Communication Avoiding All-Pairs Shortest Paths Algorithm for Sparse Graphs

Lin Zhu, Qiang-Sheng Hua✉, and Hai Jin
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, China
qshua@hust.edu.cn

ABSTRACT

In this paper, we propose a parallel algorithm for computing all-pairs shortest paths (APSP) for sparse graphs on the distributed memory system with p processors. To exploit the graph sparsity, we first preprocess the graph by utilizing several known algorithmic techniques in linear algebra such as fill-in reducing ordering and elimination tree parallelism. Then we map the preprocessed graph on the distributed memory system for both load balancing and communication reduction. Finally, we design a new scheduling strategy to minimize the communication cost. The bandwidth cost (communication volume) and the latency cost (number of messages) of our algorithm are $O(\frac{n^2 \log^2 p}{p} + |S|^2 \log^2 p)$ and $O(\log^2 p)$, respectively, where S is a minimal vertex separator that partitions the graph into two components of roughly equal size. Compared with the state-of-the-art result for dense graphs where the bandwidth and latency costs are $O(\frac{n^2}{\sqrt{p}})$ and $O(\sqrt{p}\log^2 p)$, respectively, our algorithm reduces the latency cost by a factor of $O(\sqrt{p})$, and reduces the bandwidth cost by a factor of $O(\frac{\sqrt{p}}{\log^2 p})$ for sparse graphs with $|S| = O(\frac{n}{\sqrt{p}})$. We also present the bandwidth and latency costs lower bounds for computing APSP on sparse graphs, which are $\Omega(\frac{n^2}{p} + |S|^2)$ and $\Omega(\log^2 p)$, respectively. This implies that the bandwidth cost of our algorithm is nearly optimal and the latency cost is optimal.

CCS CONCEPTS

• **Theory of computation → Design and analysis of algorithms**; **Shortest paths**; • **Mathematics of computing** → *Graph algorithms.*

KEYWORDS

parallel algorithms, sparse graphs, APSP, communication complexity

ACM Reference Format:
Lin Zhu, Qiang-Sheng Hua✉, and Hai Jin. 2021. Communication Avoiding All-Pairs Shortest Paths Algorithm for Sparse Graphs. In *50th International Conference on Parallel Processing (ICPP '21), August 9–12, 2021, Lemont, IL, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3472456.3472524

ICPP '21, August 9–12, 2021, Lemont, IL, USA

ACM ISBN 978-1-4503-9068-2/21/08…$15.00
https://doi.org/10.1145/3472456.3472524

1 INTRODUCTION

The all-pairs shortest paths (APSP) is a fundamental graph problem. The classic dynamic programming APSP algorithm was proposed by Floyd [10] and Warshall [25]. Its algorithm structure consists of a three nested-loop and performs reasonably well for dense graphs in practice. A Floyd-Warshall-based algorithm, SuperFW [22], is proposed to improve the performance on shared memory parallel machines for sparse graphs. However, as far as we know, there are few studies focusing on efficient APSP algorithms for sparse graphs in distributed memory systems. In addition, in distributed computing for large-scale data, communication cost gradually dominates. Therefore, we aim to design a parallel sparse APSP algorithm with a low communication cost.

Given a weighted undirected graph $G = (V, E)$ with $|V| = n$ vertices and $|E| = m$ edges, if the graph is dense, then on the distributed memory system with p processors, a divide-and-conquer distributed algorithm, 2D-DC-APSP [24], can be used to compute APSP. Its bandwidth cost (communication volume) and latency cost (the number of messages) (see Section 3.1) are $O(n^2/\sqrt{p})$ and $O(\sqrt{p}\log^2 p)$ respectively. This algorithm is communication-avoiding, which means it either asymptotically attains the lower bound of communication cost or reduces the communication cost compared to a conventional algorithm [4]. 2D-DC-APSP is suitable for dense graphs. However, for sparse graphs, it might be inefficient.

Designing a sparse APSP algorithm on the distributed memory system poses several challenges. The first is data layout. An efficient data layout can minimize communication, balance the work load, and can reduce the memory footprint. In our algorithm, we use a **block layout** (see Section 5.1) to balance the computation and data movement, while simultaneously minimizing communication cost. The second is the scheduling strategy. We know that algorithms with a higher degree of parallelism usually means less communication cost. Therefore, in the process of updating the distance matrix, how to design the scheduling strategy so that more processors can participate in the update task at the same time is an important issue.

There are two main contributions in this paper.

1 **We propose a communication-avoiding APSP algorithm for sparse graphs on a distributed memory system**. There are mainly three steps for our method. First, in the pre-processing stage, in order to exploit the graph sparsity, we use several known algorithmic techniques in linear algebra to reorder the vertices of the input graph

Table 1: List of Symbols

Symbol type	Symbol	Description
Matrix	n	Dimension of the matrix A
	$G = (V, E)$	Graph G with vertices V and edges E
	A_{ij}	Element in row i and column j of A
	$A(i, j)$	Block in row i and column j of A
	R_l	Updated regions of A to eliminate Q_l
Supernodes	T	Elimination tree of matrix A
	N	The number of supernodes in T
	S	Vertex separator of graph G
	h	Height of T $(O(\log p))$
	Q_l	Set of the l-th level supernodes in T
	$\mathcal{A}(a)$	Set consisting of ancestors of a
	$\mathcal{D}(a)$	Set consisting of descendents of a
	$C(a)$	Set consisting of cousins of a
Process	p	#MPI processes
	M	Per-process memory
	B	Per-process bandwidth cost
	L	Per-process latency cost

and to utilize the elimination tree to guide parallelism. Then we map the adjacency matrix of the reordered graph to the distributed memory system in a **block layout**, and analyze the advantages of this data layout. Finally, in order to maximize the parallelization of updating the matrix blocks in the distance matrix, we present a one-to-one mapping for assigning the computations needed for updating the matrix blocks to the processors. Let S denote the minimal separator of graph G, then the bandwidth and latency costs of our algorithm are $O(\frac{n^2 \log^2 p}{p} + |S|^2 \log^2 p)$ and $O(\log^2 p)$, respectively. Compared with the 2D-DC-APSP algorithm [24], the latency cost of our algorithm is reduced by a factor of $O(\sqrt{p})$, and the bandwidth cost is reduced by a factor of $O(\min(\frac{\sqrt{p}}{\log^2 p}, \frac{n^2}{|S|^2 \sqrt{p} \log^3 p}))$.

2 **We present the bandwidth and latency costs lower bounds for distributively computing APSP on sparse graphs, which are $\Omega(\frac{n^2}{p} + |S|^2)$ and $\Omega(\log^2 p)$**, respectively (See Section 6 for details).

2 RELATED WORK

The classic dynamic programming APSP algorithm was proposed by Floyd [10] and Warshall [25]. Its algorithm structure consists of a three nested-loop and performs reasonably well for dense graphs in practice. In order to increase the data locality, the blocked version was subsequently formulated [21]. For sparse graphs, Johnson's algorithm [16] is theoretically faster than Floyd-Warshall algorithm. However, due to the data-dependent structure, it is difficult to scalably parallelize the algorithm. Sao et al. proposed a sparse APSP algorithm, SuperFW [22], which is based on the Floyd-Warshall (FW) algorithm and applies the technology of sparse direct solvers to APSP. SuperFW focuses on reducing computational operations. Compared with the classic Floyd-Warshall algorithm, the amount of its computational operations is reduced by a factor of $O(\frac{n}{|S|})$, where S is the top-level seperator.

The first 2D distributed memory algorithm for APSP was proposed by Jenq and Sahni [14], which is based on the Floyd-Warshall algorithm. Since it did not employ the block structure, its latency cost could be $O(n)$. Solomonik et al. [24] proposed a divide-and-conquer distributed APSP algorithm, whose bandwidth and latency costs are $O(\frac{n^2}{\sqrt{p}})$ and $O(\sqrt{p} \log^2 p)$ respectively. These upper bounds either meet or nearly meet the corresponding bandwidth and latency costs lower bounds.

Since APSP and many linear algebra problems such as matrix multiplication and LU factorization have similar three-nested loop structure and data access patterns, many methods can apply for both problems. Carre was the first to analyze the equivalence relationship between APSP and numerical linear algebra [8]. Aho et al. proved that the cost of APSP and semiring matrix multiplication are almost equivalent in the random access machine [1]. Various researchers have given several algorithms based on this proof [21, 24]. Due to the similar computational structure, the communication lower bounds of the previously known "classical" matrix multiplication are also applicable to the APSP problem [4, 13, 15].

Solving linear algebra problems of sparse matrices under the distributed memory model has been widely studied [3, 7, 23]. Many researchers use the nested dissection (ND) method [11, 20] and the elimination tree parallel technology to reduce the communication cost [12, 23]. However, they all use block cyclic data layout, which we will introduce in section 5, to distribute data on the processor grid to alleviate load-imbalance. Compared with the communication optimal distributed algorithm for dense matrices, although the bandwidth cost is reduced, they have higher latency costs (a polynomial of n). The ND method is also used in this paper to compute the separator of a graph.

