A Supernodal All-Pairs Shortest Path Algorithm

Piyush Sao, Ramakrishnan Kannan*
Oak Ridge National Laboratory
Oak Ridge, TN
{saopk,kannanr}@ornl.gov

Prasun Gera, Richard Vuduc
Georgia Institute of technology
Atlanta, GA
{prasun.gera,richie}@gatech.edu

ABSTRACT

We show how to exploit graph sparsity in the Floyd-Warshall algorithm for the all-pairs shortest path (Apsp) problem. Floyd-Warshall is an attractive choice for Apsp on high-performing systems due to its structural similarity to solving dense linear systems and matrix multiplication. However, if sparsity of the input graph is not properly exploited, Floyd-Warshall will perform unnecessary asymptotic work and thus may not be a suitable choice for many input graphs. To overcome this limitation, the key idea in our approach is to use the known algebraic relationship between Floyd-Warshall and Gaussian elimination, and import several algorithmic techniques from sparse Cholesky factorization, namely, fill-in reducing ordering, symbolic analysis, supernodal traversal, and elimination tree parallelism. When combined, these techniques reduce computation, improve locality and enhance parallelism. We implement these ideas in an efficient shared memory parallel prototype that is orders of magnitude faster than an efficient multi-threaded baseline Floyd-Warshall that does not exploit sparsity. Our experiments suggest that the Floyd-Warshall algorithm can compete with Dijkstra's algorithm (the algorithmic core of Johnson's algorithm) for several classes sparse graphs.

KEYWORDS

graph algorithm, sparse matrix computations, shared-memory parallelism, communication-avoiding algorithms

*This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).

PPoPP '20, February 22–26, 2020, San Diego, CA, USA
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-6818-6/20/02...$15.00
https://doi.org/10.1145/3332466.3374533

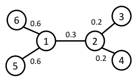
ACM Reference Format:
Piyush Sao, Ramakrishnan Kannan and Prasun Gera, Richard Vuduc. 2020. A Supernodal All-Pairs Shortest Path Algorithm. In *25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '20), February 22–26, 2020, San Diego, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3332466.3374533

1 INTRODUCTION

In this paper, we improve the performance of the classic Floyd-Warshall algorithm (Floyd-Warshall) for the all-pairs shortest path (Apsp) problem on shared memory parallel machines for *sparse* graphs. Floyd-Warshall is appropriate when the graph is dense or nearly so, in which case one can achieve good parallel scalability and high-performance by exploiting the algebraic connection between Floyd-Warshall and the Gaussian elimination process for solving linear systems [3, 6, 12]. Through that lens, Floyd-Warshall reduces to matrix-multiplication-like (level-3 BLAS-like) operations, thereby enabling fast computations of Apsp on, for instance, GPUs [5] or distributed memory platforms [38]. But if the graph is sparse, then the implementation of Floyd-Warshall must change. *Our key insight, similar to matrix multiplication, is that the full body of algorithmic techniques from sparse direct solvers can be applied [9] for Floyd-Warshall on sparse graphs.*

Formally, we consider Apsp on a weighted undirected graph $G = (V,E)$ with $n = |V|$ vertices and $m = |E|$ edges. The weights may be negative, but we preclude cycles whose sum of weights is negative. If, furthermore, the graph is dense, so that $m = O(n^2)$, then one may use the Floyd-Warshall algorithm. Its overall algorithmic structure consists of three nested-loops (Algorithm 1), each iterating over all vertices, so its sequential complexity is $O(n^3)$ operations. Throughout, it updates a matrix $\{\text{Dist}\}_{ij}$ that stores the length of the *current* shortest path between any two vertices v_i and v_j; this distance is initialized to ∞ if no path has yet been discovered. Floyd-Warshall maintains, at each iteration k, the invariant that the $\{\text{Dist}\}_{ij}$ is minimum with at most k vertices as intermediaries. Hence, as shown in Fig. 1, Floyd-Warshall discovers more paths between vertices and the number of infinite $\{\text{Dist}\}_{ij}$ entries decreases.

For sparse graphs, $m = O(n)$ and we prefer methods with better asymptotic scaling. One option is Johnson's algorithm, which scales like $O(n^2 \log n + nm)$ [21]. However, Johnson's



(a) Graph

$$\begin{bmatrix} 0 & 0.3 & \infty & \infty & 0.6 & 0.6 \\ 0.3 & 0 & 0.2 & 0.2 & \infty & \infty \\ \infty & 0.2 & 0 & \infty & \infty & \infty \\ \infty & 0.2 & \infty & 0 & \infty & \infty \\ 0.6 & \infty & \infty & \infty & 0 & \infty \\ 0.6 & \infty & \infty & \infty & \infty & 0 \end{bmatrix} \Rightarrow_{\text{after 2 iteration}} \begin{bmatrix} 0 & 0.3 & 0.5 & 0.5 & 0.6 & 0.6 \\ 0.3 & 0 & 0.2 & 0.2 & 0.9 & 0.9 \\ 0.5 & 0.2 & 0 & 0.4 & 1.1 & 1.1 \\ 0.5 & 0.2 & 0.4 & 0 & 1.1 & 1.1 \\ 0.6 & 0.9 & 1.1 & 1.1 & 0 & 1.2 \\ 0.6 & 0.9 & 1.1 & 1.1 & 1.2 & 0 \end{bmatrix}$$

(b) The initial distance matrix and after two Floyd-Warshall iteration

Figure 1: In Floyd-Warshall algorithm, the distance matrix which is initially sparse, quickly become dense if vertex ordering is not optimal

algorithm cannot effectively use the features of modern computer architecture such as long SIMD vector units or cache, and therefore, it underutilizes modern high-performing computing systems. It is natural to ask whether we can exploit sparsity in Floyd-Warshall to reduce its asymptotic complexity while retaining its parallel-scalability.

Our approach to doing so derives from the known technique of vertex reordering [26, 33]. During the execution of Floyd-Warshall, infinite values of $\{\text{Dist}\}_{ij}$ will play the role of "zero entries" in sparse numerical linear algebra, and certain operations on infinite values may be avoided. By choosing the optimal vertex order of the outer-loop of Floyd-Warshall, we can defer replacement of infinite values for more iterations of the algorithm. The so-called *fill-in reducing orderings* used in, for instance, sparse Cholesky factorization to keep the factor matrix sparse, are also optimal in the case of Floyd-Warshall. If a graph has a minimal vertex separator of size S that partitions the graph into two components of roughly equal size, then the use of a fill-in reducing ordering in Floyd-Warshall will incur only $O\left(n^2 S+S^3\right)$ operations. This can be asymptotically lower than $O\left(n^3\right)$. For instance, in a planar graph like a road network, the separator size is $S = O\left(\sqrt{n}\right)$; therefore, Floyd-Warshall would incur $O\left(n^{2.5}\right)$ operations. Even in the case of graphs with $S=O(n)$, using the fill-in reducing ordering, a constant-factor reduction in the number of operations can be substantial. Therefore, using an optimal reordering is necessary for good performance of a sparse Floyd-Warshall.

