2023 19th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob) | 979-8-3503-3667-2/23/$31.00 ©2023 IEEE | DOI: 10.1109/WiMob58348.2023.10187809

Communication-Efficient Δ-Stepping for Distributed Computing Systems

Haomeng Zhang, Junfei Xie, *Senior Member, IEEE*, Xinyu Zhang, *Senior Member, IEEE*

***Abstract*—This paper considers the single source shortest path (SSSP) problem, which is the key for many applications such as navigation, mapping, routing, and social networking. Existing SSSP algorithms are designed mostly for shared-memory systems. Nevertheless, with the prevalence of diverse smart devices like drones, there is a growing interest in deploying SSSP algorithms over distributed computing systems so that they can run efficiently onboard of smart devices via Mobile Ad Hoc Computing or at the network edges via Mobile Edge Computing. In this paper, we introduce a communication-efficient Δ-stepping algorithm for distributed computing systems. The proposed algorithm is featured by 1) a message coordination architecture for reducing message exchanges between workers, 2) a pruning technique for reducing redundant computations, and 3) an aggregation technique for further reducing message exchanges when communication delay is significant. Theoretical analyses and experimental studies on real-world graph datasets demonstrate the promising performance of proposed algorithm.**

***Index Terms*—Single source shortest path (SSSP), Δ-stepping, Distributed computing, Communication efficiency**

I. Introduction

Recent years have witnessed the prevalence of diverse smart devices with sensing, computing, and communication capabilities, such as unmanned aerial vehicles (UAVs), smartphones, wearable devices, autonomous driving vehicles, etc. With the advancement of Internet of Things (IoT) technology [1], these smart devices are able to talk to each other wirelessly, share information and resources to provide all kinds of services that transform the way we live, work, and play. Nevertheless, there is still much room for improvement of their intelligence and capability, especially their reaction to data changes.

One major problem encountered in the data analysis or decision making processes of many smart devices is the single source shortest path (SSSP) problem, which is the key for navigation, mapping, routing, and social networking applications, to name a few [2]. Solving SSSP problems is time-consuming when the problem scale is large. As smart devices, especially the UAVs, smartphones, and wearable devices, often have limited computing resources, computation-intensive tasks like SSSP are usually offloaded to the remote servers or cloud, which however may encounter many issues such as high latency, low coverage, and security vulnerability [3]. The emerging Mobile/Multi-access Edge Computing [3] and Mobile Ad Hoc Computing [4] offer promising solutions to address these issues by moving cloud resources close to users at the network edges or allowing smart devices to perform collaborative computing. To deploy SSSP algorithms on such computing paradigms with distributed computing resources, there is a need to parallelize the algorithms and make them run efficiently over distributed networks with wired or wireless connections among computing nodes.

Although the SSSP problem has been well researched in the past decades [5]–[11], majority of the studies assume the problem is solved by a single computer using one or multiple CPU cores in a sequential or parallel manner. Among existing SSSP algorithms, the Dijkstra's algorithm [5] is the most popular one, which is efficient but lacks parallelism. The Bellman-Ford [6] algorithm is more amenable to parallelization but has a considerably larger complexity than the Dijkstra's algorithm. To make the best of both worlds, the Δ-stepping algorithm [7] was then proposed, which allows us to adjust the degree of complexity and parallism by tuning the parameter Δ. Following this pioneering work, various efforts have been made to improve the implementation efficiency [8], [9] and the theoretical bound [10], [11] of Δ-stepping. Nevertheless, the proposed algorithms were designed mostly for shared-memory systems.

When deploying parallel SSSP algorithms over a distributed computing system, message exchanges are required for computing nodes to share and synchronize results. The communication delay incurred can be significant, especially in wireless networks. Therefore, existing parallel SSSP algorithms that ignore communication delay will inevitably suffer from performance degradation when they are implemented on distributed computing systems, and the impact of communication delay can be significant when large number of message exchanges are involved or when communication resources are limited.

Little attention has been paid to the efficient implementation of SSSP algorithms over distributed computing systems. In [12], the authors compared the performance of multiple popular SSSP algorithms on distributed-memory systems, but the impact of communication delay was not studied. [13] presented a new SSSP algorithm for massively parallel systems. Although distributed memory was considered, the algorithm was designed for running over multiple processors in a single machine with negligible processor-to-processor communication delay, and it requires message exchange each time when relaxing an edge. Also of relevance are the studies on using

We gratefully acknowledge support from NSF CAREER 2048266.

