Abstract This paper initiates a study of the fully dynamic all-pairs shortest paths problem (APSP) in the MPC model, with each machine having $O(n^{\alpha})$ memory. Here, n denotes the number of vertices in the graph and α represents a constant within the range $(0, 1)$. We mainly focus on the directed weighted graphs. The update operation is inserting or deleting a node and edges incident to that node. We design the first randomized parallel dynamic APSP algorithm with a worst-case update round of $O(n^{\frac{2}{3}-\frac{\alpha}{6}} \log n/\alpha)$ and a total memory usage of $O(n^{3-\alpha/2})$. Prior to our work, the fastest static algorithm for computing the shortest paths in the MPC model required $O(n)$ rounds [Hajiaghayi et al., arXiv '19]. Additionally, directly parallelizing the sequential dynamic APSP algorithm in the MPC model resulted in $\tilde{O}(n^{2+2/3})$ update rounds [Abraham et al., SODA '17], where $\tilde{O}(\cdot)$ suppresses a polylogarithmic factor. By contrast, the parallel dynamic algorithm presented in this paper requires fewer rounds to update the shortest paths or distances in the dynamic APSP problem.

Keywords dynamic graphs, all-pairs shortest paths, the MPC model

1 Introduction

In recent years, *the Massively Parallel Computation (MPC) model* has gained significant attention. Numerous distributed and parallel graph algorithms for the MPC model have been proposed [1–4]. However, most of these algorithms are designed for static graphs.

In fact, the graphs in the real world are constantly changing.Typically, the size of the real-time changes in these graphs is smaller and more localized. Employing static graph algorithms to recompute the entire graph upon each change can lead to resource inefficiency and unnecessary time overhead. To address this issue, *dynamic graph algorithms* have been designed to deal with graph changes more efficiently [5]. These algorithms can maintain a given property of a dynamic graph, answer queries quickly, and process update operations faster than recomputing. For instance, for the dynamic all-pairs shortest paths problem, it maintains the shortest path or distance between each pair of nodes in a graph. Over the years, many dynamic graph algorithms have been proposed, with most of them showing advantages over the corresponding static algorithms [6–13].

However, most studies on dynamic graph algorithms are limited to the single machine model. Moreover, a few parallel dynamic graph algorithms in the MPC model [14–16] have been proposed and shown superiority over their parallel static counterparts. For example, the round complexity for dynamic maximal matching in the MPC model is only $O(1)$ under single edge update, while the related static algorithm requires $O(\sqrt{\log n})$ rounds [16].

The *All-pairs shortest paths (APSP)* problem is a fundamental graph problem with numerous applications, such as road and transportation networks [17]. Over the years, the static version of this problem has been extensively studied in various models, including sequential, parallel, and distributed models, such as the PRAM model [18], the BSP model [19], and the MPC model [3].

1.1 Our Contributions

Despite all these mentioned above, to the best of our knowledge, there are no existing dynamic all-pairs shortest paths algorithms working in the MPC model. It will be very significant and interesting to study the dynamic APSP problem in the MPC model. We aim to design a parallel dynamic algorithm that is faster than all the existing static parallel APSP algorithms. At a high-level, our algorithm can be seen as a parallel variant of the sequential dynamic APSP algorithm proposed by Abraham et al. [7], which will be detailed in Section 1.2. The main result is as follows:

Theorem 1. Given a directed weighted graph $G = (V, E)$ with $n = |V|$ nodes and no negative cycles, in the MPC model where each processor has $O(n^{\alpha})$ memory, there is a fully dynamic randomized APSP algorithm with $O(n^{\frac{2}{3}-\frac{\alpha}{6}} \log n/\alpha)$ worst-case update rounds and $O(n^{3-\alpha/2})$ total memory, with high probability.[1)] Here, $\alpha \in (0, 1)$ is a constant. The round complexity to query the shortest path between two nodes is $O(n^{1-2\alpha})$ rounds.

We now compare our algorithm with the two most relevant results, as outlined in Table 1. Up to now, the sole exact APSP algorithm for the MPC model was proposed by Hajiaghayi et al. [22]. This algorithm computes the distance matrix of a

[1)] An event happens "with high probability" if it occurs with probability at least $1 - 1/n^c$, where $c \geq 1$ is a constant.

Received month dd, yyyy; accepted month dd, yyyy

E-mail: qshua@hust.edu.cn

Table 1 Comparing Our Parallel Fully Dynamic APSP Algorithm with the Existing Works

	Rounds	Memory	CC	EW	Query	Type
Karczmarz et al. [20]	$O(d), (d \in [1, n])$	$\tilde{O}(n^3)$	$\tilde{O}(n^3)$	Real	Distances	Deterministic
Cao et.al [21]	$O(n^{3/2+o(1)})$	$\tilde{O}(n^3)$	$\tilde{O}(n^3)$	Integer	Distances/Paths	Randomized
AEV of Hajiaghayi et al. [22]	$O(n/\alpha)$	$O(n^{3-\alpha/2})$	$O(n^3 \log n)$	Real	Distances/Paths	Deterministic
ADP of Abraham et al. [7]	$\tilde{O}(n^{2+2/3})$	$O(n^{\alpha})^*$	$\tilde{O}(n^{2+2/3})$	Real	Distances/Paths	Randomized
This work	$O(n^{\frac{2}{3}-\frac{\alpha}{6}} \log n/\alpha)$	$O(n^{3-\alpha/2})$	$O(n^3 \log n)$	Real	Distances/Paths	Randomized

Remark: (1) AEV: an extended version; ADP: a direct parallelization; CC:computation complexity; EW: edge weight (2) the round complexity of a direct parallelization of [7] in the MPC model is derived from Lemma 7.1 of [16]; (3) $*$ means that the total memory required is the sum of memory of $O(1)$ machines.

weighted graph on semiring $(+, \min)$ using the path-doubling strategy, requiring only $O(\log n)$ rounds. However, it cannot provide the shortest paths between pairs of nodes, as the path-doubling strategy only shows changes in a portion of the distance between nodes rather than all changes. If we were to extend the approach presented in [22] to compute the shortest paths, the round complexity would increase to $O(n)$.

The other work is a trivial parallel algorithm that directly parallelizes the sequential dynamic APSP algorithm of [7] in the MPC model, using the reduction method in [16]. However, the resulting algorithm would require $\tilde{O}(n^{2+2/3})$ rounds, with O(1) machines involved in each round. Furthermore, the total memory to store the data structures required in [7] is still $\tilde{O}(n^3)$. Overall, it is unrealistic and inefficient to cost $\tilde{O}(n^{2+2/3})$ rounds to update the dynamic APSP in the MPC model.

In Table 1, the computation complexity of our parallel algorithm is much higher than the parallel version of the algorithm in [7]. This is because we made a trade-off between computation complexity and rounds complexity when parallelizing the algorithm in [7]. In the MPC model, the communication overhead between processors is much more expensive than CPU computation, so we think this trade-off is reasonable.

It is important to note the tight connection between algorithms in the MPC model and the PRAM model. The round complexity of an algorithm in the MPC model is equivalent to the depth of that in the PRAM model [23]. There are several PRAM algorithms for the shortest paths problem. The most related one is a parallel APSP algorithm with $\tilde{O}(d)$ depth (where $d \in \{1, \cdots, n\}$) proposed by Karczmarz and Sankowski [20]. Although the round complexity of this algorithm in the MPC model can be as low as $O(poly(\log))$, it cannot return the path between any two nodes. This is because they used the repeated squaring algorithm (also referred to as the path-doubling strategy in [22], as mentioned earlier). Other relevant PRAM algorithms focus on the exact SSSP problem [21, 24]. They all require $O(n^{1/2+o(1)})$ depth and are limited to graphs with integer weights. Extending these parallel SSSP algorithms to solve the APSP problem in the MPC model would cause a round complexity of $O(n^{3/2+o(1)})$. By comparison, our parallel dynamic APSP algorithm is more efficient in terms of the round complexity.

Our contributions can be summarized as follows:

1. We propose the first parallel fully dynamic APSP algorithm in the MPC model (See details in Section 4).
2. Compared with the existing fastest MPC algorithm, the parallel algorithm proposed in this paper reduces the round complexity by a factor of $O(n^{\frac{1}{3}+\frac{\alpha}{6}} / \log n)$, where $\alpha \in (0, 1)$.

1.2 Technique Overview

To better understand our algorithm, we will briefly introduce Abraham et al.'s algorithm, which utilized a standard strategy from [9, 10] to design a fully dynamic APSP algorithm with worst-case update time. The algorithm consists of three components: the preprocessing procedure, the decremental procedure and the incremental procedure.

The preprocessing procedure is used to process the current graph and obtain an efficient data structure. In [7], the shortest paths between nodes were divided into $\lceil \log n \rceil$ hierarchies due to the fact that the number of edges on a path can be at most n.

In each hierarchy, a *congestion value* of zero is initially assigned to each node in the graph. *The congestion value of a node is defined as the number of shortest paths with at most 2^i edges that contain the node in the i-th hierarchy ($i \in \{1, 2, \cdots, \lceil \log n \rceil\}$).* Then, a sampling strategy introduced in [25] is employed to select a subset of nodes, forming an initial hitting set. This hitting set consists of crucial vertices used to compute the shortest path trees rooted at these nodes. These trees, in turn, are utilized to calculate the shortest distances or paths between any pair of nodes. In the i-th hierarchy, they first computed the shortest paths with at most 2^i edges from and to a node u in the sampled set with the Bellman-Ford algorithm according to the congestion value size of nodes.

Utilizing these shortest paths, the congestion value for each node of the graph and the distances between at most n^2 pairs of nodes that pass through the node u were computed. Nodes with large congestion values, not originally in the sampled set, were added to the sampled set as the final hitting set at the end of each hierarchy, after the computation of shortest paths for all sampled nodes. After iterating over all hierarchies, $\lceil \log n \rceil$ hitting sets and the corresponding distances and paths between nodes that pass through vertices of the hitting sets were obtained. Storing these distances may require up to $O(n^3 \log n)$ memory.

Then by working on the obtained data structures, the decremental procedure and the incremental procedure deal with node deletions and node insertions, respectively. The decremental procedure in [7] only needs to deal with affected nodes whose shortest paths are destroyed when nodes and their incident edges are deleted. This can be done by using the shortest paths with roots in the hitting set obtained during the preprocessing procedure above. For each affected node, a new graph with related edges is constructed, and Dijkstra's algorithm is used to compute the shortest paths from and to this node. Finally, the new distances between nodes are compared with the remaining distances to obtain the shortest distances between any two nodes.

For the insertion of nodes, a modified Floyd-Warshall algorithm is used to compute the distance matrix. Ultimately, using the method proposed in [26], Abraham et al. [7] obtained a fully dynamic APSP algorithm with a worst-case update time of $O(n^{2+2/3} \log^{4/3} n)$.

Unfortunately, employing this algorithm directly in the MPC model would result in a round complexity of $\tilde{O}(n^{2+2/3})$, which is too high and unrealistic (refer to Table 1). The main challenges are as follows:

1. The preprocessing procedure presented in [7] requires $O(n^3 \log n)$ memory to store the distances between all vertices, which is too large. Reducing the memory and round complexities while still storing these distances is a significant challenge.
2. For the decremental procedure in [7], constructing a new graph for every affected node in each hierarchy and still using Dijkstra's algorithm would result in a high round complexity.
3. In [7], there is a tight connection of the computation and comparison of shortest distances between the preprocessing procedure and the decremental algorithm. If we reduce the total memory needed for the preprocessing algorithm, it may pose new challenges to design the decremental procedure.
4. The incremental procedure in [7] uses the modified Floyd-Warshall algorithm with two-layer loops to update the node insertions one by one. However, in the MPC model, a direct implementation of this algorithm would result in an unacceptable increase in round complexity.

