CCF Transactions on High Performance Computing
https://doi.org/10.1007/s42514-025-00234-1

REGULAR PAPER

A parallel all-pairs shortest paths algorithm for dynamic graphs

Lin Zhu[1,2] · Qiang-sheng Hua[1] · Hai Jin[1]

Received: 15 June 2023 / Accepted: 22 June 2025

Abstract
The computation of the all-pairs shortest paths is an important graph algorithmic problem. When the graph changes, such as edge deletions/insertions, recalculating the shortest distance of a graph from scratch is costly. In this paper, we investigate how to quickly maintain the shortest distance of the dynamic graph in the distributed memory system. For a distributed system with p processors, the state-of-art algorithm to recompute the shortest distance of a graph with n vertices from scratch requires $O\left(n^2/\sqrt{p}\right)$ bandwidth cost and $O\left(\sqrt{p}\log^2 p\right)$ latency cost. For the insertion of k edges, we give an incremental algorithm with a bandwidth cost of $O\left(\frac{nk}{\sqrt{p}}+k^2\right)$ and a latency cost of $O(\log p)$. For typical scenarios where $k=O\left(\frac{n}{\sqrt{p}}\right)$, the bandwidth and latency costs are reduced by a factor of $O\left(\sqrt{p}\right)$ and $O\left(\sqrt{p}\log p\right)$, respectively. For the deletion of k edges, we give a decremental algorithm with a bandwidth cost of $O\left(\frac{nk}{\sqrt{p}}+k^2+\frac{n^2}{p}\log^3 p+|S|^2\log p\right)$ and a latency cost of $O\left(\log^3 p\right)$, where $|S|$ is the separator size of a constructed graph and is related to the alteration degree of the shortest path of the dynamic graph. When $k=O\left(\frac{n}{\sqrt{p}}\right)$ and $|S|=O\left(\frac{n}{\sqrt{p}}\right)$, the bandwidth and latency costs are reduced by a factor of $O\left(\sqrt{p}/\log^3 p\right)$ and $O\left(\sqrt{p}/\log p\right)$, respectively.

Keywords All-pairs shortest paths · Dynamic graph · Distributed memory model · Communication complexity

1 Introduction

The all-pairs shortest paths (APSP) is an important algorithmic problem and has wide applications in road networks, datacenter network design, and so on. Many well-known algorithms can be used to solve the shortest paths for static graphs. However, when the graph changes, especially when the change is small, it is costly to recalculate the shortest paths of the graph from scratch. Thus, it may be impractical to use these static approaches to solve dynamic graph problems.

A dynamic algorithm must be able to maintain the property of the graph (e.g., minimum spanning tree and all-pairs shortest paths) as it changes. In general, an efficient dynamic graph algorithm should maintain the property of the graph faster than recomputing it from scratch. We call a dynamic algorithm "incremental" if the algorithm can handle the edge/vertex insertions. A dynamic algorithm is "decremental" if the algorithm can handle the deletion of edges/vertices. In addition, an algorithm is partially dynamic if it can handle only one insertion or deletion. An algorithm is fully dynamic if it can handle both insertion and deletion.

Research on dynamic graph algorithms has already yielded several breakthrough results (Demetrescu and Italiano 2003; Thorup 2005; Gutenberg and Wulff-Nilsen 2020). They are concerned with maintaining the APSP of the changing graph in the sequential model and aim to reduce

✉ Qiang-sheng Hua
qshua@hust.edu.cn

Lin Zhu
linzhu@hust.edu.cn

Hai Jin
hjin@hust.edu.cn

1 National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, People's Republic of China

2 Henan Zhongtian High-tech Intelligent Technology Co, Ltd., Zhengzhou, People's Republic of China

the computational complexity. However, as the data grows, a social network graph may have over a million vertices and over a billion edges, and it might become impractical to analyze it in the sequential model due to memory limitations and excessive computation time. In this paper, instead of considering traditional models such as the RAM model and $\mathcal{CONGEST}$ model (Jia et al. 2022; Hua et al. 2021), we consider the widely used distributed memory model (Ballard et al. 2011, 2012, 2013, 2018; Demmel et al. 2013), and study how to maintain the APSP of changing graphs in this model.

1.1 The problem

In the dynamic APSP problem, the graph is dynamically changing and the data about the graph is initially evenly distributed across p processors. Our objective is to maintain the APSP of the dynamic graph undergoing batched edge insertions/deletions with the lowest inter-processor communication. Note that batched edge insertions/deletions is more general than vertex insertions/deletions since any vertex insertions/deletions can be simulated by inserting/deleting all neighboring edges of this vertex. We modify the weights of an edge from finite to infinite to represent the deletion of that edge and modify the weights of an edge from infinite to finite to indicate the insertion of that edge. We assume that the graph has n vertices and m edges.

1.2 Related work

Floyd (1962) and Warshall (1962) proposed the classical dynamic programming algorithm, which is widely employed to solve the shortest paths. It has a three-nested-loops algorithm structure similar to matrix multiplication and is usually used to handle dense graphs. The blocked Floyd-Warshall algorithm was subsequently designed to increase the locality of data (Park et al. 2004).

Based on the Floyd-Warshall algorithm, Jenq and Sahni (1987) proposed the first APSP algorithm in the distributed memory system. However, the algorithm does not exploit the block structure of the data, which causes it to have a high message transfer volume, even up to $O(n)$.

Due to the similar computational dependency structure, the communication lower bound proof technique for linear algebra problems can also be applied to the APSP problem (Ballard et al. 2014; Irony et al. 2004; Hong and Kung 1981). When no extra memory is available, that is, the total processor memory size and the input size match, the word transfer (bandwidth) lower bound for the APSP problem is $\Omega((n^2/\sqrt{p})$ and the message transfer (latency) lower bound is $\Omega\left(\sqrt{p}\right)$ (Solomonik et al. 2013). Employing the divide and conquer strategy, Solomonik et al. (2013) proposed a distributed algorithm. This algorithm requires no extra available memory and incurs $O\left(n^2/\sqrt{p}\right)$ bandwidth cost and $O\left(\sqrt{p}\log^2 p\right)$ latency cost, both of which can either reach or early reach the asymptotic lower bound. Since each shortest path contains no more than n edges, Tiskin (2001) employed the path doubling method and got the shortest distance matrix A^n by repeatedly squaring in $\log n$ matrix multiplications over the $(min, +)$ semiring.

The APSP problem for dynamic graphs on a single machine has been widely studied. Johnson (1977) proposed the first partially-dynamic algorithm. This algorithm can be used to maintain the APSP for multiple vertex insertions, and the worst update time required for each insertion is $O\left(n^2\right)$. King (1999) proposed the first fully-dynamic APSP algorithm, and when the graph is inserted or deleted an edge, this algorithm will incur $O\left(n^{2.5}\sqrt{W\log n}\right)$ amortized update time (W is the maximal weight of all edges as defined in (Even and Shiloach 1981).

Greco et al. proposed dynamic APSP algorithms for single edge insertion/deletion (Greco et al. 2016) and multiple edge insertions/deletions (Greco et al. 2018). Compared with sequential static algorithms, these algorithms work well in reducing computational complexity. However, due to strong data dependencies, running these algorithms on parallel machines can lead to high latency overheads, even reaching $O(n)$. It is challenging to design low-communication dynamic APSP algorithms in a distributed memory model.

Cicerone et al. proposed the distributed fully dynamic (Cicerone et al. 2003) and partially dynamic (Cicerone et al. 2010) APSP algorithms. These two algorithms are used to maintain the APSP of a general network with each processor representing one vertex. Several parallel dynamic graph algorithms are proposed to maintain the solutions to problems such as Maximal Matching, Graph Connectivity, and MST (Italiano et al. 2019; Nowicki and Onak 2021; Dhulipala et al. 2020). Khopkar et al. (2012) proposed an incremental algorithm for single edge insertion. However, to our best knowledge, there are few studies on how to maintain APSP for batch edge insertions and deletions on parallel machines.

1.3 Our contribution

(1) We propose the first incremental algorithm for batch edge insertions in the distributed memory model. After inserting multiple edges, for any vertex pair, its shortest distance is unchanged unless its shortest path contains at least one inserted edge. Based on this, for each vertex pair on each processor, we calculate the shortest distance of paths containing at least one inserted edge in parallel, and update its shortest distance by comparing

it with the original shortest distance. This can be achieved by constructing a graph with $O\left(\frac{n}{\sqrt{p}}\right)$ vertices on each processor. Compared to static parallel algorithms, the bandwidth and latency upper bounds of our incremental algorithm can be reduced by a factor of $O(\sqrt{p})$ and $O(\sqrt{p}\log p)$, respectively.

(2) We propose the first decremental algorithm for batch edge deletions in the distributed memory model. Our approach is divided into three stages. In the first stage, we use an idea similar to our incremental algorithm to identify those vertex pairs with the unchanged shortest distance. In the second stage, we construct a graph and use the nested dissection technique to reorder the vertices of this graph. The purpose of this stage is to rearrange the shortest distance matrix so that those vertex pairs with unchanged shortest distances are arranged in the same block, thus taking advantage of data locality. In the third stage, we use the path doubling method to update only the blocks with the changed shortest distance. Compared to static parallel algorithms, the bandwidth and latency upper bounds of our decremental algorithm can be reduced by a factor of $O(\sqrt{p}/\log^3 p)$ and $O(\sqrt{p}/\log p)$, respectively.

1.4 Paper organization

In Sect. 2, we give the problem definition, theoretical cost model, and two classical algorithms for computing APSP. In Sect. 3, we propose a dynamic incremental APSP algorithm for batch edge insertions and analyze the algorithm's costs and memory requirement. We also propose a dynamic decremental APSP algorithm for batch edge deletions in Sect. 4. In Sect. 5 we give a conclusion.

2 Preliminaries

We first give the theoretical cost model, problem description, and two classical methods for computing the APSP problem: the Floyd-Warshall (FW) algorithm and the path doubling method.

2.1 Theorem cost model

The distributed memory system is widely employed (Ballard et al. 2011, 2012, 2013, 2018; Demmel et al. 2013) and is modeled as follows. The architecture has p processors, which are connected through an all-to-all network. That is, there is a link between any two processors. Each processor is assigned a separate memory of size M. Processors communicate with other processors by sending/receiving messages, and each processor can only send/receive one message to/from another processor at a time. We measure the performance of an algorithm by the number of words transferred (bandwidth cost) and the number of messages transferred (latency cost) along the critical path (Yang and Miller 1988), that is, the two messages communicated simultaneously between different pairs of processors are only counted once.