Haomeng Zhang is with the Department of Electrical and Computer Engineering, University of California San Diego (UCSD) and San Diego State University (SDSU), San Diego, CA, 92182 (e-mail: hzhang3986@sdsu.edu, haz069@ucsd.edu).

Junfei Xie is with the Department of Electrical and Computer Engineering, SDSU, San Diego, CA, 92182 (e-mail: jxie4@sdsu.edu).

Xinyu Zhang is with the Department of Electrical and Computer Engineering, UCSD, La Jolla, CA, 92093 (e-mail: xyzhang@ucsd.edu) .

graph partitioning (GP) strategies to improve the efficiency of distributed graph algorithms [14]–[16]. These methods aim to distribute graph vertices or edges across multiple worker nodes to balance the workload and reduce communication costs. Although they are particularly effective on graphs with power law degrees, they are not optimized for sparse graphs or SSSP algorithms.

Main Contributions: This paper introduces a communication-efficient distributed Δ-stepping algorithm that achieves high efficiency in distributed computing systems, especially in wireless networks. In this algorithm, a master is introduced to coordinate massage exchanges between the workers and synchronize intermediate results. A novel pruning technique is designed to speed up computation for edge relaxations. To address high-latency scenarios, we further introduce a novel aggregation technique that significantly reduces the number of message exchanges, and subsequently the communication cost. Nevertheless, employing the aggregation technique will introduce redundant computations. To achieve a better communication-computation tradeoff, a switching mechanism that allows two techniques to be combined is further suggested. To verify the merits of the proposed algorithm, both theoretical analyses and comparative experimental studies using real-world graph datasets are conducted.

In the rest of the paper, Sec. II formulates the SSSP problem and briefly reviews the standard Dijkstra's and Δ-stepping algorithms. Sec. III describes the proposed algorithm and conducts theoretical analyses on its performance. Experimental results are then presented in Sec. IV. Sec. V concludes the paper with a brief discussion on future works.

II. Preliminaries

A. SSSP Problem

Consider an undirected graph $G = (V, E)$ composed of a set of vertices V and edges E. Let $m = |V|$ denote the number of vertices and $n = |E|$ the number of edges. Each edge $(v, u) \in E$ is associated with a non-negative weight, $w(v, u) : V \times V \to \mathbb{R}^+$. Let $\text{dist}(s, v) \geq 0$ be the weight of a shortest path from a source vertex $s \in V$ to another vertex $v \in V$. If v is unreachable, then $\text{dist}(s, v) = \infty$. Given the source vertex $s \in V$, the goal of the SSSP problem is to find the shortest path with the smallest weight $\text{dist}(s, v)$ from s to each reachable vertex $v \in V$. In the rest of the paper, we abbreviate $\text{dist}(s, v)$ to $\text{dist}(v)$ for the given s.

B. Dijkstra's Algorithm

To solve the SSSP problem, the Dijkstra's algorithm calculates a *tentative distance* $\text{tent}(v)$ for each vertex $v \in V$, which is an estimate of the shortest distance $\text{dist}(v)$ to the source s and satisfies $\text{tent}(v) \geq \text{dist}(v)$. The tentative distances are improved iteratively until $\text{tent}(v) = \text{dist}(v)$, $\forall v \in V$. Vertices with $\text{tent}(v) = \text{dist}(v)$ are said to be *settled*. To improve the tentative distances, the Dijkstra's algorithm maintains a *queue* of vertices at each iteration. Initially, only the source s with $\text{tent}(s) = 0$ is queued. The tentative distances of all other vertices are set to $\text{tent}(v) = \infty$, which are said to be *unreached*. In each iteration, the queued vertex v with the smallest tentative distance is settled and removed from the queue. In addition, all edges from v, $\forall (v, u) \in E$, are *relaxed*, i.e., each tentative distance $\text{tent}(u)$ is updated by $\min\{\text{tent}(u), \text{tent}(v) + w(v, u)\}$. If $\text{tent}(u) = \infty$ before relaxation, then u is added into the queue.