The use of fill-reducing orderings poses several challenges. First, one must design a careful data structure that can accommodate new entries in $\{\text{Dist}\}_{ij}$. Secondly, the data structure should also support *blocked* operations, to effectively use the memory hierarchies in modern architectures [17]. These issues motivate our *supernodal Floyd-Warshall*, or SuperFw, inspired by supernodal sparse Cholesky factorization [9]. In the supernodal approach, we group nodes having a similar adjacency structure into *supernodes*, and operate on supernodes instead of individual vertices, thereby leading to a blocked algorithm. To accommodate new entries in the $\{\text{Dist}\}_{ij}$, we use *symbolic analysis*, which can efficiently extract the fill-in structure of the $\{\text{Dist}\}_{ij}$ matrix. We perform symbolic analysis and extract supernodal structure to set up a supernodal block sparse matrix, which allows blocked operations while exploiting the sparsity. Finally, the operation and task dependencies in SuperFw can be represented by an *elimination tree*. Its structure indicates what operations may execute concurrently and, therefore, guides parallelism.

Table 1: List of symbols used

Symbol type	Symbol	Description
	$x \oplus y$	$min(x,y)$
	$x \otimes y$	$x+y$
	$\mathcal{P} \times \mathcal{Q}$	Cartesian Product of non-empty sets $\mathcal{P}, \mathcal{Q}$
Graphs	n	Dimension of the matrix A
	n_b	Dimension of every block of Dist matrix
	$G(V,E)$	Input Graph G with V vertices and E edges
	$Dist$	Distance matrix
Supernode	E	Elimination tree of A
	S	Top level separator of E
	C_1, C_2	Children etrees of E
	$\mathcal{A}(a)$	set of ancestors of a supernode a
	$\mathcal{D}(a)$	set of descendents of the supernode a

Applying these ideas from sparse Cholesky factorization results in an efficient shared-memory parallel SuperFw implementation. We show that it can in practice be orders of magnitude faster than an efficient implementation of Floyd-Warshall that does not exploit the sparsity (Section 5). Moreover, despite performing asymptotically more operations, SuperFw's better match to modern hardware can make it comparable to or even faster than Johnson's algorithm for many sparse graphs, as well as more scalable. Finally, given the rich literature for optimizing sparse Cholesky high-performance systems, we believe many of these techniques are also applicable to graph path problems. We discuss potential scenarios in which graph path analysis would benefit from optimization techniques in linear systems, and vice versa.

2 BACKGROUND

Many path problems in graph analysis can be described succinctly in a semiring algebra. We review this formalism and the resulting classical and blocked Floyd-Warshall algorithms for Apsp, below.

Notation and terminology. Let $G = \{V,E,W\}$ be an undirected weighted graph with a vertex set V containing $n=|V|$ vertices or nodes, edge set E with $m=|E|$ edges, and weights W, defined below. Denote the i-th vertex by v_i and an edge between v_i and v_j by $e_{i,j}$. The weights are represented by W, a sparse symmetric matrix whose entry $w_{i,j}$ denotes the distance between vertices v_i and v_j if $e_{i,j} \in E$; otherwise, $w_{i,j} = \infty$.

During the computation of Apsp, Floyd-Warshall maintains and updates a 2-D array of distances, Dist. Each entry Dist$[i,j]$ holds the current shortest distance between v_i and v_j discovered so far, with its value at termination of the algorithm being the shortest such distance. We will assume for simplicity that the graph G consists of a single connected component, in which case the Dist eventually becomes fully dense; however, our implementation will work when there are multiple connected components.

2.1 Classical Floyd-Warshall algorithm

Algorithm 1 Floyd-Warshall algorithm for Apsp

```
1: function FloydWarshall($G=(V,E)$):
2:   Let $n \leftarrow \dim(V)$
3:   Let $\mathrm{Dist}[i,j] = \begin{cases} w_{i,j} & \text{if}(i,j) \in E \\ \infty & \text{otherwise} \end{cases}$
4:   for $k = \{1, 2 \ldots, n\}$ do:
5:     for $i = \{1, 2 \ldots, n\}$ do:
6:       for $j = \{1, 2 \ldots, n\}$ do:
7:         $\mathrm{Dist}[i,j] = \min\{\mathrm{Dist}[i,j],\ \mathrm{Dist}[i,k] + \mathrm{Dist}[k,j]\}$
8:   Return Dist
```

Floyd-Warshall uses a dynamic programming approach to computing Apsp, as shown in Algorithm 1. It initializes Dist with the input weights W. Then, in the k-th iteration, it checks for all pairs of vertices v_i and v_j if there is a shorter path between them via the intermediate vertex v_k. If so, Floyd-Warshall updates $\mathrm{Dist}[i,j]$. Therefore, $\mathrm{Dist}[i,j]$ after k steps, which we denote by $\mathrm{Dist}^k(i,j)$, may be defined recursively as

$$\mathrm{Dist}^k[i,j] \leftarrow \min\left\{\mathrm{Dist}^{k-1}[i,j], \mathrm{Dist}^{k-1}[i,k] + \mathrm{Dist}^{k-1}[k,j]\right\}.$$

This computation may be done in place, with $\mathrm{Dist}^k[i,j]$ overwriting $\mathrm{Dist}^{k-1}[i,j]$. It can also be shown that after k iterations, $\mathrm{Dist}^k[i,j]$ is the shortest of all paths between v_i and v_j that use only vertices from the set $\{v_1, v_2, \ldots, v_k\}$.[1] Therefore, at the end of the n-th iteration $\mathrm{Dist}^n[i,j]$ will be the length of the shortest path between v_i and v_j.

2.2 Min-Plus Matrix Multiplication

Apsp may be understood algebraically as computing the matrix closure of the weight matrix, W, defined over the tropical semiring [12]. In more basic terms, let $\oplus$ and $\otimes$ denote the two binary scalar operators

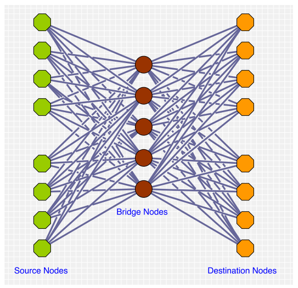
$$\begin{aligned} x \oplus y &:= \min(x,y) \\ x \otimes y &:= x + y, \end{aligned}$$

where x and y are real values or ∞. Next, consider two matrices $A \in \mathbb{R}^{m \times k}$ and $B \in \mathbb{R}^{k \times n}$. The Min-Plus product C of A and B is

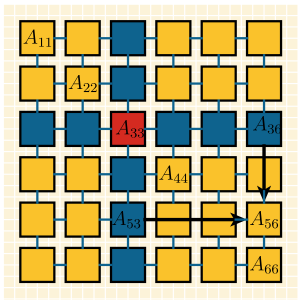
$$C_{ij} \leftarrow \sum_{k}^{\oplus} A_{ik} \otimes B_{kj} = \min_{k} \left(A_{ik} + B_{kj} \right).$$

This product is the analogue of matrix-matrix multiplication over the reals. To see its connection to graph path analysis, consider an example of the complete tripartite graph

[1]This fact holds only if there are no cycles of negative weight sum. In the presence of negative cycles, the minimum path length between any two vertices in the cycle will be ∞.



(a) MinPlus product $C = A \otimes B$



(b) Substeps of Algorithm 2

Figure 2: Fig. 2a shows the shortest path between source vertices and destination vertices that goes through bridge vertices (see Section 2.2). Fig. 2b illustrates substeps of Apsp on the diagonal subgraph, panel (block row and column) update, and outer-product update of the trailing subgraph.

in Fig. 2a. This graph has three disjoint subsets of $m+n+p$ vertices: m *source* vertices, $\{s_1, s_2 \cdots s_m\}$; n *destination* vertices, $\{d_1, d_2 \cdots d_n\}$; and p *bridge* vertices, $\left\{b_1, b_2 \cdots b_p\right\}$. Every source and destination connects to every bridge, but no vertices within each subset are adjacent. Let A_{ik} denotes the weight of the edge (s_i, b_k) and let B_{kj} be the weight of (b_k, d_j). Then $A_{ik} \otimes B_{kj} = A_{ik} + B_{kj}$ denotes the length of path from s_i to d_j via b_k. Thus, the shortest path between s_i and d_j via any vertex b_k is the minimum of $A_{ik} \otimes B_{kj}$ over all k. This interpretation of the Min-Plus product helps to understand the following blocked version of Floyd-Warshall (Algorithm 2).