The following four well-known operations are used heavily in our incremental and decremental algorithms.

- *Gather*: all processors in a processor group contribute local data to the root processor to serve as local data for the root processor.
- *Broadcast*: root processor distributes local data to every processor in a processor group to serve as their local data.
- *Allgather*: all processors in a processor group contribute local data to a concatenation, and the result is broadcast to all processors in this processor group.
- *Reduce*: all processors in a processor group contribute local data and reduce it to the root processor to serve as local data for the root processor.

Assuming that the processor group size is p and the number of words transferred is n, then the asymptotic communication costs of the above operations can be obtained by a binomial tree or butterfly scheduling (Thakur et al. 2005; Hutter and Solomonik 2019). We summarize it in Table 1.

2.2 Problem description

Table 2 describes the notations used in this paper. Given a weighted undirected graph $G = (V, E)$, where the number of vertices is $|V| = n$ and the number of edges is $|E| = m$. We denote by v_x $(x = \{1,2,\dots,n\})$ the vertex in graph G and by e_{xy} the edge between v_x and v_y. Each edge e_{xy} has a weight $\left|e_{xy}\right|$. We assume that there are no cycles with a negative sum of weights in graph G. Let π_{xy} represent a path between v_x and v_y, then π_{xy} is the shortest path if it has the minimum weight over all paths between v_x and v_y.

We denote by ΔE the set of inserted/deleted edges. When multiple edges are inserted/deleted, we indicate by $G^+ = G \cup \Delta E$ the graph that ΔE is inserted into G, and

Table 1 Costs of communication operations

#	Bandwidth cost	Latency cost
$\boldsymbol{T_{Gather}(p,n)}$	$O(n)$	$O(\log p)$
$\boldsymbol{T_{Broadcast}(p,n)}$	$O(n)$	$O(\log p)$
$\boldsymbol{T_{Reduce}(p,n)}$	$O(n)$	$O(\log p)$
$\boldsymbol{T_{Allgather}(p,n)}$	$O(n)$	$O(\log p)$

Table 2 List of notations used in this paper

Type	Notation	Description
	p	The number of processors
Processor	p_{ij}	Processor located in the i-th row and the j-th column
	M	The memory size of each processor
	$G = (V, E, W)$	The initial input graph
	ΔE	The set of inserted/deleted edges
	$G^{+} = G \cup \Delta E$	Graph after batch inserting multiple edges ΔE in G
	$G^{-} = \mathrm{G} \backslash \Delta E$	Graph after batch deleting multiple edges ΔE in G
Graph	$G(i,j), G^{-}(i,j), G^{+}(i,j)$	The subgraphs of G, G^{-} and G^{+} owned by processor p_{ij}
	$G\prime(i,j)$	The constructed graph in processor P_{ij}
	G^{*}	The constructed graph for performing vertex reordering
	π_{xy}	A path between vertex v_x and vertex v_y
	O_{xy}	The set of paths π_{xy} containing at least one edge in ΔE
Matrix	A, A^{-}, A^{+}, A^{*}	Adjacency matrices of G, G^{-}, G^{+} and G^{*} $A_{ij} = w_{ij}$
	D, D^{-}, D^{+}, D^{*}	Shortest distance matrices of G, G^{-}, G^{+} and G^{*}

indicate by $G^{-} = G/\Delta E$ the graph that ΔE is deleted from G. We denote by A, A^{+}, and A^{-} the n-by-n adjacency matrices of the graphs G, G^{+}, and G^{-}, where $A_{xx} = 0$ and $A_{xy} = \left|e_{xy}\right|$. In addition, we denote by D, D^{+}, and D^{-} the n-by-n shortest distance matrices of G, G^{+}, and G^{-}, where D_{xy} is the distance between vertex v_x and vertex v_y. Alteration degree indicates the degree of alteration of the shortest path in the dynamic graph. A high alteration degree means that the shortest path is severely altered, i.e., a large number of corresponding elements in D^{+}/D^{-} are different from those in D.

The p processors can be viewed as a $\sqrt{p} \times \sqrt{p}$ grid with each processor indexed as p_{ij}, where $i,j = \{1,2,\ldots,\sqrt{p}\}$. Initially, the adjacency matrix A and the shortest distance matrix D of graph G are known and evenly distributed over p processors in a block layout. We denote by $A(i,j)$ (or $D(i,j)$) the block of A (or D) owned by p_{ij}, which is the adjacency matrix (or shortest distance) between vertex set V_i and vertex set V_j, $V_i \subset V$, $V_j \subset V$ and $\left|V_i\right| = \left|V_j\right| = n/\sqrt{p}$. When G is inserted or deleted with multiple edges $\Delta E = \{e_{a_1 b_1}, e_{a_2 b_2} \ldots, e_{a_k b_k}\}$, if $a_m \in V_i$ and $b_m \in V_j$, then processor p_{ij} stores the data of the inserted/deleted edge $e_{a_m b_m}$.

Based on the above description, our goal is to solve for the shortest distance of each vertex pair in the graph G^{+}/G^{-} on the parallel machine with the lowest communication cost.

2.3 FW algorithm and path doubling method

Floyd-Warshall: Floyd-Warshall is a classical dynamic programming method for computing the APSP. Similar to the classic matrix multiplication algorithm, Floyd-Warshall calculates APSP in the following three-nested-loops structure.

```
for $z = 1$ to $n$
for $x = 1$ to $n$
for $y = 1$ to $n$
```

$$\mathrm{Dist(x,y)} = \min(\mathrm{Dist(x,y)}, \mathrm{Dist(x,z)} + \mathrm{Dist(z,y)})$$

The *Dist* matrix represents the shortest path between v_x and v_y discovered so far, and it is initialized by the adjacency matrix of G, i.e., $Dist(x,y) = A_{xy}$. At the z-th iteration, this algorithm checks for all vertex pairs (v_x, v_y) whether there is a shorter path via the intermediate vertex v_z. If so, $Dist(x,y)$ is updated to $Dist(x,z) + Dist(z,y)$, otherwise $Dist(x,y)$ is not updated. Intuitively, at the z-th iteration, $Dist(x,y)$ represents the shortest distance of paths between v_x and v_y that use only vertices in set $\{v_1, v_2 \ldots, v_z\}$. Eventually, at the n-th iteration, the shortest distance $Dist(x,y)$ between any two vertices v_x and v_y is obtained, and $Dist(x,y) = D_{xy}$.

Path Doubling Method: Given a graph G and its adjacency matrix A, solving the APSP of G is algebraically equivalent to solving the matrix closure $A^{*} = I \oplus A \oplus A^{2} \oplus \ldots$ of A in the $(min, +)$ semiring, where $\oplus$ is the element-wise application of *min*. Path doubling is a logarithmic-depth algorithm for computing A^{*} by repeatedly squaring the matrix over the semiring. For example, let $A^{2} = A \otimes A$ denotes the semiring matrix multiplication of A and A, then

$$A^{2}_{xy} \leftarrow \min_{z}\left(A_{xz} + A_{zy}\right)(x,y,z \in \{1,2,\ldots,n\}),$$

where A^{2}_{xy} is the shortest distance among paths of size (number of edges) no more than 2 between v_x and v_y in graph G. Since each shortest path in G contains no more than n edges, the shortest distance matrix of G can be got from $D = A^{*} = A^{n}$, which can be achieved by performing $\log n$ semiring matrix multiplications.

Tiskin proposed a modified version that can asymptotically reduce the number of semiring matrix multiplications from $O(\log n)$ to $O(\log p)$. This method is based on the fact that each shortest path in A^{*} is formed by two subpaths, where the size of the initial subpath is a multiple of q and the size of the final subpath is at most $q \le p$. Thus, A^* can be calculated by

$$A^* = A^p(q)^* \otimes A^p(q \le p), \tag{1}$$

where $A^p(q)$ consists of path distances in A^p with path size q and $A^p(q)^*$ consists of path distances whose path size is a multiple of q.

In Eq. (1), A^p can be got by iteratively performing $\log p$ semiring matrix multiplications. Let each element in A^p contains not only the path distance but also the path size, then $A^p(q)$ can be obtained from A^p as

$$A^p(q)_{xy} = \begin{cases} A^p{}_{xy}, \text{if the path size of } A^p{}_{xy} \text{ is } q \\ \infty, \textit{Otherwise} \end{cases}.$$

To compute $A^p(q)^*$, we can compute the APSP of a graph $G^p(q)$ whose adjacency matrix is $A^p(q)$. Considering that A^p contains n^2 shortest distances, there must be a q of range $q \le p$ such that the number of shortest paths with size q is at most $\frac{n^2}{p}$, i.e., $|A^p(q)| \le \frac{n^2}{p}$. Since the memory size per processor is at least $\frac{n^2}{p}$, each processor can access the matrix $A^p(q)$ via an allgather operation on p processors. Then, let each processor pick $\frac{n}{p}$ vertices in $G^p(q)$ and independently runs the dijkstra algorithm $\frac{n}{p}$ times to calculate the shortest path using these vertices as source vertices. The results of all processors' calculations are the shortest distances matrix $A^p(q)^*$ of $G^p(q)$. For more details, interested readers can refer to (Tiskin 2001).

3 Batch edge insertions

In this section, we propose an efficient incremental algorithm to maintain all-pairs shortest paths for the insertion of edges. When G is inserted with multiple edges ΔE, we give the following three observations about D and D^+.

(1) The shortest distance of each vertex pair (v_x, v_y) in graph G^+ cannot be greater than that in graph G, i.e., $D^+_{xy} \le D_{xy}$.
(2) If the shortest path between v_x and v_y in G^+ does not contain any edge in ΔE, then $D^+_{xy} = D_{xy}$.
(3) If $D^+_{xy} < D_{xy}$, then the shortest path between v_x and v_y in G^+ must contain at least one edge in ΔE.

Assuming that graph G is inserted with k edges $\Delta E = \{e_{a_1b_1}, e_{a_2b_2}, \dots, e_{a_kb_k}\}$, if $\left|e_{a_mb_m}\right| \ge D_{a_mb_m}$, where $m \in \{1,2,\dots,k\}$, then the insertion of edge $e_{a_mb_m}$ will not affect the shortest distances. Therefore, before computing the APSP of G^+ we prune the inserted edges and remove those edges with $\left|e_{a_mb_m}\right| \ge D_{a_mb_m}$ from ΔE.