3 PRELIMINARIES

In this section, we introduce the communication model, problem definition, the classic Floyd-Warshall (FW) algorithm and the blocked FW algorithm for dense matrices.

3.1 Communication Model

In this paper, we will use a distributed-memory model that has been widely employed in previous work [9] [5]. We model the machine as having p processors, which are connected via a network, and the local memory size of each processor is M. We count the communication cost in terms of bandwidth cost B (the number of words) and latency cost L (the number of messages) along the critical path as defined in [26]. That is, two messages that are communicated between separate pairs of processors simultaneously are counted only once.

We assume that (1) the architecture is homogeneous, (2) **a processor can only send/receive a message to/from one other processor at a time**, and (3) there is a link between each processor pair to avoid communication resource contention among processors.

3.2 Problem Definition

Given an undirected weighted graph $G = (V, E)$, where $|V| = n$ and $|E| = m$, each vertex in graph G is represented by $v_i (i \in \{1, 2..., n\})$, and the edge between vertex v_i and v_j is represented by e_{ij}. The weight of each edge $e_{ij} \in E$ is $|e_{ij}|$. If $e_{ij} \notin E$, then $|e_{ij}| = \infty$. We assume that $|e_{ij}|$ can be negative but there is no cycle with a negative sum of weights in graph G. Graph G can be represented by an $n \times n$ adjacency matrix A, where $A_{ii} = 0$ and $A_{ij} = |e_{ij}|$.

Computing the APSP of a graph can be seen as a continuous update of the symmetric matrix A. After each update to A, A_{ij} holds the shortest distance between vertex v_i and vertex v_j discovered so far. When the algorithm finishes running, A_{ij} holds the shortest distance between v_i and v_j.

3.3 Classical FW Algorithm and Blocked FW Algorithm

Given an undirected weighted graph G with n vertices and m edges, its adjacency matrix is A, and the ClassicalFW algorithm updates all elements A_{ij} of A according to

$$A_{ij} = A_{ij} \oplus A_{ik} \otimes A_{kj} \qquad i, j, k \in \{1, 2..., n\}$$

Here, $x \oplus y = min\{x, y\}$, $x \otimes y = x + y$, x and y can be any real or infinite values.

The BlockedFW algorithm divides the adjacency matrix A into $\frac{n}{b} \times \frac{n}{b}$ blocks and the size of each block is $b \times b$. The (i, j)-th block in matrix A is denoted by $A(i, j)$, $i, j \in \{1, 2..., \frac{n}{b}\}$. The BlockedFW algorithm updates all matrix blocks $A(i, j)$ in the following three steps.

For $k \in \{1, 2..., \frac{n}{b}\}$ and $i, j \neq k$:
diagonal update: $A(k, k) \leftarrow$ ClassicalFW $(A(k, k))$
panel update: $A(i, k) \leftarrow A(i, k) \oplus A(i, k) \otimes A(k, k)$
$A(k, j) \leftarrow A(k, j) \oplus A(k, k) \otimes A(k, j)$
minplus outer product: $A(i, j) \leftarrow A(i, j) \oplus A(i, k) \otimes A(k, j)$

4 PRE-PROCESSING

In this section, we introduce two known algorithmic techniques, fill-in reducing ordering and elimination tree parallelism, to preprocess the input graph. These techniques can take advantage of the sparsity of the graph and are widely used in sparse numerical linear algebra. Sao et al. combines these technologies with the BlockedFW algorithm to get the SuperFW algorithm [22].

4.1 Fill-in Reducing Ordering

In the BlockedFW algorithm, if a block is "empty", which means that all the entries in this block are infinite values, certain computation operations involving this block can be avoided. For example, consider the BlockedFW algorithm, for $k = 3$ and $A(4, 3)$ is empty, then the update of $A(4, :)$ can be avoided. This is because $A(4, :) \leftarrow A(4, :) \oplus A(4, 3) \otimes A(3, :)$ and $A(4, 3) \otimes A(3, :)$ is empty. However, even if the adjacency matrix A is sparse, the sparse structure of A is irregular and there may not be all infinite values in a block. We can use the known "nested-dissection"(ND) process [11] to reorder the adjacency matrix and can obtain the block-arrow structure, which can effectively reduce fill-in. Some graph partitioning tools like Metis [17] can be used to compute the **ND process**.

The initial goal of **ND process** is to compute a separator S of the graph $G = (V, E)$. S partitions V into three disjoint sets, $V = V_1 \cup S \cup V_2$. This partition satisfies the following conditions:

(1) For any vertex $v_i \in V_1$ and any vertex $v_j \in V_2$, $e_{ij} \notin E$;
(2) V_1 and V_2 are balanced, i.e., $|V_1| \approx |V_2|$;
(3) S is as small as possible.

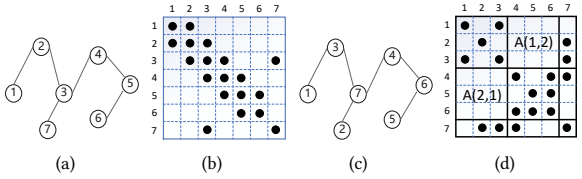

Figure 1: Fig. 1a and 1b respectively show the input graph and its adjacency matrix. Fig. 1c and 1d respectively show the reordered graph and its adjacency matrix. Each circle in Fig. 1b and Fig. 1d represents a finite value.

V_1, V_2 and S are called supernodes, and each supernode consists of vertices with similar adjacent structures. According to this partition, the indices of the vertices in V_1, V_2 and S are reordered. The vertices in each vertex set V_1, V_2 and S have consecutive indices, and the index of each vertex in S is higher than the index of any vertex in V_1 and V_2. For example, in Fig. 1a and 1b, we show the input graph G and its adjacency matrix. The **ND process** divides and reorders V into $V_1 = \{1, 2, 3\}$, $V_2 = \{4, 5, 6\}$ and $S = V_3 = \{7\}$. Fig. 1c and 1d show the reordered graph G and its adjacency matrix. $A(1, 2)$ and $A(2, 1)$ in Fig. 1d are empty. It should be noted that V_1 and V_2 can perform the **ND process** recursively to get a more fine-grained ordering of A.

Karypis and Kumar proposed a parallel algorithm that can efficiently calculate the separator of a graph [18]. Given an input graph $G = (V, E)$, the 2-way partition of V can be obtained through three phases: coarsening, initial partitioning, and uncoarsening, which divide V into two equal parts A and B to minimize the number of edges between A and B. The separator can be obtained by finding the minimum vertex cover of edges between A and B. The algorithm can be directly applied to our model with a few modifications, and the bandwidth cost and the latency cost of finding a separator are $O(\frac{n \log p}{\sqrt{p}})$ and $O(\log p)$, respectively.

For the reordered sparse adjacency matrix, different iteration orders may have a great impact on the computation cost. For example, consider that the reordered adjacency matrix A in Fig. 1d are executed in different iteration orders in the BlockedFW algorithm. When the iteration order is $\{3, 1, 2\}$, $A(1, 2)$ and $A(2, 1)$ become non-empty in the first iteration $k = 3$. When the iteration order is $\{1, 2, 3\}$, $A(1, 2)$ and $A(2, 1)$ become non-empty in the last iteration $k = 3$. The latter can reduce fill-in and can better maintain sparsity, so more operations can be avoided.

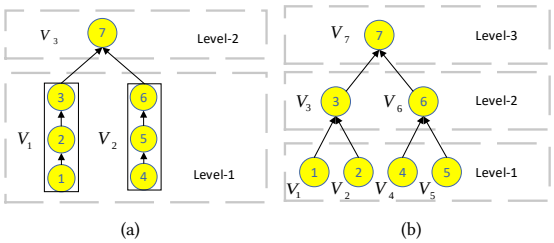

Figure 2: Fig. 2a is a 2-level eTree, and the seperator S is V_3. Fig. 2b is a 3-level eTree, obtained by recursively executing the ND process on V_1 and V_2 in Fig. 2a.

4.2 Elimination Tree

As mentioned above, the elimination order $\{1, 2, 3\}$ or $\{2, 1, 3\}$ can maintain low fill, and the elimination tree (eTree) shown in Fig.2a can be used to describe this ordering. A multilevel eTree shown in Fig.2b can be obtained by recursively perform the **ND process** on V_1 and V_2. In an eTree, if supernode V_i is V_j's parent, the parent of parent or so on, then we call V_i an ancestor of V_j and V_j a descendant of V_i. If V_i is neither the ancestor of V_j nor the descendant of V_j, then call V_i the cousin of V_j. Use $\mathcal{A}(i)$, $\mathcal{D}(i)$ and $\mathcal{C}(i)$ to represent the set of all ancestors, descendants, and cousins of V_i, respectively. In Fig. 2b, $\mathcal{A}(3) = \{7\}$, $\mathcal{D}(3) = \{1, 2\}$ and $\mathcal{C}(3) = \{4, 5, 6\}$. A proper iteration ordering is that, if V_i is a descendant of V_j in eTree, then V_i should be eliminated before V_j, and if V_i is a cousin of V_j, the elimination order of V_i and V_j can be arbitrary. Consider the elimination of supernode V_k, for all $i, j \in \mathcal{C}(k)$, the update of $A(i, k)$,$A(k, i)$ and $A(i, j)$ can be avoided.