To address the first challenge, we remove the computation of distances between nodes during the preprocessing procedure and only store the shortest path trees. This reduces the round complexity required to compute the distances between vertices, and the memory required to store the trees is only $\tilde{O}(n^2)$ since these trees can be used to calculate these distances.

In tackling the second challenge, we only concentrate on the nodes in the hitting set associated with the affected nodes. We replace the Dijkstra algorithm with the restricted Bellman-Ford algorithm proposed in [4]. Due to resource contention, the restricted Bellman-Ford algorithm (refer to Algorithm 1) can only compute the shortest path tree of a single node at a time, potentially resulting in a significant round complexity. Subsequently, we introduce an algebraic method to reduce the round complexity and update the distances and shortest paths between nodes accurately.

This algebraic method combines the blocked Floyd-Warshall algorithm [18] and the sampling strategy of [7, 25], using the short-hop distances obtained by the restricted Bellman-Ford algorithm to compute the long-hop distances between nodes. The short-hop (or long-hop) distance means that the number of edges on the shortest path of any two nodes is small (or large).

The third challenge arises from removing the calculation of distances between nodes during the preprocessing procedure, which increases the round complexity when performing this step in the decremental procedure. To address this problem, we utilize matrix multiplication on a semiring to calculate the distances between nodes within each hierarchy and compare them between these hierarchies.

Finally, in addressing the fourth challenge, we combine the blocked Floyd-Warshall algorithm [18] with an MPC algorithm of matrix multiplication on a semiring. This approach reduces the round complexity when computing both the shortest distances and paths (see details in Section 4.3).

1.3 Related Work

Dynamic graph algorithms can be categorized into two types: fully dynamic algorithms and partially dynamic algorithms. Fully dynamic algorithms can handle both edge and node deletions and insertions, while partially dynamic algorithms can only handle one of these operations. This paper focuses on fully dynamic all-pairs shortest path (APSP) algorithms. Interested readers can refer to [8, 11, 12, 27–31] for studies on partially dynamic APSP algorithms.

Throup [10] improved the local paths method in [8] and proposed the first fully dynamic APSP algorithm for directed weighted graphs, which supports node deletions and insertions with a worst-case update time of $\tilde{O}(n^{2.75})$. This bound was broken by Abraham et al. [7] who designed a simpler randomized algorithm with $\tilde{O}(n^{2+2/3})$ worst-case update time. Subsequently, Gutenberg et al. [9] presented a deterministic data structure that improved the result of [10] to $\tilde{O}(n^{2.71})$ worst-case update time. Chechik et al. [32] later designed a faster preprocessing algorithm than that of Abraham et al. [7] and Gutenberg et al. [9] to compute the shortest paths between nodes and they obtained a deterministic algorithm with $\tilde{O}(n^{2+41/61})$ worst-case update time. More recently, Mao [33] introduced a randomized algorithm with $\tilde{O}(n^{2.5})$ worst-case update time which almost met a natural $\Omega(n^{2+1/2})$ barrier.

These studies above all utilized a technique introduced in [26] to transform a decremental algorithm (i.e., ones that can only deal with node/edge deletions) into a fully dynamic algorithm. We also employ this technique in our algorithm (See details in Section 4). Among these studies, the randomized algorithm proposed by Abraham et al. [7] is the simplest one and has the minimum sequential dependency. The preprocessing algorithms designed in [10] and [7] were both based on the work of [8], which caused a running time larger than $O(n^3)$ and needed large memory to store the possible shortest paths between all nodes. The works presented in [9] and [33] used a similar framework for the decremental algorithm, but there is a strict sequential dependency in computing the congestion value for nodes between different iterations. This increases the hardness of parallelizing these serial algorithms. Therefore, we attempt to parallelize the sequential dynamic algorithm of [7].

The works on the dynamic APSP problem in the parallel and distributed setting are few and not relevant enough to the one studied in this paper, interested readers can refer [34, 35] for details.

1.4 Organization

In Section 2, we introduce the MPC model, the problem definition, and some necessary and useful techniques. Some assumptions about the storage of nodes and edges of a given graph and a list of known subroutines are provided in Section 3. In Section 4, we present a detailed description and analysis of our parallel fully dynamic APSP algorithm. Finally, the conclusion of the paper is given in Section 5.

2 Preliminaries

2.1 The MPC Model and Complexity Measurements

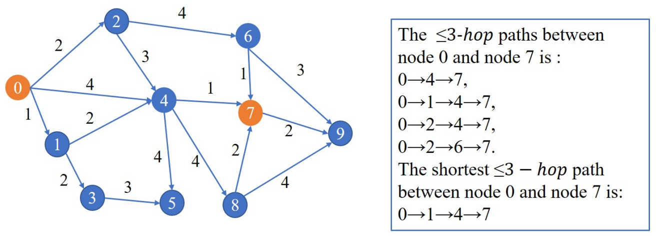
The MPC model can be traced back to the MapReduce model proposed by Karloff et al. [36], and gradually grows into the theoretical computation model [23, 37]. There are three key parameters for the MPC model: the size of input data N, the number of required processors P, and the memory size S for each processor. The relationship between these parameters is $P \cdot S = \tilde{\Theta}(N)$. In this paper, we consider the MPC model with strongly sublinear memory, which means $S = \tilde{O}(n^{\alpha})$, where $\alpha \in (0, 1)$ and n is the number of nodes in a given graph.

In the MPC model, the computation is executed in synchronous rounds. At the beginning of each round, the input data are distributed among P processors. In each round, each processor performs local computation and has no communication with other processors. At the end of each round, processors communicate with each other to obtain the data required for the next round , with the size of the message sent or received limited to at most S words in each round. Pair-wise connectivity is used for communication between processors.

The MPC model's complexity measurements include the *round* and *memory complexities*. The primary objective is reducing the round complexity of the MPC algorithms.

2.2 Problem Definitions

In this paper, we consider directed weighted graphs $G = (V, E, W)$ without negative cycles, where V denotes the set of nodes, E is the set of edges, and $W : E \to \mathbb{R}$ is a weighted function. $|V| = n$ and $|E| = m$ are the number of nodes and edges in the graph G, respectively. The weight of edge (u, v) $(u, v \in V)$ is denoted by $w(u, v)$. We focus on dense graphs where $m = \Theta(n^2)$. In the following, we give some definitions about the problem studied in this paper.



The ≤3-*hop* paths between node 0 and node 7 is :
0→4→7,
0→1→4→7,
0→2→4→7,
0→2→6→7.
The shortest ≤3 − *hop* path between node 0 and node 7 is:
0→1→4→7

Fig. 1 The left figure is a directed weighted graph with 9 nodes and 16 edges. The right figure shows the ≤ 3-hop paths between node 0 and node 7 and the shortest ≤ 3-hop path between node 0 and node 7.

Definition 1. (Parallel Fully Dynamic All Pairs Shortest Paths Problem) In the MPC model with the memory of each processor as $\tilde{O}(n^\alpha)$, given a directed weighted graph above, the update operations are deleting or inserting nodes and their incident edges. *The parallel fully dynamic all pairs shortest paths problem is to maintain the shortest distances and paths between nodes of a graph in the MPC model.*

Definition 2. Given a directed weighted graph above, $D \subseteq V$ is a subset of nodes, $G(V \backslash D)$ is a subgraph of G with edges whose nodes are both in $V \backslash D$. $G(V \cup D)$ is the extended graph that contains both nodes in V and D, along with their corresponding edges.

Definition 3. Given the directed weighted graph $G = (V, E, W)$ above, $s, t \in V$ are two different nodes, the path between s and t is defined by $\pi(s,t) = s(= v_0) \to v_1 \to \cdots \to v_{k-1} \to t(= v_k)$. The length of $\pi(s,t)$ is the sum of its edges' weights, namely $|\pi(s,t)| = \sum_{i=0}^{k-1} w(v_i, v_{i+1})$. Additionally, the shortest distance between nodes s and t is denoted as $d(s,t)$, where $d(s,t) = \min_{i=1}^{r}\{|\pi_i(s,t)|\}$ and r is the number of reachable paths between s and t.

Definition 4. Given the directed weighted graph $G = (V, E, W)$ above, $s, t \in V$ are two different nodes, the $\leq h$ hop path is defined as a path with at most h edges. The shortest $\leq h$ hop path between s and t (whose length is noted by $d^h(s,t)$) is the path with the shortest distance among all $\leq h$ hop paths from s to t.

To better understand the definition of the $\leq h$-hop path, we can refer to Fig.1 above. It is worth noting that $d(s,t)$ and $d^h(s,t)$ denote the shortest distances in graph G, while $d_{G(V\backslash D)}(s,t)$ and $d^h_{G(V\backslash D)}(s,t)$ represent the shortest distances in subgraph $G(V \backslash D)$.

2.3 Some Necessary and Useful Techniques

Next, we will introduce several techniques. Firstly, we will present the random sampling strategy, which plays a fundamental role in our parallel preprocessing procedure. This technique was originally introduced by Ullman and Yannakakis [25] and has later been widely used in designing efficient randomized algorithms for dynamic shortest paths problems [7, 11].

Lemma 1. ([25]) $G = (V, E)$ is a digraph with n nodes and $1 \leq k \leq n$. Let RS denote a random sampling subset of nodes obtained by selecting each node independently, with probability $\min\{1, \frac{c \ln n}{k}\}$ where c is a constant. Then for arbitrary path π with at most k edges in graph G, at least one of the nodes on path π belongs to RS *w.h.p.*.

Secondly, we propose a variant of the standard method [7, 9, 10, 26], which can transfer a decremental algorithm into a fully dynamic algorithm, as mentioned in Section 1:

Lemma 2. For the dynamic APSP problem in the MPC model with P processors and S words of memory per processor, the following holds: if there exists a preprocessing algorithm for the original graph $G = (V, E)$ that costs R_{pre} rounds to obtain a decremental data structure that can support any sequence of 2Δ deletions of nodes in R_{del} rounds, and an incremental algorithm can handle any sequence of Δ node insertions by updating the distance matrix and paths obtained by the decremental algorithm within R_{ins} rounds, then we can achieve a fully dynamic APSP algorithm that supports each node update in $O(R_{pre}/\Delta + R_{del} + R_{ins})$ worst-case update rounds for the MPC model.

Proof. As shown, the decremental algorithm can handle a batch of 2Δ deletions after the preprocessing algorithm. To obtain the fully dynamic algorithm, we can maintain two copies of these algorithms in parallel (which includes the preprocessing algorithm, the decremental algorithm, and the incremental algorithm). At each interval of Δ updates, we use one copy to query the result and the other one to preprocess the current graph. This means that for every Δ update, the preprocessing algorithm takes R_{pre}/Δ rounds to process the current graph, which supports at most Δ deletions for the decremental algorithm. After this copy is ready to query the results, it is Δ updates behind. Since the decremental algorithm can handle 2Δ deletions, we only need to add at most Δ deletions in the previous Δ updates to that of the current Δ. Then we can use this copy to execute the query operations and use the other copy to process the graph again.

To convert the decremental algorithm into a fully dynamic algorithm, we still have to execute an incremental algorithm that can handle Δ insertions in R_{ins} rounds. This incremental algorithm only needs to update the distance matrix and paths obtained by the decremental algorithm. Therefore, we only have to maintain this decremental data structure. □

We observe that the incremental algorithm mentioned in Lemma 2 only requires updating the distance matrix. However, we need to update the shortest paths between nodes; we will design an incremental algorithm in the MPC model that can update both the distance matrix and the shortest paths simultaneously in Section 4.3.