2.3 Blocked Floyd-Warshall algorithm

Suppose we divide Dist into $n_b \times n_b$ blocks, each of size $b \times b$ (i.e., $n_b = \frac{n}{b}$). Let A_{ij} denote the (i,j) block of A, where $1 \le i,j \le n_b$. Then a blocked version of Floyd-Warshall, called BlockedFw in Algorithm 2, can carry out the same Apsp computation as Floyd-Warshall in the following three steps, as illustrated in Fig. 2b:

- **Diagonal Update:** Perform the classic Floyd-Warshall algorithm on a diagonal block, A_{kk}.
- **Panel Update:** Update the k-th block row and column. For any block $A(k,j)$, $j \ne k$ in the block row, the update is a Min-Plus multiply with A_{kk} from the left, and for block $A(i,k)$ on the k-th block column is Min-Plus multiply with A_{kk} from right, i.e.,

  $$\begin{aligned} A(k,j) &\leftarrow A(k,j) \oplus A(k,k) \otimes A(k,j) && j \ne k \\ A(i,k) &\leftarrow A(i,k) \oplus A(i,k) \otimes A(k,k) && i \ne k \end{aligned}$$

  Here, $\oplus$ denotes element-wise application of the corresponding scalar operator, and $\otimes$ denotes Min-Plus product.
- **MinPlus Outer Product**: Perform the outer product of k-th block row and block column, and update all the remaining blocks of matrix A

  $$A(i,j) \leftarrow A(i,j) \oplus A(i,k) \otimes A(k,j) \ \ i,j \ne k.$$

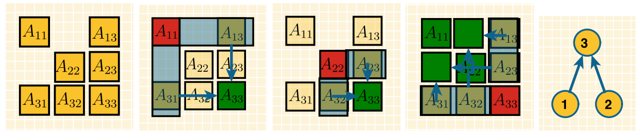


Figure 3: An illustration of BlockedFw on a 3 × 3 block-partitioned matrix.

This step is analogous to a Schur-complement update in LU or Cholesky factorization.

Algorithm 2 A blocked version of Floyd-Warshall algorithm for Apsp

```
1: function BlockedFloydWarshall($A$):
2:   for $k = \{1, 2 \ldots, n_b\}$ do:
     Diagonal Update
3:     $A(k,k) \leftarrow$ Floyd-Warshall($A(k,k)$)
     Panel Update
4:     $A(k,:) \leftarrow A(k,:) \bigoplus A(k,k) \otimes A(k,:)$
5:     $A(:,k) \leftarrow A(:,k) \bigoplus A(:,k) \otimes A(k,k)$
     MinPlus Outer Product
6:     for $i = \{1, 2 \ldots, n_b\}, i \neq k$ do:
7:       for $j = \{1, 2 \ldots, n_b\}, j \neq k$ do:
8:         $A(i,j) \leftarrow A(i,j) \bigoplus A(i,k) \otimes A(k,j)$
9:   Return $A$
```

3 SUPERNODAL FLOYD-WARSHALL

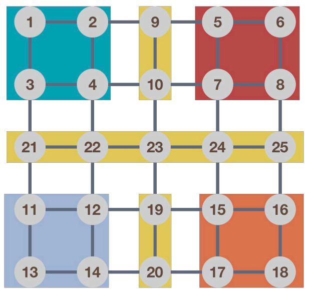
When the Dist matrix is sparse, we can use ideas from sparse direct solvers to transform the BlockedFw algorithm into one that can maintain and exploit that sparsity. There are three critical concepts: (i) *reordering*, which helps to control the dynamically evolving sparsity structure of Dist, thereby controlling the amount of asymptotic work incurred by BlockedFw; (ii) *supernodes*, which help manage and organize this structure, thereby exploiting locality; and (iii) the *elimination tree*, which helps express the operational and data dependencies, thereby exposing parallelism.

3.1 Effect of ordering on BlockedFw

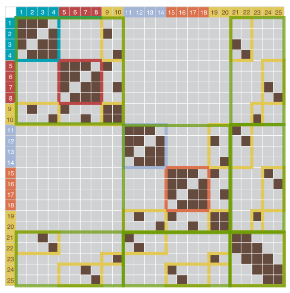
When running BlockedFw, the sparsity of Dist in any given iteration determines how that sparsity will change in subsequent iterations. We illustrate these dependencies in Fig. 3. There, we show a 3×3 block-partitioned sparse matrix with a particular sparsity pattern in which the A_{21} and A_{12} blocks are "empty," meaning that all the entries in them are infinity.

The first iteration of BlockedFw, i.e., $k = 1$, performs a DiagUpdate on the A_{11} block followed by PanelUpdate on the block A_{13} and A_{31}. Since A_{12} and A_{21} are empty, they remain empty after the PanelUpdate step. In the OuterUpdate step, the blocks A_{22}, A_{23}, and A_{32} remain unchanged since any updates thereto depend on A_{12} or A_{21}, which are empty. We only perform an update on the block A_{33} as follows:

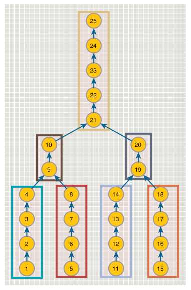
$$A_{33} \leftarrow A_{33} \oplus A_{31} \otimes A_{13}.$$



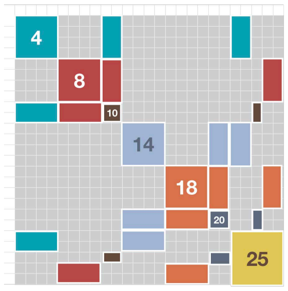
(a) A 25×25 sparse matrix



(b) Non-zero structure of the permuted matrix



(c) elimination tree



(d) supernodal matrix

Figure 4: Nested Dissection (ND) on a 5×5 grid graph. Under an ND ordering, we find a graph separator (highlighted in yellow Fig. 4a), and label the nodes in the separator in the end. Fig. 4b shows the adjacency matrix of the graph permuted in ND ordering. The elimination tree (Fig. 4c) captures the dependency in elimination of different nodes. The Fig. 4d shows the final block sparse matrix obtained after steps described in Sections 3.2 and 3.3.

Similarly, when $k = 2$, we perform DiagUpdate on the block A_{22}, and PanelUpdate on A_{23} and A_{32}, and we only perform update on the block A_{33}

$$A_{33} \leftarrow A_{33} \oplus A_{32} \otimes A_{23}.$$

In the 3rd iteration, we perform DiagUpdate on the block A_{33}, PanelUpdate from the left on A_{31} and A_{32}, and from the right on A_{23} and A_{13}. Since none of the third row and column blocks are empty, we update all the remaining blocks in the OuterUpdate step. The blocks A_{12} and A_{21} which had been empty so far finally become full blocks:

$$A_{12} \leftarrow A_{13} \otimes A_{32}$$
$$A_{21} \leftarrow A_{23} \otimes A_{31}.$$

This example reveals two essential aspects of BlockedFw.

(1) For this particular example, the block sparsity is maintained until the last iteration. That is, only when $k = 3$ does execution of the OuterUpdate step require an update on the complete matrix in which empty blocks (of all ∞ values) become finite. This change from infinite to finite values is the graph-path analog of nonzero *fill-in* in sparse linear algebra.

