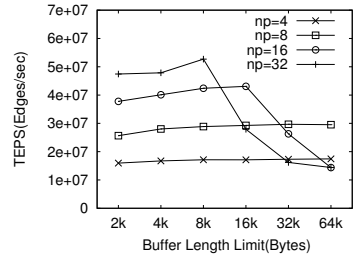


(c) Construction Time, np=32, Graph Scale=22

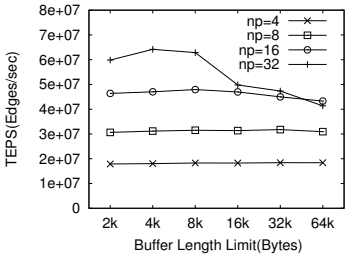
Figure 12: Comparison of BFS algorithms by varying the graph density



(a) TEPS, np=32, Graph Scale=16



(b) TEPS, np=32, Graph Scale=19



(c) TEPS, np=32, Graph Scale=22

Figure 13: Buffer length optimization

9. CONCLUSION

In this paper, we study the problem of BFS in distributed computing environments. Unlike previous methods which use uniform predefined graph partitioning methods, we propose a novel graph partitioning method based the degree of each vertex. Based on this new partitioning method, we propose a performance tuning technique which always achieves a balance between the computation and the communication. Furthermore, buffer length optimization methods are also proposed. Finally, we perform a comprehensive experimental study by comparing our algorithm against two state-of-the-art algorithms under the Graph500 benchmark with a variety of settings. The results show our algorithm outperforms the existing ones under all the settings. In the future work, we will study the problem of supporting other query types (e.g. shortest path, reachability and etc.) and platforms (e.g. Pregel [29] and etc.). The self-tuning algorithms with automatic network topology detection [28] will be another potential direction.

10. REFERENCES

[1] http://mvapich.cse.ohio-state.edu/benchmarks/.

[2] http://www.graph500.org/.

[3] C. C. Aggarwal and H. Wang. On dimensionality reduction of massive graphs for indexing and retrieval. In *ICDE*, pages 1091–1102, 2011.

[4] B. Awerbuch. An efficient network synchronization protocol. In *STOC*, pages 522–525, 1984.

[5] B. Awerbuch and R. G. Gallager. A new distributed algorithm to find breadth first search trees. *IEEE Transactions on Information Theory*, 33(3):315–322, 1987.

[6] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-mat: A recursive model for graph mining. In *SDM*, 2004.

[7] J. Cheng, Y. Ke, A. W.-C. Fu, and J. X. Yu. Fast graph query processing with a low-cost index. *VLDB J.*, 20(4):521–539, 2011.

[8] J. Cheng, Y. Ke, A. W.-C. Fu, J. X. Yu, and L. Zhu. Finding maximal cliques in massive networks by h*-graph. In *SIGMOD*, pages 447–458, 2010.

[9] J. Cheng and J. X. Yu. On-line exact shortest distance query processing. In *EDBT*, pages 481–492, 2009.

[10] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. *SIAM J. Comput.*, 32(5):1338–1355, 2003.

[11] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms (3. ed.)*. MIT Press, 2009.

[12] A. Crauser, K. Mehlhorn, U. Meyer, and P. Sanders. A parallelization of dijkstra's shortest path algorithm. In *MFCS*, pages 722–731, 1998.

[13] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1959.

[14] G. N. Frederickson. A single source shortest path algorithm for a planar distributed network. In *STACS*, pages 143–150, 1985.

[15] R. Gallager. *Distributed Minimum Hop Algorithms*. Defense Technical Information Center, 1982.

[16] J. Gao, J. X. Yu, R. Jin, J. Zhou, T. Wang, and D. Yang. Neighborhood-privacy protected shortest distance computing in cloud. In *SIGMOD*, pages 409–420, 2011.

[17] A. V. Goldberg and C. Harrelson. Computing the shortest path: search meets graph theory. In *SODA*, pages 156–165, 2005.

[18] A. Grama and V. Kumar. State of the art in parallel search techniques for discrete optimization problems. *IEEE Trans. Knowl. Data Eng.*, 11(1):28–35, 1999.

[19] Y. Han, V. Y. Pan, and J. H. Reif. Efficient parallel algorithms for computing all pair shortest paths in directed graphs. *Algorithmica*, 17(4):399–415, 1997.

[20] F. E. Heart, R. E. Kahn, S. M. Ornstein, W. R. Crowther, and D. C. Walden. The interface message processor for the arpa computer network. AFIPS, pages 551–567, 1970.

[21] N. Jin and W. Wang. Lts: Discriminative subgraph mining by learning from search history. In *ICDE*, pages 207–218, 2011.

[22] N. Jin, C. Young, and W. Wang. Gaia: graph classification using evolutionary computation. In *SIGMOD*, pages 879–890, 2010.

[23] R. Jin, L. Liu, B. Ding, and H. Wang. Distance-constraint reachability computation in uncertain graphs. *PVLDB*, 4(9):551–562, 2011.

[24] A. Khan, N. Li, X. Yan, Z. Guan, S. Chakraborty, and S. Tao. Neighborhood based fast graph search in large networks. In *SIGMOD*, pages 901–912, 2011.

[25] P. N. Klein and S. Subramanian. A randomized parallel algorithm for single-source shortest paths. *J. Algorithms*, 25(2):205–220, 1997.

[26] C. E. Leiserson. Fat-trees: Universal networks for hardware-efficient supercomputing. In *ICPP*, pages 393–402, 1985.

[27] J. Leskovec, J. M. Kleinberg, and C. Faloutsos. Graphs over time: densification laws, shrinking diameters and possible explanations. In *KDD*, pages 177–187, 2005.

[28] B. Lowekamp, D. R. O'Hallaron, and T. R. Gross. Topology discovery for large ethernet networks. In *SIGCOMM*, pages 237–248, 2001.

[29] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, pages 135–146, 2010.

[30] M. Potamias, F. Bonchi, C. Castillo, and A. Gionis. Fast shortest path distance estimation in large networks. In *CIKM*, pages 867–876, 2009.

[31] H. Shang, X. Lin, Y. Zhang, J. X. Yu, and W. Wang. Connected substructure similarity search. In *SIGMOD*, pages 903–914, 2010.

[32] H. Shang, Y. Zhang, X. Lin, and J. X. Yu. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *PVLDB*, 1(1):364–375, 2008.

[33] X. Yan, H. Cheng, J. Han, and P. S. Yu. Mining significant graph patterns by leap search. In *SIGMOD*, pages 433–444, 2008.

[34] X. Yan, P. S. Yu, and J. Han. Substructure similarity search in graph databases. In *SIGMOD*, pages 766–777, 2005.

[35] A. Yoo, E. Chow, K. W. Henderson, W. M. III, B. Hendrickson, and Ü. V. Çatalyürek. A scalable distributed parallel breadth-first search algorithm on bluegene/l. In *SC*, page 25, 2005.