Finally, we introduce a useful algebraic result for matrix multiplication on semirings in the MapReduce model with strongly sublinear memory through the following lemma:

Lemma 3. ([22]) In the MapReduce model with memory of each processor as $O(n^{\alpha})$, where $0 \leq \alpha \leq 2$, the following results hold:

- The multiplication of any two $n \times n$ matrices A and B over semiring $(min, +)$ can be computed in $1 + \lceil (1 - \alpha/2)/\alpha \rceil$ rounds with $O(n^{3(1-\alpha/2)})$ processors.
- When the parameter $z \geq \alpha$, the multiplication of a matrix A with size $n \times n^z$ and a matrix B with size $n^z \times n$ over semiring $(min, +)$ can be computed in $O(1)$ rounds with $O(n^{2+z-\frac{3\alpha}{2}})$ processors.

3 Some Assumptions and Basic Subroutines

Before presenting our parallel fully dynamic APSP algorithm in detail, we specify certain assumptions regarding the storage of nodes and edges in the MPC model. Additionally, we introduce several known subroutines that can facilitate the design of our fully dynamic APSP algorithm in the MPC model.

3.1 Some Assumptions

In the MPC model, we assume that there are P processors, which are identified by the IDs $P_1, P_2, \ldots, P_P$. We assume that the IDs of processors are numbered in ascending order, i.e., P_1 has the smallest ID, and P_P has the largest ID. The nodes of a directed weighted graph $G = (V, E, W)$ are numbered sequentially as $1, 2, \ldots, n$. For each node $v \in V$, we use $N_{in}(v)$ and $N_{out}(v)$ to denote the sets of incoming and outgoing edges, respectively. Edges in $N_{in}(v)$ (or $N_{out}(v)$) are sorted by the ID of their endpoints, except node v. For example, if $(u_1, v), (u_2, v), (u_3, v) \in N_{in}(v)$, then the order of these edges in $N_{in}(v)$ is determined by the IDs of the nodes u_1, u_2, and u_3.

We use P_{nodes} to denote the set of processors (the number of which is $O(n^{1-2\alpha})$) that store the nodes of G and the information associated with each node $v \in V$. Nodes with smaller IDs are stored in processors with smaller IDs. The information for each node $v \in V$ is represented by a tuple $(i_v, P(v), P_{in}(v), P_{out}(v))$, which includes the ID of the node, the ID of the processor that stores it, and the ID ranges of the processors that store the edges in $N_{in}(v)$ and $N_{out}(v)$. We use a continuous set of processors to store the edges incident to each node v, so $P_{in}(v)$ includes the ID of the first processor that stores the first edge of $N_{in}(v)$ and the ID of the last processor that stores the last edge of $N_{in}(v)$. The same goes for $P_{out}(v)$. In Section 4.1, we will describe the other information about nodes stored in the set P_{nodes}.

Furthermore, the information about each edge in E includes the IDs of the nodes it connects, the ID of the processor that stores it, and its weight. Each edge is stored on two processors since it connects two nodes. For example, if $(u, v) \in N_{in}(v)$, then the information of edge (u, v) can be represented by a tuple $((u, v), P_{in}(v, (u, v)), i_u, i_v, P(v), P(u), \omega(u, v))$. If $(u, v) \in N_{out}(u)$, then the tuple for the information of edge (u, v) is $((u, v), P_{out}(u, (u, v)), i_v, i_u, P(u), P(v), \omega(u, v))$. Here, $P_{in}(v, (u, v))$ and $P_{out}(u, (u, v))$ indicate the ID of the processor that stores the edge (u, v) in the sets $N_{in}(v)$ and $N_{out}(u)$, respectively.

3.2 The Basic Subroutines

Now, we present a list of useful subroutines for further designing our fully dynamic APSP algorithm in the MPC model.

The Sorting (M, a, b) Algorithm [23]: Given a set M of n comparable values, the aim is to sort these values among processors with the IDs from a to b that store these values. Also, the processor with a smaller ID stores smaller values. The sorting algorithm (M, a, b, n) can be performed in $O(1/\alpha)$ rounds in the MPC model with $S = \tilde{O}(n^{\alpha})$.

The Find Minimum(M, a, b) Algorithm or **the Find Maximum(M, a, b) Algorithm** [4]: Given a set M of n comparable values which are stored in a continuous set of processors with IDs ranging from a to b. The goal of these two algorithms is to find the minimum and maximum values, respectively, which can be performed in $O(1/\alpha)$ rounds in the MPC model with $S = \tilde{O}(n^{\alpha})$.

Broadcast(x, a, b, c) [4]: Broadcasts a message x from the processor with ID a to a continuous set of n processors of

Algorithm 1 Restricted Bellman-Ford $(G, s, h, MPC(n^{\alpha}))$

```
Input: A directed graph $G = (V, E, W)$ ($|V| = n, |E| = m$) distributed among processors $P_1, P_2, \cdots, P_P$ and a source $s$.
Output: A $\leq h$ hop restricted distances to all nodes $v \in V$ from source $s$ and a tree rooted at $s$.
1: for $i = 1$ to $h$ do
2:     for $v \in V$ do
3:         Compute $d(s, v) = \min_{u \in N_{in}(v)}\{d(s, u) + w(u, v)\}$ and set $v.\pi = u$ ($u$ is the parent of node $v$)
4:         Broadcast $d(s, v)$ to processors (a continuous processors set $P_{out}(v)$) that store the outgoing edges of vertex $v$
5:                                        .                    ▷ Broadcast$(x, a, b, c)$ in Section 3.2
6:     end for
7: end for
```

the ID range(b, c). Broadcast(x, a, b, c) can be performed in $O(1/\alpha)$ rounds in the MPC model with $S = \tilde{O}(n^{\alpha})$.

Finally, we highlight the restricted Bellman-Ford algorithm in the MPC model, designed in [4], which is an essential component for our parallel preprocessing and decremental procedures. We have modified this algorithm to output the tree rooted at source s while maintaining the same round complexity.

Lemma 4. ([4]) $G = (V, E)$ is a digraph with source node $s \in V$, in the MPC model with $S = \tilde{O}(n^{\alpha})$, the restricted Bellman-Ford algorithm (Algorithm 1) can compute distances $d^h(s, v)$ of the shortest $\leq h$ hop paths from s to all $v \in V$ in $O(h/\alpha)$ rounds.

4 Fully Dynamic APSP Algorithm in the MPC model

In Section 1.2, we provide a brief overview of the fully dynamic APSP algorithm proposed in [7]. We then analyze the challenges of parallelizing this sequential algorithm in the MPC model and present our strategies for overcoming these challenges. In this section, we will provide a detailed description of our parallel fully dynamic APSP algorithm, including the design of the algorithm and complexity analysis. Specifically, the supporting subroutines and the parallel preprocessing algorithm will be presented in Section 4.1. In Section 4.2, we will design the supporting subroutines and the parallel decremental algorithm for the deletion of nodes and edges incident to these nodes. Finally, the parallel incremental algorithm will be presented in Section 4.3 for the insertion of nodes and their incident edges.

4.1 The Parallel Preprocessing Procedure

We retain the structure for solving the shortest paths trees in the preprocessing algorithm presented in [7], except for calculating the distances between nodes. In Section 1.2, we describe how the computation of shortest paths trees is divided into $\lceil \log n \rceil$ iterations. Although the iterations are independent of each other, it is unrealistic to perform these *logn* iterations simultaneously due to resource contention between processors. Thus, our main task is to parallelize the computation within each iteration of the preprocessing algorithm in the MPC model to reduce the round complexity.

To achieve this, we designed subroutines that can parallelize some important steps in Section 4.1.1. These subroutines consist of several tasks: random sampling of nodes, initializing node congestion values, identifying nodes with the largest congestion values, and computing node congestion values using the obtained shortest path trees. Additionally, we designed a subroutine in the MPC model for constructing a new graph for nodes with the largest congestion values.

We provide the details of our parallel preprocessing procedure in Section 4.1.2 with the designed subroutines. In Section 4.1.3, we analyze the round complexity of these subroutines and the preprocessing algorithm in the MPC model.

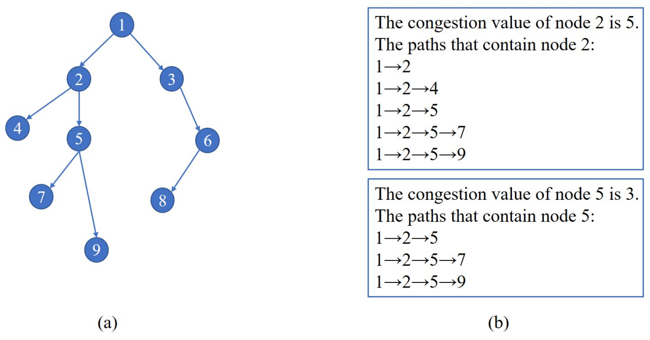
4.1.1 A Set of Supporting Subroutines

Before presenting the preprocessing procedure, we first provide a set of supporting subroutines that are crucial components of the parallel preprocessing procedure designed in Section 4.1.2.

NodesRandomSample(M, h_i, a, b): Given a set M of $|M|$ nodes from a continuous set of processors with the ID ranging from a to b (where nodes with smaller IDs are stored in processors with smaller IDs), it returns a sample node set by setting $k = h_i = 2^i$ in Lemma 1 ($1 \leq i \leq \lceil \log n \rceil$) during the i-th iteration. The selected nodes are marked as i_C, which form the hitting set in the i-th iteration.

The subroutine **NodesRandomSample**(M, h_i, a, b) is used to obtain the initial hitting set, as discussed earlier in Section 1.2. The following subroutines all involve an important concept: the congestion value of a node. Therefore, we restate the definition of the congestion value of a node, and we provide a simple example in Fig. 2 for better understanding.

Fig. 2 (a) shows a tree denoted as T_1 with a root node of 1. Fig. 2 (b) illustrates the congestion values of nodes 2 and 5 in tree T_1. To calculate the congestion value of a node in tree T_1, one can simply count the number of child nodes it has (excluding the root node 1) and add 1.



The congestion value of node 2 is 5.
The paths that contain node 2:
1→2
1→2→4
1→2→5
1→2→5→7
1→2→5→9

The congestion value of node 5 is 3.
The paths that contain node 5:
1→2→5
1→2→5→7
1→2→5→9

Fig. 2 (a) The tree T_1; (b) The congestion values of node 2 and 5 in the tree T_1.

The congestion value of a node v is the total number of shortest paths with at most $h_i = 2^i$ hops that contain v in the i-th iteration of the preprocessing procedure. In Fig. 2, we show how to use a tree rooted at node 1 to compute the congestion values of all nodes except node 1. In fact, in the i-th iteration of the preprocessing procedure, we will obtain several shortest path trees rooted at different nodes. Therefore, we must compute the congestion value of a node (apart from these root nodes) with all these shortest path trees one by one. The reason of computing the congestion value of a node will be illustrated after the description of the subroutine **Access**$(v, P(v))$.

InitiaCongestValue(M, cv, i, a, b): For a given set M of nodes among a continuous set of processors with IDs from a to b, it initializes the congestion value $cv(v)$ of each node v in the set M to 0 at the beginning of the i-th iteration.

FindMaximum(M, a, b): Uses the Find Maximum (M, a, b) algorithm in Section 3.2 to find the node v with the maximum congestion value in the nodes set M among processors with ID range (a, b). This node v is marked as i_R in the i-th iteration.

CompCongValue(T, v, cv, a, b): First, it computes the congestion values cv of nodes based on the shortest $\leq h_i$ hop paths tree T rooted at node v among processors with ID range (a, b). Then, it counts the number of children nodes of a node in the tree T to determine its congestion value. Sort the nodes in M based on their ID size using the sorting (M, a, b) algorithm. Finally, it sends the congestion values of nodes to processors that store the corresponding nodes and compute the new congestion values using the received data.