5 THE PARALLEL ALGORITHM

In this section, we present a parallel sparse APSP algorithm. We start with the mapping from the supernodal block sparse matrix to the processors, then we design a novel scheduling strategy to reduce communication cost.

5.1 Data Layout

In this subsection, we discuss how to map the supernodal block sparse matrix obtained in the pre-processing to the processor grid. As far as we know, there are few studies on APSP algorithms with low communication complexity for sparse matrices. In the most of the previous related studies, the matrix is distributed on the processor grid in a **block cyclic layout**, such as 2D-DC-APSP [24] and SuperLU_DIST [23]. The block cyclic distribution divides the matrix A into $\frac{n}{b} \times \frac{n}{b}$ blocks of size $b \times b$, and each block is distributed on p processors in a **block layout**. The **block layout** divides the matrix A into $\sqrt{p} \times \sqrt{p}$ blocks of size $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$, and each processor owns one block.

The 2D-DC-APSP algorithm [24] is a distributed APSP algorithm described in a divide-and-conquer (DC) approach on the distributed memory system . The bandwidth and latency costs of the algorithm are $O(\frac{n^2}{\sqrt{p}})$ and $O(\sqrt{p}\log^2 p)$ respectively, and it is a communication avoiding algorithm for dense matrices. SuperLU_DIST is a distributed algorithm for solving LU factorization of sparse matrix, which is used in LAPACK [2]. This algorithm applies several of the aforementioned algorithmic techniques to the popular right-looking LU algorithm.

We map the supernodal block sparse matrix A onto the processors in **block layout** instead of block cyclic. There are two reasons why we do not use the **block cyclic layout**.

First, the latency cost of using **block cyclic layout** is $O(n/\sqrt{p})$, which is even higher than the latency cost $O(\sqrt{p}\log^2 p)$ of 2D-DC-APSP. Suppose we adopt the **block cyclic layout**, for all $k \in \{1, 2..., N\}$, $A(k, k)$ is distributed on $\sqrt{p}$ diagonal processors, then there is one processor who will store at least $\frac{N}{\sqrt{p}}$ blocks $A(k, k)$. Consider the **diagonal update** of supernode k , the processor that stores $A(k, k)$ must send at least one message. Therefore, for the processor that stores at least $\frac{N}{\sqrt{p}}$ blocks, the number of messages it must send is at least $N/\sqrt{p}$.

Second, unlike the 2D-DC-APSP and SuperLU_DIST algorithms with **block layout**, performing our algorithm does not suffer from load-imbalance. This is because our algorithm is based on Floyd-Warshall algorithm. For the k-th iteration, BlockedFW algorithm with **block layout** updates all blocks $A(i, j)$, where $i, j \in \{1, 2..., \sqrt{p}\}$. The update of A is specified by

$$A(i,j) = \begin{cases} \text{ClassicalFW}(A(k,k)) & i,j = k, \\ A(i,j) \oplus A(i,k) \otimes A(k,j) & i \neq k \text{ or } j \neq k. \end{cases}$$

which shows that all blocks $A(i, j)$ are updated in each iteration, where $i, j \in \{1, 2..., \sqrt{p}\}$. If **block layout** is adopted, then all processors are active in each iteration. However, the SuperLU_DIST uses the right-looking scheme, the update of A is specified by

$$A(i,j) = \begin{cases} L(k,k)U(k,j) & i = k, j \in \{k..., \sqrt{p}\}, \\ L(i,k)U(k,k) & j = k, i \in \{k+1..., \sqrt{p}\}, \\ A(i,j) - L(i,k)U(k,j) & i,j \in \{k+1..., \sqrt{p}\}. \end{cases}$$

It can be found that with **block layout**, those processors P_{ij} with $i = \{1, 2..., k-1\}$ or $j = \{1, 2..., k-1\}$ will be idle, which causes load-imbalance. Considering the 2D-DC-APSP algorithm, using a **block layout** will result in at least three-quarters of the processors being idle at any point in the algorithm. Therefore, SuperLU_DIST and 2D-DC-APSP use **block cyclic layout** to alleviate load-imbalance, while our algorithm can utilize **block layout**.

In order to map the supernodal block sparse matrix A to a $\sqrt{p} \times \sqrt{p}$ grid in a **block layout**, we specify the number of recursions of the **ND process** such that $N = \sqrt{p}$. Denote h to represent the number of levels of eTree, then $\sum_{l=1}^{h} 2^{h-l} = \sqrt{p}$ and $h = \log(\sqrt{p} + 1)$. For $i, j \in \{1, 2..., \sqrt{p}\}$, each processor P_{ij} owns a block $A(i, j)$.

5.2 The Scheduling Strategy

With the eTree and block layout, in this section we discuss task scheduling on the processor grid. Since the supernodes in the same level are cousins, the elimination of supernodes in the same level is independent. We eliminate supernodes level by level from bottom to top. For ease of expression, we relabel the supernodes in this order, as shown in Fig. 3a.

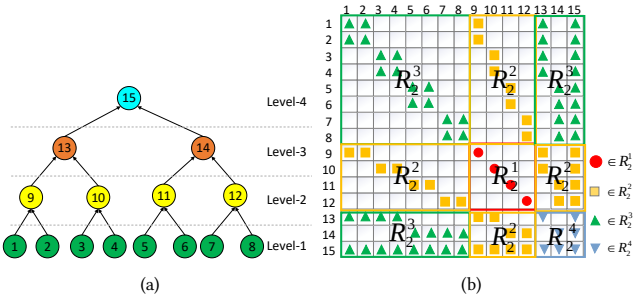

Figure 3: Fig. 3a shows a 4-level eTree labeled from bottom to top. Fig. 3b shows the regions of R_2^1, R_2^2, R_2^3 and R_2^4.

Consider an eTree of level h, we use Q_l to denote the collection of the l-th level supernodes, and use R_l to denote the updated region of A during the elimination of the l-th level supernodes. Then

$$R_l = \bigcup_{k \in Q_l} (k \cup \mathcal{A}(k) \cup \mathcal{D}(k), k \cup \mathcal{A}(k) \cup \mathcal{D}(k)),$$

and for each $(i, j) \in R_l$, $A(i, j)$ is updated by

$$A(i, j) = A(i, j) \oplus \sum_{k}^{\oplus} A(i, k) \otimes A(k, j), \tag{1}$$

where $k \in (\mathcal{A}(i) \cup \mathcal{D}(i)) \cap (\mathcal{A}(j) \cup \mathcal{D}(j)) \cap Q_l$. We define each $A(i, k) \otimes A(k, j)$ as a computing unit.

We can divide R_l into four subsets:

(1) $R_l^1 = \bigcup_{k \in Q_l} (k, k)$
(2) $R_l^2 = \bigcup_{k \in Q_l} (\mathcal{A}(k) \cup \mathcal{D}(k), k) \cup (k, \mathcal{A}(k) \cup \mathcal{D}(k))$
(3) $R_l^3 = \bigcup_{k \in Q_l} (\mathcal{A}(k) \cup \mathcal{D}(k), \mathcal{D}(k)) \cup (\mathcal{D}(k), \mathcal{A}(k))$
(4) $R_l^4 = \bigcup_{k \in Q_l} (\mathcal{A}(k), \mathcal{A}(k))$

Where R_l^1 corresponds to **diagonal update**, R_l^2 corresponds to **panel update**, R_l^3 and R_l^4 correspond to **minplus outer product**. For example, for the eTree in Fig. 3a and $l = 2$, the regions of these four subsets are shown in Fig. 3b. We update these four subsets sequentially.

5.2.1 *The Update of R_l^1, R_l^2 and R_l^3.*

For the update of R_l^1, it is obvious that each P_{kk} updates $A(k, k)$ locally, and these updates are performed in parallel. There are no communication between processors.

For R_l^2, we first consider the update of $\bigcup_{k \in Q_l} (\mathcal{A}(k) \cup \mathcal{D}(k), k)$. Each processor P_{kk} broadcasts $A(k, k)$ to all processors P_{ik} in this column, where $i \in \mathcal{A}(k) \cup \mathcal{D}(k)$ and P_{ik} updates $A(i, k)$. For each $k \in Q_l$, each broadcast operation is executed on a different processor column, so these operations can be executed in parallel. The update of subset $\bigcup_{k \in Q_l} (k, \mathcal{A}(k) \cup \mathcal{D}(k))$ is similar.