(2) The OuterUpdate step in the first iteration does not involve updating any block of second block row or column, i.e. A_{22}, A_{23} and A_{23}, and vice versa. Therefore, the DiagUpdate and PanelUpdate of iterations $k=1$ and $k=2$ do not have any data dependencies, and may therefore proceed in parallel. However, both iterations update the A_{33} block. Nevertheless, since Min-Plus Matrix Addition is associative, the updates from the two iterations can be done in any order. Thus, the final A_{33} after the two updates will be given by

$$A_{33} \leftarrow A_{33} \oplus A_{32} \otimes A_{23} \oplus A_{31} \otimes A_{13}.$$

This dependency between operations in iteration $k=3$ on those from $k=1$ and $k=2$ can be described by a tree, as shown in Fig. 3 (last).

3.2 Nested-Dissection Ordering

From the theory and practice of sparse matrix reordering, it is well understood that "arrow" patterns for Gaussian elimination work well in reducing fill-in. Indeed, we can reorder the adjacency matrix of the graph G to obtain block-arrow structure through a process known as *nested-dissection* (ND) [14], which graph partitioning tools like Metis or Scotch can be used to obtain [22, 31].

The ND process may be summarized as follows. Our initial goal is to compute a *vertex separator* $S \subset V$ that partitions the vertices of the graph G into three disjoints sets, $V = C_1 \cup S \cup C_2$, so that following holds:

(1) there are no edges between any vertex in C_1 to any vertex in C_2;
(2) $|C_1|$ and $|C_2|$ are roughly equal; and
(3) S is as small as possible.

Using this partition, we can reorder the adjacency matrix A so that the vertices within each set C_1 and C_2 have consecutive indices; and vertices in S have a higher index than any vertex in C_1 and C_2. For example, in Fig. 4, we show a grid graph G, a separator, its adjacency matrix, and the 3×3 block-arrow matrix obtained by reordering the matrix as described above. This process may be performed recursively within C_1 and C_2 to obtain a more fine-grained ordering for the entire matrix.

3.3 Supernodal structure extraction

The goal of this step is to obtain a blocked sparse matrix as shown in Fig. 4b that we shall call supernodal matrix, from the permuted matrix using ND ordering (shown in Fig. 4d); and to calculate the so-called *elimination tree* (shown in Fig. 4c) that guides the scheduling and parallelism of our algorithm. We borrow the so-called *symbolic analysis* used in sparse direct solvers to do so.

***S*ymbolic analysis:** Symbolic analysis is the calculation of the exact *fill-in* structure in sparse Cholesky factorization. It enables preallocation of memory for any fill-ins.

***E*limination tree:** Recall the 3×3 block sparse matrix shown in Fig. 3. Elimination can proceed in either $\{1, 2, 3\}$ or $\{2, 1, 3\}$ order while maintaining low fill. Such an elimination ordering can be described using a tree Fig. 3, called elimination tree or eTree. The eTree is also calculated during the symbolic analysis step. ND ordering leads to a multilevel eTree as shown in Fig. 4c.

***S*upernodes:** A supernode is a collection of vertices or nodes that have a similar fill-in structure. When that occurs, these vertices may be grouped together to obtain block-sparse matrix. The supernodal partition is obtained by performing vertex contraction on the eTree, yielding a supernodal eTree. For subsequent discussion, eTree refers to a supernodal eTree.

***A*ncestor and descendant supernodes:** Further, we define the ancestors of a supernode as the set of supernodes that consist of its parent, the parent of parent, and so on. Similarly, if supernode a is an ancestor of another supernode b, then we say b is a descendant of a. In the eTree representation, the ancestors of a node occupy a higher spot than that node and descendants appear below it. We denote ancestors and descendants of a supernode v by $\mathcal{A}(v)$ and $\mathcal{D}(v)$, respectively.

3.4 The SuperFw algorithm

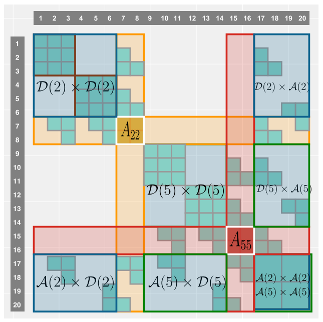
Algorithm 3 The SuperFw algorithm

```
1: $n_s$ := Number of supernodes
2: function SuperFw($G = (V, E)$):
3:     for $k = \{1, 2 \ldots, n_s\}$ do:
   Diagonal Update
4:         $A(k,k) \leftarrow$ Floyd-Warshall($A(k,k)$)
   Panel Update
5:         for $i \in \mathcal{A}(k) \cup \mathcal{D}(k)$ do
6:             $A(i,k) \leftarrow A(i,k) \oplus A(i,k) \otimes A(k,k)$
7:             $A(k,i) \leftarrow A(k,i) \oplus A(k,k) \otimes A(k,i)$
   MinPlus Outer Product
8:         for $(i,j) \in \{\mathcal{A}(k) \cup \mathcal{D}(k)\} \times \{\mathcal{A}(k) \cup \mathcal{D}(k)\}$ do:
9:             $A(i,j) \leftarrow A(i,j) \oplus A(i,k) \otimes A(k,j)$
```

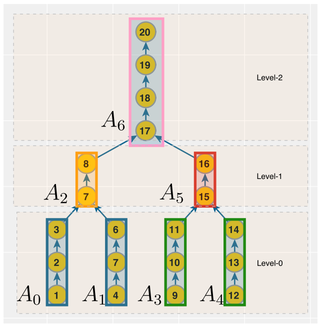
Algorithm 3 describes the sequential SuperFw algorithm. At a high-level, it performs Floyd-Warshall iterations on the supernodal matrix. However, this algorithm exhibits some subtle behaviors.

Consider the elimination of a supernode v. Its elimination only requires updating blocks corresponding to $\mathcal{A}(v)$, $\mathcal{D}(v)$, and v itself. The DiagUpdate and PanelUpdate are performed in place. The regions updated in the OuterUpdate step can be divided into four subsets:

(1) $\mathcal{D}(v) \times \mathcal{D}(v)$ (top-left region, relative to a),
(2) $\mathcal{D}(v) \times \mathcal{A}(v)$ (top-right region),


$\mathcal{D}(2)\times\mathcal{D}(2)$
$\mathcal{D}(2)\times\mathcal{A}(2)$
A_{22}
$\mathcal{D}(5)\times\mathcal{D}(5)$
$\mathcal{D}(5)\times\mathcal{A}(5)$
A_{55}
$\mathcal{A}(2)\times\mathcal{D}(2)$
$\mathcal{A}(5)\times\mathcal{D}(5)$
$\mathcal{A}(2)\times\mathcal{A}(2)$
$\mathcal{A}(5)\times\mathcal{A}(5)$

(a) Parallel elimination of supernodes 2 and 5



(b) The eTree view

Figure 5: Parallel OuterUpdate of two cousin supernodes. The regions that updated in the OuterUpdate step of elimination of second and fifth supernode are highlighted under blue and green transparency, respectively.

(3) $\mathcal{A}(v)\times\mathcal{D}(v)$ (bottom-left region), and
(4) $\mathcal{A}(v)\times\mathcal{A}(v)$ (bottom-right region).