Algorithm 1: Δ-Stepping Algorithm

```
1  Input: G = (V, E, w), Δ, s
2  Output: tent(v), ∀v ∈ V
3  relax(s, 0); i ← 0;
4  while ¬isEmpty(B) do
5      Sett ← ∅;        ▷ Initialize the set of settled vertices
6      while B[i] ≠ ∅ do
7          Reqs ← {(u, tent(v) + w(v,u)) : v ∈ B[i] and
             (v,u) ∈ light(v)}; ▷ Generate edge relaxation
             requests
8          Sett ← Sett ∪ B[i]; B[i] ← ∅;
9          foreach (u, d) ∈ Reqs do
10             relax(u, d);                ▷ Relax light edges
11     Reqs ← {(u, tent(v) + w(v,u)) : v ∈ Sett and
         (v,u) ∈ heavy(v)}
12     foreach (u, d) ∈ Reqs do
13         relax(u, d);                    ▷ Relax heavy edges
14     i ← i + 1;
15 return tent(v), ∀v ∈ V.
16 Function relax(u, d):
17     if d < tent[u] then
18         tent(u) ← d;          ▷ Update tentative distance
19         B[⌊tent(u)/Δ⌋] ← B[⌊tent(u)/Δ⌋] \ {u};
20         B[⌊d/Δ⌋] ← B[⌊d/Δ⌋] ∪ {u};
```

C. Δ-stepping

Different from the Dijkstra's algorithm, the Δ-stepping [7] maintains an array of *buckets*, where each bucket $B[i]$ stores a set of queued vertices with $\text{tent}(v) \in [i\Delta, (i+1)\Delta)$. $\Delta > 0$ is a parameter known as the step or bucket width. A suggested choice is $\Delta = O(1/d)$ for a random weighted graph with a maximum degree of d [7].

As shown in Alg. 1, in each *phase* i (iteration of the outer while-loop) (Lines 5-14), the algorithm settles all vertices in bucket $B[i]$ (vertices are removed if settled) and relaxes all edges emanating from the vertices in the bucket. To settle each vertex v in bucket $B[i]$, it only considers the outgoing *light* edges from v, denoted by $\text{light}(v) \in E$, that have weight $w(v, u) \leq \Delta$, where $(v, u) \in \text{light}(v)$. Specifically, in each iteration of the inner while-loop, the current bucket $B[i]$ is emptied and all outgoing light edges are relaxed (Lines 7-10). Note that the relaxation may cause vertices deleted previously being *reinserted* or new vertices being added into the current bucket (Lines 19-20). Therefore, multiple iterations are usually

needed to settle all vertices in the bucket. After all vertices in $B[i]$ have been settled, the algorithm relaxes all outgoing *heavy* edges from each vertex v in $B[i]$, denoted by heavy(v), that have weight $w(v,u) > \Delta$, where $(v,u) \in$ heavy(v) (Lines 12-13). Note that relaxing heavy edges won't cause associated tentative distances being updated to fall within the scope of the current bucket, and hence no vertices will be inserted into the already emptied bucket.

Traditionally, the Δ-stepping is parallelized by randomly assigning the vertices in the current bucket to multiple processing units (PUs) at each iteration [7]. Each PU then scans the adjacency list of the assigned vertices and relaxes all edges emanating from these vertices. Referring to Alg. 1, the relaxation of light edges (Lines 9-10) and the heavy edges (Lines 12-13) are computed in a parallel manner.

III. Communication-Efficient Distributed Δ-Stepping

In this section, we describe the communication-efficient distributed Δ-stepping algorithm that runs efficiently in distributed computing systems.

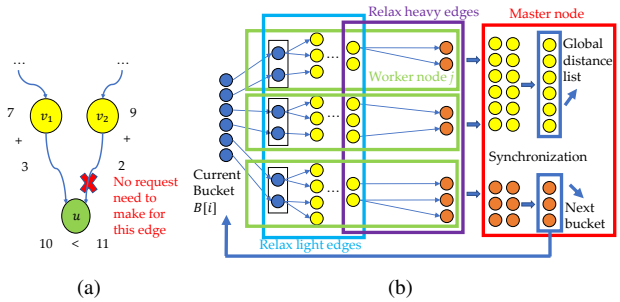
A. Limitation of Traditional Parallel Implementation

In the traditional parallel Δ-stepping algorithm [7], when relaxing the light edges light(v) (Lines 7-10 in Alg. 1), the PUs need to update the tentative distances in the shared memory whenever smaller values are found. However, in a distributed computing system, computing nodes do not have access to their peers' memories. Message exchanges between computing nodes are thus required to synchronize the shortest tentative distances found. If we directly implement the parallel Δ-stepping in a distributed computing system, the number of message exchanges involved will be huge, as any update to a tentative distance requires one message exchange. When the communication delay incurred for each message exchange cannot be ignored, the accumulated delay can be significant.