Recalling the problem definition described in Sect. 2, the distance matrix D of graph G is initially evenly distributed over p processors and p_{ij} owns the block D(i, j), which is the shortest distance between V_i and V_j, $V_i \subset V$, $V_j \subset V$ and $|V_i| = |V_j| = n/\sqrt{p}$. We denote by $D^+(i,j)$ the shortest distance between V_i and V_j in G^+. Since D^+ consists of all blocks $D^+(i,j)$, $i,j = \{1,2,\dots,\sqrt{p}\}$, D^+ can be obtained if all blocks $D^+(i,j)$ are calculated correctly. Our method is divided into two phases, and eventually, each processor p_{ij} gets the block $D^+(i,j)$ of the shortest distance between V_i and V_j.

In the first phase, each processor p_{ij} locally constructs a new graph $G\prime(i,j)$, where the vertex set $V\prime(i,j)$ contains the vertex sets V_i and V_j. To construct $G\prime(i,j)$, p_{ij} accesses the required data of vertices and edges from other processors. In the second phase, each processor p_{ij} locally performs the Floyd-Warshall algorithm on $G\prime(i,j)$ to get the shortest distance in $G\prime(i,j)$. We prove in Theorem 1 that the shortest distance between V_i and V_j in G^+ and $G\prime(i,j)$ is equal.

3.1 The constructed graph on each processor

As shown in Fig. 1, the graph $G\prime(i,j)$ constructed on processor p_{ij} has a total of $2\left(n/\sqrt{p} + k\right)$ vertices, which consists of four vertex sets, i.e., V_i, $\{a_1, a_2 \dots, a_k\}$, $\{b_1, b_2 \dots, b_k\}$ and V_j. Let the edges between the four vertex sets be set as follows.

- The edges between V_i and $\{a_1, a_2 \dots, a_k\}$: There exists an edge between any two vertices in V_i and $\{a_1, a_2 \dots, a_k\}$, and the edge weight is equal to the shortest distance between those two vertices in graph G, which is a known element in D.
- The edges between V_i and V_j: There exists an edge between any two vertices in V_i and V_j, and the edge weight is equal to the shortest distance between those two vertices in graph G.

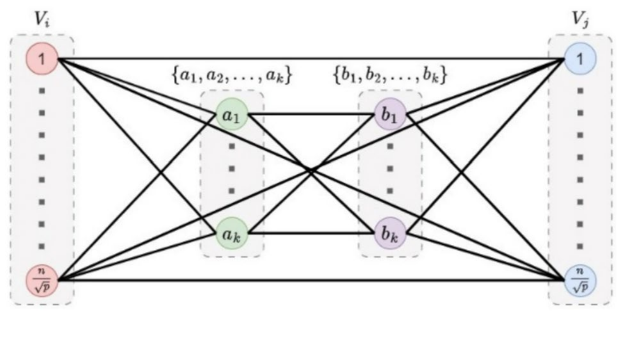

Fig. 1 The constructed graph $G\prime(i,j)$ on p_{ij} for edge insertions

- The edges between $\{a_1, \dots, a_k\}$ and $\{b_1, \dots, b_k\}$: There exists an edge between any two vertices in $\{a_1, \dots, a_k\}$ and $\{b_1, \dots, b_k\}$, where the edge weight between a_m and b_m is equal to $\left|e_{a_m b_m}\right|$ $(1 \le m \le k)$, and the edge weight between a_m and b_r is equal to the shortest distance between a_m and b_r in G $(m \ne r)$.
- The edges between $\{b_1, b_2 \dots, b_k\}$ and V_j: For any two vertices in $\{b_1, b_2 \dots, b_k\}$ and V_j, there exists an edge, and the edge weight is equal to the shortest distance between $\{b_1, b_2 \dots, b_k\}$ and V_j in graph G.

We denote by $D\prime(i,j)$ the shortest distance matrix between V_i and V_j in $G\prime(i,j)$, then the matrix $D\prime(i,j)$ has size $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$. $D\prime(i,j)$ can be obtained by locally executing the Floyd-Warshall algorithm on processor p_{ij} for the constructed graph $G\prime(i,j)$. Actually, the shortest distance between V_i and V_j in G^+ and $G\prime(i,j)$ is equal. We prove it in Theorem 1.

Theorem 1 *Given the initial graph, the shortest distance matrix of, and the set of inserted edges, then the distance between and in and is equal.*

$$GDG\Delta E = \{e_{a_1b_1}, e_{a_2b_2} \dots, e_{a_kb_k}\}V_iV_jG\prime(i,j)G^+$$

Proof. For any two vertices $v_x \in V_i$ and $v_y \in V_j$, we show that $D\prime(i,j)_{xy} = D^+(i,j)_{xy}$.

(1) $D\prime(i,j)_{xy} \ge D^+(i,j)_{xy}$: For any one edge e of graph $G\prime(i,j)$, we can find a corresponding path π in G^+ where e and π have the same endpoints and equal weights. Therefore, for any path between v_x and v_y in $G\prime(i,j)$, we can find a path in G^+ with equal weights and the same endpoints. Hence $D\prime(i,j)_{xy} \ge D^+(i,j)_{xy}$.

(2) $D^+(i,j)_{xy} \ge D\prime(i,j)_{xy}$: We use π^+_{xy} to represent the shortest path between v_x and v_y in G^+, then $\left|\pi^+_{xy}\right| = D^+(i,j)_{xy}$. No loss of generality, we suppose that π^+_{xy} contains m edges in ΔE, $m \in \{0,1,\dots,k\}$, and the order of these m edges in π^+_{xy} is $\Delta E_m = \{e^+_{a_{r_1}b_{r_1}}, e^+_{a_{r_2}b_{r_2}} \dots, e^+_{a_{r_m}b_{r_m}}\}$, where $r_1, r_2, \dots, r_m \in \{1,2,\dots,k\}$. Then the weight of π^+_{xy} is

$$\left|\pi^+_{xy}\right| = \left|\pi^+_{xa_{r_1}}\right| + \left|e_{a_{r_1}b_{r_1}}\right| + \left|\pi^+_{b_{r_1}a_{r_2}}\right| + \left|e_{a_{r_2}b_{r_2}}\right|$$

$$+\left|\pi^+_{b_{r_2}a_{r_3}}\right| + \dots + \left|e_{a_{r_m}b_{r_m}}\right| + \left|\pi^+_{b_{r_m}y}\right|,$$

where $\pi^+_{xa_{r_1}}, \pi^+_{b_{r_1}a_{r_2}}, \dots, \pi^+_{b_{r_m}y}$ are also shortest paths since π^+_{xy} is the shortest. Notice that the paths $\pi^+_{xa_{r_1}}, \pi^+_{b_{r_1}a_{r_2}}, \dots, \pi^+_{b_{r_m}y}$ do not contain any edge of ΔE, else π^+_{xy} contains at least $m+1$ edges in ΔE. From the previous observation (2), we have $\left|\pi^+_{xa_{r_1}}\right| = D^+_{xa_{r_1}} = D_{xa_{r_1}}, \dots, \left|\pi^+_{b_{r_m}y}\right| = D^+_{b_{r_m}y} = D_{b_{r_m}y}$.

If we can find a path $\pi_{xy}\prime$ between v_x and v_y in $G\prime(i,j)$ so that $\left|\pi_{xy}\prime\right| = \left|\pi^+_{xy}\right|$, then $D\prime(i,j)_{xy} \le \left|\pi^+_{xy}\right|$ holds due to the fact that $D\prime(i,j)_{xy} \le \left|\pi_{xy}\prime\right|$. Consider such a path $\pi_{xy}\prime$ consisting of edges $\{e_{xa_{r_1}}\prime, e_{a_{r_1}b_{r_1}}\prime, \dots, e_{b_{r_m}y}\prime\}$, where $e_{xa_{r_1}}\prime$ is the edge between v_x and a_{r_1} in $G\prime(i,j)$ and $e_{a_{r_1}b_{r_1}}\prime, \dots, e_{b_{r_m}y}\prime$ are similar. The weight of $\pi_{xy}\prime$ is

$$\left|\pi_{xy}\prime\right| = \left|e_{xa_{r_1}}\prime\right| + \left|e_{a_{r_1}b_{r_1}}\prime\right| + \dots + \left|e_{b_{r_m}y}\prime\right|.$$

According to the description of $G\prime(i,j)$, we have $\left|e_{xa_{r_1}}\prime\right| = D_{xa_{r_1}} = \left|\pi^+_{xa_{r_1}}\right|, \dots, \left|e_{b_{r_m}y}\prime\right| = D_{b_{r_m}y} = \left|\pi^+_{b_{r_m}y}\right|$. From this, we get $\left|\pi_{xy}\prime\right| = \left|\pi^+_{xy}\right|$ and thus $D\prime(i,j)_{xy} \le \left|\pi^+_{xy}\right| = D^+(i,j)_{xy}$.

3.2 A scheme to construct the graph

To construct $G\prime(i,j)$, p_{ij} needs to access the data of vertices and edges from other processors. Below we give a scheduling scheme.

