

Article

Distributed Single-Source Shortest Path with Only Local Relaxation

Jianing Tang, Shufeng Gong *, Yanfeng Zhang, Chong Fu  and Ge Yu 

School of Computer Science and Engineering, Northeastern University, Shenyang 110169, China; 2201863@stu.neu.edu.cn (J.T.); zhangyf@mail.neu.edu.cn (Y.Z.); fuchong@mail.neu.edu.cn (C.F.); yuge@mail.neu.edu.cn (G.Y.)

* Correspondence: gongsf@mail.neu.edu.cn

Abstract: Finding the shortest path from a source vertex to any other vertices on a graph (single-source shortest path, SSSP) is used in a wide range of applications. With the rapid expansion of graph data volume, graphs are too large to be stored and processed in a standalone machine. Therefore, performing SSSP distributively in the computer clusters becomes an inevitable way. We found that the performance of existing distributed SSSP algorithms is limited by the communication cost between workers, which is caused by global relaxation. To eliminate the expensive communication cost, we propose an efficient distributed SSSP algorithm LR-SSSP that replaces global relaxation with local relaxation. Furthermore, we propose two optimizations, i.e., lazy synchronization and forward relaxation, to reduce invalid synchronization and communication. Our results show that LR-SSSP can achieve up to 6–20 $\times$ speedup over the state-of-the-art Δ -stepping++ algorithm.

Keywords: single-source shortest path; local relaxation; distributed computation



Citation: Tang, J.; Gong, S.; Zhang, Y.; Fu, C.; Yu, G. Distributed Single-Source Shortest Path with Only Local Relaxation. *Electronics* **2024**, *13*, 2502. <https://doi.org/10.3390/electronics13132502>

Academic Editors: Sungyong Park and Gordon Euhyun Moon

Received: 4 June 2024

Revised: 21 June 2024

Accepted: 24 June 2024

Published: 26 June 2024



Copyright: © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

Single-source shortest path (SSSP) aims to find the shortest paths between a given source vertex s and all other vertices in the graph. As a fundamental problem, it has been widely used in many real applications, such as network routing [1], online car-hailing scheduling [2], approximation of betweenness measures [3], and program performance profiling [4]. To compute the shortest distance of each vertex v , it is necessary to obtain the shortest distance of its incoming neighbors and the length of edges from incoming neighbors to it. Then, we compare the current shortest distance of v and that of incoming neighbors plus the length of incoming edges. Finally, the shortest distance of each vertex is updated with the minimum value. This operation is called edge relaxation.

As a classical and important problem, SSSP has been studied for decades of years. Many efficient SSSP algorithms [5–9] have been proposed. However, these efficient SSSP algorithms are designed based on the standalone worker and try to reduce the number of edge relaxations by scheduling the order of edges to relax. For example, Dijkstra’s algorithm prefers to select the vertex with the smallest known distance and relax its outgoing edges to update the shortest paths to its outgoing neighbors. Then, the shortest distance of each vertex will be calculated in order from closest to farthest from the source vertex. They assume that the whole graph is hosted on only one computer, which makes it difficult to process large-scale graphs. With the rapid expansion of graph volume, graph data are becoming too large to be stored and processed in a standalone machine, and have to be stored in distributed cluster machines.

To perform SSSP on large-scale graphs, designing the distributed SSSP algorithm to reduce the computational burden is desired. Meyer et. al proposed a distributed SSSP algorithm, Δ -stepping [10]. Then Chakaravarthy et. al augmented Δ -stepping with several optimizations [11], e.g., hybrid push/pull edge relaxation, edge classification, and load balance; we call it Δ -stepping++. However, a distributed runtime system may suffer from

security risks [12,13] and communication overhead [14]. In this paper, we mainly focus on reducing the communication overhead of the distributed SSSP algorithm. We found that, in distributed SSSP algorithms, most of the runtime is spent on communication between workers rather than computation. When vertex v and its incoming neighbor u are not on the same computer, the edge relaxation on (u, v) requires communication between computers (global relaxation), i.e., the sum of u 's shortest distance and the length of edge (u, v) should be transformed from u 's host computer to v 's host computer.

Figure 1 shows the normalized breakdown runtime of Δ -stepping and Δ -stepping++, where the computation time is counted by running Δ -stepping and Δ -stepping++ on a single computer using four cores, the communication time is calculated by subtracting the runtime on a single computer using four cores from runtime on a distributed cluster using one core on each computer. We can see that more than 93% runtime of Δ -stepping is spent on communication. Even in Δ -stepping++, the communication time still takes up 85~94% of runtime. This is because Δ -stepping++ only focuses on improving computation effectiveness and aims to avoid unnecessary computations. Although some communication decreases with reduced unnecessary computation, it does not pay enough attention to reducing communication costs. The performance of both algorithms is greatly limited by a large amount of communication.

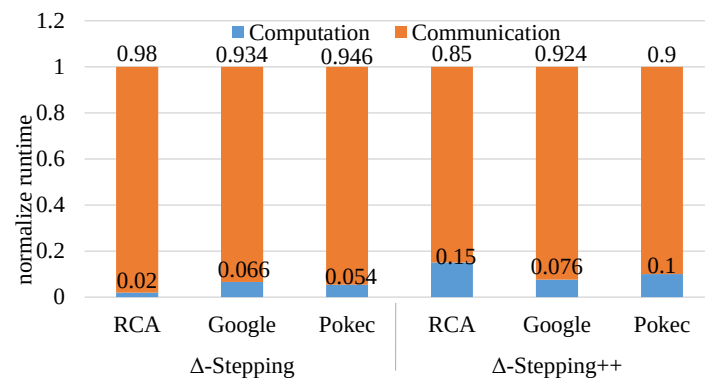


Figure 1. The normalized communication time and computation time of Δ -Stepping and Δ -Stepping++ on a 4-node cluster. RCA, Google, and Pokec are three graph datasets from [15].

Since the bottleneck of state-of-the-art distributed SSSP algorithms is communication between computers, reducing the amount of communication is the key to improving the performance of distributed SSSP. In distributed SSSP, the communication overhead is caused by relaxation on edges whose vertices are not on the same machine. Though we can reduce communication by employing the smart graph partition method, graph partitioning is an NP-hard problem [16]. It is difficult to find an effective partition that results in little communication, especially for power-law graphs.

Our Solution. In this paper, we propose an efficient distributed SSSP algorithm by replacing global relaxation with local relaxation, LR-SSSP. In LR-SSSP, if a vertex and its outgoing neighbor are not in the same computer, we locally create a replica of the neighbor, then the shortest distance of the neighbor's replica can be updated locally (local relaxation).

However, reducing communication by simply replacing global relaxation with local relaxation can cause other problems. (1) The shortest distances of different replicas are inconsistent, which results in incorrect results. (2) The inconsistency between replicas may lead to invalid computation, e.g., a replica has obtained the shortest distance, while other replicas are still updating.