B. Proposed Algorithm

1) Message Coordination: Exchanging messages between computing nodes in a decentralized manner like the parallel Δ-stepping algorithm will lead to a high communication cost due to the large number of message exchanges. To address this issue, we assign one computing node, called *master*, to coordinate the communications between other nodes, called *workers* (or *slaves*). The workers perform edge relaxations and send intermediate results all to the master for synchronization. No communication between workers is thus needed.

2) Pruning: Since the master is responsible for synchronizing all intermediate results obtained by the workers, the computation cost for synchronization increases compared with the decentralized synchronizations happening in the traditional parallel Δ-stepping algorithm, where all computing nodes will perform synchronizations. To reduce master's workload, we introduce a pruning technique. The idea is to detect and prune redundant edge relaxation requests. As illustrated in Fig. 1(a), when the outgoing edges of two vertices, v_1 and v_2, in the current bucket intersect at the same vertex u, if one edge (v_1,u) with a shorter tentative distance has been relaxed, the other edge (v_2,u) does not need to be relaxed. To implement this idea, at the beginning of each iteration, all neighbors u of vertices v in the current bucket with $(v,u) \in E$ and $v \in B[i]$ are found and divided into M subsets, $\{U_1, U_2, \ldots, U_M\}$, according to the allocation scheme specified by the master, where M is the total number of workers. Here we assume all workers have the same computing capability, so the set is divided equally. Each subset $U_j = \{u\}$ is assigned to a worker j, who then performs edge relaxations (Line 4-10 in Alg. 3). Redundant edge relaxation requests will be identified and pruned by workers. This differs from the traditional parallel Δ-stepping which partitions the current bucket $B[i]$ among PUs. As workers relax different edges, different relaxation-induced updates will be made to the current bucket. The updated bucket, denoted by $B_j[i]$ for worker j, and associated tentative distances are sent back to the master for synchronization at the end of each iteration (Line 6 & 9 in Alg. 2).



(a) (b)

Fig. 1: Illustration of the (a) pruning and (b) aggregation methods.

3) Aggregation: As relaxing each bucket requires multiple message exchanges (the inner while loop), the communication cost can still be high when the delay incurred for each message exchange is relatively large, e.g., in wireless networks. To address this scenario, we propose another technique to further reduce message exchanges. The idea is to aggregate the multiple iterations performed in each phase. As illustrated in Fig. 1(b), the current bucket $B[i]$ is divided evenly into M sub-buckets with each sub-bucket $B_j[i]$ assigned to a worker j. Each worker j settles all vertices in the assigned sub-bucket $B_j[i]$ by relaxing all light and heavy edges (Line 12-14 in Alg. 3). The updated tentative distances are then sent back to the master for synchronization. After each synchronization, the settled vertices and associated shortest distances are stored in the memory at the master (Line 10-13 in Alg. 2). Although this method significantly reduces message exchanges, it introduces redundant computations due to less-frequent synchronizations. The amount of redundancy incurred is highly related to the density of the graph, as more repeatedly relaxed edges may exist in dense graphs than sparse graphs.

4) Switching between Pruning and Aggregation: The pruning method works well when the communication delay for each message exchange is small or when the graph is dense, as we will show in Sec. IV. On contrary, the aggregation method works well when the communication delay is large or when

the graph is sparse. Although they cannot be implemented in the same phase, they can be combined and implemented in different phases to achieve a communication-efficiency tradeoff and address more complicated scenarios, e.g., mobile and wireless networks with dynamic communication delays or graphs with large density variations. This can be realized by using a switching mechanism that chooses the best method to use at the beginning of each phase based on the network condition and graph density. In Sec. IV, we will suggest an empirical switching criteria based on observations from the experiments.

Algorithm 2: Proposed Algorithm - Master

Input: $G = (V, E, w)$, Δ, s.
Output: tent$(v), \forall v \in V$ relax$(s, 0)$; $i \leftarrow 0$;
Broadcast allocation mechanism to all workers;
while $\neg$*isEmpty*(B) **do**
  Select edge relaxation method;
  Broadcast $\{(v, \text{tent}(v)) : v \in B[i]\}$;
  $B[i] \leftarrow \emptyset$;
  foreach $j \in \{1, 2, \ldots, M\}$ **do**
    Recv $\{(v, \text{tent}(v)) : v \in B_j[i] \cup Sett_j\}$;
    if $\neg$*isEmpty*$(Sett_j)$ **then**
      foreach $(v, d) \in Sett_j$ **do**
        if $d < glob\_dis(v)$ **then**
          $glob\_dis(v) \leftarrow d$;
  $B[i+1] \leftarrow B[i+1] \cup B_j[i]$;
  $i \leftarrow i + 1$;