1. The data for the edges between V_i and $\{a_1, a_2, \dots, a_k\}$: This part of the data is the distance between V_i and $\{a_1, a_2 \dots, a_k\}$ in graph G, which is a $\frac{n}{\sqrt{p}} \times k$ submatrix of D. Note that for each processor located in the same row $p_{i:}$, they have the same V_i and $\{a_1, a_2 \dots, a_k\}$ in the constructed graph. Therefore, the processors located in the same row require the same submatrix. Note that the processor row $p_{i:}$ stores an $\frac{n}{\sqrt{p}} \times n$ submatrix of D, that is, the shortest distance between V_i and V in G. Therefore, as shown in Fig. 2(a), by performing an allgather operation on $p_{i:}$, the shortest distance between V_i and $\{a_1, a_2 \dots, a_k\}$ can be obtained by each processor on $p_{i:}$. Similar operations can be performed in parallel on other processor rows. From Table 1, this operation has a bandwidth cost of $O\left(\frac{nk}{\sqrt{p}}\right)$ and a latency cost of $O(\log p)$.
2. The data for the edges between V_i and V_j: This part of the data is initially stored on the processor p_{ij} and does not need to be acquired.
3. The data for the edges between $\{a_1, \dots, a_k\}$ and $\{b_1, \dots, b_k\}$: This part of the data is a $k \times k$ matrix which contains the edge weights between a_m and b_m and between a_m and b_r, $m, r \in \{1,2,\dots,k\}$ and $m \ne r$. Notice that the graph constructed by each processor has the same $\{a_1, a_2 \dots, a_k\}$ and $\{b_1, b_2 \dots, b_k\}$, so each processor needs the same $k \times k$ matrix, which is initially stored on p processors. Therefore, as shown in Fig. 2(c), by

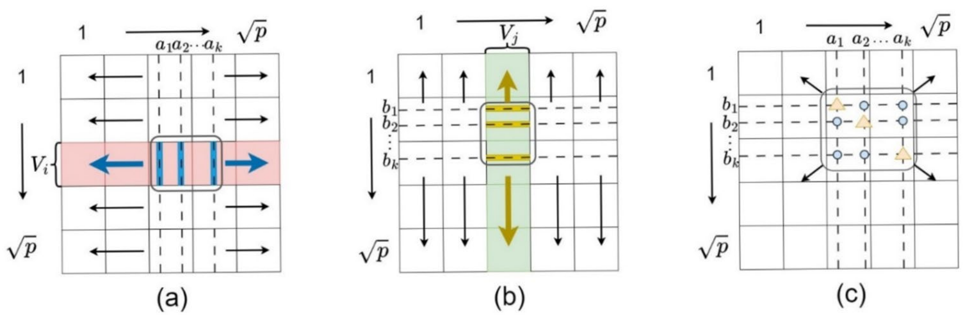

Fig. 2 **a** and **b** show the allgather operations performed in parallel on each processor row and each processor column, respectively. For example, in Fig. 2(a), processor row $p_{i:}$ collects k columns and broadcasts it to all processors in that row, where each column is a $|V_i| \times 1$ shortest distance matrix. Figure 2(c) shows the allgather operation performed on the $\sqrt{p} \times \sqrt{p}$ processor grid, where the diagonal triangles are the inserted edges' weights and the non-diagonal circles are the shortest distances between a_m and b_r in graph G ($m \neq r$)

performing an allgather operation on p processors, the $k \times k$ matrix can be obtained by each processor. From Table 1, this operation has a bandwidth cost of $O(k^2)$ and a latency cost of $O(\log p)$.

4. The data for the edges between $\{b_1, b_2 \ldots, b_k\}$ andV_j: The acquisition of this part of the data is similar to(1). For each processor in the same processor column$p_{:j}$, they need the same $k \times \frac{n}{\sqrt{p}}$ submatrix of D, which is stored on processor column$p_{:j}$. Therefore, as shown in Fig. 2(b), by parallel execution of an allgather operation on each processor column, every processor can get the data it needs. From Table 1, the bandwidth cost and latency cost of this operation are $O\left(\frac{nk}{\sqrt{p}}\right)$ and$O(\log p)$, respectively.

3.3 Algorithm and asymptotic cost analysis

As previously described, our method has two phases.

First, each processor p_{ij} acquires data and constructs local graph $G\prime(i,j)$ in parallel. Only communication cost is incurred in this phase. According to the construction scheme described in subSect. 3.2, the total bandwidth cost is $O\left(\frac{nk}{\sqrt{p}} + k^2\right)$ and the total latency cost is $O(\log p)$. Since $|V\prime(i,j)| = 2\left(n/\sqrt{p} + k\right)$ and $|E\prime(i,j)| = O\left(\frac{n^2}{p} + k^2\right)$, the memory requirement per processor is $M = O\left(\frac{n^2}{p} + k^2\right)$.

In the second phase, each processor p_{ij} locally executes the Floyd-Warshall algorithm to calculate the APSP of $G\prime(i,j)$. According to Theorem 1, $D^+(i,j) = D\prime(i,j)$ for all $i,j \in \{1,2,\ldots,\sqrt{p}\}$. Since D^+ consists of all blocks $D^+(i,j)$, p processors can directly obtain the shortest distance matrix D^+ of the updated graph G^+ in this phase. Note that only computation cost is incurred in this phase. Since there are $2\left(\frac{n}{\sqrt{p}} + k\right)$ vertices in $G\prime(i,j)$, the total computation cost is $O\left(\left(\frac{n}{\sqrt{p}} + k\right)^3\right)$.

We give the communication cost, computational cost, and memory requirements of the incremental algorithm in Theorem 2.

Theorem 2. *Given the initial graph G, its shortest distance matrix D, and the set of k edges ΔE inserted into graph G, our incremental algorithm can maintain the APSP of G^+. This algorithm has a bandwidth cost of $O\left(\frac{nk}{\sqrt{p}} + k^2\right)$, a latency cost of $O(\log p)$, a computation cost of $O\left(\left(\frac{n}{\sqrt{p}} + k\right)^3\right)$, and a memory requirement of $O\left(\frac{n^2}{p} + k^2\right)$.*

Corollary 1. *Given the initial graph G, its shortest distance matrix D, and the set of k edges ΔE inserted into the graph G, if $k = O\left(\frac{n}{\sqrt{p}}\right)$, our incremental algorithm can maintain the APSP of G^+. This algorithm has a bandwidth of $O\left(\frac{n^2}{p}\right)$, a latency cost of $O(\log p)$, a computation cost of $O\left(\frac{n^3}{p^{3/2}}\right)$ computation cost, and a memory requirement of $O\left(n^2/p\right)$.*

Since initially the $n \times n$ shortest distance matrix D is evenly stored on p processors, the memory requirement per processor is at least $\Omega\left(\frac{n^2}{p}\right)$. Therefore, when $k = O\left(\frac{n}{\sqrt{p}}\right)$ edges are inserted, our incremental algorithm correctly solves the shortest distance of the updated graph G^+ without the need for additional available memory. Compared to computing the APSP of G^+ from scratch, both the bandwidth and computation upper bounds of our incremental algorithm are lowered $O(\sqrt{p})$ times, and the latency upper bound is lowered $O(\sqrt{p}\log p)$ times.

4 Batch edge deletions

In this section, for the deletion of edges $\Delta E = \{e_{a_1b_1}, e_{a_2b_2}, \ldots, e_{a_kb_k}\}$, we propose an efficient decremental algorithm to maintain the shortest distance of G^-. Next, we first give three observations about D and D^- for edge deletions.

1) The shortest distance of each vertex pair (v_x, v_y) in graph G^- cannot be less than that in graph G, i.e., $D^-_{xy} \geq D_{xy}$.
2) If the shortest distance between v_x and v_y in G does not contain any edge in ΔE, then $D^-_{xy} = D_{xy}$.
3) If $D^-_{xy} > D_{xy}$, then the shortest distance between v_x and v_y in G contains no less than one edge in ΔE.

Consider that for edge insertions, the shortest path of any vertex pair in the updated graph G^+ is either unchanged or contains no less than one of the inserted edges. However, for edge deletions, it does not hold and the shortest path in G^- may be arbitrary. Therefore, it is more difficult to maintain the APSP for edge deletions. In addition, for the deleted edge $e_{a_mb_m}$, even if $\left|e_{a_mb_m}\right| \geq D_{a_mb_m}$, we cannot prune $e_{a_mb_m}$ from ΔE because the deletion of ΔE may cause the path $D_{a_mb_m}$ unavailable, resulting in $e_{a_mb_m}$ being the shortest distance between a_m and b_m in G^-.

Our method has three phases. At a high level, in the first phase, we identify those unchanged elements in D^-, i.e., $D^-_{xy} = D_{xy}$; second, we rearrange the structure of D^- so that those unchanged elements are arranged into the same blocks, resulting in some unchanged blocks consisting of unchanged elements; finally, we use a variant of the path doubling method to compute D^-, where the recomputation of unchanged blocks in D^- can be avoided.

More specifically, in the first phase, by a graph construction similar to that introduced in Sect. 3.1, each processor p_{ij} constructs a graph $G\prime(i,j)$ (cf. Figure 3). Then each processor obtains the distance $D\prime(i,j)$ between V_i and V_j by locally computing the APSP of $G\prime(i,j)$. The goal of constructing such a graph is to make the distance $D\prime(i,j)_{xy}$ between v_x and v_y in $G\prime(i,j)$ equal to the shortest distance of the paths between v_x and v_y in G that contains at least one edge in ΔE. If this goal is achieved, then from observation (2), the inequality $D\prime(i,j)_{xy} \neq D(i,j)_{xy}$ implies that the shortest distance between v_x and v_y won't be changed, i.e., $D^-(i,j)_{xy} = D(i,j)_{xy}$. We give the proof in Lemma 1. Let $F(i,j)$ be the set of vertex pairs with the unchanged shortest distance, if the above description holds, then.

$$F(i,j) = \{(x,y) | D^-(i,j)_{xy} \neq D(i,j)_{xy}.$$

In the second phase, since data is stored on p processors in a block layout, and data movement between processors is in blocks, we increase temporary data locality to improve concurrency. The goal of this phase is to arrange those vertex pairs with the unchanged shortest distance into the same block in D^-. Specifically, each processor p_{ij} locally generates a graph $G^*(i,j)$ where $V^*(i,j) = V_i \cup V_j$ and $E^*(i,j) = \{e^*{}_{xy} | (x,y) \in F(i,j)\}$. Let $G^* = (V^*, E^*)$ be the generated graph on p processors consisting of all $G^*(i,j)$, where $V^* = \bigcup_{i,j\in\{1,2,\ldots,\sqrt{p}\}} V^*(i,j)$ and $E^* = \bigcup_{i,j\in\{1,2,\ldots,\sqrt{p}\}} E^*(i,j)$, we perform vertex reorder on G^* by a process known as nested-dissection (ND) (George 1973). Assuming that each element in a block is unchanged, we can avoid recalculating this block. If this process is not used, then each block contains both changed and unchanged elements, which causes each block to be recomputed.

In the third phase, we compute the APSP of graph G^- by the path doubling method, i.e. $D^- = (A^-)^n$. This method performs $\log n$ semiring matrix multiplications. We use a variant of this method that only needs to perform $\log p$ semiring matrix multiplications. During each semiring matrix multiplication, the recomputation of those blocks with unchanged shortest distances obtained in the second phase can be avoided.

Below we introduce these three phases in detail.