For R_l^3, the processors in R_l^2 send data to the processors in R_l^3, and the processors in R_l^3 receive the data and update the local matrix block. For each $(i, j) \in R_l^3$, according to Equation (1) and there is some $k \in Q_l$ such that $i \in \mathcal{D}(k)$ or $j \in \mathcal{D}(k)$, we can get $|(\mathcal{A}(i) \cup \mathcal{D}(i)) \cap (\mathcal{A}(j) \cup \mathcal{D}(j)) \cap Q_l| = 1$, which means that $A(i, j)$ can be updated through a computing unit $A(i, k) \otimes A(k, j)$. We can describe the update of $A(i, j)$ in R_l^3 in three steps: (1) For each $(i, k) \in R_l^2$, P_{ik} broadcasts $A(i, k)$ to all processors P_{ij} with $j \in \mathcal{A}(k) \cup \mathcal{D}(k)$; (2) For each $(k, j) \in R_l^2$, P_{kj} broadcasts $A(k, j)$ to all processors P_{ij} with $i \in \mathcal{A}(k) \cup \mathcal{D}(k)$; (3) For each $(i, j) \in R_l^3$, P_{ij} receives $A(i, k)$ and $A(k, j)$ and updates $A(i, j) = A(i, j) \oplus A(i, k) \otimes A(k, j)$.

5.2.2 *The Update of R_l^4.*

In the process of eliminating each level of supernodes, the update of R_l^4 in the distance matrix is the most important and complex one. We will elaborate it in this subsection.

According to Equation (1) and $(i, j) \in R_l^4$, we can get $|(\mathcal{A}(i) \cup \mathcal{D}(i)) \cap (\mathcal{A}(j) \cup \mathcal{D}(j)) \cap Q_l| > 1$, which means that the update of each $A(i, j)$ requires multiple computing units. Suppose a block $A(i, j)$ is updated by q computing units, its update is $A(i, j) \leftarrow A(i, j) \oplus A(i, 1) \otimes A(1, j) \oplus \cdots \oplus A(i, q) \otimes A(q, j)$. A trivial strategy is to perform those computing units sequentially on processor P_{ij}, which is used in SuperLU_DIST [19]. That is, for all $k \in \{1, 2..., q\}$, P_{ij} receives messages sequentially from P_{ik} and P_{kj} and computes $A(i, k) \otimes A(k, j)$ to update $A(i, j)$. With this strategy, processor P_{ij} needs to receive $2q$ messages to update $A(i, j)$. An optimized strategy is to allocate q computing units that update $A(i, j)$ to q different processors, and then those processors reduce results to P_{ij}. That is, for each $k \in \{1, 2..., q\}$, processors P_{ik} and P_{kj} send local data to the corresponding processor in parallel, then each processor performs the computation $A(i, k) \otimes A(k, j)$ in parallel. Finally, the q processors reduce to P_{ij}. In this way, each processor only needs to transmit $O(\log q)$ messages.

However, it should be noted that the elimination of the l-th level supernodes needs to update all blocks $A(i, j)$ in R_l^4. If two computing units that update two blocks in R_l^4 are assigned to the same processor, then these two blocks can only be updated sequentially. Therefore, in order to update the blocks in R_l^4 with a maximum degree of parallelization, the optimal strategy is to allocate each computing unit that updates R_l^4 to a separate processor one-to-one.

Lemma 5.1. *If all the computing units required to update all blocks in R_l^4 can be mapped to the processor grid one-to-one, then all blocks in R_l^4 can be updated in parallel.*

According to Equation (1) and $(i, j) \in R_l^4$, updating the block $A(i, j)$ in R_l^4 requires computing all $A(i, k) \otimes A(k, j)$, where $k \in Q_l \cap \mathcal{D}(i) \cap \mathcal{D}(j)$. We know that the number of processors is p, therefore, if the number of computing units is less than p, then a one-to-one mapping from the computing units to processors exists. We prove it in Lemma 5.2.

Lemma 5.2. *The number of computing units required to update R_l^4 is $O(p)$.*

Proof. We divide the blocks $A(i, j)$ in R_l^4 into $(h - l)$ subsets. We use $R_l^4(a)$ to denote the subset consisting of blocks $A(i, j)$ with $min(level(i), level(j)) = a$, then $R_l^4 = \bigcup_{a=l+1}^{h} R_l^4(a)$. We calculate the number of computing units required to update all blocks $A(i, j)$ in $R_l^4(a)$.

First, we calculate the number of computing units needed to update each block $A(i, j)$ in the subset $R_l^4(a)$. We know that updating $A(i, j)$ needs to compute all $A(i, k) \otimes A(k, j)$, where $k \in Q_l \cap \mathcal{D}(i) \cap \mathcal{D}(j)$. Since $\min(level(i), level(j)) = a$, thus $|Q_l \cap \mathcal{D}(i) \cap \mathcal{D}(j)| =$

2^{a-l}. The number of computing units needed to update each $A(i,j)$ in the subset $R_l^4(a)$ is 2^{a-l}.

Next, we calculate the number of blocks $A(i,j)$ in $R_l^4(a)$. If $level(i) < level(j)$, then $i \in Q_a$ and $j \in \mathcal{A}(i)$. Since $|Q_a| = 2^{h-a}$ and for each $i \in Q_a$, $|\mathcal{A}(i)| = h - a$, thus the number of blocks $A(i,j)$ is $(h-a)2^{h-a}$. Similarly, if $level(i) > level(j)$, the number of blocks $A(i,j)$ is also $(h-a)2^{h-a}$. If $level(i) = level(j)$, then the number of blocks $A(i,j)$ is 2^{h-a}. Therefore, the total number of blocks $A(i,j)$ in the subset $R_l^4(a)$ is $(2h-2a+1)2^{h-a}$.

As a result, the total number of computing units required to update R_l^4 is $\sum_{a=l+1}^{h}(2h-2a+1)2^{h-l} = O(h^2 2^h)$. Since $h^2 = O(2^h)$ and $2^h = O(\sqrt{p})$, thus the number of computing units required to update R_l^4 is $O(p)$. □

Below, we analyze how to get such a one-to-one mapping. For those blocks $A(i,j)$ in $R_l^4(a)$, because the distance matrix A is symmetric, we only consider the update of blocks $A(i,j)$ with $level(i) \geq level(j)$ in $R_l^4(a)$. According to the level of j in the eTree, we further divide $R_l^4(a)$ into $(h-a+1)$ subsets. We use $R_l^4(a,c)$ to denote each subset consisting of all $A(i,j)$ with $level(i) = a$ and $level(j) = c$ where $c \in \{a, a+1..., h\}$, then $R_l^4(a) = \bigcup_{c=a}^{h} R_l^4(a,c)$. For each subset $R_l^4(a,c)$, we assign the computing units needed to update the blocks $A(i,j)$ in $R_l^4(a,c)$ to a row of processors. And for any two subsets, their computing units are allocated to different processor rows. We prove in Lemma 5.3 that this strategy can enable each computing unit be performed on a separate processor.

Lemma 5.3. *R_l^4 is divided into multiple subsets, and each subset $R_l^4(a,c)$ consisting of blocks $A(i,j)$ with $level(i) = a$ and $level(j) = c$, where $a \in \{l+1, l+2..., h\}$, $c \in \{a, a+1..., h\}$. If the computing units needed to update the two subsets are assigned to two different processor rows, then the mapping from the computing units to the processors can be one-to-one.*

Proof. First, because the number of processor rows is $\sqrt{p}$, we prove that the number of subsets $R_l^4(a,c)$ is less than $\sqrt{p}$. For each $a \in \{l+1, l+2..., h\}$, the number of subsets $R_l^4(a,c)$ of $R_l^4(a)$ is $h-a+1$, so the number of subsets is $\sum_{a=l+1}^{h}(h-a+1) < (h-l)^2$. Since $h = \log(\sqrt{p}+1)$, thus the number of subsets $R_l^4(a,c)$ is less than $\sqrt{p}$.

Second, because the number of processors in each row is $\sqrt{p}$, we prove that the number of computing units required to update each subset $R_l^4(a,c)$ is less than $\sqrt{p}$. Since the number of computing units to update each block $A(i,j)$ is 2^{a-l} and the number of blocks $A(i,j)$ in each subset is 2^{h-a}, so the number of computing units required to update each subset $R_l^4(a,c)$ is 2^{h-l}. Since $h = \log(\sqrt{p}+1)$, thus the number of computing units required to update each subset $R_l^4(a,c)$ is less than $\sqrt{p}$. □

Below, we discuss the one-to-one mapping from subsets to processor rows. There are many satisfied one-to-one mappings, and here we give one of them. That is, for each subset $R_l^4(a,c)$, we assign the computing units that update blocks $A(i,j)$ in $R_l^4(a,c)$ to the processor row $f = \sum_{b=h+a-c}^{h-1} 2^b + (a-l)$. We prove in Lemma 5.4 that this allocation method can map each subset $R_l^4(a,c)$ to a separate processor row.