We can solve the above two problems by synchronizing the shortest distance of all replicas. However, frequent synchronization operations will bring a lot of communication overhead. For this, we proposed lazy synchronization and forward relaxation to avoid problems caused by inconsistencies in the shortest distances of different replicas and a large number of communication problems caused by frequent synchronization operations.

In summary, this paper has the following contributions:

- We propose an efficient distributed SSSP algorithm, LR-SSSP, which reduces communication by only performing local edge relaxation.
- We further propose lazy synchronization and forward relaxation to prevent invalid synchronization and computation.
- We provide theoretical analysis to prove the effectiveness and correctness of our proposed LR-SSSP.
- We perform extensive experiments to evaluate LR-SSSP. It is shown that our LR-SSSP can achieve about $6\text{--}20\times$ faster speed than existing distributed SSSP algorithms.

The rest of this paper is organized as follows. In Section 2, we first introduce some preliminary knowledge about SSSP and distributed SSSP. Then we propose LR-SSSP in Section 3. The experimental evaluation is shown in Section 4.

2. Preliminaries

Before introducing our distributed SSSP algorithm, we first give some basic definitions and knowledge about SSSP and distributed SSSP. Then we discuss some related works.

2.1. Basic Definition

Let $G = \{V, E\}$ be a directed (the undirected graph can be treated as a directed graph by treating an undirected edge into two directed edges) and weighted graph with a set of vertices V and edges E . Each edge $e_{u,v} = \{u, v, \omega\} \in E$ is associated with a positive weight ω , which is referred to as the length of edge $e_{u,v}$, i.e., $L(e_{u,v}) = \omega$. For edge $e_{u,v}$, u is the incoming neighbor of v , and v is the outgoing neighbor of u . $OUT(v)$ and $IN(v)$ represent the outgoing neighbors and incoming neighbors of v .

Path and its length. A path from v_1 to v_k in G , $P_{v_1,v_k} = \{v_1, v_2, \dots, v_k\}$, is composed of a sequence of vertices $v_i \in V$ and their edges $e_{v_i,v_{i+1}} \in E$. The distance of the P_{v_1,v_k} is defined as the sum of the length of edges in path P_{v_1,v_k} , i.e., $L(P_{v_1,v_k}) = \sum_{i=0}^{i=k-1} L(e_{v_i,v_{i+1}})$.

Single-Source Shortest Path, SSSP. Given a source vertex $s \in V$, single-source shortest path computation is used to find the path $P_{s,v}^*$ from s to any other vertex $v \in V$ with minimal length $L(P_{s,v}^*)$, i.e., the length of any path from s to v , $P_{s,v}$ is larger than that of $P_{s,v}^*$, $L(P_{s,v}) > L(P_{s,v}^*)$.

Edge relaxation. The solution to solve the SSSP problem is edge relaxation. Given edge $e_{u,v} = \{u, v, \omega\}$, the edge relaxation is defined as follows:

$$L(P_{s,v}) = \min\{L(P_{s,v}), L(P_{s,u}) + L(e_{u,v})\}$$

Initially, the distance of the shortest path from s to any other vertex v is $L(P_{s,v}) = +\infty$, and the $L(P_{s,s}) = 0$. Then we relax the edges in E iteratively. If the distance of all vertices no longer changes, i.e., the vertices are converged, the shortest distance of all vertices reaches the minimum value. When relaxing edge $e_{u,v}$, we record the predecessor of v . According to the predecessor of each vertex, it is easy to reconstruct the shortest path from s to v .

Distributed SSSP. In general, the real graphs are too large to store in a standalone machine. Thus, the big graph is partitioned and stored on a cluster of machines. Let $G_i = \{V_i, E_i\}$ be a partition of the big graph G , where V_i is the set of vertices in G_i and E_i is the set of edges whose source vertices in V_i . For the given source vertex $s \in V$, we aim to find the shortest path $P_{s,v}$, where s and v may be on different machines.

2.2. Related Work

As a fundamental problem, considerable attention has been devoted to solving SSSP in sequential memory-/disk-based as well as distributed settings. In this paper, we mainly focus on improving the performance of the distributed SSSP algorithm.

Memory-Based. The most famous sequential memory-based SSSP algorithm is Dijkstra's algorithm [7]. It can be performed in $O(n \log n + m)$ time, where n is the number of vertices

and m is the number of edges, by employing the priority queue implemented with heap or tree. In the beginning, all the vertices are set to nonconvergent and push source vertex s into the priority queue. Then Dijkstra's algorithm processes vertices iteratively. In each iteration, Dijkstra's algorithm selects the nonconvergent vertex v with minimum distance (the first element in the priority queue) and sets it as a convergent vertex. Then v 's outgoing edges are relaxed and its outgoing neighbors' distances are updated. Meanwhile, v 's outgoing neighbors will be inserted into the priority queue if they are not in the queue. Iteration is repeated until all vertices are set as convergent. Since Dijkstra's algorithm requires loading the whole graph into memory, Dijkstra's algorithm is poorly suited for large graphs. There are also some more efficient memory-based SSSP algorithms [17–21] that are accelerated by building indexes in advance. However, these algorithms require the entire graph when building and using indexes, so it is difficult to apply them in distributed SSSP.

Disk-Based. In order to process large graphs, some disk-based SSSP algorithms are proposed [8,9,22,23]. Disk-based SSSP algorithms focus on designing an effective index to reduce the inefficient edge relaxation or disk I/O. Ref. [22] was the first work to design a disk-based Dijkstra's algorithm. They tried to reduce disk I/O by replacing the priority queue with the tournament tree. Ref. [23] proposed an efficient index, VC-Index, which can be treated as multiple reduced versions of the graph. Each reduced graph contains some important vertices and the distances between them. Then the SSSP can be accelerated based on precomputed distances. Ref. [8] introduced a tunable hash index to reduce the scale of wasteful data loaded from the disk. Then they proposed a new iterative mechanism and designed an Across-step Message Pruning (ASMP) policy to deal with the disk I/O. Ref. [9] presented HoD index. HoD augments the input graph with a set of auxiliary edges and exploits them to reduce disk I/O.

Distributed. The traditional Bellman–Ford algorithm can be executed on a distributed cluster. Different from Dijkstra's algorithm, in each iteration, the Bellman–Ford algorithm selects all the vertices whose distances are changed in the previous iteration to relax their outgoing edges and update their outgoing neighbors. The computation and communication costs are huge due to a large number of invalid relaxations. Preferring selecting the vertices with shorter distances and relaxing their outgoing edges can effectively reduce ineffective relaxation [7]. For example, Dijkstra's algorithm uses a priority queue to select the vertex with the shortest distance and relaxes its outgoing edges.