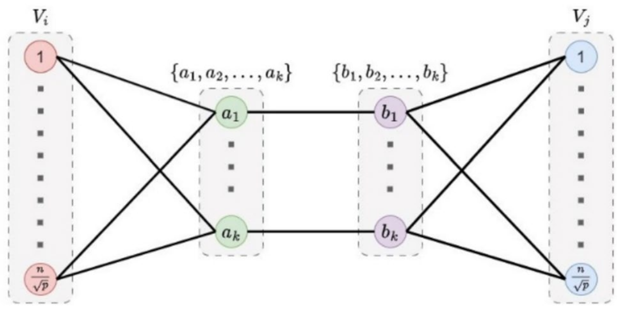

Fig. 3 The constructed graph $G\prime(i,j)$ on p_{ij} for edge deletions

4.1 The graph constructed on each processor

For edge deletions, the graph $G\prime(i,j)$ constructed on processor p_{ij} can be seen as a subgraph of the construction graph shown in Fig. 2. See Fig. 3, $G\prime(i,j)$ has a total of $2(n/\sqrt{p} + k)$ vertices, which consists of four vertex sets, i.e., V_i, $\{a_1, a_2 \dots, a_k\}$, $\{b_1, b_2 \dots, b_k\}$ and V_j. Let the edges between the four vertex sets be set as follows:

- For any two vertices $v_x \in V_i$ and $a_m \in \{a_1, a_2 \dots, a_k\}$, there exists an edge, and the edge weight is $\left|e\prime(i,j)_{xa_m}\right| = D_{xa_m}$.
- For any two vertices $a_m \in \{a_1, a_2 \dots, a_k\}$ and $b_m \in \{b_1, b_2 \dots, b_k\}$, there exists an edge, and the edge weights is $\left|e\prime(i,j)_{a_m b_m}\right| = \left|e_{a_m b_m}\right|$.
- For any two vertices $v_y \in V_j$ and $b_m \in \{b_1, b_2 \dots, b_k\}$, there exists an edge, and the edge weight is $\left|e\prime(i,j)_{b_m y}\right| = D_{b_m y}$.

Since the constructed graph shown in Fig. 3 is a subgraph of Fig. 2, constructing it on each processor requires less communication. Therefore, for edge deletions, the bandwidth cost of constructing the graph is $O\left(\frac{nk}{\sqrt{p}} + k^2\right)$ and the latency cost is $O(\log p)$.

As mentioned earlier, unlike the case of edge insertions, $D^-(ij)$ cannot be got by computing the distance $D\prime(i,j)$ between V_i and V_j in $G\prime(i,j)$. Instead, we use $D\prime(i,j)$ to determine those vertex pairs whose shortest distances are unchanged. Let O_{xy} be the path set between v_x and v_y in G that contain no less than one edge in ΔE, i.e., $O\_\{xy\} = \{\pi\_\{xy\} | \pi\_\{xy\} \cap \Delta E \neq \emptyset\}$, and let $min\left(O_{xy}\right)$ be the shortest path in O_{xy}. We prove in Theorem 3 that $D\prime(i,j)_{xy} = \left|min\left(O_{xy}\right)\right|$.

Theorem 3 $D\prime(i,j)_{xy} = \left|\min\left(O_{xy}\right)\right|$

Proof. (1) $D\prime(i,j)_{xy} \geq \left|\min\left(O_{xy}\right)\right|$: Since there is no edge between V_i and V_j in $G\prime(i,j)$, the shortest path $\pi_{xy}\prime$ between v_x and v_y contains no less than one edge in ΔE.

From the construction of $G\prime(i,j)$, for each edge in $\pi_{xy}\prime$, we can find a corresponding path in G with equal weights and the same endpoints. Thus there exists a path $\pi_{xy} \in O_{xy}$ in G such that $\left|\pi_{xy}\right| = \left|\pi_{xy}\prime\right|$. From $|\min(O_{xy})| \leq |\pi\_\{xy\}|$ and $\left|\pi_{xy}\prime\right| = D\prime(i,j)_{xy}$, we have $D\prime(i,j)_{xy} \geq \left|\min\left(O_{xy}\right)\right|$.

(2)$D\prime(i,j)_{xy} \leq \left|\min\left(O_{xy}\right)\right|$: Without loss of generality, let the path π_{xy} be $\min\left(O_{xy}\right)$ and $e_{a_m b_m} \in \Delta E$ be an edge contained inπ_{xy}. Obvious$\left|\pi_{xy}\right| = D_{xa_m} + \left|e_{a_m b_m}\right| + D_{b_m y}$. Consider such a path$\pi_{xy}\prime in G\prime(i,j)$, which consists of three edges:$e\prime(i,j)_{xa_m}$, $e\prime(i,j)_{a_m b_m}$, and$e\prime(i,j)_{b_m y}$. From the construction of$G\prime(i,j)$, we have$\left|e\prime(i,j)_{xa_m}\right| = D_{xa_m}$,$\left|e\prime(i,j)_{a_m b_m}\right| = \left|e_{a_m b_m}\right|$, and $\left|e\prime(i,j)_{b_m y}\right| = D_{b_m y}$. Therefore,$\left|\pi_{xy}\prime\right| = \left|\pi_{xy}\right|$. From $D_{xy}\prime(i,j) \leq \left|\pi_{xy}\prime\right|$ and $\left|\pi_{xy}\right| = \left|\min\left(O_{xy}\right)\right|$, we get$D\prime(i,j)_{xy} \leq \left|\min\left(O_{xy}\right)\right|$. □

By Theorem 3, $D(i,j)_{xy} \neq D\prime(i,j)_{xy}$ means that the shortest distance between v_x and v_y in graph G contains no edge in ΔE. Therefore, from observation (2), deleting ΔE won't change the distance between v_x and v_y. From this, we give in Lemma 1 the set F(i, j) of vertex pairs whose shortest distance does not change.

Lemma 1 $F(i,j) = \{(x,y) | D(i,j)_{xy} \neq D'(\mathrm{i,j})_{\mathrm{xy}}\}$.

4.2 Vertex reordering

For each vertex pair $(x,y) \in F(i,j)$, its shortest distance D^-_{xy}, which is an element in D^-, does not need to be recomputed when computing the APSP of G^-. However, D^- is distributed over the p processors with block layout and each block consists of changed elements and unchanged elements. This causes each block in D^- to be recomputed even if it contains some unchanged elements. To increase data locality, we use the vertex reordering technique to

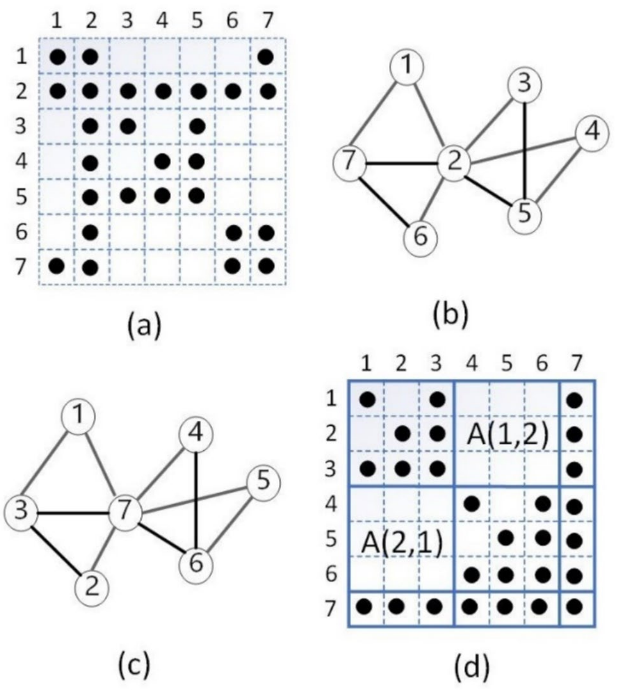

Fig. 4 **a** shows the two sets F and $\overline{F}$ that have changed shortest distance and unchanged shortest distance when ΔE is deleted from graph G. The circles represent the changed elements. **b** demonstrates the graph G^* generated from Fig. 4(a). **c** demonstrates the graph after performing the ND process and reordering the vertices on G^*. **d** shows the adjacency matrix of the reordered graph in Fig. 4(c)

rearrange the elements in D^- so that the unchanged elements are arranged to the same block.

With slight abuse of notation, we denote by $F = \bigcup_{i,j \in \{1,2,\ldots,\sqrt{p}\}} F(i,j)$ all vertex pairs with the unchanged shortest distance, and denote by $\overline{F} = \{(x,y)|(x,y) \notin F\}$ all vertex pairs with changed shortest distance (see Fig. 4(a)). Consider such a generated graph G^* where $V^* = V$ and for each.

$(x,y) \in \overline{F}$, there is an edge e^*_{xy} in G^* with weight 1 (see Fig. 4(b)). Then the adjacency matrix A^* for graph G^* is.

$$A^*_{xy} = \begin{cases} 1 \mathit{If}\,(x,y) \in \overline{F}\left(\text{in this case}, D^-_{xy} \neq D_{xy}\right) \\ \infty \mathit{If}\,(x,y) \in F\left(\text{in this case}, D^-_{xy} = D_{xy}\right), \end{cases}$$

Which can be considered to have a similar structure to D^-.

Due to the similar structure, we can rearrange the elements of A^* and D^- by performing a vertex reordering of graph G^*, which can be achieved by a process called "nest-dissection" (ND) (George 1973). This process can be obtained by certain graph partitioning tools, such as Metis (Karypis and Kumar 1998). Specifically, the ND process recursively solves the vertex separator S of G^*, where S partitions V^* into three disjoint sets, $V^* = V_1 \cup V_2 \cup S$. The partition of V using the ND process can satisfy three conditions:

(1) V_1 and V_2 are not connected by edges. That is, if $v_x \in V_1$ and $v_y \in V_2$, then $e_{xy} \notin E^*$;
(2) $V_1$and V_2 have approximately equal size, i.e., $|V_1| \approx |V_2|$;
(3) The size of the separator S is small.

Karypis and Kumar presented a method for solving the separator of a graph in parallel (Karypis and Kumar 1988), which has $O\left(n\log p/\sqrt{p}\right)$ bandwidth cost and $O(\log p)$ latency cost. Briefly, this method first solves the 2-way partition of the input graph $G = (V,E)$, which divides V into two disjoint subsets A and B with equal size and minimizes the number of edges in E with two vertices in A and B, respectively. Then, the method finds the minimum vertex cover S of the set of edges between A and B. For any edge between A and B, it has at least one vertex in S, so the vertex set V of G is separated by S.

In Fig. 4(b), $S = \{2\}$ is a separator of G^* and $V_1 = \{1,6,7\}$, $V_2 = \{3,4,5\}$. It can be found that there are no edges between V_1 and V_2, which means that D^-_{xy} is an unchanged element for all $v_x \in V_1$ and $v_y \in V_2$.