Algorithm 3: Proposed Algorithm - Worker j

Listen to the channel;
if *Recv* $B[i]$ *and tent*(v), $\forall v \in B[i]$, *from Master* **then**
  if *Pruning method is used* **then**
    if $B[i] \neq \emptyset$ **then**
      Generate **requests** and **relax** light edges $(v, u_j) \in \text{light}(v)$ for $v \in B[i]$, where $u_j \in U_j$;
      Send $\{(v, \text{tent}(v)) : v \in B_j[i]\}$ to master;
    else
      Generate **requests** and **relax** heavy edges $(v, u_j) \in \text{heavy}(v)$ for $v \in Sett_j$;
      Send $\{(v, \text{tent}(v)) : v \in B_j[i] \cup Sett_j\}$;
      $Sett_j \leftarrow \emptyset$;
  else if *Aggregation method is used* **then**
    while $B_j[i] \neq \emptyset$ **do**
      Generate **requests** and **relax** light edges $(v, u) \in \text{light}(v)$ for $v \in B_j[i]$;
    Generate **requests** and **relax** heavy edges $(v, u) \in \text{heavy}(v)$ for $v \in Sett_j$;
    Send $\{(v, \text{tent}(v)) : v \in B_j[i] \cup Sett_j\}$;
    $Sett_j \leftarrow \emptyset$;

C. Analysis on Communication Cost

Define $L := \max\{\text{dist}(v) : \text{dist}(v) < \infty\}$ as the maximum shortest path weight. Define Δ-*path* as a path that has a weight no larger than Δ and no edge repetitions. Let C_Δ denote a set of all vertex pairs $\langle u, v\rangle$ connected by a Δ-path. Lemma 1 and 2 provide the amount of communication cost required by the proposed algorithm when the pruning and aggregation methods are used in all phases, respectively.

Lemma 1: The amount of communication cost required by the proposed distributed Δ-stepping algorithm when the pruning method is used in all phases is $O(\frac{LMl_\Delta}{\Delta})$, where $l_\Delta = 1 + \max_{\langle u,v\rangle \in C_\Delta} \min\{|A| : A = (u, ..., v) \text{ is a minimum weight } \Delta\text{-path}\}$.

Lemma 2: The amount of communication cost required by the proposed distributed Δ-stepping algorithm when the aggregation method is used in all phases is $O(\frac{LM}{\Delta})$.

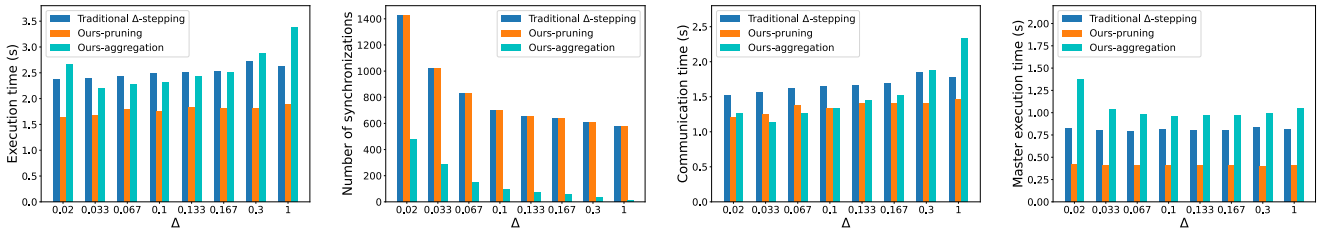
Both lemmas can be proved according to [7]. Particularly, l_Δ computes the number of iterations required for calculating a nonempty bucket. As the array B contains up to $\lceil L/\Delta \rceil$ buckets, $\sum_{i=1}^{\lceil L/\Delta \rceil} 2M(l_\Delta + 1)$ transmissions are required for the proposed algorithm when pruning is used in all phases. The amount of communication cost is hence $O(\frac{LMl_\Delta}{\Delta})$. For the proposed algorithm with aggregation used in all phases, as settling vertices in a bucket only requires a single synchronization, $\sum_{i=1}^{\lceil L/\Delta \rceil} 2M$ transmissions are required and the required communication cost is hence $O(\frac{LM}{\Delta})$.