Δ -Stepping [10] uses a distributed bucket array as the distributed priority queue to schedule edge relaxations, and the length of each bucket is Δ . Each computer m_i maintains an array of buckets, shown in Figure 2, and the vertex v in m_i is assigned to $\lfloor \frac{L(P_s, v)}{\Delta} \rfloor$ -th bucket. Then, Δ -Stepping relaxes edges and updates the shortest distance of vertices distributively on same-level buckets. Until all the shortest distance of vertices in current buckets on all workers are not changed, Δ -Stepping will not step to the next higher-level buckets. Δ -Stepping++ [11] tries to improve the Δ -Stepping with some optimizations, e.g., hybrid push/pull edge relaxation, edge classification, and load balance. However, both of them suffer from high communication overhead.

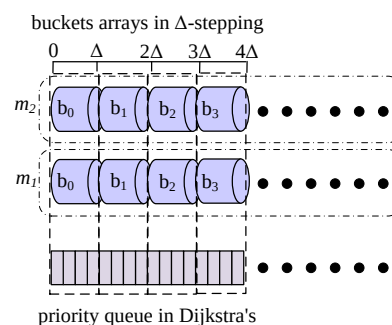


Figure 2. Distributed buckets array.

3. LR-SSSP

As we discussed in Section 1, the bottleneck of the existing distributed SSSP algorithm is network communication. In this section, we propose a simple but efficient distributed SSSP algorithm, LR-SSSP. Similar to Δ -Stepping, LR-SSSP also uses the distributed buckets as a global priority queue. Different from Δ -Stepping, LR-SSSP uses local edge relaxation to replace global edge to reduce communication and employs two optimizations, lazy relaxation and forward relaxation, to prevent invalid communication and computation.

3.1. From Global Relaxation to Local Relaxation

Global and Local relaxation. In distributed SSSP, if two vertices of an edge are on different computers, then the shortest distance information will be sent through the network. We call this edge relaxation global relaxation, e.g., relaxation on edge $e_{u,v}$ in Figure 3. If the edges' two vertices are in the same computer, the relaxation on them is called local relaxation, e.g., the relaxation on edge $e_{t,v}$ in Figure 3.

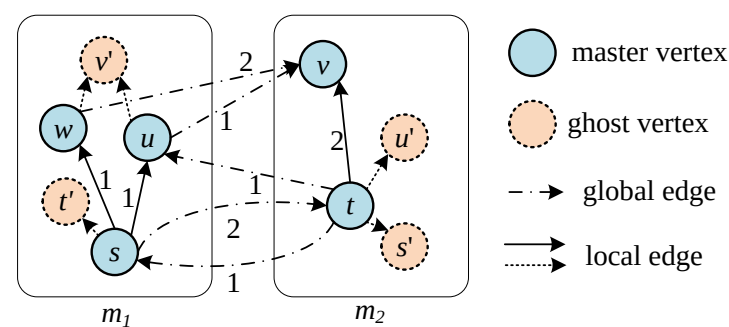


Figure 3. Global relaxation and local relaxation; the edge relaxation on the global edge is global relaxation while on the local edge it is local relaxation. The number on the edge represents the length or weight of the edge.

An intuitive solution to prevent communication caused by these global edge relaxations is to replace global edge relaxations with local edge relaxations, which can be achieved by creating ghost vertices. The ghost vertex is a replication of the master on the remote computer and does not have a follower–leader relationship. If two vertices u, v of the edge $e_{u,v}$ are in different computers, we create a ghost of the v, v' , on the host of the u and connect u and v' with a new edge whose weight is the same as the edge $e_{u,v}$. Then, we relax the edge $e_{u,v'}$ and update the shortest distance of v' on the host computer of u , which is performed locally. In this way, the global edge relaxations are replaced by local edge relaxations.

Example 1. As shown in Figure 3, the relaxation on edge $e_{u,v}$ is global relaxation since u and v are in different computers. We create a ghost vertex of v, v' , on computer m_1 , and connect u and v' with $e_{u,v'}$, where $L(e_{u,v'}) = L(e_{u,v})$. Then we relax edge $e_{u,v'}$ on m_1 , $L(P_{s,v'}) = \min\{L(P_{s,v}), L(P_{s,u}) + L(e_{u,v'})\}$.

However, because of the creation of ghost vertices, we obtain the shortest distance for each ghost vertex and master vertex, which may cause incorrect results. We have to synchronize the shortest values of the ghost vertices and master vertex. Though we prevent the communication cost caused by global edge relaxation by replacing the global edge relaxation with local relaxation, the synchronization of ghost and master vertices will lead to additional communication costs. We can see that the communication of vertices synchronization is less than that of global relaxation based on the following theorem. Note that, in LR-SSSP, the synchronization operation is to send the shortest values of ghost vertices to the master vertex. There are no synchronizations between ghost vertices.

Theorem 1. Given g in-neighbors of vertex v hosted on $k - 1$ machines, in a round of iteration, there are g global relaxations on g edges with the same destination v or g local relaxations and $k - 1$ synchronizations. The communication of global relaxations is C_g , and the communication of synchronization after local relaxations is C_l . Then we have

$$C_g \geq C_l \quad (1)$$

Proof. The communication of a global relaxation and a vertex synchronization is the same, c , because they send the same size message that contains a vertex id and a distance value through a network. Then the communication of g global relaxations is $C_g = g \cdot c$. The communication of synchronization after g local relaxations on $k - 1$ machines is $C_l = (k - 1) \cdot c$. Since the in-neighbors of v located on $k - 1$ machines, $k - 1 \leq g$. Finally, we have $C_g \geq C_l$. $\square$

As shown in Figure 3, after relaxing $e_{w,v'}$ and $e_{u,v'}$ on m_1 , we synchronization the v' and v . There is only one message passing through the network, while the global relaxations on $e_{w,v}$ and $e_{u,v}$ require two messages to pass through the network. Thus, the local edge relaxation results in less communication than global relaxation.

Furthermore, creating ghost vertices not only reduces the communication cost by replacing global relaxation with local edge relaxation but also reduces the number of relaxations. For example, in Figure 4, when we relax edge $e_{c,d'}$, we find that the $L(P_{s,d'}) < L(P_{s,c})$, then it is unnecessary to perform local edge relaxation on $e_{c,d'}$, because according to the triangle inequality, we have $L(P_{s,d'}) < L(P_{s,c}) + L(e_{c,d'})$.