However, those unchanged elements are not consecutively arranged in a block. To solve the problem, we reorder the vertices in V^* so that the vertices within each set V_1 and V_2 have consecutive indices, and the vertices in S have a higher index than any vertex in V_1 and V_2. For example, Fig. 4(c) shows the vertex indexes after vertex reordering, where $S = \{7\}, V_1 = \{1,2,3\}$ and $V_2 = \{4,5,6\}$. As shown in Fig. 4(d), after vertex reordering, there are two blocks in D^- that consist of unchanged elements, i.e., $A(1,2)$ and $A(2,1)$.

More unchanged blocks can be got by recursively solving the separator of V_1 and V_2. For example, V_1 can be divided into $S_1 = \{3\}$, $V_{11} = \{1\}$ and $V_{12} = \{2\}$, then the shortest distance between V_{11} and V_{12} is also an unchanged block.

The elimination tree (eTree) can be used to describe the vertex reordering. As shown in Fig. 5(a), the separator S is the parent of V_1 and V_2 in the eTree. Figure 5(b) shows a multilevel eTree obtained by recursively solving separators on V_1 and V_2. We denote by $\mathcal{A}(\mathrm{i})$ the ancestor node set of V_i, where $\mathcal{A}(i)$ consists of V_i's parent, parent of parent or so on in the eTree. Similarly, we denote by $\mathcal{D}(i)$ the descendant node set of V_i, where $\mathcal{D}(i)$ consists of V_i's child, child of child or so on in the eTree. We denote by $\mathcal{C}(i)$ the cousin node set of V_i, where $\mathcal{C}(i)$ consists of all nodes that are neither ancestors nor descendants of V_i. For example, for the eTree in Fig. 5(b), $\mathcal{A}(6) = \{7\}$, $\mathcal{D}(6) = \{4,5\}$ and $\mathcal{C}(6) = \{1,2,3\}$. In an eTree, if two nodes V_i and V_j are cousins of each other, then there is no edge between V_i and V_j in G^*, which means that the shortest distance matrix block between V_i and V_j is unchanged. Conversely, if V_i is an ancestor or descendant of V_j, then the distance matrix block between V_i and V_j is changed. Figure 5(c) shows the block structure obtained from the eTree shown in Fig. 5(b), where each circle represents a changed block. Based on this, we can compute the number of changed blocks in the rearranged matrix D^-.

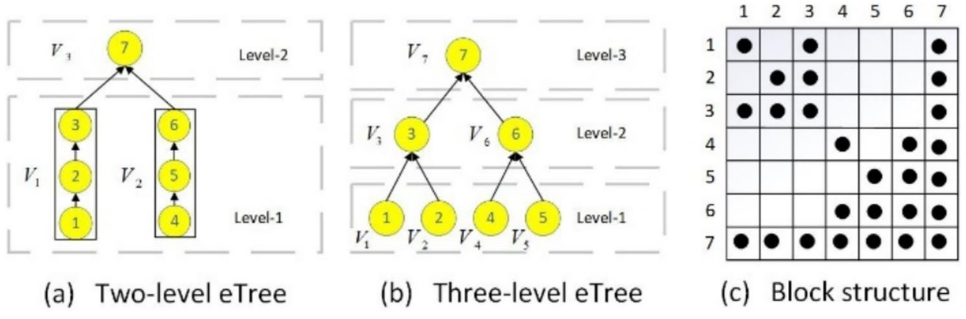

Fig. 5 **a** is a two-level eTree, where V_3 is the separator. **b** shows a three-level eTree, which is got by recursively solving the separator of V_1 and V_2 in (**a**). **c** shows the matrix block structure obtained from the eTree shown in (**b**). If $i \in \mathcal{A}(j) \cup \mathcal{D}(j)$, $D^-(i,j)$ is a changed block and is denoted by a circle

Theorem 4. *By vertex reordering, the shortest distance matrix D^- can be rearranged into p blocks, with $O\left(\sqrt{p}\log p\right)$ changed blocks.*

Proof. An eTree of height h can be obtained by recursively solving the separator $h-1$ times and has $\sum_{l=1}^{h} 2^{h-l}$ nodes (see Fig. 5(a) and (b)). The shortest distance between any two nodes V_i and V_j in the eTree corresponds to a block $D^-(i,j)$ in D^- (see Fig. 4(d) and 5(a)). To distribute the shortest distance matrix to $\sqrt{p}\times\sqrt{p}$ processors with block layout, we specify $\sum_{l=1}^{h} 2^{h-l}=\sqrt{p}$ so that the eTree has $\sqrt{p}$ nodes and D^- has p blocks. Thus the height of the eTree is obtained as $h=\log\left(\sqrt{p}+1\right)$.

Recall that for any two nodes V_i and V_j in the eTree, if V_i is the ancestor or descendant of V_j, then the block $D^-(i,j)$ is changed. Considering that in an eTree, the number of nodes at l-th level is 2^{h-l}, and each node has $h-l$ ancestors and $\sum_{k=1}^{l-1} 2^k = 2^l-2$ descendants. Thus, for the l-th level, the number of changed blocks is $2^{h-l}\cdot\left(h-l+2^l-2\right)$, which is less than $h\cdot 2^{h-l}+2^h$. Therefore, the amount of changed blocks in D^- is less than $O\left(\sum_{l=1}^{\mathrm{h}} h\cdot 2^{h-l}+2^h\right)=O\left(\sqrt{p}logp\right)$. □

We assume the size of separators is monotonic with the graph size, that is,

$$n_1 \ge n_2 \Rightarrow S\left(n_1\right) \ge S\left(n_2\right),$$

where $S(n)$ is the separator size of a graph with n vertices. Given an eTree of height h, the nodes from the 2-th level to the h-th level are separators obtained by calculation, while the nodes in the first level are not. For example, in the three-level eTree shown in Fig. 5(b), only V_5, V_6, and V_7 are separators obtained by the ND process. Among these separators, the separator of the h-th level has the largest size. From this, we give the size per block in the reordered matrix in Lemma 2.

Lemma 2. *The size per block in the reordered matrix D^- is $O\left(\frac{n^2}{p}+|S|^2\right)$.*

Proof. For two nodes V_i and V_j in the eTree, the size of the distance block between V_i and V_j is $|D^-(i,j)|=\left|V_i\right|\cdot\left|V_j\right|$. Suppose S is the separator of G^*, then S is the h-th level separator in the eTree, and the size of any separator from the 2-th level to the h-th level in the eTree does not exceed $|S|$. In addition, because the total number of vertices is $|V^*|=n$ and the number of nodes in the first level is $2^{h-1}=\Theta\left(\sqrt{p}\right)$, each node in the first level contains at most $O\left(\frac{n}{\sqrt{p}}\right)$ vertices. Therefore, the size of each node in the eTree is $O\left(\frac{n}{\sqrt{p}}+|S|\right)$, and the size of each block in D^- is $O\left(\frac{n^2}{p}+|S|^2\right)$.

Recall that the bandwidth cost and latency cost of solving the separator of a graph with n vertices are $O\left(\frac{n\log p}{\sqrt{p}}\right)$ and $O(\log p)$, respectively. To get the 2^{h-l} nodes in the l-th level of the eTree, the processors and graph are both divided into 2^{h-l} parts to compute these nodes in parallel. Specifically, each $\frac{p}{2^{h-l}}$ processors computes the separator of a graph with about $\frac{n}{2^{h-l}}$ vertices. Therefore, the total bandwidth cost of solving all separators in the eTree of height $h=O(\log p)$ is

$$O\left(\sum_{l=1}^{h}\frac{n\log\frac{p}{2^{h-l}}}{2^{h-l}\sqrt{\frac{p}{2^{h-l}}}}\right)=O\left(\frac{n\log^2 p}{\sqrt{p}}\right),$$

and the total latency cost is

$$O\left(\sum_{l=1}^{h}\log\frac{p}{2^{h-l}}\right)=O\left(\log^2 p\right).$$

To summarize, we get an eTree with $\sqrt{p}$ nodes by vertex reordering. For any two nodes V_i and V_j in the eTree, $D^-(i,j)$ is a changed block in D^- if $i\in\mathcal{A}(j)\cup\mathcal{D}(j)$, and $D^-(i,j)$ is an unchanged block if $i\in\mathcal{C}(j)$. There are $O\left(\sqrt{p}\log p\right)$ changed blocks in D^- that need to be recomputed, and the recomputation of other unchanged blocks can be avoided. Below we give a scheme to efficiently update the changed blocks by exploiting the known unchanged shortest distances.

4.3 Path doubling method

For edge deletions, by combining the strategies above, we present a modified version of the path doubling method described in Sect. 2.3. We do not adopt the Floyd-Warshall algorithm because it updates each diagonal block $D^-(i,i)$ sequentially, which results in a latency cost of at least $\Omega\left(\sqrt{p}\log p\right)$. Although the latency cost of using path doubling to compute the APSP of G^- from scratch is also $\Omega\left(\sqrt{p}\log p\right)$, we can reduce it to $O\left(\log^3 p\right)$ by not computing the unchanged blocks.

Review the path doubling method described in Sect. 2.3, from Eq. (1), the shortest distance matrix of G^- can be obtained as

$$(A^-)^*=(A^-)^p(q)^*\otimes(A^-)^p(q\le p). \tag{2}$$

In Eq. (2), calculating $(A^-)^p$ requires about $\log p$ semiring matrix multiplications, i.e., $(A^-)^2,(A^-)^4,\dots,(A^-)^p$, which dominates the bandwidth and latency costs. For the unchanged block (i,j), our goal is to avoid calculating the (i,j)-th block in the output of each semiring matrix multiplication, i.e., $(A^-)^2(i,j), (A^-)^4(i,j), \ldots, (A^-)^p(i,j)$, thus reducing bandwidth and latency. We give a simple strategy to achieve this goal. Specifically, if $D^-(i,j)$ is an unchanged block, during each matrix multiplication,

$(A^-)^2(i,j), (A^-)^4(i,j), \ldots, (A^-)^p(i,j)$ are not computed but directly replaced by $D^-(i,j)$. Such a strategy does not affect the correctness of the algorithm. This is since.

each element in $(A^-)^2(i,j), (A^-)^4(i,j), \ldots, (A^-)^p(i,j)$ is replaced by a shorter path, which only causes the distance matrix to converge faster.

From the above description, for each semiring matrix multiplication, the (i,j)-th block of the output needs to be computed only if $D^-(i,j)$ is a changed block, and there are $O(\sqrt{p}\log p)$ such blocks (cf. Theorem 4). For each semiring matrix multiplication, we give below an efficient scheme to compute these $O(\sqrt{p}\log p)$ blocks with low communication cost.