Access$(v, P(v))$: Sets node v and the edges incident to it as inactive, and notify the processors in the set $P(v)$, which store the edges incident to v, to set these edges as inactive as well. It then uses the Broadcast(x, a, b, c) algorithm in Section 3.2 to send a message to processors whose nodes (except for node v) are endpoints of edges incident to v. Finally, it notifies the processors that store edges connecting to node v to set these edges as inactive.

The subroutine **Access**$(v, P(v))$ is used to deactivate the edges incident to node v, which prevents the restricted Bellman-Ford algorithm executed from another node u from visiting these edges. This greedy strategy limits the maximum congestion value of each node and minimizes the impact of updates, which is the significance of setting a congestion value for each node.

4.1.2 The Preprocessing Procedure in the MPC Model

In Section 4.1.1, it is necessary to assign labels to the randomly sampled nodes and set a congestion value for each node, in addition to the information outlined in Section 3.

In the following, we will describe how to mark the congestion value $cv(v)$ for a given node v, as well as the sampling process results for each node in V. As the preprocessing procedure runs independently between iterations, and $cv(v)$ is initialized to 0 at the beginning of each iteration, the symbol $cv(v)$ can be utilized to represent the congestion value of node v for $\lceil \log n \rceil$ iterations. However, we require $\lceil \log n \rceil$ markings (as outlined in **NodesRandomSample**(M, h_i, a, b)) for the random sampling of each node. This is because these markings will be used to obtain the final hitting set. The total memory required for storing the nodes in set V is $\tilde{O}(n)$, and we can simply utilize $O(n^{1-2\alpha})$ processors to store the information for all nodes.

Algorithm 2 presents the pseudo-code for our parallel preprocessing algorithm. At each iteration, we consider the shortest paths with at most $h_i = 2^i$ edges and set an empty node set R_i which is used to get the final hitting set. Firstly, we execute **NodesRandomSample**(V, h_i, P_{nodes}) among the processors set P_{nodes} to obtain a sample node set C_i. These sampled nodes are marked as i_C in the i-th iteration. The nodes in the sampled node set C_i form the initial hitting set. Subsequently, we initialize the congestion values cv for all nodes in V with **InitialCongestValue**(V, cv, i, P_{nodes}).

Next, we construct the new hitting set R_i for each iteration

Algorithm 2 The Preprocessing Procedure in MPC(n^α)

```
Input: A directed graph $G = (V, E, W)$.
Output: The new hitting set $R_i$ and the shortest $\le h_i$ paths trees $T_i^{v\rightarrow}$ and $T_i^{\leftarrow v}$ for $1 \le i \le \lceil \log n \rceil$, $v \in R_i$ ($T_i^{v\rightarrow}$ and $T_i^{\leftarrow v}$ is out tree and in tree with at most $h_i$
    edges rooted at node $v$)
 1: for $i = 1$ to $\lceil \log n \rceil$ do
 2:     $h_i \leftarrow 2^i$,$R_i \leftarrow \emptyset$
 3:     $C_i \leftarrow$NodesRandomSample($V, h_i, P_{nodes}$);            ▷ Get a sampled node set
 4:     Execute InitiaCongestValue($V, cv, i, P_{nodes}$)            ▷ Initialize the congestion values for all nodes
 5:     while $C_i \backslash R_i \neq \emptyset$ do
 6:         $v \leftarrow$ FindMaximum($C_i \backslash R_i, P_{nodes}$)            ▷ Choose the sampled vertex with the largest congestion value
 7:         $R_i \leftarrow R_i \cup \{v\}$
 8:         ($T_i^{v\rightarrow}, P(T_i^{v\rightarrow})$) ← Restricted Bellman-Ford ($G(V\backslash R_i), v, h_i, MPC(n^\alpha)$)            ▷ Compute the shortest path tree from $v$
 9:         ($T_i^{v\leftarrow}, P(T_i^{v\leftarrow})$) ← Restricted Bellman-Ford ($\overleftarrow{G}(V\backslash R_i), v, h_i, MPC(n^\alpha)$)            ▷ Compute the shortest path tree to $v$
10:         Execute CompCongValue($T_i^{v\rightarrow}, v, cv, P(T_i^{v\rightarrow})$)            ▷ Compute the congestion value for each node
11:         Execute CompCongValue($T_i^{v\leftarrow}, v, cv, P(T_i^{v\leftarrow})$)            ▷ Compute the congestion value for each node
12:         Access($v, P(v)$)            ▷ Make the edges incident to node $v$ inactive
13:         if $V\backslash(C_i \cup R_i) \neq \emptyset$ then
14:             $v \leftarrow$ FindMaximum($V\backslash(C_i \cup R_i), P_{nodes}$)            ▷ Choose the non-sampled node with the biggest congestion value
15:             $R_i \leftarrow R_i \cup \{v\}$
16:             ($T_i^{v\rightarrow}, P(T_i^{v\rightarrow})$) ← Restricted Bellman-Ford ($G(V\backslash R_i), v, h_i, MPC(n^\alpha)$)
17:             ($T_i^{v\leftarrow}, P(T_i^{v\leftarrow})$) ← Restricted Bellman-Ford ($\overleftarrow{G}(V\backslash R_i), v, h_i, MPC(n^\alpha)$)
18:             Execute CompCongValue($T_i^{v\rightarrow}, v, cv, P(T_i^{v\rightarrow})$)
19:             Execute CompCongValue($T_i^{v\leftarrow}, v, cv, P(T_i^{v\leftarrow})$)
20:             Access($v, P(v)$)
21:         end if
22:     end while
23: end for
```

and compute the shortest path trees from and to nodes in R_i. We start by selecting the node v in the sampled node set C_i with the largest congestion value by executing **FindMaximum(**$C_i \backslash R_i$, P_{nodes}**)** (refer to line 5 in Algorithm 2). We then add this sampled node v to R_i and compute the $\le h_i$ hop shortest path trees $T_i^{v\rightarrow}$ and $T_i^{v\leftarrow}$ from and to node v using the **Restricted Bellman-Ford(**$G(V\backslash R_i), v, h_i, MPC(n^\alpha)$**)** algorithm (refer to Algorithm 1) and the **Restricted Bellman-Ford(**$\overleftarrow{G}(V\backslash R_i), v, h_i$, $MPC(n^\alpha)$**)** algorithm (refer to Algorithm 1), respectively. This process is from line 7 to line 9 in Algorithm 2. Here, $\overleftarrow{G} = (V, \overleftarrow{E})$ denotes the reverse graph of G. The resulting tree data structures $T_i^{v\rightarrow}$ and $T_i^{v\leftarrow}$ are then stored in a continuous set $P(T_i^{v\rightarrow})$ and $P(T_i^{v\leftarrow})$ of processors.

After that, we execute the two subroutines **ComputeCongValue(**$T_i^{v\leftarrow}, v, cv, P(T_i^{v\leftarrow})$**)** and **ComputeCongValue(**$T_i^{v\rightarrow}, v, cv$, $P(T_i^{v\rightarrow})$**)** to compute the congestion values for all nodes except those in R_i using the shortest path trees computed above (refer to line 10 - line 11 in Algorithm 2). Here, $P(T_i^{v\rightarrow})$ (or $P(T_i^{v\leftarrow})$) is a continuous set of processors that stores tree $T_i^{v\rightarrow}$ (or $T_i^{v\leftarrow}$). Finally, we deactivate the sampled node v and its incident edges using the subroutine **Access(**$v, P(v)$**)** (refer to line 12 in Algorithm 2). This operation can restrict the size of the congestion value of a node and reduce the influence caused by the deletion and insertion of nodes.

As mentioned above, the new hitting set R_i comprises both the sampled and non-sampled nodes. Similar to selecting the sampled nodes, we select the non-sampled node with the largest congestion value and add it to R_i (refer to line 14 - line 15 in Algorithm 2). The subsequent steps for the non-sampled node are the same as those for the sampled node (refer to line 16 - line 20 in Algorithm 2).

After visiting the shortest path trees for all sampled nodes, we obtain the new hitting set R_i and the corresponding shortest path trees at the end of each iteration. Additionally, we store extra information for each tree, such as the node ID, the processor ID that stores the node, the processor ID that stores edges within the tree, and the distances from the source v to other nodes in tree $T_i^{v\rightarrow}$ (or $T_i^{v\leftarrow}$). This information is necessary for updating the distances and paths in the parallel decremental algorithm presented in Section 4.2.

Compared to the sequential fully dynamic APSP algorithm described in [7], we can reduce the total memory from $\tilde{O}(n^3)$ of [7] to $\tilde{O}(n^2)$ in the MPC model. This is achieved by deferring the computation of shortest distances and the sorting of distances for each pair of nodes.

4.1.3 Complexity Analysis

We now analyze the round complexity of Algorithm 2 (the preprocessing procedure). To begin, we present a lemma regarding the round complexity of the supporting procedures outlined in Section 4.1.1 below:

Lemma 5. (The Round Complexity of The supporting subroutines) For Algorithm 2 in the MPC model with strongly sublinear memory ($S = \tilde{O}(n^\alpha)$), the number of rounds for **NodesRandomSample**(V, h_i, P_{nodes}) and **InitiaCongestValue**(V, cv, i, P_{nodes}) are $O(1)$, while the round complexity for **FindMaximum** (M, a, b), **CompCongValue**(T, v, cv, a, b), and **Access**($v, P(v)$) is $O(1/\alpha)$ in each iteration of each *while* loop.

Proof. In each iteration of Algorithm 2, the subroutines **NodesRandomSample**(V, h_i, P_{nodes}) and **InitiaCongestValue**(V, cv, i, P_{nodes}) can be executed locally in a single round. We now analyze the round complexity of **CompCongValue**(T, v, cv, a, b) within each while loop of Algorithm 2. As we utilize a continuous set of processors to store the paths, it is possible to compute the congestion values of nodes in the shortest paths locally. Sorting the nodes based on their IDs requires $O(1/\alpha)$ rounds and sending congestion values of nodes in the shortest paths to the processors set P_{nodes} only takes one round. The **Access**($v, P(v)$) subroutine, which is employed to make certain edges inactive, only requires $O(1/\alpha)$ rounds by the Broadcast operation in Section 3.2. Additionally, **FindMaximum**(M, a, b) can be obtained by executing the Find Maximum(M, a, b) algorithm in Section 3.2, which has a round complexity of $O(1/\alpha)$ rounds. □

Lemma 6. (The round complexity of the parallel preprocessing procedure) The preprocessing procedure (Algorithm 2) designed for a fully dynamic APSP problem in the MPC model takes $O(n \log^2 n/\alpha)$ rounds to process a weighted digraph G with non-negative cycles with high probability ($w.h.p.$), where n is the number of nodes in graph G.

Proof. As we can see, the subroutines **NodesRandomSample**(V, h_i, P_{nodes}) and **DeleteNodes**(D, P_{nodes}) (refer to Lemma 5) require $O(1)$ rounds and Algorithm 2 performs $\lceil \log n \rceil$ iterations. The round complexity of the **NodesRandomSample**(V, i, P_{nodes}) and **DeleteNodes**(D, P_{nodes}) subroutines (refer to Lemma 5) used in Algorithm 2 is $O(\log n)$.

In each iteration, there is a *while* loop in the preprocessing procedure that computes the hitting set R_i and the shortest $\leq h_i$ hop paths trees. The size of the hitting set R_i is $|R_i| \leq 2|C_i| = O(n \log n/h_i)$, as shown in [7], where C_i is the set of sampled nodes (refer to Lemma 1). The supporting procedures used within this *while* loop are **Access**($v, P(v)$), **CompCongValue**(T, v, cv, a, b), **FindMaximum**(M, a, b), and the Restricted Bellman-Ford algorithm. Among these subroutines, in the i-th iteration, the last one requires $O(h_i/\alpha)$ rounds (refer to Lemma 4), while the others require $O(1/\alpha)$ rounds (refer to Lemma 5). Thus, the while loop within each iteration requires $O(|R_i| \times h_i/\alpha) = O(n \log n/\alpha)$ rounds.