IV. Experimental Studies

This section conducts experiments to evaluate the performance of the proposed algorithm. All experiments are conducted on a desktop with Intel i9 12900KF CPU that has a base frequency of 3.20 GHz and 16 GB memory.

A. Experiment setup

To implement the proposed algorithm, we use the open-source packet `mpi4py` in Python for the Message Passing Interface (MPI), which is a standard distributed computing interface. Each CPU core represents a computing node with distributed memory. In the experiment, four cores are used among which three are the workers and one is the master. To evaluate the performance of the two edge relaxation methods (i.e., pruning and aggregation), we let all phases use only one of the methods. The resulting algorithms are referred to as **Ours-pruning** if pruning is used in all phases and **Ours-aggregation** otherwise. For comparison, we also implement the sequential Dijkstra's algorithm using priority queue on a single core (referred to as **Dijkstra**), **distributed Dijkstra**'s algorithm [17], **Bellman-Ford** [6], and the traditional parallel Δ-stepping (referred to as **Traditional Δ-stepping**) implemented over multiple cores with a master-slave message coordination architecture and distributed memory. To evaluate their performance, we consider six real-world graphs obtained from SNAP [18] that have different sizes and densities. A synthetic graph created by the Networkx expected degree graph generator [19] is also considered. The characteristic of each graph is described in Tab. I.



(a) (b) (c) (d)

Fig. 2: (a) Execution time, (b) number of synchronizations, (c) total communication time, and (d) master execution time of different distributed Δ-stepping algorithms at different values of Δ when running on the graph **PA**.

Name	Type	Source	m	n	Avg. deg.	Max. deg.
PA	Road	Pennsylvania	1 million (M)	1.5M	1.5	9
CA	Road	California	2M	2.8M	1.4	12
TX	Road	Texas	1.4M	1.9M	1.35	12
YO	Social	Youtube	1.1M	3M	2.7	28754
DB	Web	Google	0.3M	1M	3.3	343
AM	Social	Twitch	0.3M	0.9M	3	549
SY	Synthetic	Networkx	1M	7.2M	7.2	38

TABLE I: Graphs used in the experiments.

Four performance metrics are used, including 1) *execution time* spent to solve the SSSP problem; 2) *number of synchronizations* taken to solve the problem; 3) *total communication time* spent for exchanging intermediate results; and 4) *master execution time* taken by the master to process intermediate results. To reduce experimental uncertainty, each experiment is repeated for 10 times and the average values are recorded.

B. Experiment Results

In the following experiments, we first analyze the impact of key parameter Δ on distributed Δ-stepping algorithms using graph **PA**. We then conduct comparison studies using different graphs. The impact of communication delay and graph density on the proposed algorithm are then investigated.

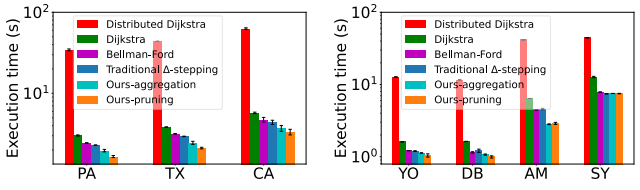
1) Impact of Δ: Fig. 2 shows the performance of the three distributed Δ-stepping algorithms as Δ varies. Fig. 2(a) shows that our algorithm with pruning achieves the best performance for all Δ setups due to significantly reduced edge relaxation requests. Our algorithm with aggregation is generally more efficient than the traditional Δ-stepping but it achieves a worse performance at very small or very large Δ values, due to the additional redundant computations introduced. Its optimal performance is achieved at $\Delta = 0.33$. The optimal performance of the other two algorithms is achieved at $\Delta = 0.02$.

From Fig. 2(b), we see that all algorithms require fewer rounds of synchronizations to solve the SSSP problem as Δ increases. Our algorithm with aggregation requires the fewest synchronizations as it only synchronizes once in each phase. Of interest, as Fig. 2(c) shows, the total communication time does not decrease with the increasing Δ and more synchronizations. This is because the amount of data to be transmitted for each synchronization increases with the increase of Δ, as Δ directly impacts the size of the buckets. In terms of master execution time, as shown in Fig. 2(d), all algorithms are generally robust to the change of Δ, except our algorithm with aggregation at very small or very large Δ values due to introduced redundant computations. Meanwhile, we can observe that our algorithm with pruning achieves the least master execution time, indicating its effectiveness in pruning redundant edge relaxation requests.