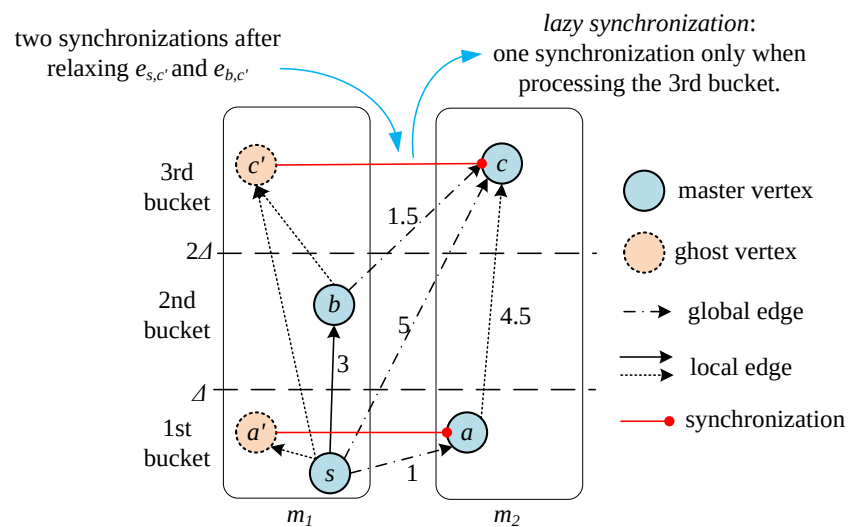


Figure 4. An example of Lazy Synchronization. Without lazy synchronization, there are two synchronizations between c and c' after relaxing $e_{s,c'}$ and $e_{b,c'}$. With lazy synchronization, there is one synchronization only when processing the 3rd bucket. The number on the edge represents the length or weight of the edge.

3.2. Lazy Synchronization

Compared with global edge relaxation, local edge relaxation helps reduce communication. To obtain the correct results of SSSP, it is required to synchronize the ghosts and the master. If the shortest distances of two ghosts of a vertex are inconsistent, the shortest distances of the two ghosts are different. If the distance of one of the ghosts is not the real shortest, the distance of the successor of this ghost is also not the shortest, which results in incorrect SSSP results. However, there is still a large number of synchronizations when performing distributed SSSP for obtaining the correct and exact result. And the synchronization between ghosts and the master requires communication costs. However, we find that most of the synchronizations are invalid during the iterative computation.

Example 2. As shown in Figure 4, we may synchronize the c' and c two times. The first synchronization is caused by local relaxation on edge $e_{s,c'}$ when processing the vertices located in the 1st bucket. The second synchronization is caused by local relaxation on edge $e_{b,c'}$ when processing the vertices located in the 2nd bucket. However, the synchronization caused by local relaxation on edge $e_{s,c'}$ is invalid, because after relaxing edge $e_{b,c'}$, we will obtain a shorter path from s to c' .

As illustrated in Example 2, some synchronization is unnecessary as the shortest distance values of ghost vertices will become smaller in the later relaxation, so postponing the synchronization may avoid some early invalid synchronizations. However, it is difficult to determine which synchronization is valid and when to perform synchronization. Too-early synchronization is too feeble to prevent invalid synchronization, while too-late synchronization may result in an incorrect result. In this subsection, we propose a lazy synchronization method to prevent most of the invalid synchronization by exploiting the local relaxation and the distributed buckets array.

In LR-SSSP, we also use distributed buckets array to help schedule edge relaxation, which can reduce a lot of invalid relaxation. We first relax the edges whose source vertices are in lower-level buckets, as the vertices in lower-level buckets have a shorter path from s to them than vertices in higher-level buckets, because preferring processing vertices close to the source vertex s always avoids some invalid relaxations [7,10]. When performing local relaxation on the outgoing edge of the vertex in a lower-level bucket, if the updated ghost falls into a higher-level bucket, then it is unnecessary to synchronize the updated ghost with its master, because this synchronization may be invalid if the ghost is updated again later. Based on this intuition, we propose lazy synchronization.

In lazy synchronization, when processing the t -th buckets, only the vertices whose ghosts' shortest distances are in $((t-1) \cdot \Delta, t \cdot \Delta]$ are synchronized. After synchronization, we perform the local relaxation on edges whose source vertices' current shortest distances are in $((t-1) \cdot \Delta, t \cdot \Delta]$. Then, there may be new ghosts created and their shortest distances are in $((t-1) \cdot \Delta, t \cdot \Delta]$ or some ghosts' / masters' shortest distance becomes smaller than $t \cdot \Delta$; then we reperform synchronization between these ghosts and masters until there is no ghosts / masters whose distance are changed. For the ghosts whose shortest distances are larger than $t \cdot \Delta$, we delay their synchronization until we step to the corresponding buckets.

As shown in Figure 4, where $\Delta = 2$, when processing the 1st buckets, we not only relax $e_{s,b}$, but also create a' and c' for relaxing $e_{s,a}$ and $e_{s,c}$ in computer m_1 , since the $L(P_{s,a'}) < \Delta$, a' and a are synchronized immediately. While $L(P_{s,a'}) > \Delta$, we delay the synchronization between c' and c . In m_2 , after a is updated with synchronization, we relax edge $e_{a,c}$ and update the shortest distance of c . Then, if there are no ghost vertices created or changed in the 1st buckets, we step to the 2nd buckets. We relax $e_{b,c'}$ and update c' . Next, we step to the 3rd buckets. We synchronize c and c' and obtain the correct shortest path of c . It can be seen that the synchronization between c and c' is delayed and one synchronization is reduced.

With the lazy synchronization strategy, we do not synchronize ghosts until the shortest distance of the ghosts will be obtained. In this way, the useless synchronization will be filtered. Furthermore, this lazy synchronization method has no side effect on SSSP's result according to the following theorem, i.e., the SSSP result still is exact and correct.

Theorem 2. Given the source vertex s and a vertex v in G , the shortest distance from s to v is $L(P_{s,v}^*)$, in which u is the predecessor of v on path $P_{s,v}^*$. With the lazy synchronization, we still can obtain the shortest path of u , i.e., $L(P_{s,u}^*) = L(P_{s,v}^*) + L(e_{v,u})$.

Proof. If u and v are in the same computer, after obtaining the shortest distance of u from s , $L^*(P_{s,v})$, we can obtain the exact shortest distance of u by relaxing edge $e_{v,u}$, $L^*(P_{s,u}) = L^*(P_{s,v}) + L(e_{v,u})$ since the v is the predecessor of u in the shortest distance path from s to u .

If u and v are in different machines, after obtaining the shortest distance of u from s , $L^*(P_{s,v})$, we will create the ghost of u , u' , and relax edge $e_{v,u'}$ on v 's host machine. Then we have $L(P_{s,u'}) = L(P_{s,v}) + L(e_{v,u'})$. There are two cases, (i) u' and v are in the same level bucket, and (ii) u' and v are in different level bucket. For the first case, we will synchronize u' and u immediately and obtain $L(P_{s,u}^*) = L(P_{s,u'})$. For the second case, we will synchronize u' and u when LR-SSSP steps to the buckets where u' are located and obtain $L(P_{s,u}^*) = L(P_{s,u'}) = L(P_{s,v}^*) + L(e_{v,u})$. $\square$

According to Theorem 2, we can see that if the shortest path from s to v is obtained, then we can correctly obtain the shortest path from s to the vertices whose predecessor is v with lazy synchronization. Since the predecessor of all vertices can be traced back to vertex s , and the shortest distance of source vertex s has been obtained, i.e., $L^*(P_{s,s}) = 0$, then we can obtain the shortest distance from s to all reached vertices.