Since there are $\lceil \log n \rceil$ iterations in Algorithm 2 and we use the random sampling strategy, the number of rounds required for the preprocessing procedure to process a given graph G is $O(n \log^2 n/\alpha)$ with high probability.

Regarding the computation complexity of Algorithm 2, the most expensive operation is executing the restricted Bellman-Ford algorithm for each node in the final hitting set R. In the i-th iteration, $|R_i| = O(n \log n/h_i)$, the computation complexity required is $O(|R_i| n^2 h_i) = O(n^3 \log n)$. Therefore, the total computation complexity for Algorithm 2 is $O(n^3 \log^2 n)$. □

4.2 The Parallel Decremental Procedure

As described in Section 1.2, the primary objective of the parallel decremental algorithm is to identify the affected nodes whose shortest paths have been destroyed and then recompute the shortest paths for those nodes. Next, we compute the distances by using the remaining shortest paths obtained from the parallel preprocessing algorithm and compare them to obtain the shortest distances and paths between the nodes.

To minimize the number of rounds required, we utilize an algebraic method which utilizes the short-hop distances obtained from the restricted Bellman-Ford algorithm to compute the long-hop distances between nodes. We then leverage an MPC algorithm that computes matrix multiplication on the semiring (as explained in Lemma 3) to compute and compare the distances between the nodes.

4.2.1 A Set of Supporting Subroutines

Similar to the preprocessing procedure described above, we will represent some supporting subroutines for the decremental procedure of the dynamic APSP problem below:

Delete($D, P(D)$): Given a set D of nodes, $P(D)$ represents the set of processors that stores edges incident to nodes in D, the goal is to remove the information of edges incident to nodes in D. Then processors in $P(D)$ notify the other processors that store the corresponding edges to delete them.

DeleteNodes(D, P_{nodes}): Deletes the information of nodes in the set D from the set P_{nodes} of processors.

We require information on the deleted nodes to facilitate the design of the parallel decremental algorithm, so we separate the deleted nodes and their incident edges.

Check(R_i, i, d_i, A_i): In the i-th iteration, it checks whether the set R_i contains deleted nodes and if the trees rooted at nodes in R_i also contain deleted nodes (except the nodes in R_i). Specifically, if the set R_i contains deleted nodes, it removes those nodes along with their associated shortest path trees, and finally records the number of deleted nodes in R_i as d_i (where d_i is initially 0). If the tree rooted at a node v in R_i contains deleted nodes (except the node v), it removes the deleted nodes and their children in the tree $T_i^{v\rightarrow}$ (or $T_i^{v\leftarrow}$) and adds the node v to the new set A_i (where A_i is initially an empty set). If no deleted nodes are found, the trees remain unchanged. To facilitate this process, the nodes can be sorted based on their IDs and sent from the processor set P_{nodes} to the relevant processors for checking in the corresponding trees $T_i^{v\rightarrow}$ (or $T_i^{v\leftarrow}$). This is easier since each node knows its father node in the tree.

The purpose of **Check**(R_i, i, d_i, A_i) is to identify the affected nodes and remove them from the shortest path trees that have their roots in the hitting set during each iteration.

ActivateEdges$(G(V\backslash D))$: Activates edges and nodes in $G(V\backslash D)$.

In Section 4.1, we temporarily deactivate certain edges during the execution of Algorithm 2 (the parallel preprocessing algorithm) to construct the hitting set R_i ($i \in 1, \cdots, \lceil \log n \rceil$). However, when designing our parallel decremental algorithm, we need to reactivate these edges to use them for computation.

QueryDistanceMatrix(D, A, B, C): Queries the distance matrix D whose rows and columns correspond to the IDs of nodes in the set A and the set B and get a matrix C with size $|A| \times |B|$, which stores the distances from the nodes in the set A to the nodes in the set B.

BlockedFloydWarshall(A): Processors that store matrix A communicate with each other to compute APSP using the blocked Floyd-Warshall algorithm (c.f. [18]) in the MPC model.

MMOnSemiring(A, B, C): Computes matrix multiplication of two matrices A and B on semiring in the MPC model using the algorithm presented in Lemma 3. We then compare the resulting distance matrix with matrix C using the min operation on semiring. The final distance matrix is denoted as C.

To enhance comprehension of the subroutine **MMOnSemiring**(A, B, C), we provide a basic example below.

Example 1. Given a directed weighted graph G with nodes $\{v_1, v_2, v_3, v_4, v_5\}$, the hitting set for this graph is $R_i = \{v_3, v_4\}$ in a iteration. The restricted Bellman-Ford algorithm is then executed to generate the distance matrix A below:

$$\begin{array}{c|ccccc} & v_1 & v_2 & v_3 & v_4 & v_5 \\ \hline v_1 & 0 & \infty & 1 & 3 & \infty \\ v_2 & \infty & 0 & 2 & \infty & \infty \\ v_3 & \infty & \infty & 0 & 5 & 1 \\ v_4 & \infty & 4 & \infty & 0 & 3 \\ v_5 & \infty & \infty & \infty & \infty & 0 \end{array}$$

$$A = \begin{bmatrix} 0 & \infty & 1 & 3 & \infty \\ \infty & 0 & 2 & \infty & \infty \\ \infty & \infty & 0 & 5 & 1 \\ \infty & 4 & \infty & 0 & 3 \\ \infty & \infty & \infty & \infty & 0 \end{bmatrix}$$

$$D = A \star A = \begin{bmatrix} 0 & 7 & 1 & 3 & 2 \\ \infty & 0 & 2 & 7 & 3 \\ \infty & 9 & 0 & 5 & 1 \\ \infty & 4 & 6 & 0 & 3 \\ \infty & \infty & \infty & \infty & 0 \end{bmatrix}$$

The distances from the nodes in the hitting set $R_i = \{v_3, v_4\}$ to the nodes in the graph G are $d(v_3, v_4) = 5, d(v_3, v_5) = 1$, $d(v_4, v_2) = 4$, and $d(v_4, v_5) = 3$. On the other hand, the distances to the nodes in the hitting set $R_i = \{v_3, v_4\}$ from the nodes in graph G are $d(v_1, v_3) = 1, d(v_2, v_3) = 2, d(v_1, v_4) = 3$, and $d(v_3, v_4) = 5$.

By computing $A \star A$, where $\star$ represents the computation on semiring with the operation (min, +), the distance matrix D between 5 nodes is obtained. This is equivalent to computing $d(v_i, v_j) = \min_k \{d(v_i, v_k), d(v_k, v_j)\}$ ($i, j \in \{1, 2, 3, 4, 5\}$, $k \in \{3, 4\}$).

The example above clearly demonstrates that computing the distances between nodes using matrix multiplication on a semiring is more convenient. This method outperforms the direct computation of distances between each pair of nodes individually, leading to a reduction in the round complexity needed for our decremental algorithm in the MPC model.

4.2.2 The Decremental Procedure in the MPC Model

In this section, we will present the parallel decremental procedure for solving the dynamic APSP problem below:

Algorithm 3 **The Decremental Procedure in MPC(n^{α})**

```
Input: A directed graph $G = (V, E, W)$, the new hitting set $R_i$ and the shortest $\leq h_i$ paths trees $T_i^{v\rightarrow}$ and $T_i^{\leftarrow v}$ for $1 \leq i \leq \lceil \log n \rceil$, deleted nodes set $D$.
Output: The shortest paths distance matrix $Dist$
 1: $h = \sqrt{n^{1-\alpha/2} \log n / |D|}$
 2: Execute Delete($D, P(D)$)    ▷ Delete the incident edges of nodes in set $D$
 3: Execute ActiveEdges($G(V \backslash D)$)
 4: for $i = 1$ to $\lceil \log h \rceil$ do
 5:     $h_i \leftarrow 2^i$
 6:     Execute Check($R_i, i, d_i, A_i$)    ▷ Check the affected nodes by the shortest paths with roots in the hitting set $R_i$
 7:     if $d_i > 0$ then
 8:         $B_i \leftarrow$ NodesRandomSample($V \backslash (R_i \cup D), d_i, P_{nodes}$)    ▷ Sample the nodes to complete the hitting set
 9:     end if
10:     for each node $v \in B_i \cup A_i$ do
11:         $(T_i^{v\rightarrow}, P(T_i^{v\rightarrow})) \leftarrow$ Restricted Bellman-Ford ($G(V \backslash D), v, h_i, MPC(n^{\alpha})$)
12:         $(T_i^{s\leftarrow}, P(T_i^{s\leftarrow})) \leftarrow$ Restricted Bellman-Ford ($\overleftarrow{G}(V \backslash D), v, h_i, MPC(n^{\alpha})$)
13:     end for
14:     Execute MMOnSemiring($M_i, M_i, Dist$)
15: end for
16: $H \leftarrow$ NodesRandomSample($V \backslash D, h, P_{nodes}$)
17: Execute QueryDistanceMatrix($Dist, H, H, Dist_H$)
18: Execute BlockedFloydWarshall($Dist_H$)    ▷ Compute $(Dist_H)^{\star|H|}$
19: Execute QueryDistanceMatrix($Dist, V \backslash D, H, Dist_{V \backslash D, H}$) and QueryDistanceMatrix($Dist, H, V \backslash D, Dist_{H, V \backslash D}$)
20: Set a null matrix $N$ with size $|V \backslash D| \times |H|$
21: Execute MMOnSemiring($Dist_{V \backslash D, H}, Dist_H, N$)    ▷ Compute $Dist_{V \backslash D, H} \star Dist_H$
22: Execute MMOnSemiring($N, Dist_{H, V \backslash D}, Dist$)    ▷ Compute $Dist_{V \backslash D, H} \star Dist_{H, V \backslash D}$
23: for $i = \lceil \log h \rceil + 1$ to $\lceil \log n \rceil$ do
24:     Execute Check($R_i, i, d_i, A_i$)    ▷ Find the shortest paths trees remained caused by the opeartion of node deletions
25:     MMOnSemiring($M_i, M_i, Dist$)
26: end for
27: Execute DeleteNodes($D, P_{nodes}$)
```

To begin, we set $h = \sqrt{n^{1-\alpha/2} \log n/|D|}$, and execute **Delete**(D, $P(D)$) to delete the correlated edges incident to these nodes. For the initial $\lceil \log h \rceil$ iterations, we recompute the shortest paths between nodes to update the distance matrix $Dist$ (refer to lines 4-15 in Algorithm 3).

Specifically, in each iteration, we begin by checking the affected nodes and trees using the subroutine **Check**(R_i, i, d_i, A_i) (refer to line 6 of Algorithm 3). Then, we resample nodes in $V\backslash(R_i \cup D)$ to avoid the number of deleted nodes in R_i is too large and execute **Restricted Bellman-Ford** ($G(V\backslash D), v, h_i, MPC(n^\alpha)$) (refer to Algorithm 1) and **Restricted Bellman-Ford** ($\overleftarrow{G}(V\backslash D), v, h_i, MPC(n^\alpha)$) (refer to Algorithm 1), which yields the $\leq h_i$ shortest path trees $T_i^{v\rightarrow}$ and $T_i^{v\leftarrow}$.