Theese different regions associated with the elimination of a node appear in Fig. 5a. In conventional Cholesky factorization, we only need to update $\mathcal{A}(v)\times\mathcal{A}(v)$, also called the trailing matrix. This operation involves only sparse blocks, and it updates the supernodal block sparse matrix. The OuterUpdate on $\mathcal{D}(v)\times\mathcal{D}(v)$, $\mathcal{D}(v)\times\mathcal{A}(v)$ and $\mathcal{A}(v)\times\mathcal{D}(v)$ directly updates the distance matrix, which is dense. At the end of the computation, the supernodal matrix contains the *semiring equivalent* of Cholesky factors and the dense distance matrix contains final pairwise lengths of all shortest paths.

3.5 Parallel SuperFw algorithm

Recall from the baseline BlockedFw algorithm that, in the k-th OuterUpdate step, all the updates on all the A_{ij} blocks could be performed in parallel. Relative to that available parallelism, the enhancement in SuperFw comes from exploiting the eTree.

eTree guided scheduling and parallelism. We say a supernode a is a cousin of a supernode b if $\mathcal{D}(a)\cap\mathcal{D}(b)=\emptyset$. For instance any two leaf nodes in the eTree are cousins. The DiagUpdate and PanelUpdate of any two cousin nodes can be done in parallel as they operate on distinct regions of the matrix. Next, consider the dependencies in the OuterUpdate step of two cousins. Recall the four sets of blocks updated in the OuterUpdate: $\mathcal{D}(v)\times\mathcal{D}(v)$, $\mathcal{D}(v)\times\mathcal{A}(v)$, $\mathcal{A}(v)\times\mathcal{D}(v)$, and $\mathcal{A}(v)\times\mathcal{A}(v)$. Any block $A(i,j)$ in the first three subsets will have at least one of i and j in $\mathcal{D}(v)$. If two supernodes v and u are cousins, then by definition $\mathcal{D}(v)\cap\mathcal{D}(u)=\emptyset$. Therefore, the first three subsets of OuterUpdate of two cousin supernodes are disjoint and can be updated concurrently. But $\mathcal{A}(v)\times\mathcal{A}(v)$ and $\mathcal{A}(u)\times\mathcal{A}(u)$ can have some common blocks. Therefore, those blocks are updated sequentially.

To expose maximum parallelism, we perform a bottom-up topological sort of the eTree, which partitions the eTree into levels as shown in Fig. 5b. Since all the supernodes in the given level are cousins to one other, their elimination can be done in parallel. We refer to such eTree-guided scheduling as eTree parallelism.

4 ASYMPTOTIC ANALYSIS

Table 2: Asymptotic work(W), depth(D) and concurrency(C)

Algorithm	$W(n)^{\dagger}$	$D(n)^{\dagger}$	$C(n)^{\ddagger}$
BlockedFw	$O(n^3)$	$O(n)$	$O(n^2)$
SuperFw	$O(n^2\lvert S\rvert)$	$O(\lvert S\rvert\log^2 n)$	$O\left(\frac{n^2}{\log^2 n}\right)$
Dijkstra	$O(n^2\log n+nm)$	$O(n\log n+m)$	$O(n)$
PathDoubling [40]	$O(n^3)$	$O(\log n)$	$O\left(\frac{n^3}{\log n}\right)$

$^{\diamond}$ n: #vertices, m: #edges, $|S|$: size of the top level separator.
† lower is better. ‡ higher is better.

We use the work-depth model[4] to quantify the asymptotic sequential work and the available parallelism for different algorithms. The work is the total number of operations performed and the depth is the length of the longest sequential chain of data dependencies. Formally, if $T(n,p)$ denotes the time of execution of a parallel algorithm on p processors for a problem of size n, then work $W(n)$ and depth $D(n)$ are defined as follows:

$$W(n)=T(n,p=1) \tag{1}$$
$$D(n)=\lim_{p\to\infty}T(n,p) \tag{2}$$

Using $W(n)$ and $D(n)$, the average available parallelism, or concurrency $C(n)$, is defined as

$$C(n)=\frac{W(n)}{D(n)}.$$

The concurrency $C(n)$ indicates the average number of processors an algorithm can fruitfully utilize.

We assume a parallel random access memory (PRAM) model [1] of parallel execution that supports concurrent read exclusive write (CREW). In this model, all processors can access a memory location simultaneously, and only one processor can write at a location at a time. In terms of their depth costs, performing $x_i\leftarrow\alpha y_i+\beta$ has a depth of $O(1)$; and reduction operation $y\leftarrow\sum_{i=1:n}x_i$ has a depth of $O(\log n)$.

4.1 Asymptotic Work

If $|S|$ is the size of the top-level separator of graph G with n vertices, then the cost of running SuperFw is $O(n^2|S|)$. The asymptotic cost reduction from $O(n^3)$ to $O(n^2|S|)$ comes from the ND reordering. The asymptotic cost of ND reordering and other equivalent reordering schemes can be found elsewhere as well [26]. Given the algebraic equivalence of SuperFw with sparse Gaussian elimination (Cholesky for the symmetric or undirected case), its costs are the same; here, we sketch the derivation of that cost and its implication for SuperFw below.

Let $V = \{C_1 \cup S \cup C_2\}$ be a nested dissection vertex partitioning of the graph $G = (V,E)$. Let $|S|$, $|C_1|$, and $|C_2|$ denote the number of vertices in each set. In the following discussion, we assume the following. First, the separator is small, i.e., $|S| \ll n$ and the partition is balanced, i.e., $|C_1| = |C_2|$. Therefore,

$$|C_1| = |C_2| \approx n/2.$$

Further we assume that size of separators as a function of number of vertices in the graph denoted by $S(n)$, is monotonic, i.e.:

$$n_1 > n_2 \implies S(n_1) > S(n_2).$$

Consider the elimination of the separator S, which is the last iteration of SuperFw. It involves three steps: DiagUpdate, PanelUpdate and OuterUpdate. The cost of DiagUpdate is equal to the cost of running Floyd-Warshall on a graph with $|S|$ vertices, or $O(|S|^3)$. The PanelUpdate step involves Min-Plus matrix multiplication of the two separator panels of size $|S| \times (n - |S|))$ with a $|S| \times |S|$ matrix; thus, its cost is $|S|^2 \times (n - |S|))$. The OuterUpdate step involves computing outer product of the two panels; $(n - |S|) \times |S|$ and $|S| \times (n - |S|)$; thus, its cost is $(n - |S|)^2 \times |S|$. Since $|S| \ll n$, the outer product cost will dominate and so the total cost of elimination of the top level separator is given by

$$W^0(n) \approx n^2 S(n).$$

Here zero denotes the level of separator from the top, i.e., root has a level zero and leaves have level $h-1$, where h is the height of the separator tree.

Now consider the elimination of the two first level separators in the eTree. Again, OuterUpdate dominates the cost of elimination. Recalling our assumption that the partitions are approximately balanced, then OuterUpdate involves computing outer product whose dimensions are $(n/2 + |S(n)/2|)) \times |S(n/2)|$ and $|S(n/2)| \times (n/2 + |S(n)/2|)$, so that the cost of elimination of the two first level separator is

$$W^1(n) \approx 2\left(\frac{n}{2}\right)^2 S\left(\frac{n}{2}\right) = \frac{n^2}{2} S\left(\frac{n}{2}\right).$$

Similarly, the total cost of elimination of all the i-th level separator is given by

$$W^i(n) \approx 2^i \left(\frac{n}{2^i}\right)^2 S\left(\frac{n}{2^i}\right) = \frac{n^2}{2^i} S\left(\frac{n}{2^i}\right).$$

There are approximately $\log n$ levels of the separator. Therefore, summing over all levels yields the final cost,

$$W(n) \approx \sum_{i=0}^{\log n} \frac{n^2}{2^i} S\left(\frac{n}{2^i}\right) = n^2 S(n) \sum_{i=0}^{\log n} \frac{S(n/2^i)}{2^i S(n)}.$$

Since $S(n)$ is monotonic, so $\frac{S(n/2^i)}{S(n)} \leq 2$, the coefficient of $n^2 S(n)$ in $W(n)$ is $\leq \sum_{i=0}^{\log n} 1/2^i \leq 1$. Hence, the total work of SuperFw on a graph with n vertices is given by

$$W(n) = n^2 |S|. \tag{3}$$

4.2 Asymptotic Depth

The depth of the baseline Floyd-Warshall algorithm is $O(n)$ since each of n vertices of the graph is eliminated sequentially. Within the elimination of a single vertex, DiagUpdate, PanelUpdate and OuterUpdate will each have depth $O(1)$ using $O(n^2)$ processors. If the SuperFw does use the etree parallelism and perform sequential elimination of vertex, then it will have the same depth $O(n)$ as the baseline Floyd-Warshall.