Take the calculation of $(A^-)^2$ as an example, i.e., $(A^-)^2 = A^- \otimes A^-$, each changed block of $(A^-)^2$ is updated as

$$(A^-)^2(i,j) = \min_k (A^-(i,k) \otimes A^-(k,j)), \tag{3}$$

where $k = \{1,2,\ldots,\sqrt{p}\}$. For each k, we consider $A^-(i,k) \otimes A^-(k,j)$ as a computational unit that can be executed locally by a single processor, then the computation of a block requires $\sqrt{p}$ computational units, which can be computed in parallel on $\sqrt{p}$ processors. Thus p processors can compute $\sqrt{p}$ blocks in parallel at a time. Based on this, we divide the $\sqrt{p}\log p$ blocks that need to be computed into $\log p$ parts and each part has $\sqrt{p}$ blocks. For the computation of each part, let the computational tasks of a single block be assigned to a single processor row, and each processor in that row executes one computation unit. Note that before the processor locally executes the computation unit, it has to get the required data from other processors to execute that computation unit.

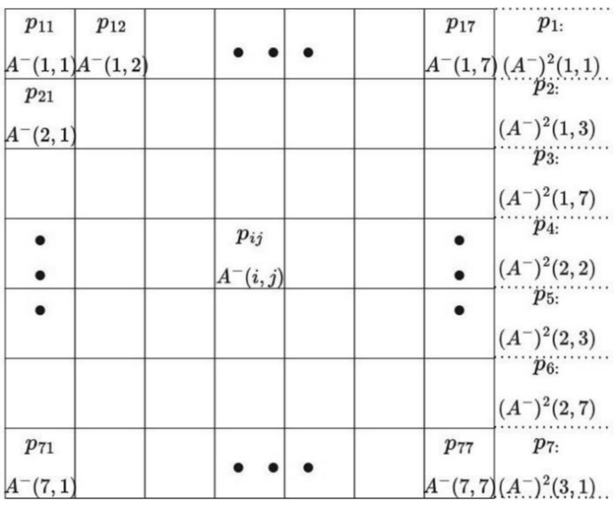

Fig. 6 The data layout on the 7×7 processor grid and the data being computed on each processor row

Here we take the calculation of the first $\sqrt{p}$ changed blocks as an example. Assume that matrix A^- has the block structure shown in Fig. 5(c), where each circle denotes a changed block. Since the grid is 7×7, we can get that the first 7 blocks being calculated are the set $\{(1,1),(1,3),(1,7),(2,2),(2,3),(2,7),(3,1)\}$. As illustrated in Fig. 6, each processor p_{ij} maintains a local matrix block $A^-(i,j)$ and let the i-th block be computed on processor row $p_{i:}$. On the right side of the figure are the matrix blocks that need to be computed for each processor row. For example, the computation of the second block $(A^-)^2(1,3)$ is

$$(A^-)^2(1,3) = \min_k (A^-(1,k) \otimes A^-(k,3)), k = \{1,\ldots,\sqrt{p}\},$$

which is executed on $p_{2:}$. Thus $p_{2:}$ needs to get the first row $A^-(1,:)$ and the third column $A^-(:,3)$ of matrix A^-. Specifically, for each $k \in \{1,2\ldots,\sqrt{p}\}$, processor p_{2k} needs to access $A^-(1,k)$ and $A^-(k,3)$ for performing the computing unit $A^-(1,k) \otimes A^-(k,3)$. The calculations of the other six blocks are similar. For the computation of each $\sqrt{p}$ blocks, each processor can get the required data through row broadcast and column broadcast, as shown in Figs. 7(a) and (b).

Row broadcast: Processor rows $p_{1:}$, $p_{2:}$, and $p_{3:}$ perform the calculations of blocks (1,1), (1,3), and (1,7), respectively, all of which require the first row of data $A^-(1,:)$ of A^-. Therefore, the first row $A^-(1,:)$ is broadcast by processor row $p_{1:}$ to the processor group $\{p_{1:}, p_{2:}, p_{3:}\}$ (see Fig. 7(a)). Specifically, processor p_{1k} broadcasts $A^-(1,k)$ to the processor group $\{p_{1k}, p_{2k}, p_{3k}\}$. By a similar analysis, $A^-(2,:)$ is broadcast from $p_{2:}$ to the processor group $\{p_{4:}, p_{5:}, p_{6:}\}$ and $A^-(3,:)$ is broadcast from $p_{3:}$ to $\{p_{7:}\}$.

Column broadcast: Processor rows $p_{1:}$ and $p_{7:}$ perform the calculations of blocks (1,1) and (3,1), respectively, both of which require the first column of data $A^-(:,1)$ of A^-. Therefore, the first column $A^-(:,1)$ is broadcast by processor column $p_{:1}$ to the processor group $\{p_{1:}, p_{7:}\}$ (see. Figure 7(b)). Specifically, processor p_{k1} broadcasts $A^-(k,1)$ to the processor group $\{p_{1k}, p_{7k}\}$. By a similar analysis, $A^-(:,2)$ is broadcast from $p_{:2}$ to $\{p_{4:}\}$, $A^-(:,3)$ is broadcast from $p_{:3}$ to $\{p_{2:}, p_{5:}\}$, and $A^-(:,7)$ is broadcast from $p_{:7}$ to $\{p_{3:}, p_{6:}\}$.

With the above two operations, each processor row gets all the data needed to compute a block $(A^-)^2(i,j)$. Next, each processor in that row executes the local computation unit and then attributes the result to the processor P_{ij} by a reduction operation such that P_{ij} gets $(A^-)^2(i,j) = \min_k(A^-(i,k) \otimes A^-(k,j))$.

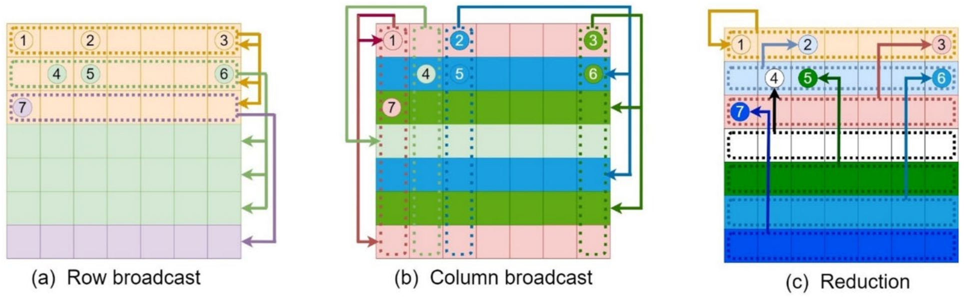

Fig. 7 **a** and **b** illustrate row and column broadcast operations, respectively, so that each processor row $p_{q:}$ acquires the data needed to compute the q-th block (the data in the dashed boxes are transferred). **c** illustrates the reduction operation in each processor row such that p_{ij} get the changed block $(A^{-})^2(i,j)$

Reduction: Note that each processor row computes one block. For the processor row that computes block $(A^{-})^2(i,j)$, the computation results are reduced to the processor p_{ij}. For example, as shown in Fig. 7(c), processor rows $p_{1:}, p_{2:}, \dots, p_{7:}$ reduce local computation results to processors $p_{11}, p_{13}, p_{17}, p_{22}, p_{23}, p_{27}$ and p_{31}, respectively. From the above analysis, p processors can compute $\sqrt{p}$ blocks by letting each processor participate in only $O(1)$ broadcast and reduction operations. Since there are $O\left(\sqrt{p}\log p\right)$ blocks that need to be computed for each semiring matrix multiplication, it can be achieved by $O(\log p)$ broadcast and reduction operations. In addition, calculating $(A^{-})^n$ requires $O(\log p)$ semiring matrix multiplications (cf. Equation (2)), thus a total of $O\left(\log^2 p\right)$ broadcast and reduction operations need to be performed. From this, we give the bandwidth cost and latency cost of computing $(A^{-})^n$ in Theorem 5.

Theorem 5. *The computation of* $(A^{-})^n$ *has a bandwidth cost of* $O\left(\frac{n^2}{p}\log^3 p + |S|^2\log p\right)$ *and a latency cost of* $O\left(\log^3 p\right)$.

Proof. Note that in each semiring matrix multiplication, the input and output matrices have a similar block structure as the rearranged matrix D^{-}. Therefore, the size of each block in the broadcast and reduction operations is $O\left(\frac{n^2}{p} + |S|^2\right)$ (cf. Lemma 2). Computing $(A^{-})^n$ requires $\mathrm{O}\left(\log^2 \mathrm{p}\right)$ broadcast and reduction operations. Therefore, from Table 1, the bandwidth and latency costs are $O\left(n^2\log^3 p/p + |S|^2\log p\right)$ and $O\left(\log^3 p\right)$, respectively. □

From Theorem 5, we get the communication cost of the third phase. Note that when $k = O\left(\frac{n}{\sqrt{p}}\right)$, the communication cost of the first two phases is dominated by the communication cost of the third phase. Hence, the asymptotic communication cost of our decremental algorithm is equal to that of its third phase.

Lemma 3. *Assuming that* $k = O\left(\frac{n}{\sqrt{p}}\right)$ *and the size of the separator obtained in the second phase is* $|S| = O\left(\frac{n}{\sqrt{p}}\right)$, *then the bandwidth and latency costs of our decremental algorithm are* $O\left(\frac{n^2\log^3 p}{p}\right)$ *and* $O\left(\log^3 p\right)$.

Proof. This is obvious since $|S| = O\left(\frac{n}{\sqrt{p}}\right)$ and the decremental algorithm has an equal asymptotic communication cost with its third phase for $k = O\left(\frac{n}{\sqrt{p}}\right)$, which is given in Theorem 5.

Since p processors initially store the distance matrix D of the initial graph G, the memory requirement per processor is at least $\Omega\left(\frac{n^2}{p}\right)$. When $k = O\left(\frac{n}{\sqrt{p}}\right)$ edges are deleted and the separator size is $|S| = O\left(\frac{n}{\sqrt{p}}\right)$, our decremental algorithm correctly solves the shortest distance of the updated graph G^{-} with no need of extra available memory. Compared to computing the APSP of G^{-} from scratch, the bandwidth upper bound of our decremental algorithm is asymptotically reduced $O\left(\sqrt{p}/\log^3 p\right)$ times, and the latency upper bound is asymptotically reduced $O\left(\sqrt{p}/\log p\right)$ times.