Lemma 5.4. *For each subset $R_l^4(a,c)$, where $a \in \{l+1, l+2..., h\}$ and $c \in \{a, a+1..., h\}$, if its corresponding processor row is $f = \sum_{b=h+a-c}^{h-1} 2^b + (a-l)$, then this mapping from subsets to processor rows is one-to-one.*

Proof. To prove that the mapping is one-to-one, we only need to prove that it satisfies two conditions.

First, for each subset $R_l^4(a,c)$, since the processor grid is $\sqrt{p} \times \sqrt{p}$, f must not be greater than $\sqrt{p}$. Because $f = \sum_{b=h+a-c}^{h-1} 2^b + (a-l)$ and $\sqrt{p} = \sum_{b=0}^{h-1} 2^b$, $\sqrt{p} - f = \sum_{b=0}^{h+a-c-1} 2^b - (a-l) > 2^{h+a-c-1} - (a-l)$. Since $c \leq h$, then $2^{h+a-c-1} \geq 2^{a-1}$, thus $2^{h+a-c-1} - (a-l) \geq 2^{a-1} - (a-l) \geq 0$. Therefore, $f \leq \sqrt{p}$.

Second, for any two different subsets $R_l^4(a_1,c_1)$ and $R_l^4(a_2,c_2)$, in order to ensure that the mapping is one-to-one, $f_1 \neq f_2$ must hold. If $a_1 = a_2$ and $(a_1 - c_1) \neq (a_2 - c_2)$, then obviously $f_1 \neq f_2$. If $a_1 \neq a_2$ and $(a_1 - c_1) = (a_2 - c_2)$,then obviously $f_1 \neq f_2$. If $a_1 \neq a_2$ and $(a_1 - c_1) \neq (a_2 - c_2)$, without loss of generality, we assume $(a_1 - c_1) < (a_2 - c_2)$, then

$$f_1 > \sum_{b=h+a_1-c_1}^{h-1} 2^b \geq \sum_{b=h+a_2-c_2-1}^{h-1} 2^b = \sum_{b=h+a_2-c_2}^{h-1} 2^b + 2^{h+a_2-c_2-1} > \sum_{b=h+a_2-c_2}^{h-1} 2^b + (a_2 - l) = f_2.$$

Therefore, for any two different subsets, $f_1 \neq f_2$.

□

For those computing units that update the blocks $A(i,j)$ in $R_l^4(a,c)$, we discuss how to assign each computing unit to a separate processor in row $f = \sum_{b=h+a-c}^{h-1} 2^b + (a-l)$. According to the proof of Lemma 5.3, we know that the number of computing units needed to update each subset is 2^{h-l}, where each computing unit corresponds to a supernode $k \in Q_l$. We let these 2^{h-l} computing units be performed on the processor set $\{P_{f,1}, P_{f,2}..., P_{f,2^{h-l}}\}$, and each processor performs one computing unit. According to the labels of supernodes in subsection 5.2, the set of $k \in Q_l$ is $\{\sum_{b=h-l+1}^{h-1} 2^b + 1, \sum_{b=h-l+1}^{h-1} 2^b + 2..., \sum_{b=h-l+1}^{h-1} 2^b + 2^{h-l}\}$. Then for each computing unit $A(i,k) \otimes A(k,j)$ that updates the subset $R_l^4(a,c)$, it corresponds to a separate processor P_{fg}, where $f = \sum_{b=h+a-c}^{h-1} 2^b + (a-l)$ and $g = k - \sum_{b=h-l+1}^{h-1} 2^b$.

Corollary 5.5. *For each $A(i,j)$ with $level(i) \geq level(j)$ and $(i,j) \in R_l^4$, the update of $A(i,j)$ needs to compute all $A(i,k) \otimes A(k,j)$, where $k \in Q_l \cap \mathcal{D}(i) \cap \mathcal{D}(j)$. And each computing unit can be assigned to a separate processor P_{fg}, where $f = \sum_{b=h+a-c}^{h-1} 2^b + (a-l)$ and $g = k - \sum_{b=h-l+1}^{h-1} 2^b$.*

With the above mapping, each block $A(i,j)$ with $level(i) \geq level(j)$ in R_l^4 can be updated in parallel, and each computing unit $A(i,k) \otimes A(k,j)$ can be performed on a separate processor P_{fg}. In order to compute $A(i,k) \otimes A(k,j)$, the processors in R_l^2 send $A(i,k)$ and $A(k,j)$ to P_{fg}, and then each P_{fg} computes $A(i,k) \otimes A(k,j)$. For each $A(i,j)$ with $level(i) \geq level(j)$ in R_l^4, it is updated by

all computing units $A(i,k) \otimes A(k,j)$, where $k \in Q_l \cap \mathcal{D}(i)$. We regard the processors P_{fg} performing these computing units as a group, and reduce the computation results to P_{ij}. Below we give the communication process from "R_l^2 to P_{fg}" and "P_{fg} to R_l^4".

R_l^2 to P_{fg}: For each P_{ik} with $k \in Q_l$ and $i \in \mathcal{D}(k)$, its local data $A(i,k)$ is needed to compute all $A(i,k) \otimes A(k,j)$, where $j \in i \cup \mathcal{A}(i)$. Each computing unit is performed on a separate processor P_{fg}. Therefore, P_{ik} sends $A(i,k)$ to all the processors P_{fg} corresponding to these computing units through the broadcast operation. Formally, for each P_{ik} with $k \in Q_l$ and $i \in \mathcal{A}(k) \cap Q_a$, where $a \in \{l+1, l+2..., h\}$, it broadcasts $A(i,k)$ to all processors P_{fg} where $f = \sum_{b=h+a-c}^{h-1} 2^b + (a-l)$, $g = k - \sum_{b=h-l+1}^{h-1} 2^b$ and $c \in \{a, a+1..., h\}$. Similarly, for each P_{kj} with $k \in Q_l$ and $j \in \mathcal{A}(k) \cap Q_c$, where $c \in \{l+1, l+2..., h\}$, it broadcasts $A(k,j)$ to all processors P_{fg} where $f = \sum_{b=h+a-c}^{h-1} 2^b + (a-l)$, $g = k - \sum_{b=h-l+1}^{h-1} 2^b$ and $a \in \{l+1, l+2..., c\}$.

P_{fg} to R_l^4: We know that for each block $A(i,j)$ in R_l^4, it is updated by all computing units $A(i,k) \otimes B(k,j)$, where $k \in Q_l \cap \mathcal{D}(i)$. And each computing unit is executed on a separate processor P_{fg}. Therefore, we regard those processors P_{fg} as a group, and reduce their computation results to P_{ij}.

5.3 The Pseudocode of 2D Sparse APSP Algorithm

The pseudocode of the 2D sparse APSP appears in Algorithm 1. T is the eTree, h is the height of T, and Q_l is the supernodes in the l-th level. For $i, j \in \{1, 2..., \sqrt{p}\}$, each processor P_{ij} owns a block $A(i,j)$. The elimination sequence is from the bottom $l = 1$ to the top $l = h$, and the elimination of the l-th level sequentially updates the four regions R_l^1, R_l^2, R_l^3 and R_l^4.

For all $k \in Q_l$, the update of R_l^1 is performed on P_{kk} (line 4). Each processor P_{kk} in R_l^1 broadcasts $A(k,k)$ to those processors P_{ik} and P_{kj} in R_l^2 (lines 5-6), where $i, j \in \mathcal{A}(k) \cup \mathcal{D}(k)$. And the processors $P(i,k)$ and $P(k,j)$ in R_l^2 update the local matrix blocks $A(i,k)$ and $A(k,j)$, respectively (lines 7-8). In order to update the blocks in R_l^3, the processors in R_l^2 broadcast local data to the processors in R_l^3. The communication operations are row BROADCAST (line 9) and column BROADCAST (line 10). The update of blocks in R_l^3 is in line 11.

The blocks $A(i,j)$ with $level(i) \le level(j)$ in R_l^4 are updated in parallel (lines 13-24). According to Corollary 5.5, we know that each computing unit that updates these blocks $A(i,j)$ is allocated to a separator processor P_{fg}, and lines 13-18 make each P_{fg} get the data needed to perform the computation unit. Each block $A(i,j)$ needs to be updated by all computing units $A(i,k) \otimes A(k,j)$, where $k \in Q_l \cap \mathcal{D}(i)$. These computing units are calculated in parallel in lines 19-22 and are sent to P_{ij} through a reduce operation in line 23. Since the distance matrix A is symmetric, the blocks $A(i,j)$ with $level(i) \le level(j)$ in R_l^4 can be updated in line 25.