3.3. Forward Relaxation

Although we have prevented most of the invalid synchronizations with the help of lazy synchronization, there may be some invalid local relaxations due to the lack of global information. These invalid local relaxations may further cause invalid synchronizations.

Example 3. As shown in Figure 5, we set $\Delta = 1$. When processing the second bucket in computer m_2 , we relax $e_{t,u}$ by creating the ghost of u , u' , and relaxing $e_{t,u'}$. But we have the shortest path from s to u when processing the first buckets in m_1 ; thus, this local relaxation is invalid. Furthermore, when processing the third bucket, the synchronization between u' and u will be triggered, which is an invalid synchronization.

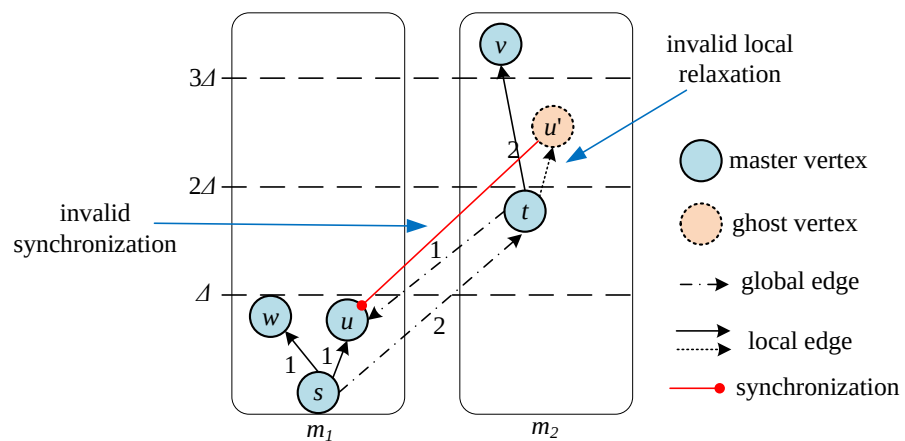


Figure 5. Invalid relaxations and invalid synchronizations. The number on the edge represents the length or weight of the edge.

This invalid local edge relaxation and synchronization are caused by the lack of global information about which vertices have obtained the shortest path. The higher the level of bucket we process, the more invalid relaxation there will be. It is noted that invalid relaxations are not caused by local relaxation. Even in global relaxation, there also are invalid global edge relaxations due to the lack of global information. Ref. [11] used a pull-and-push model to prevent invalid global relaxations. However, the pull-and-push model is not suitable for local edge relaxations. Thus, based on the following theorem, we propose a simple but efficient pruning method to prevent invalid local edge relaxation and synchronization.

Theorem 3. Given source s , vertex v , and its ghosts $v'_1, \dots, v'_k$, when processing t -th buckets, if $(t-1) \cdot \Delta < \min\{L(P_{s,v}), L(P_{s,v'_1}), \dots, L(P_{s,v'_k})\} \leq t \cdot \Delta$, then we have

$$(t-1) \cdot \Delta < L(P_{s,v}^*) \leq t \cdot \Delta \quad (2)$$

Proof. If $\min\{L(P_{s,v}), L(P_{s,v'_1}), \dots, L(P_{s,v'_k})\} \leq t \cdot \Delta$, according to the definition of SSSP, the shortest distance from s to v must be smaller than $t\Delta$, i.e., $L^*(P_{s,v}) \leq t \cdot \Delta$.

To prove $(t-1) \cdot \Delta < L(P_{s,v}^*)$, we first assume that $L(P_{s,v}^*) \leq (t-1) \cdot \Delta$. We set $(t'-1) \cdot \Delta < L^*(P_{s,u}) \leq t' \cdot \Delta$ and the predecessor of v on the shortest path from s to v is u , where $t' \leq t-1$. Since u is the predecessor of v , $L(P_{s,u}^*) \leq t' \cdot \Delta$. When processing t' -th buckets, we have relaxed the edge $e_{u,v}$ or $e_{u,v'}$. After that, we have had $L(P_{s,v}) \leq t' \cdot \Delta$ or $L(P_{s,v'}) \leq t' \cdot \Delta$ before processing the t -th buckets, which contradicts the condition $(t-1)\Delta < \min\{L(P_{s,v}), L(P_{s,v'_1}), \dots, L(P_{s,v'_k})\}$. Thus, $(t-1)\Delta < L(P_{s,v}^*)$. $\square$

According to Theorem 3, after processing t -th buckets, we can make sure that the vertices whose shortest distances are in $((t-1)\Delta, t\Delta]$, i.e., we have obtained the shortest path from source to these vertices. Thus, if there is an edge from the vertex in higher-level buckets to the vertex in lower-level buckets, then the relaxation on this edge (referred to as backward relaxation) is invalid. We can prevent invalid local edge relaxation and synchronization by broadcasting the *id* of vertices that have obtained the shortest path to other machines to avoid backward relaxation.

As shown in Figure 3, after processing the first buckets, we broadcast the *id* of u to m_2 , then when performing local relaxation on $e_{t,u}$, we find that u has obtained the shortest path, then the local edge relaxation and synchronization will be avoided.

Since we only broadcast the vertices' *ids* that become converged in t -th buckets, the additional communication costs are insignificant but reduce a large amount of invalid relaxation and synchronization.

3.4. Algorithm Detail

Based on the above discussion and the proposed local edge relaxation, lazy synchronization, and forward relaxation strategy, we propose the distributed SSSP algorithm, LR-SSSP.

Algorithm 1 shows the details of our LR-SSSP on computer m_i . In LR-SSSP, we also use a distributed bucket array as a priority queue to manage vertices according to their current shortest distance, and each computer m_i contains an array of buckets $B_i[]$.

Initially, we set the shortest distance of each vertex as positive infinity and set all the buckets to empty (line 1–4). To prevent backward relaxation, we use a set S to store the vertices that have obtained the shortest path (line 5) and synchronize it after processing each bucket (line 23–25). For the source vertex s , we set the current shortest distance of s as 0 and insert it to the 0th bucket (line 6–8) on the host of s . Then we select vertices in distributed bucket arrays from bottom to top to relax their outgoing edges until all vertices have obtained the shortest path, i.e., the buckets are empty (line 9–25).