The above studies provide guidelines for selecting proper Δ values for distributed Δ-stepping algorithms. In the following experiments, we follow this procedure to configure Δ in the distributed Δ-stepping algorithms.

2) Comparison Studies: Fig. 3 compares the mean and standard deviation of execution times across different algorithms and graphs. As we can see, our algorithms consistently outperform all benchmark algorithms on all graphs. The one with pruning achieves the best efficiency on most graphs, as it involves the least amount of redundant work. However, as the graph density increases, its superiority diminishes compared to the algorithm with aggregation (Fig. 3(b)). This is because, for sparse graphs, the amount of light edges to relax at each iteration is small, making it more costly to perform one synchronization than one computation iteration. Our method that aggregates the computation over multiple iterations will hence significantly reduce the communication time. Moreover, as not much redundant computation will be introduced during the aggregation due to the sparsity of graphs, the overall execution time will also be reduced.

3) Impact of Data Transmission Rate: Communication delay can greatly impact performance in distributed computing systems. To evaluate its effect, we tested our algorithms under various data transmission rates. The data rate in the wired condition was measured to be 2.2 Gbps. To simulate various data rates, we added a time delay proportional to the wired communication cost. Specifically, we tested our algorithms at data rates of 1100, 550, 220, 70, and 20 Mbps. The results show that the aggregation method outperforms the pruning method on the sparse **PA** graph (Fig. 4(a)), while the opposite trend is observed on the dense **YO** graph (4(b)). They also suggest that data rate has a greater impact on our algorithms' performance for sparse graphs than for dense graphs.



(a) (b)

Fig. 3: Execution time of different algorithms for different (a) sparse graphs and (b) dense graphs.

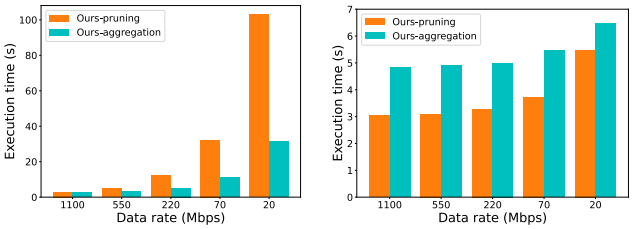


Fig. 4: Execution time of different algorithms at data rates for (a) sparse graph **PA** and (b) dense graph **YO**.

4) Impact of Graph Density: To better understand the impact of graph density, we generate random graphs of varying density using the Erdos-Renyi [20] method, which takes two parameters 1) the number of vertices m and 2) the probability for an edge to exist between two vertices $p = \tilde{d}/m$. Here $\tilde{d}$ specifies the graph density. In the experiments, we set $m = 1$M and vary the value of $\tilde{d}$. If the generated graphs are not connected, the largest connected subgraphs are used. Fig. 5 plots a heatmap that shows how much the aggregation method outperforms the pruning method in terms of the execution time when both $\tilde{d}$ and data rate vary. The improvements in percentage are marked on the grids. Negative values indicate the pruning method performs better. This figure provides an empirical switching criteria that allows us to make a proper selection between the two methods based on network condition and graph density, where the boundary that separates positive percentages from negative ones specifies the decision surface.

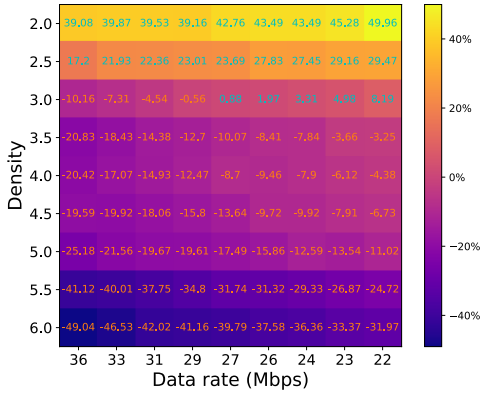


Fig. 5: The efficiency improvement percentages of the aggregation method compared with the pruning method.