5.4 Algorithm Analysis

5.4.1 Memory Requirements.

Using the **block layout**, we construct an eTree with $h = \log(\sqrt{p}+1)$, and each processor P_{ij} owns a block $A(i,j)$. For $l \in$

Algorithm 1: 2D Sparse APSP algorithm

1 **function** 2D-SPARSE-APSP(A, T):
▷ Eliminate supernodes from bottom to top (lines 2-27)
2 **for** $l = \{1, 2..., h\}$ **do**:
▷ Update R_l (lines 3-26)
3 **for** all $k \in Q_l$ and $i, j \in \mathcal{A}(k) \cup \mathcal{D}(k)$ **do**
▷ Update R_l^1 (line 4)
4 $A_{kk} \leftarrow$ ClassicalFW($A(k,k)$)
▷ R_l^1 BROADCAST to R_l^2 (lines 5-6)
5 P_{kk} broadcasts $A(k,k)$ to all P_{ik}
6 P_{kk} broadcasts $A(k,k)$ to all P_{kj}
▷ Update R_l^2 (lines 7-8)
7 $A(i,k) \leftarrow A(i,k) \oplus A(i,k) \otimes A(k,k)$
8 $A(k,j) \leftarrow A(k,j) \oplus A(k,k) \otimes A(k,j)$
▷ R_l^2 BROADCAST to R_l^3 (lines 9-10)
9 P_{ik} broadcasts $A(i,k)$ to all P_{ij}
10 P_{kj} broadcasts $A(k,j)$ to all P_{ij}
▷ Update R_l^3 (line 11)
11 $A(i,j) \leftarrow A(i,j) \oplus A(i,k) \otimes A(k,j)$
12 **end for**
▷ Update R_l^4 (lines 13-26)
▷ Parallel update $A(i,j)$ with $level(i) \le level(j)$ in R_l^4 (lines 13-24)
▷ R_l^2 to P_{fg} (lines 13-18)
13 **parallel for** each P_{ik} **where** $k \in Q_l$, $i \in \mathcal{A}(k) \cap Q_a$, $a \in \{l+1, l+2..., h\}$;
14 P_{ik} broadcasts $A(i,k)$ to all P_{fg} **where** $g = k - \sum_{b=h-l+1}^{h-1} 2^b$, $f = \sum_{b=h+a-c}^{h-1} 2^b + (a-l)$; $c \in \{a, a+1..., h\}$;
15 **end parallel for**
16 **parallel for** each P_{kj} **where** $k \in Q_l$, $j \in \mathcal{A}(k) \cap Q_c$, $c = \{l+1, l+2..., h\}$ **do**:
17 P_{kj} broadcasts $A(k,j)$ to all P_{fg} **where** $g = k - \sum_{b=h-l+1}^{h-1} 2^b$, $f = \sum_{b=h+a-c}^{h-1} 2^b + (a-l)$; $a \in \{l+1, l+2..., c\}$;
18 **end parallel for**
▷ P_{fg} to R_l^4 (lines 19-24)
19 **parallel for** each P_{ij} **where** $i \in \mathcal{A}(k), j \in i \cup \mathcal{A}(i), k \in Q_l$ **do**:
20 $a = level(i)$, $c = level(j)$;
21 **parallel for** each P_{fg} **where** $f = \sum_{b=h+a-c}^{h-1} 2^b + (a-l)$, $g = k - \sum_{b=h-l+1}^{h-1} 2^b$; $k \in Q_l \cap \mathcal{D}(i)$ **do**:
22 P_{fg} computes $A(i,k) \otimes A(k,j)$;
23 P_{fg} reduces to P_{ij}.
24 **end parallel for**
▷ Update $A(i,j)$ with $level(i) \ge level(j)$ in R_l^4 (line 25)
25 P_{ij} sends $A(i,j)$ to P_{ji}.
26 **end parallel for**
27 **end for**

$\{h, h-1..., 2\}$, the supernodes of the l-level are separators obtained by performing the **ND process**. The size of the h-level separator is $|S(n)| = |S|$. Since $|S|$ is a monotonic function of n, the size of all separators from level 2 to level $(h-1)$ are smaller than $|S|$. Consider that the first level has 2^{h-1} supernodes, so the size of each supernode in the first level is $O(\frac{n}{\sqrt{p}})$. The blocks in A can be divided into three categories according to their sizes.

(1) The size of $A(k,k)$ is $O(\frac{n^2}{p})$, for all $k \in Q_1$.
(2) The size of $A(i,k)$ and $A(k,j)$ is $O(\frac{n|S|}{\sqrt{p}})$, for all $k \in Q_1$, $i, j \in Q_l$ ($l \in \{2, 3..., h\}$).
(3) The size of $A(i,j)$ is $O(|S|^2)$, for all $i, j \in Q_l$ ($l \in \{2, 3..., h\}$).

Therefore, the size of each block $A(i,j)$ in the supernodal block sparse matrix A is $O(\frac{n^2}{p} + \frac{n|S|}{\sqrt{p}} + |S|^2)$. Since $\frac{n|S|}{\sqrt{p}} = O(\frac{n^2}{p} + |S|^2)$, the memory requirement of each processor in our model is $M = O(\frac{n^2}{p} + |S|^2)$.

5.4.2 Latency Cost.

Considering that our algorithm eliminates the supernodes of each level sequentially from $l = 1$ to h. Let L_l denote the number of messages transmitted for the elimination of the l-th level supernodes, then the total latency cost is

$$L = \sum_{l=1}^{h} L_l \tag{2}$$

Lemma 5.6. $L_l = O(\log p)$

Proof. The sum of the latency cost of updating R_l^1, R_l^2, R_l^3 and R_l^4 is L_l. We analyze the updates of these four subsets separately.

(1) R_l^1 (line 4): There is no communication operation, so the latency cost is 0.
(2) R_l^2 (lines 5-8): The communication operations are BROADCAST (lines 5-6), and the latency cost is $O(\log|\mathcal{A}(k) \cup \mathcal{D}(k)|)$. Since $|\mathcal{A}(k)| = h - l$ and $|\mathcal{D}(k)| = 2^l - 2$, thus the latency cost is $O(\log(2^l + h - l - 2))$, i.e., $O(\log p)$.
(3) R_l^3 (lines 9-11): The communication operations are BROADCAST (lines 9-10), and the latency cost is $O(\log|\mathcal{A}(k) \cup \mathcal{D}(k)|)$, which is $O(\log p)$.
(4) R_l^4 (lines 13-26): We first discuss the lines 13-24 of the algorithm. The communication process can be divided into two parts, one is BROADCAST from R_l^2 to P_{fg} (lines 14, 17), and the other is REDUCE from P_{fg} to R_l^4 (line 23). In line 14, each processor P_{ik} broadcasts $A(i,k)$ to all P_{fg} to compute $A(i,k) \otimes A(k,j)$, where $j \in Q_c \cap \mathcal{A}(i) \cup i$ and $c \in \{a, a+1..., h\}$. Since $|Q_c \cap \mathcal{A}(i) \cup i| = h - a + 1$, so the latency cost of this operation is $O(\log(h-a+1))$, which is $O(\log p)$. Similarly, in line 17, each processor P_{kj} broadcasts $A(k,j)$ to all P_{fg} to compute $A(i,k) \otimes A(k,j)$, where $i \in Q_a \cap \mathcal{A}(k) \cap \mathcal{D}(j) \cup i$ and $a \in \{l+1, l+2..., c\}$. Since $|Q_a \cap \mathcal{A}(k) \cap \mathcal{D}(j) \cup i| = c - l$, thus the latency cost of this operation is $O(\log(c-l))$, which is $O(\log p)$. In line 23, all computation results $A(i,k) \otimes A(k,j)$ are reduced to the processor P_{ij}, where $k \in Q_l \cap \mathcal{D}(i)$. Since $|Q_l \cap \mathcal{D}(i)| \leq 2^{a-l}$, thus the latency cost is $O(\log 2^{a-l})$, which is $O(\log p)$. In line 25, each P_{ij} sends $A(i,j)$ to P_{ji} in parallel, so the latency cost is $O(1)$. So the latency cost of updating R_l^4 is $O(\log p)$.

Therefore, $L_l = O(\log p)$. □

Theorem 5.7. *The latency cost of our algorithm is* $O(\log^2 p)$.

Proof. According to Equation (2) and Lemma 5.6, the latency cost of our algorithm is $L = O(\log^2 p)$. □

5.4.3 Bandwidth Cost.

Bandwidth cost can be obtained by summing the number of words in each message. In our algorithm, each message is a block of matrix A.

Let B_1 denote the bandwidth cost for the elimination of the first level supernodes, and B_l is used to denote the bandwidth cost for the elimination of the l-th level supernodes, where $l \in \{2, 3..., h\}$, then the total bandwidth cost B is

$$B = B_1 + \sum_{l=2}^{h} B_l. \tag{3}$$

Lemma 5.8. $B_1 = O(\frac{n^2 \log p}{p} + \frac{n|S|\log^2 p}{\sqrt{p}})$

Proof. (1) R_1^1: there is no communication cost.
(2) R_1^2: the number of messages is $O(\log p)$ and the per message size of BROADCAST is $O(\frac{n^2}{p})$ (lines 5-6), so the bandwidth cost is $O(\frac{n^2 \log p}{p})$.
(3) R_1^3: the number of messages is $O(\log p)$ and per message size of BROADCAST is $O(\frac{n|S|}{\sqrt{p}})$ (lines 9-10), so the bandwidth cost is $O(\frac{n|S|\log p}{\sqrt{p}})$.
(4) R_1^4: the number of messages is $O(\log p)$ and per message size is $O(\frac{n|S|}{\sqrt{p}} + |S|^2)$ (lines 14, 17, 23, 25), so the bandwidth cost is $O(\frac{n|S|\log p}{\sqrt{p}} + |S|^2 \log p)$.