Next, by executing **MMOnSemiring**($M_i, M_i, Dist$), we obtain the final distance matrix between the remaining $(n - |D|)$ pairs of nodes in each iteration. M_i represents the distance matrix obtained by combining the results of the **Restricted Bellman-Ford** ($G(V\backslash D), v, h_i, MPC(n^\alpha)$) algorithm, the **Restricted Bellman-Ford** ($\overleftarrow{G}(V\backslash D), v, h_i, MPC(n^\alpha)$) algorithm, and the shortest paths trees with at most h_i edges from and to the nodes in the set $B_i \cup A_i$. Moreover, for the procedure **MMOnSemir-ing**($M_i, M_i, Dist$), extra processors are invoked to compute the matrix multiplication on semiring (see details in Section 4.2.3).

After that, we get the shortest path trees with at most h edges and the corresponding distance matrix $Dist$ between $|V\backslash D|$ nodes. Then, we utilize this distance matrix $Dist$ to compute the distance between any two nodes in $V\backslash D$, with the number of edges between the two nodes is at least h (refer to lines 16-22 of Algorithm 3).

Concretely, we are inspired by the method in [38], which uses short-hop distances between nodes to calculate long-hop distances. First, we sample a random vertex subset H with size $O(nlogn/h)$ through **NodesRandomSample**($V\backslash D$, h, P_{nodes}). Then, we execute **QueryDistanceMatrix**($Dist, H, H, Dist_H$) to query the distance matrix $Dist$, obtained from the previous $\lceil \log h \rceil$ iterations, to get the shortest distance submatrix $Dist_H$ with at most h edges between all node pairs in H. Next, we compute $(Dist_H)^{\star|H|}$ (we still use $Dist_H$ to denote the finial result) by **BlockedFloydWarshall**($Dist_H$), to obtain the shortest distances between all node pairs in H.

Subsequently, we utilize **QueryDistanceMatrix**($Dist, V\backslash D, H, Dist_{V\backslash D,H}$) to query the distance matrix $Dist$ to obtain the distance submatrix $Dist_{V\backslash D,H}$ using at most h edges, from nodes in set $V\backslash D$ to nodes in H. Similarly, we execute **QueryDistanceMatrix**($Dist, H, V\backslash D, Dist_{H,V\backslash D}$) and obtain the distance submatrix $Dist_{H,V\backslash D}$ with at most h edges, from nodes in H to nodes in $V\backslash D$. In line 20 of Algorithm 3, we set an empty matrix N of size $|V\backslash D| \times |H|$ to facilitate the computation of line 21 in Algorithm 3. To compute the shortest distance between all node pairs in V using at least h edges, we sequentially execute **MMOnSemiring**($Dist_{V\backslash D,H}, Dist_H, N$) and **MMOnSemiring**($N, Dist_{H,V\backslash D}, Dist$). These subroutines are used to compute $Dist_{V\backslash D,H} \star Dist_H \star Dist_{H,V\backslash D}$ (refer to Lemma 1) and compare it with the previously existing distance matrix $Dist$.

For the remaining iterations from lines 23-26 in Algorithm 3, we compute and compare the distances using the shortest path trees left by the preprocessing algorithm. At the end of the decremental procedure in Algorithm 3, the procedure **DeleteNodes**(D, P_{nodes}) is performed to delete the information of nodes in set D. This completes the whole process of the node deletion operations.

We will present a detailed analysis of the round complexity and the total memory required for Algorithm 3 in Section 4.2.3.

Furthermore, to determine the *shortest paths* between any two nodes, we initialize an empty matrix Π_{Dist} with size $n \times n$ to update the shortest paths after the execution of lines 14 and 18 of Algorithm 3. This can be achieved with $O(1 + \lceil(1 - \alpha/2)/\alpha\rceil)$ rounds and $O(n^{3-\alpha/2})$ total memory, since we only need to know the intermediate node that minimizes the distance between two nodes. Thus, the memory required for storing Π_{Dist} is $O(n^2)$. (The shortest path trees rooted at these intermediate nodes are already known during the computation of the decremental procedure in Section 4.2.)

Additionally, to obtain the shortest paths between nodes of lines 16-22 of Algorithm 3, we set $|H|$ empty matrices with size $|H|\times|H|$. These empty matrices are used to record the intermediate nodes during the execution of **BlockedFloydWarshall**($Dist_H$), which causes $O(|H|^3)$ memory (the number of rounds and memory are determined later in Section 4.2.3). When executing lines 21 and 22 of Algorithm 3, we also set two empty matrices with size $n \times n$ to store the intermediate nodes for each pair of nodes in V, which induces $O(1 + \lceil(1 - \alpha/2)/\alpha\rceil)$ rounds and $O(n^{3-\alpha/2})$ total memory.

4.2.3 Complexity Analysis

Lemma 7. For Algorithm 3 in the MPC model with strongly sublinear memory $S = \tilde{O}(n^\alpha)$, the following results hold:

- **DeleteNodes**(D, P_{nodes}) has a round complexity of $O(1)$.

- **Delete**$(D, P(D))$, **Check**(R_i, i, d_i, A_i), and **ActivateEdges**$(G(V\backslash D))$ have a round complexity of $O(1/\alpha)$.
- **MMOnSemiring**(A, B, C) has a round complexity of $O(1 + \lceil(1-\alpha/2)/\alpha\rceil)$ in each iteration.
- **QueryDistanceMatrix**(D, A, B, C) has a round complexity of $O(1/\alpha)$.
- **BlockedFloydWarshall**$(Dist_H)$ has a round complexity of $O(n^{1-\alpha/2} logn/(h\alpha))$

Proof. As the subroutine **DeleteNodes**(D, P_{nodes}) can be locally completed within one round during each iteration of Algorithm 3, its round complexity is $O(1)$. On the other hand, the main operations for **Delete**$(D, P(D))$, **Check**(R_i, i), and **ActivateEdges**$(G(V\backslash D))$ are all sorting and broadcast (as explained in Section 3.2), which cause a round complexity of $O(1/\alpha)$. The round complexity for **MMOnSemiring**(A, B, C) can be obtained by applying Lemma 3.

As for **QueryDistanceMatrix**(D, A, B, C), the IDs of the sampled nodes in A and B are determined by the processors set P_{nodes}, which can be completed in $O(1/\alpha)$ rounds. Then, to query the distances from nodes in A to nodes in B, we construct two matrices M_A with size $|A| \times n$ and M_B with size $n \times |B|$. In Section 3.1, we numbered vertices in V as $1, 2, \cdots, n$. We sort the nodes in A (or B) in ascending order. If the node with ID i ($i \in \{1, 2, \cdots, n\}$) in A is in the i_A position, then the node corresponds to the i_A row in M_A. Then, we set the element of the i_A-th row and i-th column of M_A as 1; otherwise, it is zero.

The construction of matrix M_B is similar to matrix M_A. Therefore, we only need to compute the distance submatrix $C = M_A \times D \times M_B$ using Lemma 3 (the general matrix multiplication is the same as the matrix multiplication on semiring). The round complexity is $O(1/\alpha)$.

The subroutine **BlockedFordWarshall**$(Dist_H)$ divides the matrix $Dist_H$ with size $|nlogn/h| \times |nlogn/h|$ into submatrices with size $n^{\alpha/2} \times n^{\alpha/2}$ (the memory of each processor in the MPC model is $O(n^\alpha)$). In this case, we require $n^{1-\alpha/2} \log n/h$ iterations to complete the computation of this subroutine. Within each iteration, as the submatrix multiplication on semiring can be computed in a single processor, there are only broadcast operations that send a submatrix to other processors. Therefore, the round complexity for **BlockedFloydWarshall**$(Dist_H)$ is $O(n^{1-\alpha/2} \log n/(\alpha h))$. □

To better understand the process of **QueryDistanceMatrix**(D, A, B, C), we provide a straightforward example below:

Example 2. Given a directed weighted graph G with nodes $\{u_1, u_2, u_3, u_4, u_5\}$ and the distance matrix D between the five nodes below:

$$D = \begin{array}{c c} & \begin{array}{ccccc} u_1 & u_2 & u_3 & u_4 & u_5 \end{array} \\ \begin{array}{c} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \end{array} & \begin{pmatrix} 0 & 2 & 1 & 3 & 5 \\ 3 & 0 & 2 & 4 & 3 \\ 6 & 1 & 0 & 5 & 1 \\ 3 & 4 & 5 & 0 & 3 \\ 2 & 1 & 4 & 2 & 0 \end{pmatrix} \end{array}$$

The IDs of the five nodes are numbered as their subscripts. We want to query the distances from nodes in set $A = \{u_2, u_3\}$ to nodes in set $B = \{u_1, u_2, u_4\}$. According to the proof of the Lemma 7, the matrices M_A and M_B constructed are as following:

$$M_A = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}, \quad M_B = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

Then we can compute the distance submatrix $C = M_A \times D \times M_B$ from nodes in A to nodes in B as following:

$$C = \begin{bmatrix} 3 & 0 & 3 \\ 6 & 1 & 1 \end{bmatrix}$$

Lemma 8. (The round complexity of the parallel decremental procedure) Given a weighted digraph G, the decremental procedure (Algorithm 3) can process a set of deletion nodes D in the MPC model with $O(\sqrt{n^{1-\alpha/2}|D| \log n}/\alpha)$ rounds for the fully dynamic APSP problem, where n is the number of nodes in G, $h = \sqrt{n^{1-\alpha/2} \log n/|D|}$, and a is a constant. Additionally, the total memory required to execute Algorithm 3 is $O(n^{3-\alpha/2})$.

Proof. Algorithm 3 comprises $\lceil \log n \rceil$ iterations, with only $\lceil \log h \rceil$ iterations used to recompute the shortest paths for the hitting set R_i. During the initial $\lceil \log h \rceil$ iterations, the most costly operation is the Restricted Bellman-Ford algorithm, which costs $O(h_i/\alpha)$ rounds for each node $v \in B_i \cup A_i$ in each iteration. Since $|B_i \cup A_i| \leq |D|$ (the affected nodes in R_i cannot be larger than the number of deleted nodes), the number of round complexity to compute these trees rooted at these affected nodes is at most $O(|D|h_i/\alpha)$ rounds in each iteration. Although the round complexity for **MMOnSemiring**(A, B, C) is $O(1 + \lceil(1-\alpha/2)/\alpha\rceil)$, it comes at the expanse of increasing the total memory to $O(n^{3-\alpha})$ ($\alpha \in (0, 1)$) according to Lemma

3. For the initial $\lceil \log h \rceil$ iterations, the round complexity is $O(\sum_{i=0}^{\lceil \log h \rceil} |D| h_i/\alpha) = O(|D|h/\alpha)$.

In lines 16-22 of Algorithm 3, by Lemma 7, the most costly operation is executing **BlockedFloydWarshall**($Dist_H$), which requires $O(n^{1-\alpha/2} \log n/(\alpha h))$ rounds. Starting from iteration $i = \lceil \log h \rceil$ and continuing until $i = \lceil \log n \rceil$, the round complexity is $O(\log n/\alpha)$, which is caused by **MMOnSemiring**(A, B, C) and **Check**(R_i, i), ect. (See Lemma 8.)

In summary, by setting $h = \sqrt{\frac{n^{1-\alpha/2} \log n}{|D|}}$, the decremental procedure requires $O(\sqrt{n^{1-\alpha/2}|D| \log n}/\alpha)$ rounds.

Now, we analyze the computation complexity of Algorithm 3. On the one hand, as we recompute the shortest paths tree for the affected nodes of R_i in the i-th iteration, the computation complexity to execute the restricted Bellman-Ford algorithm for the initial $\lceil \log h \rceil$ iterations is $O(\sum_{i=0}^{\lceil \log h \rceil} |D| n^2 h_i) = O(n^{5/2-\alpha/4} |D|^{1/2} \log^{1/2} n)$. On the other hand, the computation complexity to execute the matrix multiplication over semiring $(min, +)$ is $O(n^3 \log n)$ in total.