To calculate the depth of SuperFw with etree parallelism, we consider the elimination of top-level separator first. The elimination of top-level separator uses the blocked Floyd-Warshall algorithm, thus its depth is $S(n)$. In the first level, we eliminate two separators each of size $\approx S(n/2)$ in parallel.

For the elimination of two separators in the first level, we can perform DiagUpdate, PanelUpdate, and OuterUpdate involving regions of $\mathcal{A} \times \mathcal{D}$, $\mathcal{D} \times \mathcal{A}$ and $\mathcal{D} \times \mathcal{D}$, in parallel. However, for performing OuterUpdate involving $\mathcal{A} \times \mathcal{A}$, we may update the same block from OuterUpdate step of either separator. In general, in the elimination of separators of the i-th level, multiple processes might try to update the same block in $\mathcal{A} \times \mathcal{A}$. Notice that the update is a reduction operation, hence if p process try to update the same block, we can perform the update using tree-reduction with a depth of $\log_2 p - 1 = O(\log p)$. Since any block in $\mathcal{A} \times \mathcal{A}$ will be updated by at most $O(2^i)$ processors at level i, hence the depth of updating any block in OuterUpdate of update of any block in the $\mathcal{A} \times \mathcal{A}$ will be at most $\log(2^i) = i$. Therefore, the depth of performing elimination in the i-the level of the separator tree will be $iS(n/2^i)$. So the total depth of performing SuperFw is given by :

$$D(n) = \sum_{i=0}^{\log n} iS(n/2^i) \leq \sum_{i=0}^{\log n} \log n S(n) = S(n) \log^2 n \tag{4}$$

Therefore, the depth of performing SuperFw with etree parallelism is $O(S(n)\log^2 n)$ or just $O(|S|\log^2 n)$. Verily, we can express the available parallelism or concurrency as :

$$C(n) = \frac{W(n)}{D(n)} = O\left(\frac{n^2}{\log^2 n}\right) \tag{5}$$

4.3 Discussion

In Table 2, we summarize the work, depth, and concurrency of SuperFw and BlockedFw, along with two notable general Apsp algorithms a) Dijkstra which is work optimal for

Table 3: Test sparse matrices used in experiments

Name	Source	n	$\frac{nnz}{n}$	$\frac{n}{\lvert S\rvert}$
USpowerGrid	Power network	4.9e3	2.66	6.2e2
OPF_6000	Power network	2.9e4	9.1	1.4e3
nd6k	3D	1.8e4	383	5.8
oilpan	structural	7.3e4	29.1	1.7e2
finan512	Optimization	7.5e4	7.9	1.5e3
net4-1	Optimization	8.8e4	28	2.9e3
c-42	Optimization	1.0e4	10.58	1.5e2
c-69	Optimization	6.7e4	9.24	2.0e2
lpl1	Optimization	3.2e4	10	4.8e2
onera_dual	Structural	8.5e4	4.9	1.5e2
email-Enron	SNAP	3.7e4	9.9	52
delaunay_n14	DIMACS10	1.6e4	5.99	1.7e2
delaunay_n16	DIMACS10	6.5e4	5.99	1.7e2
fe_sphere	DIMACS10	1.6e4	5.99	8.5e1
luxembourg_osm	DIMACS10	1.1e5	2.1	6.7e3
fe_tooth	DIMACS10	7.8e4	11.6	88
wing	DIMACS10	6.2e4	3.9	1.0e2
t60k	DIMACS10	6.0e4	3.0	1.1e3
G67	Random	1e4	4	5.0e1
EB_8192_256	Barabasi - Albert	8.1e3	256	2.5e0
EB_16384_64	Barabasi - Albert	1.63e4	64	2.6e0
rgg2d_14	Random Geometric	1.63e4	128.17	1.6e1
rgg3d_14	Random Geometric	1.63e4	910	2.57
hypercube_14	hypercube Graph	1.6e4	28	5.0e0

sparse graphs but offers only $O(n)$ concurrency; and b) Path-Doubling is a known theoretical variant of Floyd-Warshall algorithm with best known parallel complexity. Per Eq. (3), SuperFw lowers the work complexity of Floyd-Warshall by a factor of $O\left(\frac{n}{S(n)}\right)$, while also reducing the asymptotic depth by $O\left(\frac{n}{S(n)\log^2 n}\right)$. Hence SuperFw improves the asymptotic work complexity with little exacerbation of available parallelism. In contrast, Dijkstra's algorithm has a lower asymptotic work complexity, but it has a concurrency of $O(n)$. Further, Dijkstra uses the priority queue data structure, which may not effectively utilize modern architectural features such as on chip cache memory and large SIMD units found on modern processors.

Per Eq. (3), a small vertex separator implies an asymptotic reduction in the cost of SuperFw relative to the naïve (dense) cost of $O\left(n^3\right)$. The class of graphs with small separators largely falls into the category of *geometric* graphs, as well as additional classes of graphs derived from geometric graphs.Informally, a geometric graph is one that can be embedded into a $d \ll n$-dimensional grid [10]. For a d-dimensional grid graph, there exists a separator of size $O\left(n^{1-\frac{1}{d}}\right)$. The best-known example is a planar graph, which has a separator of size $O\left(\sqrt{n}\right)$ [27]. Additionally, there are graphs with $O(n)$ separators also known as expander graphs [20]. Many random graphs tend to become expanders as number of edges increase.

5 RESULTS

We have implemented a shared memory multicore version of the SuperFw algorithm using OpenMP. The aim of our evaluation is to quantify the impact of each algorithmic techniques from sparse Cholesky on Apsp, though these techniques are not specific to shared memory; Section 6 briefly discusses candidate implementations for other programming models and frameworks, such as distributed memory. We present performance of SuperFw for both the sequential and multithreaded implementation on different datasets listed in Table 3.

5.1 Experimental Setup

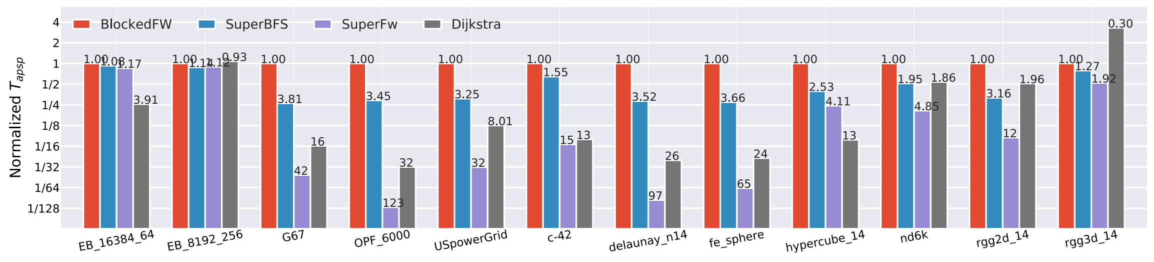
In this section, we present the details of the Test Bed, Baselines and the test graphs that we use for experiments.