Therefore, $B_1 = O(\frac{n^2 \log p}{p} + |S|^2 \log p)$. □

Lemma 5.9. $B_l = O(\frac{n|S|\log p}{\sqrt{p}} + |S|^2 \log p)$, *where* $l = \{2, 3..., h\}$.

Proof. The calculation of B_l is similar to the calculation of B_1. The bandwidth cost of updating R_l^1, R_l^2, R_l^3 and R_l^4 are 0, $O(|S|^2 \log p)$, $O(\frac{n|S|\log p}{\sqrt{p}} + |S|^2 \log p)$ and $O(\frac{n|S|\log p}{\sqrt{p}} + |S|^2 \log p)$, respectively. Therefore, $B_l = O(\frac{n|S|\log p}{\sqrt{p}} + |S|^2 \log p)$. □

Theorem 5.10. $B = O(\frac{n^2 \log^2 p}{p} + |S|^2 \log^2 p)$.

Proof. According to Equation (3), Lemma 5.8 and Lemma 5.9, obviously $B = O(\frac{n^2 \log^2 p}{p} + |S|^2 \log^2 p)$. □

5.4.4 The cost of solving the separators.

Given a graph $G = (V, E)$ with n vertices and p processors, the bandwidth and latency costs to compute a separator are $O(\frac{n \log p}{\sqrt{p}})$ and $O(\log p)$, respectively. This is the cost of computing the top-level separator. Consider the eTree with height $O(\log P)$, after computing the top level separator, the processors are divided into two parts, and each part has $p/2$ processors and stores a subgraph with $n/2$ vertices. These two parts of the processors can compute the separators of the two subgraphs in parallel, and the bandwidth and latency costs are $O(\frac{\frac{n}{2} \log \frac{p}{2}}{\sqrt{p/2}})$ and $O(\log \frac{p}{2})$, respectively. The separators of each level can be obtained similarly. Therefore, the bandwidth and latency cost to compute the separators from level h to 1 are $O(\frac{n \log^2 p}{\sqrt{p}})$ and $O(\log^2 p)$, respectively, which is subsumed by the cost of computing the APSP.

5.5 Discussion

Our algorithm lowers the latency cost of 2D-DC-APSP by a factor of $O(\frac{\sqrt{p}}{\log p})$, and lowers the bandwidth cost of 2D-DC-APSP by a factor of $O(\min(\frac{\sqrt{p}}{\log^2 p}, \frac{n^2}{|S|^2 \sqrt{p} \log^3 p}))$. It can be found that the latency cost is independent of $|S|$, and a small separator means an asymptotic reduction in the bandwidth cost (for example, $|S| = O(\sqrt{n})$). Although our algorithm can be used to compute the APSP of any graph, our algorithm is more efficient for sparse graphs with a small separator.

6 COMMUNICATION LOWER BOUND

In this section, we prove the communication lower bound of distributively computing APSP for sparse graphs. Before proving the lower bound, we first introduce the 3NL computation model and its communication lower bound [4].

6.1 3NL Computation model

Many classical linear algebra problems can be expressed in a three nested-loop (3NL) way. For example, the following pseudocode is for multiplying two $n \times n$ matrices.

for $i = 1 \leftarrow n$
  for $j = 1 \leftarrow n$
    for $k = 1 \leftarrow n$
      $C_{ij} = C_{ij} + A_{ik} B_{kj}$

The computation of matrix multiplication can be expressed as $C_{ij} = \sum_k A_{ik} B_{kj}$, and a computation like this is called a 3NL computation. This computation model is formally defined as follows:

Definition 6.1 (3NL computation [4]). Let $\mathcal{M}$ denote the set of slow memory locations on a single machine or a location in some processor's memory on a parallel machine. Let $S_a, S_b, S_c \subseteq \{1, 2..., n\} \times \{1, 2..., n\}$ denote the indices of matrices A, B, and C respectively, and $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{c}$ denote the mapping from indices to memory $S_a \rightarrow \mathcal{M}$, $S_b \rightarrow \mathcal{M}$ and $S_c \rightarrow \mathcal{M}$ respectively. Let $Mem(\mathbf{a}(i,k))$ denote the value of memory location $\mathbf{a}(i,k) \in \mathcal{M}$, i.e., A_{ik}. Similarly, $Mem(\mathbf{b}(k,j))$ is B_{kj} and $Mem(\mathbf{c}(i,j))$ is C_{ij}. A computation is considered to be a 3NL if it includes computing, for all $(i,j) \in S_c$, $S_{ij} \subseteq \{1, 2, ...n\}$ and $k \in S_{ij}$,

$$Mem(\mathbf{c}(i,j)) = f_{ij}(\{g_{ijk}(Mem(\mathbf{a}(i,k)), Mem(\mathbf{b}(k,j)))\})$$

where

1) $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$ are one-to-one mappings

2) functions f_{ij} and g_{ijk} depend nontrivially on their arguments.

Lemma 6.2 (Ballard et al. [6]). *On a p-processor parallel machine, the bandwidth and latency lower bounds of a 3NL computation are $\Omega(\frac{F}{pM^{1/2}})$ and $\Omega(\frac{F}{pM^{3/2}})$ respectively, where F is the number of computation operations and M is the per-process memory size.*

Lemma 6.2 can be applied to almost all linear algebra problems for dense or sparse matrices, such as solving triangular systems, Cholesky and LU factorization. We apply it to the APSP problem.

6.2 Lower Bound

We assume that data is distributed on the distributed memory system in **block layout**. For a given sparse graph $G = (V, E)$ with n vertices, we first prove that computing APSP of a sparse graph is a 3NL computation, and then give the bandwidth and latency lower bounds under the communication model.

Lemma 6.3. *Computing the APSP of a sparse graph is a 3NL computation.*

Proof. We know that the update of the distance matrix A_{ij} in the APSP problem is specified as

$$A_{ij} = \min_k (A_{ik} + A_{kj}). \tag{4}$$

We let the function f_{ij} be defined as Equation (4) and g_{ijk} is defined as a scalar addition operation. Obviously f_{ij} and g_{ijk} depend nontrivially on their arguments. Since the input and output in Equation (4) are both A, we make the correspondences that A_{ij} is stored in a certain processor location $\mathbf{a}(i,j) = \mathbf{b}(i,j) = \mathbf{c}(i,j)$, and $\mathbf{a}, \mathbf{b}, \mathbf{c}$ are one-to-one mappings to the processor memory. Furthermore, using the elimination tree technique to compute the APSP of the sparse graph, we can get the set S_{ij} of k. Assuming that the elimination tree has N supernodes, we know that each supernode is a collection of nodes with similar adjacency structures. Let $S(u)$ denote the vertex set of the u-th supernode, and $S(u \cup v)$ denote the vertex set of the u-th supernode and the v-th supernode, then $\bigcup_{u=1}^{N} S(u) = \{1, 2..., n\}$. Consider the update of A_{ij}, assuming $i \in S(u)$ and $j \in S(v)$, if w is a cousin node of u or v in eTree, then the elimination of supernode w will not update block $A(u,v)$. This is because $A(u,w)$ or $A(w,v)$ is empty and the computation $A(u,w) \otimes A(w,v)$ can be avoided. Therefore, for all $k \in S(w)$, $A_{ik} + A_{kj}$ can be avoided. We can get the set

$$S_{ij} = \{S(u \cup \mathcal{A}(u) \cup \mathcal{D}(u))\} \cap \{S(v \cup \mathcal{A}(v) \cup \mathcal{D}(v))\}, \tag{5}$$

which means that for each A_{ij}, $S_{ij} \subseteq \{1, 2..., n\}$.

□

Lemma 6.4. *The total number of operations to compute the APSP is $\Omega(n^2|S|)$.*

Proof. According to Definition 6.1, $F = \sum_{(i,j) \in S_c} |S_{ij}|$. Based on the labeling of supernodes in subsection 5.2, the top-level supernode is the N-th supernode in eTree. We only consider those computations to update A_{ij} with $i, j \in \bigcup_{u=1}^{N-1} S(u)$ during the elimination of the N-th supernode, which is part of the total computations.