□

4.3 The Parallel Incremental Procedure

In Section 4.2, we assumed that the number of deleted nodes is $|D|$. Therefore, the number of inserted nodes is also $|D|$, but with different nodes and edges. As outlined in Section 1.2, parallelizing the modified Floyd-Warshall algorithm with two-layer loops poses a challenge. Even if the modified Floyd-Warshall algorithm can be executed in constant rounds, the round complexity can be as large as $O(|D|)$. To address this, we propose a method to design the parallel incremental procedure using the blocked Floyd-Warshall algorithm [18] and an MPC algorithm of matrix multiplication on semiring.

In Section 4.2, we obtain the distance matrix $Dist$ with size $(n - |D|) \times (n - |D|)$ for $O(n - |D|)$ nodes and the shortest trees whose roots are affected nodes or belong to the hitting set. After inserting $|D|$ nodes and their incident edges, we must update a distance matrix with size $n \times n$ and the shortest paths between these n nodes. Specifically, we need to compute the distances for the remaining $n|D|$ pairs of nodes, which comprises $|D|^2$ pairs of inserted nodes and $n|D| - |D|^2$ pairs of nodes between the inserted nodes and the existing nodes. Then, we utilize these computed distances to update the distance matrix $Dist$. For an intuitive understanding of the update process resulting from the inserted nodes, refer to Fig. 3 and Fig. 4.

Fig. 3 and Fig. 4 illustrate the update process of the distance matrix with size $n \times n$. They show how we use the distance matrix obtained by the decremental procedure in Section 4.2 to obtain the final distance matrix for n^2 pairs of nodes in two iterations.

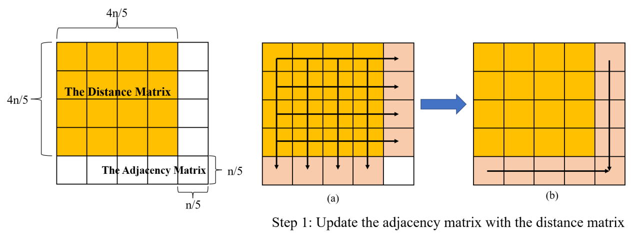

4n/5
4n/5
The Distance Matrix
The Adjacency Matrix
n/5
n/5
(a)
(b)
Step 1: Update the adjacency matrix with the distance matrix

Fig. 3 The process of updating the adjacency matrix with the distance matrix in the first step.

Specifically, the leftmost figure in Fig. 3 depicts the distance matrix of $16n^2/25$ pairs of nodes (the yellow part) and the adjacency submatrices that need to be updated (the white part). The right figures illustrate the update process in the first iteration, where the distance matrix is utilized to update the horizontal and vertical adjacency matrices in Fig. 3 (a). In Fig. 3 (b), the updated parts are employed to update the white part in Fig. 3 (a), causing its color to change as shown.

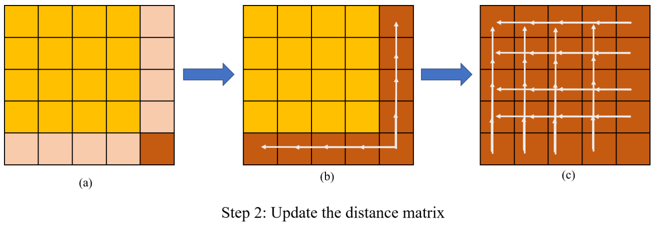


Fig. 4 The process of updating the distance matrix in the second step.

Fig. 4 illustrates the update process of the distance matrix in the second iteration. The deep orange matrix in Fig. 4 (a) represents the last diagonal update, which is the final distance matrix for the last $n^2/25$ pairs nodes. In Fig. 4 (b), this distance matrix is utilized to update the horizontal and vertical distance matrices (the deep orange part). In Fig. 4 (c), the distance matrix of $16n^2/25$ pairs of nodes is updated.

4.3.1 The Incremental Procedure In The MPC Model

In contrast to the blocked Floyd-Warshall algorithm employed in the distributed memory model [18, 19, 39], where the submatrix can be stored in a single processor, the size of each distance submatrix in our work is too large to be stored in a single processor. This increases the difficulty of designing an efficient algorithm for the dynamic APSP problem in the MPC

Algorithm 4 **The Incremental Procedure in the MPC(n^{α})**

```
Input: The distance matrix $Dist$ obtained by the decremental procedure, the adjacency matrix $A_{n/|D|,n/|D|}$ of the inserted nodes, the adjacency matrices $A_{i,n/|D|}$, and $A_{n/|D|,j}$ of edge weights between the existing nodes and the inserted nodes ($i, j \in \{1, \dots, n/|D| - 1\}$).
Output: The shortest paths distance matrix $D$
 1: The first iteration:                                   ▷ Figure (a) and figure (b) in Figure 3
 2: for $i = 1$ to $n/|D| - 1$ do
 3:     Processors that contain submatrix $Dist_{ii}$ broadcast their matrices to processor that contain horizontal adjacency matrix $A_{i,n/|D|}$ or vertical adjacency matrix $A_{n/|D|,i}$
 4:     Processors that contain submatrix $Dist_{ij} (i \neq j, j \in \{1, \dots, n/|D| - 1\})$ send their matrices to processors that contain vertical adjacency matrix $A_{n/|D|,j}$ as in Figure 3
 5:     Processors that contain submatrix $Dist_{ji} (i \neq j, j \in \{1, \dots, n/|D| - 1\})$ send their matrices to processors that contain horizontal adjacency matrix $A_{j,n/|D|}$ as in Figure 3
 6:     Execute MMOnSemiring($Dist_{ij}, A_{n/|D|,j}, A_{n/|D|,j}$) and MMOnSemiring($Dist_{ji}, A_{j,n/|D|}, A_{j,n/|D|}$) ($j \in \{1, \dots, n/|D| - 1\}$)   ▷ See details in Section 4.2
 7: end for
 8: for $i = 1$ to $n/|D| - 1$ do
 9:     Processors that contain submatrices $A_{i,n/|D|}$ or $A_{n/|D|,i}$ send their matrices to processors that contain $A_{n/|D|,n/|D|}$
10:     Execute MMOnSemiring($A_{\frac{n}{|D|},i}, A_{i,\frac{n}{|D|}}, A_{\frac{n}{|D|},\frac{n}{|D|}}$)
11: end for
12: The second iteration:                                  ▷ See Figure 4
13: Processors with matrix $A_{n/|D|,n/|D|}$ communicate with each other to compute APSP with blocked Floyd-Warshall algorithm and get the distance submatrix $D_{n/|D|,n/|D|}$
14: for $i = 1$ to $n/|D| - 1$ do
15:     Processors with matrix $D_{n/|D|,n/|D|}$ broadcast their matrix to processors with matrix $A_{n/|D|,i}$ and $A_{i,n/|D|}$
16:     Execute MMOnSemiring($A_{i,\frac{n}{|D|}}, D_{\frac{n}{|D|},\frac{n}{|D|}}, A_{i,\frac{n}{|D|}}$) and MMOnSemiring($D_{\frac{n}{|D|},\frac{n}{|D|}}, A_{\frac{n}{|D|},i}, A_{\frac{n}{|D|},i}$) and use $D_{i,n/|D|}$ and $D_{n/|D|,i}$ to replace $A_{i,n/|D|}$ and $A_{n/|D|,i}$, respectivety.   ▷ See details in Section 4.2
17: end for
18: for $i = 1$ to $n/|D| - 1$ do
19:     Processors that contain matrix $D_{i,n/|D|}$ broadcast their matrix to processor with matrix $D_{n/|D|,j}$ ($j \in \{1, \dots, n/|D| - 1\}$);
20:     Processors that contain matrix $Dist_{i,j}$ send their matrices to processors that contain matrix $D_{n/|D|,j}$ ($j \in \{1, \dots, n/|D| - 1\}$);
21:     Execute MMOnSemiring($D_{i,n/|D|}, D_{n/|D|,j}, Dist_{i,j}$) on processors that contain matrix $D_{n/|D|,j}$ ($j \in \{1, \dots, n/|D| - 1\}$) and use $D_{i,j}$to replace $Dist_{i,j}$▷ See details in Section 4.2
22:     Processors that contain matrix $D_{n/|D|,j}$ ($j \in \{1, \dots, n/|D| - 1\}$) send $D_{i,j}$ to processors that contain matrix $Dist_{i,j}$ (which would be replaced by $D_{i,j}$) ($j \in \{1, \dots, n/|D| - 1\}$)
23: end for
```

model after inserting nodes and incident edges. However, we can make a tradeoff between the round complexity and the total memory required in the MPC model. By leveraging the MPC algorithm of matrix multiplication on semiring in Lemma 3 (see Section 2.3), we can obtain a parallel incremental algorithm with low round complexity in Algorithm 4.

Since the size of the inserted nodes set is $|D|$, the adjacency matrix of the inserted nodes constitutes a proportion of $\frac{|D|^2}{n^2}$ of the $n \times n$ final distance matrix. Therefore, we divide the $n \times n$ matrix into $\frac{n}{|D|} \times \frac{n}{|D|}$ parts. The distance matrix obtained by the decremental procedure in Section 4.2 is divided into $(n/|D| - 1)^2$ same parts, with each submatrix of size $|D| \times |D|$ stored in $O(|D|^2/n^\alpha)$ processors, where the memory of each processor is $S = \tilde{O}(n^\alpha)$. The symbols $Dist_{i,j}$, where $i, j \in \{1, \ldots, n/|D| - 1\}$, are employed to denote these distance submatrix. $A_{i,n/|D|}$ and $A_{n/|D|,j}$ (where $i, j \in \{1, \ldots, n/|D| - 1\}$) denote the adjacency matrices between the existing nodes and the inserted nodes. Furthermore, $A_{n/|D|,n/|D|}$ denotes the adjacency matrices between the inserted nodes. To reduce the round complexity for the incremental procedure, additional processors are invoked to compute the final distance matrix.

Compared to the preprocessing procedure and the decremental procedure, when using additional processors, the main challenge is to reduce the total memory required rather than the round complexity. To update the distance matrix D between n nodes, according to [22], only $O(|D|^{3(1-\alpha)})$ extra processors are needed.

Referring to Fig. 3 (a), to update the adjacency matrices in the horizontal and the vertical direction, we send the distance matrices $Dist_{i,j}$ (or $Dist_{j,i}$, where $i \in \{1, \cdots, n/|D|\}$), to the processors storing $A_{n/|D|,j}$ (or $A_{j,n/|D|}$), as shown in line 2-line 7 of Algorithm 4. We then perform matrix multiplication on semiring using Lemma 3 for $O(n/|D|)$ times since $i \in \{1, \ldots, n/|D| - 1\}$. The update of the adjacency matrix $A_{n/|D|,n/|D|}$ from line 8 to line 11 (which can refer to Fig. 3 (b)) can be completed similarly. To update the distance matrix $Dist_{i,j}$ in the second iteration, we first update the matrix $A_{n/|D|,n/|D|}$ using the algorithm in Lemma 3 (which is in line 13 of Algorithm 4).

The subroutine **Broadcast**($A_{n/|D|,n/|D|}$, $P(A_{n/|D|,n/|D|})$, $P(A_{i,n/|D|})$, $P(A_{n/|D|,i})$) (where $i \in \{1, \cdots, n/|D|\}$) in Section 3.2 is employed to update the distances between the existing nodes and the inserted nodes. (refer to lines 14-17 in Algorithm 4.) The update of the distance matrix *Dist* is divided into $O(n/|D|)$ steps. As executing **MMOnSemiring**($D_{i,n/|D|}$, $D_{n/|D|,j}$,$Dist_{i,j}$) requires additional processors, we send the data required for updating the matrix $Dist_{i,j}$ to the processors that store the original adjacency matrix $A_{i,n/|D|}$, $i \in \{1, \cdots, n/|D|\}$ (refer to lines 18-23 in Algorithm 4). Then we complete the matrix multiplication on semiring.