5 Conclusion

For the insertion and deletion of multiple edges, we propose two novel parallel algorithms to maintain the shortest distances of the dynamic graph. Compared to the static parallel APSP algorithm, our algorithm asymptotically reduces bandwidth, latency, and computational costs without extra available memory.

The incremental algorithm hinge upon the observation that the shortest path is either unchanged or contains at least one of the inserted edges. The decremental algorithm first calculates the shortest distances that will change. Then these changed elements are rearranged to the same block using the vertex reordered technique. Finally, we recompute these changed blocks using the path doubling principle. Further, we propose a new scheduling such that the recomputation of those changed blocks is evenly distributed over the processor rows, thus balancing the computation and data movement. A disadvantage of our decremental algorithm is that the communication cost is affected by the separator size $|S|$, which means it may not be as efficient when the shortest path of a dynamic graph is heavily altered.

Several issues seem worthy of further investigation. First, while additional memory in a distributed memory model can undoubtedly reduce communication, an exploration of the trade-offs between the two is missing. Second, can we efficiently solve the shortest distance in parallel when the dynamic graphs are inserted and deleted multiple edges at the same time?

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

Data availability This paper is a theoretical proof paper and does not involve the available data.

Declarations

Conflict of interest On behalf of all authors, the corresponding author states that there is no conflict of interest.

References

Ballard, G., Carson, E.C., Demmel, J., Hoemmen, M., Knight, N., Schwartz, O.: Communication lower bounds and optimal algorithms for numerical linear algebra. Acta Numer. **23**, 1–155 (2014)

Ballard, G., Demmel, J., Holtz, O., Schwartz, O.: Graph expansion and communication costs of fast matrix multiplication. In Proc. the 23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures, San Jose, CA, USA, 4–6 June, pp. 1–12, ACM, (2011)

Ballard, G., Demmel, J., Holtz, O., Lipshitz, B., Schwartz, O.: Communication-optimal parallel algorithm for Strassen's matrix multiplication. In Proc. the 24th Annual ACM Symposium on Parallelism in Algorithms and Architectures, Pittsburgh, PA, USA, 25–27 June, pp.193–204, ACM, (2012)(

Ballard, G., Buluç, A., Demmel, J., Grigori, L., Lipshitz, B., Schwartz, O., Toledo, S.: Communication optimal parallel multiplication of sparse random matrices. In Proc. the 25th Annual ACM Symposium on Parallelism in Algorithms and Architectures, Montreal, QC, Canada, 23–25 July, pp.222–231ç, ACM, (2013)

Ballard, G., Demmel, J., Grigori, L., Jacquelin, M., Knight, N.: A 3d parallel algorithm for QR decomposition. In Proc. the 30rd on Symposium on Parallelism in Algorithms and Architecture}, Vienna, Austria, 16–18 July, pp.55–65, ACM, (2018)

Cicerone, S., Stefano, G.D., Frigioni, D., Nanni, U.: A fully dynamic algorithm for distributed shortest paths. Theor. Comput. Sci. **297**(1–3), 83–102 (2003)

Cicerone, S., D'Angelo, G., Stefano, G.D., Frigioni, D.: Partially dynamic efficient algorithms for distributed shortest paths. Theor. Comput. Sci. **411**(7–9), 1013–1037 (2010)

Demetrescu, C., Italiano, G. F.: A new approach to dynamic all pairs shortest paths. In Proc. the 35th Annual ACM Symposium on Theory of Computing, San Diego, CA, {USA}, 9–11 June, pp.159–166, ACM, (2003)

Demmel, J., Eliahu, D., Fox, A., Kamil, S., Lipshitz, B., Schwartz, O., Spillinger, O.: Communication-optimal parallel recursive rectangular matrix multiplication. In Proc. the 27th IEEE International Symposium on Parallel and Distributed Processing, Cambridge, MA, USA 20–24 May, pp.261–272, IEEE Computer Society, (2013)

Dhulipala, L., Durfee, D., Kulkarni, J., Peng, R., Sawlani, S., Sun, X.: Parallel batch-dynamicgraphs: Algorithms and lower bounds. In Proc. the 2020 ACM-SIAM Symposium on Discrete Algorithms, Salt Lake City, UT, USA, 5–8 January, pp.1300–1319, SIAM, (2020)

Even, S., Shiloach, Y.: An on-line edge-deletion problem J. . ACM **28**(1), 1–4 (1981)

Floyd, R.W.: Algorithm 97: Shortest path. Commun. ACM **5**(6), 345 (1962)

George, A.: Nested dissection of a regular finite element mesh. SIAM J. Numer. Anal. **10**(2), 345–363 (1973)

Greco, S., Molinaro, C., Pulice, C.: Efficient maintenance of shortest distances in dynamic graphs. IEEE Trans. Knowl. Data Eng. **30**(3), 474–487 (2018)

Greco, S., Molinaro, C., Pulice, C.: Efficient maintenance of all-pairs shortest distances. In Proc. the 28th International Conference on Scientific and Statistical Database Managemen, Budapest, Hungary, 18–20 July, pp.9:1–9:12, ACM, (2016)

Gutenberg, M. P., Wulff-Nilsen, C.: Fully-dynamic all-pairs shortest paths: Improved worst-case time and space bounds. In Proc. the 2020 ACM-SIAM Symposium on Discrete Algorithms, Salt Lake City, UT, USA, 5–8 January, pp. 2562–2574, SIAM, (2020)

Hong, J., Kung, H. T.: I/O complexity: The red-blue pebble game. In Proc. The 13rd Annual ACM Symposium on Theory of Computing, Milwaukee, Wisconsin, USA, 11–13 May, pp.326–333, ACM, (1981)

Hua, Q., Qian, L., Yu, D., Shi, X., Jin, H.: A nearly optimal distributed algorithm for computing the weighted girth. Sci. China Inf. Sci **64**(11), 1–15 (2021)

Hutter, E., Solomonik, E.: Communication avoiding cholesky-qr2 for rectangular matrices. In Proc. the 2019 IEEE International Parallel and Distributed Processing Symposium, Hilton Copacabana, Rio de Janeiro, Brazil, 20–24 May, pp.89–100, IEEE, (2019)

Irony, D., Toledo, S., Tiskin, A.: Communication lower bounds for distributed-memory matrix multiplication. J. Parallel Distributed Comput **64**(9), 1017–1026 (2004)

Italiano, G. F., Lattanzi, S., Mirrokni, V. S., Parotsidis, N.: Dynamic algorithms for the massively parallel computation model. In Proc. The 31st ACM on Symposium on Parallelism in Algorithms and Architectures, Phoenix, AZ, USA, 22–24 June, pp.49–58, ACM, (2019)

Jenq, J., Sahni, S.: All pairs shortest paths on a hypercube multiprocessor. In Proc. International Conference on Parallel Processing, Park, PA, USA, 18–21 August, pp.713–716, Pennsylvania State University Press, (1987)

Jia, L., Hua, Q., Fan, H., Wang, Q., Jin, H.: Efficient distributed algorithms for holistic aggregation functions on random regular graphs. Sci. China Inf. Sci **65**(5), 1–19 (2022)

Johnson, D.B.: Efficient algorithms for shortest paths in sparse networks. J. ACM **24**(1), 1–13 (1977)

Karypis, G., Kumar, V.: A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM J. Sci. Comput. **20**(1), 359–392 (1998)

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 Proc. the 2012 International Conference on Advances in Social Networks Analysis and Mining, Istanbul, Turkey, 26–29 August, pp.1144–1148, IEEE Computer Society, (2012)

King, V.: Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. In Proc. the 40th Annual Symposium on Foundations of Computer Science, New York, NY, USA, 17–18 October, pp.81–91, IEEE Computer Society, (1999)

Nowicki, K., Onak, K.: Dynamic graph algorithms with batch updates in the massively parallel computation model. In Proc. the 2021 ACM-SIAM Symposium on Discrete Algorithms, Virtual Conference, 10–13 January, pp.2939–2958, SIAM, (2021)

Park, J., Penner, M., Prasanna, V.K.: Optimizing graph algorithms for improved cache performance. IEEE Trans. Parallel Distributed Syst **15**(9), 769–782 (2004)

Solomonik, E., Buluç, A., Demmel, J.: Minimizing communication in all-pairs shortest paths. In Proc. the 27th IEEE International Symposium on Parallel and Distributed Processing, Cambridge, MA, USA, 20–24 May, pp.548–559, IEEE Computer Society, (2013)

Thakur, R., Rabenseifner, R., Gropp, W.: Optimization of collective communication operations in MPICH. Int. J. High Perform. Comput. Appl. **19**(1), 49–66 (2005)

Thorup, M.: Worst-case update times for fully-dynamic all-pairs shortest paths. In Proc. the 37th Annual ACM Symposium on Theory of Computing}, Baltimore, MD, USA, 22–24 May, pp. 112–119, ACM, (2005)

Tiskin, A.: All-pairs shortest paths computation in the BSP model. In Proc. Automata, Languages and Programming, 28th International Colloquium, Crete, Greece, 8–12 July, pp.178–189, Springer, (2001)

Warshall, S.: A theorem on boolean matrices. J. ACM **9**, 11–12 (1962)

Yang, C., Miller, B. P.: Critical path analysis for the execution of parallel and distributed programs. In Proc. the 8th International Conference on Distributed Computing Systems, San Jose, California, USA, 13–17 June, pp.366–373, IEEE Computer Society, (1988)

Lin Zhu received his Ph.D. degree in computer science from Huazhong University of Science and Technology, Wuhan. His research interests include parallel algorithms anddistributed computing.

Qiang-Sheng Hua received his B.S. and M.S. degrees in computer science from Central South University, Changsha, in 2001 and 2004, respectively, and his Ph.D. degree in computer science from The University of Hong Kong, Hong Kong, in 2009. He is is currently a professor with Huazhong University of Science and Technology, Wuhan. He is interested in the algorithmic aspects of parallel and distributed computing.

Hai Jin received the PhD in computer engineering from HUST in 1994. He is a chair professor of computer science and engineering with the Huazhong University of Science and Technology (HUST) in China. In 1996, he was awarded a German Academic Exchange Service fellowship to visit the Technical University of Chemnitz in Germany. He worked with The University of Hong Kong between 1998 and 2000, and as a visiting scholar with the University of Southern California between 1999 and 2000. He was awarded Excellent Youth Award from the National Science Foundation of China in 2001. He is a fellow of CCF, a fellow of IEEE, and a life member of the ACM. He has coauthored more than 20 books and published more than 1000 research papers. His research interests include computer architecture, parallel and distributed computing, Big Data processing, data storage, and system security.