5.1.1 *Test Bed.* We conducted our experiment in a shared memory system that contains 32 cores as a dual-socket 16-core Intel E5-2698 v3 "Haswell" processors. Each socket has 40-MB shared L3 cache. It has a total of 128 GB DDR4 2133 MHz memory arranged in four 16 GB DIMMs per socket. Each core can support two hyperthreads, thus 64 threads in total.

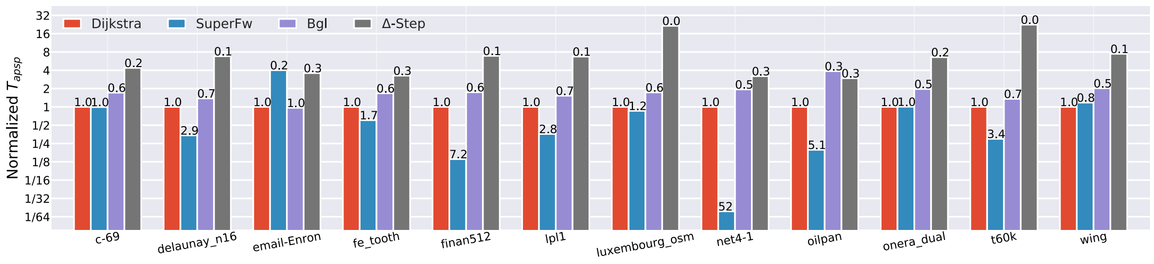
5.1.2 *Competing Apsp implementations .* We compare the **SuperFw** implementation that uses the three optimizations (a) ND ordering (b) Supernodal structure and (c) elimination tree parallelism with the following three baselines.

- **BlockedFw** : this is an efficient multithreaded implementation of Algorithm 2 using OpenMP. This implementation does not exploit the sparsity of the graph. This would perform n^3 operations.
- **SuperBfs**: This algorithm does not use the optimal ND ordering. However, it does perform symbolic factorization and set-up supernodal data structure. We perform BFS from the vertex-0 and use the order in which vertices were discovered as the ordering, to ensure that initial ordering of the matrix has some structure, so supernodal approach might still exploit the sparsity. This would perform $O\left(n^3\right)$ operations, but coefficients might be much smaller than BlockedFw
- **Dijkstra**: This algorithm performs a single-pair shortest path from all the vertices. It has the lowest asymptotic complexity of all the methods considered herein, and is the core of Johnson's algorithm when there are no negative edge weights. (That is, Johnson's algorithm uses Dijkstra as a subroutine and would be more expensive than Dijkstra for graphs with only positive edge weights.)
- **BoostDijkstra**: Apsp implementation using off-the-shelf implementation of Dijkstra's algorithm from popular Boost Graph Library (BGL) [37].[2]

[2]BGL also provides an implementation of both Floyd-Warshall and Johnson's algorithm, but their performance is not competitive to BoostDijkstra, and thus not considered.

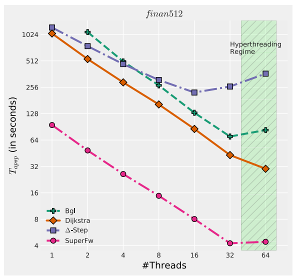


(a) Small Graphs

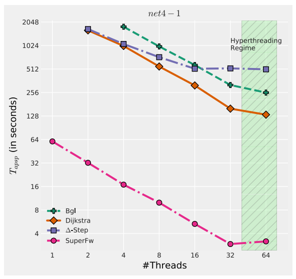


(b) Large Graphs

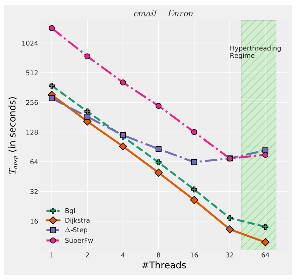
Figure 6: Performance of different multithreaded Apsp algorithms for small and large graphs. Each bar shows normalized execution time for a given algorithm and graph. Time is normalized over the baseline BlockedFw and Dijkstra algorithm for small and large graphs respectively. Each text label over each bar denotes the speed-up over the reference algorithm.



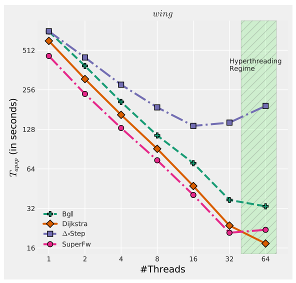
(a) finan512



(b) net4-1



(c) email-Enron



(d) wing

Figure 7: Shared-memory scaling of different Apsp implementations for large graphs on a intel "Haswell" dual-socket system with 32 physical cores.

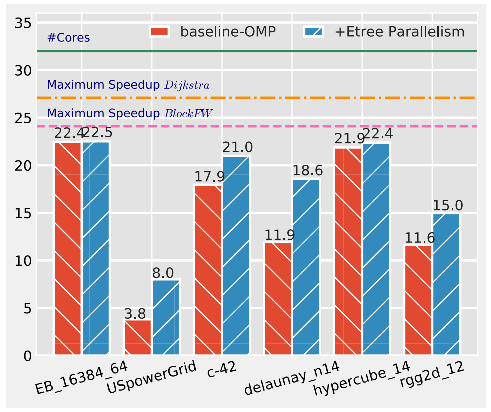


Figure 8: Impact of etree parallelism (Section 3.5) on the scaling of SuperFw algorithm on 32 cores (see Section 5.2.3)

- **Δ-Step**: We use the parallel Δ-stepping variant of Dijkstra's algorithm [30] for computing the single-source-shortest path in Johnson's algorithm. We also use the parallel Δ-stepping algorithm from the Galois Graph library [32]. The Δ-stepping requires tuning the Δ parameter for each graph. Our Δ-Step-based Apsp is autotuned, i.e., it tries different values of Δ of first few SSSP calls and picks the best Δ for rest of the execution.

The core of all three Floyd-Warshall algorithms BlockedFw, SuperBfs and SuperFw use the same semiring double precision matrix-multiplication kernel SemiringGemm. The SemiringGemm kernel achieves 10.2 Gflop/sec per core which is

28% of the theoretical peak of 36.8 Gflop/sec (18.4 Gflop without FMA instructions) per core. The BlockedFw achieves a maximum of 244 Gflop/sec on 32 cores. The operand sizes can vary significantly in SuperFw, thus per core flop rate varies between graphs and ranges from 7.6-9.4 Gflop/sec.

5.1.3 Test Graphs. We used graphs from three different categories (a) Real world Graphs (b) Graphs from DIMACS competition and (c) Random graph generators with different parameters. The details of the set of matrices such as the size and density used for experiments are listed in Table 3. The graph luxembourg_osm has 114k vertices, which requires 105GB of memory to store the distance matrix, is the largest graph we could successfully try. The Djikstra's and Δ-Step algorithms works on graphs with positive edge weights, so we modify the adjacency matrices from real world and synthetic to have only positive entries.