V. Conclusion

This paper presents a new communication-efficient Δ-stepping algorithm for distributed computing systems. It adopts a master-slave architecture for coordinating the communication between computing nodes, which significantly reduces message exchanges. To further speed up computation, two edge relaxation methods are designed, including the pruning method that reduces redundant edge relaxation requests and the aggregation method that reduces synchronizations. A switching mechanism is also suggested that allows two methods to be combined for achieving communication-efficiency tradeoff. Experiment results demonstrate the high efficiency of proposed algorithms compared with existing methods. They also suggest ways for selecting a proper edge relaxation method. Particularly, pruning method is preferred when communication delay is small or graph is dense. Otherwise, the aggregation method is preferred. In the future, we will consider more complicated distributed computing systems with heterogeneous and moving computing nodes. We will also test our algorithm in the wireless environment.

References

[1] X. Sun and N. Ansari, "Edgeiot: Mobile edge computing for the internet of things," *IEEE Communications Magazine*, vol. 54, no. 12, pp. 22–29, 2016.

[2] N. Meenakshi, V. Pandimurugan, L. Sathishkumar *et al.*, "Optimal routing methodology to enhance the life time of sensor network," *Materials Today: Proceedings*, vol. 46, pp. 5894–5900, 2021.

[3] N. Abbas, Y. Zhang, A. Taherkordi, and T. Skeie, "Mobile edge computing: A survey," *IEEE Internet of Things Journal*, vol. 5, no. 1, pp. 450–465, 2017.

[4] I. Yaqoob, E. Ahmed, A. Gani, S. Mokhtar, M. Imran, and S. Guizani, "Mobile ad hoc cloud: A survey," *Wireless Communications and Mobile Computing*, vol. 16, no. 16, pp. 2572–2589, 2016.

[5] E. W. Dijkstra, "A note on two problems in connexion with graphs:(numerische mathematik, 1 (1959), p 269-271)," 1959.

[6] R. Bellman, "On a routing problem," *Quarterly of applied mathematics*, vol. 16, no. 1, pp. 87–90, 1958.

[7] U. Meyer and P. Sanders, "δ-stepping: a parallelizable shortest path algorithm," *Journal of Algorithms*, vol. 49, no. 1, pp. 114–152, 2003.

[8] Y. Zhang, A. Brahmakshatriya, X. Chen, L. Dhulipala, S. Kamil, S. Amarasinghe, and J. Shun, "Optimizing ordered graph algorithms with graphit," *arXiv preprint arXiv:1911.07260*, 2019.

[9] L. Dhulipala, G. Blelloch, and J. Shun, "Julienne: A framework for parallel graph algorithms using work-efficient bucketing," in *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*, 2017, pp. 293–304.

[10] X. Dong, Y. Gu, Y. Sun, and Y. Zhang, "Efficient stepping algorithms and implementations for parallel shortest paths," *arXiv preprint arXiv:2105.06145*, 2021.

[11] G. E. Blelloch, Y. Gu, Y. Sun, and K. Tangwongsan, "Parallel shortest paths using radius stepping," in *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, 2016, pp. 443–454.

[12] T. Panitanarak and K. Madduri, "Performance analysis of single-source shortest path algorithms on distributed-memory systems," in *SIAM Workshop on Combinatorial Scientific Computing (CSC)*. Citeseer, 2014, p. 60.

[13] V. T. Chakaravarthy, F. Checconi, P. Murali, F. Petrini, and Y. Sabharwal, "Scalable single source shortest path algorithms for massively parallel systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 7, pp. 2031–2045, 2016.

[14] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, 2012, pp. 17–30.

[15] C. Xie, L. Yan, W.-J. Li, and Z. Zhang, "Distributed power-law graph computing: Theoretical and empirical analysis," *Advances in neural information processing systems*, vol. 27, 2014.

[16] G. Ma, Y. Xiao, T. Willke, N. Ahmed, S. Nazarian, and P. Bogdan, "A distributed graph-theoretic framework for automatic parallelization in multi-core systems," *Proceedings of Machine Learning and Systems*, vol. 3, pp. 550–568, 2021.

[17] A. Pradhan and G. Mahinthakumar, "Finding all-pairs shortest path for a large-scale transportation network using parallel floyd-warshall and parallel dijkstra algorithms," *Journal of Computing in Civil Engineering*, vol. 27, no. 3, pp. 263–273, 2013.

[18] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.

[19] A. Hagberg, P. Swart, and D. S Chult, "Exploring network structure, dynamics, and function using networkx," Los Alamos National Lab.(LANL), Los Alamos, NM (United States), Tech. Rep., 2008.

[20] P. Erdős, A. Rényi *et al.*, "On the evolution of random graphs," *Publ. Math. Inst. Hung. Acad. Sci*, vol. 5, no. 1, pp. 17–60, 1960.