The lazy synchronization strategy requires that before processing each bucket, we first synchronize the ghost vertices with their master vertices (line 13). It is noted that the synchronization of ghost vertices and master vertices is a one-way transmission, i.e., the ghosts send their distance and predecessor to their masters (see Algorithm 2), because it does not matter whether the ghost vertices have the correct result. Then, we relax the outgoing edges of each vertex in the bucket with local relaxation (line 17–20). According to Theorem 3, each vertex in the current bucket will obtain the shortest path. Thus, we record them with $Sset$ (line 11 and 14) and clear bucket (line 16). When relaxing edges, there may be new ghosts created or vertices whose shortest paths are changed and inserted into current buckets (line 16 in Algorithm 3). Until the current buckets of all computers are empty, LR-SSSP steps to the next bucket.

Algorithm 1: LR-SSSP (M_i)

Input: Graph partition $G_i = (V_i, E_i)$, source vertex s , Δ value
Output: Shortest distance of each vertex $v.l$ and its predecessor $v.p$

```

1 for each  $v \in V_i$  do
2    $v.l = +\infty$ ; //  $v.l$  is the shortest distance from  $s$  to  $v$ 
3    $v.p = \text{null}$ ; //  $v.p$  record the predecessor of  $v$ 
4  $B_i[] = \emptyset$ ; // local bucket array
5  $S = \emptyset$ ; // vertices set that have obtained shortest path
6 if  $s \in V_i$  then
7    $s.l = 0$ ;
8    $B_i[0].\text{insert}(s)$ ;
9  $j = 0$ ; // the index of  $B_i[]$ 
10 while all workers'  $B_i$  are not empty do
11    $Sset = \emptyset$  // vertices set will be converged
12   while all workers'  $B_i[j]$  is not empty do
13     LazySync ( $B_i[j]$ ); // synchronize ghosts and masters
14      $Sset = Sset \cup B_i[j]$ ;
15      $Rset = B_i[j]$ ; //  $Rset$  is the set of vertices whose outgoing edges
        will be relaxed
16      $B_i[j] = \emptyset$ ; // clear  $B_i[j]$ 
17     for each vertex  $v \in Rset$  do
18       for each  $u \in OUT(v)$  do
19         if  $u \notin S$  then
20           //  $u$  has not obtained shortest path
21           LocalRelax ( $v, u, e_{v,u}, B_i, \Delta$ );
22   LazySync ( $B_i[j]$ );
23    $j++$ ;
24    $S = S \cup Sset$ ;
25   broadcast  $Sset$  to other workers;
26   receive  $Sset$  from other workers and merge them with  $S$ ;

```

Algorithm 2: LazySync ($B_i[j]$)

```

1 Function LazySync ( $B_i[j]$ )
2   for each ghost  $v'$  in  $B_i[j]$  do
3     send  $v'$  to the host computer of its master;
4   for each received ghost  $v'$  of  $v$  do
5     if  $v.l > v'.l$  then
6        $v.l = v'.l, v.p = v'.p$ ;
7        $B_i[\lfloor v.l / \Delta \rfloor].\text{remove}(v)$ 
8        $B_i[j].\text{insert}(v)$ ;

```

Algorithm 3: LocalRelax ($v, u, e_{v,u}, B_i, V_i, \Delta$)

```

1 Function LocalRelax( $v, u, e_{v,u}, B_i, V_i, \Delta$ )
2   if  $u \notin V_i$  then
3     //  $v$  and  $u$  are not in the same computer
4     if  $u' \notin V_i$ ; then
5       // there is no ghost of  $u$  in  $V_i$ 
6       create  $u$ 's ghost  $u'$  and connect  $v$  and  $u'$  with edge  $e_{v,u'}$ ;
7        $u'.l = +\infty$ ;
8        $L(e_{v,u'}) = L(e_{v,u})$ ;
9        $V_i = V_i \cup u'$ ;
10      Relax( $v, u', e_{v,u}, B, \Delta$ );
11   else
12     Relax( $v, u, e_{v,u}, B, \Delta$ );
13
14 Function Relax( $v, u, e_{v,u}, B_i, \Delta$ )
15   if  $u.l > v.l + L(e_{v,u})$  then
16      $B_i[\lfloor u.l / \Delta \rfloor].\text{remove}(u)$ ;
17      $u.l = v.l + L(e_{v,u})$ ;
18      $u.p = v$ ;
19      $B_i[\lfloor u.l / \Delta \rfloor].\text{insert}(u)$ ;

```

4. Experimental Evaluation

In this section, we present the experimental evaluation of our distributed SSSP algorithm, LR-SSSP, which improves performance by replacing the global relaxation with only Local Relaxation.

4.1. Preparation

Competitor. We compare our proposed LR-SSSP with the traditional Bellman–Ford [5,6] algorithm and the state-of-the-art Δ -Stepping [10] and Δ -Stepping++ [11].

Datasets. We evaluate our LR-SSSP with eight datasets, including four real graphs, WiKi, HollyWood, LiveJournal, Orkut, and four synthetic graphs, Graph500-x, where x is 19–22. The real graphs are from SNAP of Stanford [15], and Graph500-x are from GraphChallenge of MIT [24]. The detailed information of these datasets is shown in Table 1.

Table 1. Datasets.

Graph	Abbrev.	# of Vertices	# of Edges	Avg. Degree
WiKi	WK	4,206,785	101,500,998	24
HollyWood	HW	1,139,905	116,050,145	101
LiveJournal	LJ	5,363,260	87,681,082	16
Orkut	OK	3,072,441	117,185,083	38
Graph500-19	×	335,318	15,458,507	46
Graph500-20	×	645,820	31,361,098	48
Graph500-21	×	1,243,072	63,462,385	51
Graph500-22	×	2,393,285	128,192,964	53

Experimental cluster. Our experiments are conducted both on local cluster and AliCloud. Our local cluster consists of four machines, and each machine runs Ubuntu 18.04 LTS and is equipped with Intel I5 4590 3.3 GHz CPU, 8 GB memory, and 1000 Mbps network card. The AliCloud cluster consists of 16 ecs.sn2.medium instances.

4.2. Performance of LR-SSSP on Different Datasets

To test the effectiveness of our LR-SSSP, we compare the LR-SSSP with the traditional Bellman–Ford and state-of-the-art Δ -Stepping and Δ -Stepping++. For fairness, we set the same Δ value in LR-SSSP, Δ -Stepping, and Δ -Stepping++ algorithms.

Figure 6 shows the runtime and communication of LR-SSSP and its competitors on four graphs. Note that the y-axis is log-scale. We can see that the traditional Bellman–Ford algorithm always results in the longest runtime and highest communication cost since there are no scheduling and pruning strategies to reduce invalid communication and relaxations. It is noted that the communication of Bellman–Ford on HW dataset is smaller than Δ -Stepping; we found that in this dataset, there are redundant relaxations when relaxing heavy edges globally. The Δ -Stepping++ performs better than Δ -Stepping because Δ -Stepping++ reduces some invalid global relaxation by employing the edge classification and direction-optimization. Our LR-SSSP outperforms other algorithms and achieves 6–20 $\times$ faster speed than Δ -Stepping++ and reduces communication by 87–96%.