To obtain the corresponding *shortest paths*, we begin by initializing an empty predecessor matrix Π_D of size $n \times n$ and create n empty trees rooted at n nodes (which include the nodes remaining after executing Algorithm 4 and the inserted nodes). These empty trees can reduce the overhead compared to setting n empty matrices to the record nodes on the shortest paths between nodes. During the computation of the incremental procedure, whenever updates occur in the predecessor matrix Π_D, we concurrently update these n trees.

4.3.2 Complexity Analysis

Lemma 9. (The round complexity of the parallel incremental procedure) Given the distance matrix *Dist* for graph $G\backslash D$, an empty predecessor matrix Π_D, and n empty trees rooted at each node, the incremental procedure proposed for a fully dynamic APSP problem can handle a batch of the inserted nodes with size $|D|$ to obtain the shortest paths between these n nodes in $O(\frac{|D|}{n^{\alpha/2}} + \frac{n}{\alpha|D|})$ rounds for the MPC model.

Proof. In Section 4.3, we use extra $O((n/|D|)^{3-\alpha})$ processors to reduce the round complexity of computing the distances and shortest paths between updated nodes. In Algorithm 4, **MMOnSemiring**(A, B, C) (See Lemma 3) costs $O(1 + \lceil(1 - \alpha/2)/\alpha\rceil)$ rounds, and the broadcast operation executed by processors costs $O(1/\alpha)$ rounds in each *for* loop of Algorithm 4. As the input matrix size is $|D| \times |D|$ and the memory of each processor is $\tilde{O}(n^\alpha)$, the shortest paths between the inserted nodes can be completed in $O(\frac{|D|}{\alpha n^{\alpha/2}})$ rounds. Then, the total number of rounds for Algorithm 4 is $O(\frac{|D|}{\alpha n^{\alpha/2}} + \frac{n}{\alpha|D|})$.

Furthermore, it is evident that the computation complexity required to execute lines 1-7 and lines 18-23 of Algorithm 4 is the same, both of which are $O((\frac{n}{|D|} - 1)^2|D|^3)$. Similarly, the computation complexity for lines 8-11 and lines 14-17 of Algorithm 4 is $O((\frac{n}{|D|} - 1)|D|^3)$. The computation complexity to execute line 13 of Algorithm 4 is $O(|D|^3)$. Therefore, the computation complexity of Algorithm 4 is $O(|D|n^2)$ since $|D| \leq n$. □

4.4 The Proof of Theorem 1

In the following section, we will analyze the round complexity of the parallel algorithm proposed in this paper for the dynamic APSP problem.

The Proof of Theorem 1: By utilizing Lemma 2, a parallel decremental algorithm can be transformed into a parallel fully dynamic algorithm. Combining this with Lemma 6, Lemma 8, and Lemma 9, the worst-case update round complexity for our parallel algorithm can be expressed as $O(\frac{n\log^2 n}{\alpha|D|} + \sqrt{n^{1-\alpha/2}|D|\log n}/\alpha + \frac{|D|}{\alpha n^{\alpha/2}} + \frac{n}{\alpha|D|})$. To minimize the round complexity of updating the shortest paths, we set $|D| = O(n^{\frac{1}{3}+\frac{\alpha}{6}}\log n)$. This yields the results specified in Theorem 1. Moreover, the number of computation operations for our parallel fully dynamic algorithm is $O(n^3\log n/|D| + n^{5/2-\alpha/4}|D|^{1/2}\log^{1/2} n + n^3\log n + |D|n^2) = O(n^3\log n)$.

Currently, to query the shortest path between any two nodes, we have to combine and compare the shortest path of any two nodes from the trees solved during the parallel incremental procedure in our parallel decremental procedure. Therefore, we query the shortest path of any two nodes twice: once from the parallel decremental procedure and another from the parallel incremental procedure. This can be achieved by the sorting and broadcast operations, we can obtain the final shortest path in $O(\alpha^{-1})$ rounds, instead of computing the final shortest paths between all n nodes simultaneously, which would be prohibitively expensive.

The query of the shortest path between any two nodes requires at most $O(n/n^{2\alpha}) = O(n^{1-2\alpha})$ rounds, as the number of nodes on the shortest path is limited to at most n. □

5 Conclusion

In this paper, we propose the first fully dynamic parallel algorithm for solving the all-pairs shortest path problem in the MPC model. Specifically, our algorithm achieves a worst-case update complexity of $O(n^{\frac{2}{3}-\frac{\alpha}{6}}\log n/\alpha)$ rounds for directed weighted graphs with non-negative cycles. Our algorithm consists of three main components: first, we present a preprocessing algorithm that constructs a decremental data structure. Second, we design a decremental algorithm to update the data structure under node deletions, which yields the distance matrix and the shortest path trees. These structures are then used by our parallel incremental algorithm to handle node insertions. Finally, we compare our algorithm with the existing static APSP algorithms in the MPC model, demonstrating the effectiveness of our approach.

Acknowledgements This work was supported in part by National Natural Science Foundation of China under Grants 61972447 and 61832006.

References

1. Mirzasoleiman B, Karbasi A, Sarkar R, Krause A. Distributed submodular maximization: Identifying representative elements in massive data. In: Neural Information Processing Systems. 2013
2. Im S, Moseley B, Sun X. Efficient massively parallel methods for dynamic programming. In: Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017. 798–811
3. Biswas A S, Dory M, Ghaffari M, Mitrovic S, Nazari Y. Massively parallel algorithms for distance approximation and spanners. In: 33rd ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2021. 118–128
4. Dinitz M, Nazari Y. Massively parallel approximate distance sketches. In: 23rd International Conference on Principles of Distributed Systems, OPODIS 2019. 35:1–35:17
5. Martin D P. Dynamic shortest path and transitive closure algorithms: A survey. 2017
6. Zhu X, Li W, Yang Y, Wang J. Incremental algorithms for the maximum internal spanning tree problem. Sci. China Inf. Sci., 2021, 64: 152103:1–152103:8
7. Abraham I, Chechik S, Krinninger S. Fully dynamic all-pairs shortest paths with worst-case update-time revisited. In: Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017. 440–452
8. Demetrescu C, Italiano G F. A new approach to dynamic all pairs shortest paths. In: Proceedings of the 35th Annual ACM Symposium on Theory of Computing, 2003. 159–166
9. Gutenberg M P, Wulff-Nilsen C. Fully-dynamic all-pairs shortest paths: Improved worst-case time and space bounds. In: Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA 2020. 2562–2574
10. Thorup M. Worst-case update times for fully-dynamic all-pairs shortest paths. In: Proceedings of the 37th Annual ACM Symposium on Theory of Computing, 2005. 112–119
11. Baswana S, Hariharan R, Sen S. Improved decremental algorithms for maintaining transitive closure and all-pairs shortest paths. In: Proceedings on 34th Annual ACM Symposium on Theory of Computing, 2002. 117–123
12. King V. Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. In: 40th Annual Symposium on Foundations of Computer Science, FOCS '99. 81–91
13. Xin S, Wang G. New method in information processing for maintaining an efficient dynamic ordered set. Sci. China Ser. F Inf. Sci., 2009, 52(8): 1292–1301
14. Dhulipala L, Durfee D, Kulkarni J, Peng R, Sawlani S, Sun X. Parallel batch-dynamic graphs: Algorithms and lower bounds. In: Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA 2020. 1300–1319
15. Nowicki K, Onak K. Dynamic graph algorithms with batch updates in the massively parallel computation model. In: Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021. 2939–2958

16. Italiano G F, Lattanzi S, Mirrokni V S, Parotsidis N. Dynamic algorithms for the massively parallel computation model. In: The 31st ACM on Symposium on Parallelism in Algorithms and Architectures, SPAA 2019. 49–58
17. Madkour A, Aref W G, Rehman F U, Rahman M A, Basalamah S. A survey of shortest-path algorithms. 2017
18. Sao P, Kannan R, Gera P, Vuduc R W. A supernodal all-pairs shortest path algorithm. In: PPoPP '20: 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. 250–261
19. Solomonik E, Buluç A, Demmel J. Minimizing communication in all-pairs shortest paths. In: 27th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2013. 548–559
20. Karczmarz A, Sankowski P. In: A Deterministic Parallel APSP Algorithm and its Applications. A Deterministic Parallel APSP Algorithm and its Applications, 255–272
21. Cao N, Fineman J T. Parallel exact shortest paths in almost linear work and square root depth. In: Proceedings of the 2023 ACM-SIAM Symposium on Discrete Algorithms, SODA 2023,. 4354–4372
22. Hajiaghayi M, Lattanzi S, Seddighin S, Stein C. Mapreduce meets fine-grained complexity: Mapreduce algorithms for apsp, matrix multiplication, 3-sum, and beyond. CoRR, 2019, abs/1905.01748
23. Goodrich M T, Sitchinava N, Zhang Q. Sorting, searching, and simulation in the mapreduce framework. In: Algorithms and Computation - 22nd International Symposium, ISAAC 2011. 374–383
24. Ashvinkumar V, Bernstein A, Cao N, Grunau C, Haeupler B, Jiang Y, Nanongkai D, Su H. Parallel and distributed exact single-source shortest paths with negative edge weights. CoRR, 2023, abs/2303.00811
25. Ullman J D, Yannakakis M. High-probability parallel transitive-closure algorithms. SIAM J. Comput., 1991, 20(1): 100–125
26. Henzinger , MR , King . Maintaining minimum spanning forests in dynamic graphs. SIAM J COMPUT, 2001
27. Even S, Shiloach Y. An on-line edge-deletion problem. J. ACM, 1981, 28(1): 1–4
28. King V, Thorup M. A space saving trick for directed dynamic transitive closure and shortest path algorithms
29. Cormen T H, Leiserson C E, Rivest R L, Stein C. Introduction to Algorithms, 3rd Edition. MIT Press, 2009
30. Khopkar S S. Incremental algorithms for centrality metric calculations in social network analysis. PhD thesis, State University of New York at Buffalo.;, 2010
31. Forster S, Nazari Y, Probst Gutenberg M. Deterministic incremental apsp with polylogarithmic update time and stretch. In: Proceedings of the 55th Annual ACM Symposium on Theory of Computing, STOC 2023. 1173–1186
32. Chechik S, Zhang T. In: Faster Deterministic Worst-Case Fully Dynamic All-Pairs Shortest Paths via Decremental Hop-Restricted Shortest Paths. Faster Deterministic Worst-Case Fully Dynamic All-Pairs Shortest Paths via Decremental Hop-Restricted Shortest Paths, 87–99
33. Mao X. Fully-dynamic all-pairs shortest paths: Likely optimal worst-case update time. CoRR, 2023, abs/2306.02662
34. Cicerone S, D'Angelo G, Stefano G D, Frigioni D. Partially dynamic efficient algorithms for distributed shortest paths. Theor. Comput. Sci., 2010, 411(7-9): 1013–1037
35. Khopkar S S, Nagi R, Nikolaev A G. An efficient map-reduce algorithm for the incremental computation of all-pairs shortest paths in social networks. In: International Conference on Advances in Social Networks Analysis and Mining, ASONAM 2012. 1144–1148
36. Karloff H J, Suri S, Vassilvitskii S. A model of computation for mapreduce. In: Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010. 938–948
37. Beame P, Koutris P, Suciu D. Communication steps for parallel query processing. In: Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2013. 273–284
38. Fully dynamic all pairs shortest paths with real edge weights. Journal of Computer and System Sciences, 2006, 72(5): 813–837
39. Tiskin A. All-pairs shortest paths computation in the BSP model. In: Automata, Languages and Programming, 28th International Colloquium, ICALP 2001. 178–189