Table 2: Asymptotic memory, bandwidth, latency for 2D-DC-APSP and 2D-SPARSE-APSP and communication lower bound

Parameter	2D-DC-APSP [24]	2D-SPARSE-APSP (here)	Lower bound	
			dense graphs [6]	sparse graphs (here)
Per-process memory (M)	$O(\frac{n^2}{p})$	$O(\frac{n^2}{p} + \lvert S\rvert^2)$	$\Omega(\frac{n^2}{p})$	$\Omega(\frac{n^2}{p})$
Bandwidth cost (B)	$O(\frac{n^2}{\sqrt{p}})$	$O(\frac{n^2 \log^2 p}{p} + \lvert S\rvert^2 \log^2 p)$	$\Omega(\frac{n^2}{\sqrt{p}})$	$\Omega(\frac{n^2}{p} + \lvert S\rvert^2)$
Latency cost (L)	$O(\sqrt{p} \log^2 p)$	$O(\log^2 p)$	$\Omega(\sqrt{p})$	$\Omega(\log^2 p)$

Since $|S(N)| = |S| \ll n$, the number of A_{ij} with $i, j \in \bigcup_{u=1}^{N-1} S(u)$ is $(n - |S|)^2 \approx n^2$. For the update of each A_{ij}, we compute $|S_{ij}|$ during the elimination of N. Since the top-level supernode N is the ancestor of all the other supernodes in the eTree, according to Equation (4), $S(N) \subseteq S_{ij}$ and $|S(N)| \leq |S_{ij}|$.

Therefore, the number of operations to compute these A_{ij} is at least $n^2|S|$, and the total number of operations is $\Omega(n^2|S|)$.

□

Theorem 6.5. *On a distributed memory machine with p processors and per-process memory size is $M = O(\frac{n^2}{p} + |S|^2)$, the bandwidth and latency lower bounds for solving the APSP of sparse graphs are $\Omega(\frac{n^2}{p} + |S|^2)$ and $\Omega(\log^2 p)$, respectively.*

Proof. If $\frac{n}{\sqrt{p}} \geq |S|$, then $M = O(\frac{n^2}{p})$. According to Lemma 6.2 and $F = \Omega(n^2|S|)$, the lower bound of bandwidth cost is $\Omega(\frac{n|S|}{\sqrt{p}})$, which is $\Omega(|S|^2)$ since $\frac{n}{\sqrt{p}} \geq |S|$. The lower bound of latency cost is $\Omega(1)$. If $\frac{n}{\sqrt{p}} \leq |S|$, then $M = O(|S|^2)$. According to Lemma 6.2 and $F = \Omega(n^2|S|)$, the lower bound of bandwidth cost is $\Omega(\frac{n^2}{p})$. The lower bound of latency cost is $\Omega(1)$. Therefore, the bandwidth lower bound is $\Omega(\frac{n^2}{p} + |S|^2)$ and latency lower bound is $\Omega(1)$. The latency lower bound we get according to Lemma 6.2 is trivial.

However, with the **block layout**, we suppose the height of the eTree is h. Then for $k \in Q_1$, the size of the block $A(k, k)$ is $O(\frac{n}{2^h} \times \frac{n}{2^h})$, which should be smaller than M, thus $h = \Omega(\log p)$. Let N denote the top-level supernode, then all nodes in the eTree are descendants of N. Consider the elimination of the l-th level supernodes, the update of $A(N, N)$ needs to compute all $A(N, k) \otimes A(k, N)$, where $k \in Q_l$ and $|Q_l| = 2^{h-l}$. Due to the **block layout**, 2^{h-l} blocks $A(N, k)$ are stored on 2^{h-l} different processors. In our model, each processor can only transmit one message with another processor at a time, therefore, updating the $A(N, N)$ requires at least $\log 2^{h-l}$ messages to be transmitted. Therefore, the latency lower bound is $\sum_{l=1}^{h} h - l$, which is $\Omega(\log^2 p)$.

□

ACKNOWLEDGMENTS

This work is supported in part by the National Natural Science Foundation of China Grant No. 61972447.

REFERENCES

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. 1974. *The Design and Analysis of Computer Algorithms.* Addison-Wesley.
[2] Ed Anderson, Zhaojun Bai, Christian H. Bischof, L. Susan Blackford, James Demmel, Jack J. Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven Hammarling, A. McKenney, and Danny C. Sorensen. 1999. *LAPACK Users' Guide, Third Edition.* SIAM.
[3] Grey Ballard, Aydin Buluç, James Demmel, Laura Grigori, Benjamin Lipshitz, Oded Schwartz, and Sivan Toledo. 2013. Communication optimal parallel multiplication of sparse random matrices. In *SPAA '13.*
[4] Grey Ballard, Erin C. Carson, James Demmel, Mark Hoemmen, Nicholas Knight, and Oded Schwartz. 2014. Communication lower bounds and optimal algorithms for numerical linear algebra. *Acta Numer.* 23 (2014), 1–155.
[5] Grey Ballard, James Demmel, Olga Holtz, Benjamin Lipshitz, and Oded Schwartz. 2012. Communication-optimal parallel algorithm for strassen's matrix multiplication. In *SPAA '12.*
[6] Grey Ballard, James Demmel, Olga Holtz, and Oded Schwartz. 2011. Minimizing Communication in Numerical Linear Algebra. *SIAM J. Matrix Anal. Appl.* 32, 3 (2011), 866–901.
[7] Grey Ballard, Alex Druinsky, Nicholas Knight, and Oded Schwartz. 2016. Hypergraph Partitioning for Sparse Matrix-Matrix Multiplication. *ACM Trans. Parallel Comput.* 3, 3 (2016), 18:1–18:34.
[8] Bernard A Carré. 1971. An algebra for network routing problems. *IMA Journal of Applied Mathematics* 7, 3 (1971), 273–294.
[9] James Demmel, David Eliahu, Armando Fox, Shoaib Kamil, Benjamin Lipshitz, Oded Schwartz, and Omer Spillinger. 2013. Communication-Optimal Parallel Recursive Rectangular Matrix Multiplication. In *IPDPS '13.*
[10] Robert W. Floyd. 1962. Algorithm 97: Shortest path. *Commun. ACM* 5, 6 (1962), 345.
[11] Alan George. 1973. Nested dissection of a regular finite element mesh. *SIAM J. Numer. Anal.* 10, 2 (1973), 345–363.
[12] Anshul Gupta, George Karypis, and Vipin Kumar. 1997. Highly Scalable Parallel Algorithms for Sparse Matrix Factorization. *IEEE Trans. Parallel Distributed Syst.* 8, 5 (1997), 502–520.
[13] Dror Irony, Sivan Toledo, and Alexandre Tiskin. 2004. Communication lower bounds for distributed-memory matrix multiplication. *J. Parallel Distributed Comput.* 64, 9 (2004), 1017–1026.
[14] Jing-Fu Jenq and Sartaj Sahni. 1987. All Pairs Shortest Paths on a Hypercube Multiprocessor. In *ICPP '87.*
[15] Hong Jia-Wei and Hsiang-Tsung Kung. 1981. I/O complexity: The red-blue pebble game. In *STOC '81.*
[16] Donald B. Johnson. 1977. Efficient Algorithms for Shortest Paths in Sparse Networks. *J. ACM* 24, 1 (1977), 1–13.
[17] George Karypis and Vipin Kumar. 1998. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM J. Sci. Comput.* 20, 1 (1998), 359–392.
[18] George Karypis and Vipin Kumar. 1998. A Parallel Algorithm for Multilevel Graph Partitioning and Sparse Matrix Ordering. *J. Parallel Distributed Comput.* 48, 1 (1998), 71–95.
[19] Xiaoye S. Li and James Demmel. 2003. SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems. *ACM Trans. Math. Softw.* 29, 2 (2003), 110–140.
[20] Richard J Lipton, Donald J Rose, and Robert Endre Tarjan. 1979. Generalized nested dissection. *SIAM journal on numerical analysis* 16, 2 (1979), 346–358.
[21] Joon-Sang Park, Michael Penner, and Viktor K. Prasanna. 2004. Optimizing Graph Algorithms for Improved Cache Performance. *IEEE Trans. Parallel Distributed Syst.* 15, 9 (2004), 769–782.
[22] Piyush Sao, Ramakrishnan Kannan, Prasun Gera, and Richard W. Vuduc. 2020. A supernodal all-pairs shortest path algorithm. In *PPoPP '20.*
[23] Piyush Sao, Xiaoye Sherry Li, and Richard W. Vuduc. 2018. A Communication-Avoiding 3D LU Factorization Algorithm for Sparse Matrices. In *IPDPS '18.*
[24] Edgar Solomonik, Aydin Buluç, and James Demmel. 2013. Minimizing Communication in All-Pairs Shortest Paths. In *IPDPS '13.*
[25] Stephen Warshall. 1962. A Theorem on Boolean Matrices. *J. ACM* 9, 1 (1962), 11–12.
[26] Cui-Qing Yang and Barton P. Miller. 1988. Critical Path Analysis for the Execution of Parallel and Distributed Programs. In *ICDCS '88.*