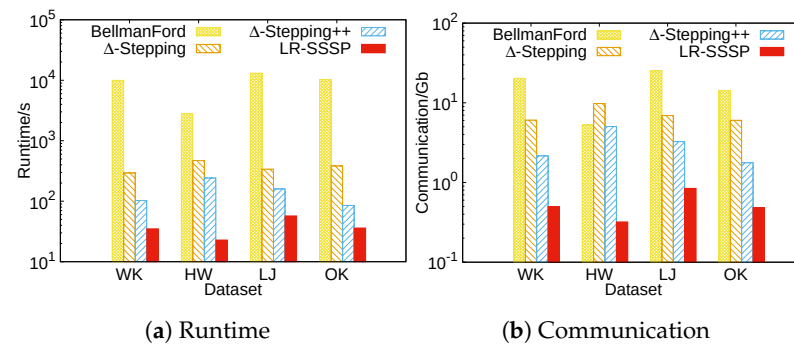


Figure 6. The performance of LR-SSSP.

4.3. Scaling Performance

Scalability is very important for distributed algorithms. Therefore, it is necessary to evaluate the performance of LR-SSSP when processing big graph datasets on large-scale clusters. The larger the cluster size is, the larger the amount of communication caused due to more edge cuts, while the bigger graph always results in more communication and computation.

Since the Bellman–Ford algorithm is inefficient and cannot return results in a reasonable time, we only run Δ -Stepping, Δ -Stepping++, and LR-SSSP on 2-node, 4-node, 8-node, 12-node, and 16-node AliCloud clusters to test the performance of distributed SSSP algorithms when varying the cluster sizes. The runtime and communication are shown in Figure 7. It is shown that the runtime of all the algorithms first decreases and then increases with the increasing size of the computer cluster, while the communication is increasing continuously. However, LR-SSSP outperforms other algorithms when varying the size of the cluster.

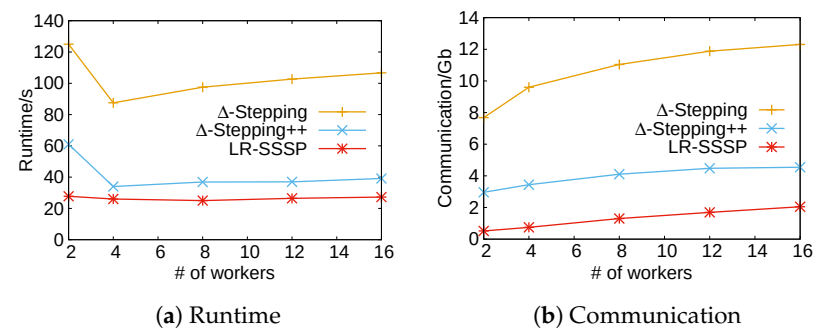


Figure 7. The performance of LR-SSSP when varying size of worker.

Furthermore, we also use the local cluster to perform Δ -Stepping, Δ -Stepping++ and LR-SSSP on Graph500-19 (15 M edges), Graph500-20 (31 M edges), Graph500-21 (63 M edges), and Graph500-22 (128 M edges) to verify the efficiency of LR-SSSP when varying the size of the graph. Figure 8 shows the runtime and communication when varying the size of the graph. We can see that when the dataset becomes larger, the runtime and communication of Δ -Stepping and Δ -Stepping++ increase dramatically, while our LR-SSSP increases slowly. In other words, LR-SSSP is more suitable for large graphs.

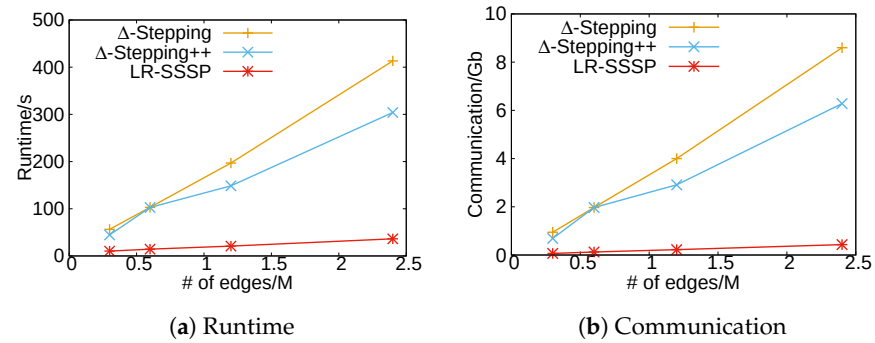


Figure 8. The performance of LR-SSSP when varying size of dataset.

4.4. Pruning Strategies Effectiveness

Since communication is the bottleneck of the distributed SSSP algorithm, we propose two strategies to reduce the communication of LR-SSSP. To test the effectiveness of the proposed pruning strategies, we first perform the LR-SSSP without pruning strategy (NP), then we turn to LR-SSSP with lazy synchronization (LS) and LR-SSSP with forward relaxation (LS+FR). Figure 9 shows the runtime and communication of LR-SSSP with different optimizations. We can see that our proposed optimizations can reduce the communication and runtime significantly, especially the forward relaxation.

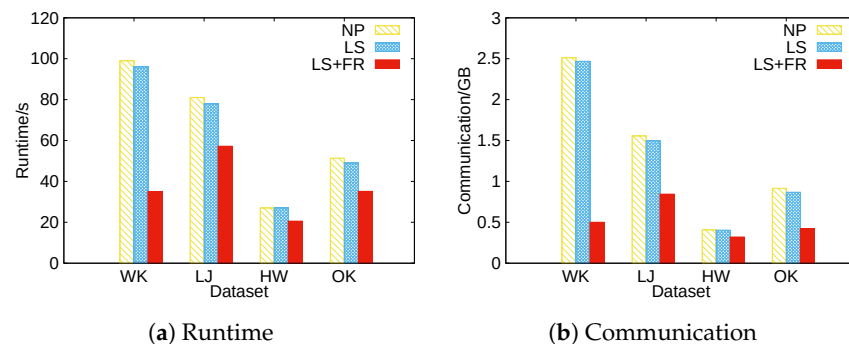


Figure 9. The efficiency of pruning method.

4.5. Parameter Studies

The value of parameter Δ affects the performance of LR-SSSP, Δ -Stepping, and Δ -Stepping++. A small Δ value will result in a large number of global barriers, which decreases the performance of the distributed SSSP. A large Δ value means fewer buckets, and the effectiveness of buckets is reduced. Figure 10 shows the runtime and communication when varying the Δ value from 0.4L to 1.2L, where L is the max length of all edges. It shows that the runtime and communication of LR-SSSP are stable, while those of Δ -Stepping and Δ -Stepping++ fluctuate. It means that LR-SSSP is insensitive to Δ .

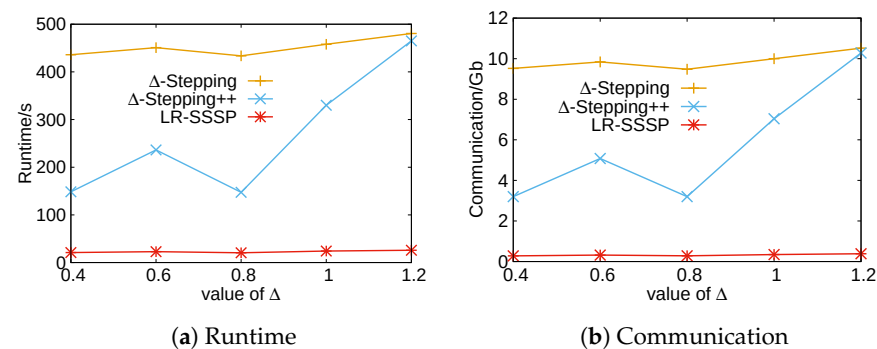


Figure 10. The performance of LR-SSSP when varying Δ .

Author Contributions: Methodology, S.G.; Software, J.T. and S.G.; Writing S.G., Y.Z., G.Y. and C.F. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by Fundamental Research Funds for the Central Universities (N2416011).

Data Availability Statement: Data are contained within the article.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Dantzig, G.B. On the shortest route through a network. *Manag. Sci.* **1960**, *6*, 187–190. [CrossRef]
2. Sanders, P. *Fast Route Planning*; Technical Report. Available online: <https://www.youtube.com/watch?v=-0ErpE8tQbw> (accessed on 20 May 2023).
3. Bader, D.A.; Kintali, S.; Madduri, K.; Mihail, M. Approximating betweenness centrality. In *Algorithms and Models for the Web-Graph, Processings of the 5th International Workshop, WAW 2007, San Diego, CA, USA, 11–12 December 2007*; Proceedings 5; Springer: Berlin/Heidelberg, Germany, 2007; pp. 124–137.
4. Li, B.; Su, P.; Chabbi, M.; Jiao, S.; Liu, X. DJXPerf: Identifying memory inefficiencies via object-centric profiling for Java. In *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization, Montréal, QC, Canada, 25 February–1 March 2023*; pp. 81–94.
5. Bellman, R. On a routing problem. *Q. Appl. Math.* **1958**, *16*, 87–90. [CrossRef]
6. Ford, L.R., Jr. *Network Flow Theory*; Technical Report; RAND Corp.: Santa Monica, CA, USA, 1956.
7. Dijkstra, E.W. A note on two problems in connexion with graphs. *Numer. Math.* **1959**, *1*, 269–271. [CrossRef]
8. Wang, Z.; Gu, Y.; Zimmermann, R.; Yu, G. Shortest Path Computation over Disk-Resident Large Graphs Based on Extended Bulk Synchronous Parallel Methods. In *Database Systems for Advanced Applications, Processings of the 18th International Conference, DASFAA 2013, Wuhan, China, 22–25 April 2013*; Proceedings, Part II 18; Springer: Berlin/Heidelberg, Germany, 2013; pp. 1–15.
9. Zhu, A.D.; Xiao, X.; Wang, S.; Lin, W. Efficient single-source shortest path and distance queries on large graphs. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Chicago, IL, USA, 11–14 August 2013*; pp. 998–1006.
10. Meyer, U.; Sanders, P. Δ -stepping: A parallelizable shortest path algorithm. *J. Algorithms* **2003**, *49*, 114–152. [CrossRef]
11. Chakaravarthy, V.T.; Checonci, F.; Murali, P.; Petrini, F.; Sabharwal, Y. Scalable single source shortest path algorithms for massively parallel systems. *IEEE Trans. Parallel Distrib. Syst.* **2016**, *28*, 2031–2045. [CrossRef]
12. Teng, F.; Ban, Z.; Li, T.; Sun, Q.; Li, Y. A Privacy-Preserving Distributed Economic Dispatch Method for Integrated Port Microgrid and Computing Power Network. *IEEE Trans. Ind. Inform.* **2024**. [CrossRef]
13. Lee, K.; Lam, M.; Pedarsani, R.; Papailiopoulos, D.; Ramchandran, K. Speeding up distributed machine learning using codes. *IEEE Trans. Inf. Theory* **2017**, *64*, 1514–1529. [CrossRef]
14. Wu, X.; Zhang, J.; Wang, F.Y. Stability-based generalization analysis of distributed learning algorithms for big data. *IEEE Trans. Neural Netw. Learn. Syst.* **2019**, *31*, 801–812. [CrossRef] [PubMed]
15. Available online: <https://snap.stanford.edu/data/> (accessed on 20 May 2023).
16. Gong, S.; Zhang, Y.; Yu, G. HBP: Hotness balanced partition for prioritized iterative graph computations. In *Proceedings of the 2020 IEEE 36th International Conference on Data Engineering, Dallas, TX, USA, 20–24 April 2020*; pp. 1942–1945.
17. Delling, D.; Sanders, P.; Schultes, D.; Wagner, D. Engineering Route Planning Algorithms. In *Algorithmics of Large and Complex Networks: Design, Analysis, and Simulation*; Springer: Berlin/Heidelberg, Germany, 2009; pp. 117–139.
18. Sommer, C. Shortest-path queries in static networks. *ACM Comput. Surv.* **2014**, *46*, 1–31. [CrossRef]

19. Zhu, A.D.; Ma, H.; Xiao, X.; Luo, S.; Tang, Y.; Zhou, S. Shortest path and distance queries on road networks: Towards bridging theory and practice. In Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, New York, NY, USA, 22–27 June 2013; pp. 857–868.
20. Ouyang, D.; Qin, L.; Chang, L.; Lin, X.; Zhang, Y.; Zhu, Q. When hierarchy meets 2-hop-labeling: Efficient shortest distance queries on road networks. In Proceedings of the 2018 International Conference on Management of Data, Houston, TX, USA, 10–15 June 2018; pp. 709–724.
21. Li, Y.; U, L.H.; Yiu, M.L.; Kou, N.M. An experimental study on hub labeling based shortest path algorithms. *Proc. VLDB Endow.* **2017**, *11*, 445–457. [[CrossRef](#)]
22. Kumar, V.; Schwabe, E.J. Improved algorithms and data structures for solving graph problems in external memory. In Proceedings of the SPDP'96: 8th IEEE Symposium on Parallel and Distributed Processing, New Orleans, LA, USA, 23–26 October 1996; pp. 169–176.
23. Cheng, J.; Ke, Y.; Chu, S.; Cheng, C. Efficient processing of distance queries in large graphs: A vertex cover approach. In Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, Scottsdale, AZ, USA, 20–24 May 2012; pp. 457–468.
24. Available online: <https://graphchallenge.mit.edu/data-sets> (accessed on 20 May 2023).

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.