5.1.4 Pre-processing overhead. The worst-case pre-processing cost is 18% of the multithreaded SuperFw execution time, even though pre-processing (i.e. graph partitioning via Metis) is not multithreaded. In sequential or single-node case, the pre-processing step is not a a bottleneck for even for sparse Cholesky (which is a $O(S^3)$ the operation, whereas SuperFw performs $O(n^2S)$ operations), thus many efforts are towards improving the performance of the numerical factorization step. In the subsequent performance and scalability analysis, the time shown does not include the pre-processing costs.

5.2 Observations

5.2.1 Small Graphs. For small graphs, we compare the performance of SuperFw against BlockedFw, SuperBfs and Dijkstra, shown in Fig. 6a. The SuperFw is faster over BlockedFw by upto 123×, and it represents the impact of all the optimizations combined. The SuperBfs is faster over BlockedFw by upto 3.9×. We highlight the following three key observations.

- ***Impact of ND ordering:*** In the case of the test matrices, delauney_n14, OPF_6000, and fe_sphere, ND ordering yielded significant benefit because of smaller separators. However, there are many real world cases with smaller separators that can benefit from SuperFw.
- ***Impact of Supernodal structure:*** The asymptotic cost of SuperBfs is still $O(n^3)$, but it does exploit sparsity to some extent, unlike blockFW. As in Fig. 6a, this offers an advantage of 1-3.9× on many real world and synthetic graphs. We also observe that in the case of hypercube_14 with $\frac{n}{\log n}$ separator, reordering cannot reduce the asymptotic cost. Yet, by using a supernodal data structure, we get a performance improvement of 4.1x speedup over BlockedFw.
- ***Adversarial Cases:*** Finally, there are synthetic graphs like extended_barbasi, where neither ND ordering nor supernodal structure offer any improvement. More generally, there are graph classes, like expander graphs, that are sparse yet well connected. Such graphs won't have good separators and we would not expect SuperFw to provide any advantage over BlockedFw. There are number of random graph generators, such as Erdos-Renyi, Watts-Strogatz, that have similar properties for graphs with large number of vertices.

We expect that the performance gap between BlockedFw and SuperFw will increase with increasing in problem size due to asymptotic difference in the time-complexity, whereas performance gap between BlockedFw and SuperBfs will remain similar for larger graphs.

5.2.2 Large Graphs. For large graphs, we compare the performance of SuperFw with Dijkstra, BoostDijkstra, and Δ-Step and leave out $O(n^3)$ algorithms shown in Fig. 6b. The SuperFw is faster by 0.2-52× than the Dijkstra's. We expect that for larger graphs Dijkstra will outperform SuperFw, nevertheless, SuperFw is competitive to Dijkstra for planar graphs of sizes on the order of 100k vertices, e.g., luxembourg_osm. The BoostDijkstra and Dijkstra are algorithmically very similar, yet BoostDijkstra is often slower than our implementation of Dijkstra. This difference mainly stems from BoostDijkstra's adjacency list data structure for storing graphs vs compressed-sparse-row storage used by Dijkstra. The Δ-Step is neither work-optimal and nor scalable, thus not competitive to either Dijkstra or SuperFw.

5.2.3 Scalability.

- **Impact of eTree Parallelism:** In Fig. 8, we show the relative performance of two implementations of SuperFw, with and without eTree parallelism on cores over sequential performance. eTree parallelism can improve the scalability of SuperFw by 2×. The impact of eTree parallelism is more visible for small-graphs, e.g., USPowerGrid, where SuperFw performs very little per-iteration work. For larger graphs eTree parallelism has a little impact of performance as they already enough parallelism in each iteration. Hence eTree-parallelism is essential for strong scaling.
- **Scalability of different Apsp implementations:** The scalability of Dijkstra, BoostDijkstra, Δ-Step, and SuperFw are compared in Figs. 7a to 7d. Our SuperFw implementation scales linearly up to 32 threads (= number of physical cores) achieving a parallel efficiency of 74%. The Dijkstra and BoostDijkstra can effectively use hyperthreading to hide the latencies of the priority queue data structure, thus they can scale to 64 threads. The Δ-Step method only parallelizes each SSSP call, thus it requires significantly more inter-thread synchronizations and scales poorly compared to the other three implementations.

6 RELATED WORK

This work builds upon principles from several different areas including semiring algebra, graph algorithms and sparse direct solvers.

A number of other problems are equivalent to Apsp, e.g. metricity, minimum-weight triangle, second shortest path, etc. [43, 44]. While Apsp is the semiring equivalent of matrix inversion, no no truly subcubic algorithm (i.e., equivalent of Strassen's algorithm) for Apsp is known. The best known complexity of Apsp for the dense case is $O\left(\frac{n^3}{n^{o1}}\right)$ [43], and $O\left(\frac{mn}{\log n}\right)$ for sparse graphs [7]; for the parallel case, it is $O(\log n)$ [40].

The equivalence between finding the shortest path and solving a system of linear equations goes back to the work of Carre [3, 6]. He gave many interpretation of linear algebra operations, including LU factorization and Sherman-Morrison Woodbury formula for graph updates. Modern treatments of this subject can also be found elsewhere [16, 28].

The method of nested dissection (ND) was discovered by George [13] for solving linear system of equations from finite element meshes. Its generalization by Lipton et al. [26], and the planar separator theorem [27, 39] has had a large impact on a number of graph and sparse linear algebra algorithms. In particular, several algorithms for path problems on planar graphs are based on ND and the planar separator theorem [8, 11, 42].

Lastly, sparse direct solvers have been studied in great detail in the context of parallel computing. Depending on scheduling, there are other variants namely, left-looking, right-looking, multifrontal, and Crout's variant. The effect of different scheduling strategies on performance can be found at [19, 34]. The proposed SuperFw closely resembles the right-looking variant. Similarly, a number of works have focused on improving scalability on accelerators such as GPU [15, 23–25, 29, 36, 41, 46] and distributed memory [2, 18, 35, 45]. Most distributed algorithms rely on some form eTree parallelism for reducing communication and data distribution [18, 35].

7 CONCLUSION

This paper is the first practical demonstration of how to exploit the algebraic structure of the all-pairs shortest paths problem to create a parallel APSP algorithm for sparse graphs, applying all of the machinery we know from sparse direct solvers for linear systems. Although the observation about the similarity of linear solver and shortest path is known, ours is the first attempt to create a practical implementation and conduct an empirical analysis. We believe that the algebraic interpretation of APSP is important to explore, and our study is just the first of many that we or others could carry out, now that we have done this one. In particular, consider linear systems. In that setting, there is a rich "hierarchy" of methods that trade-off generality and robustness for speed and asymptotic optimality, with "dense LU" at one end and "multigrid" at the other. Sparse Cholesky/LU is in the middle of that spectrum. For APSP, we do not know yet fully understand what the analogous hierarchy might look like. This study would be just one a series of future studies that could try to fill in the other analogous methods.

8 ACKNOWLEDGEMENTS

This manuscript has been authored by UT-Battelle,LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. This research is partially support by Oak Ridge National Laboratory Director's Research and Development Funding. Additional support comes from the Defense Advanced Research Projects Agency (DARPA) contract FA8750-18-2-0108, under the DARPA MTO Software Defined Hardware program, and the National Science Foundation (NSF) under Award 1710371. The publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).

REFERENCES

[1] Alok Aggarwal, Ashok K Chandra, and Marc Snir. Communication complexity of prams. *Theoretical Computer Science*, 71(1):3–28, 1990.
[2] CC Ashcraft. A taxonamy of distributed dense LU factorization methods. *Engineering Computing and Analysis Technical Report ECA-TR-161, Boeing Computer Services(1991)*, 1991.
[3] Roland C Backhouse and Bernard A Carré. Regular algebra applied to path-finding problems. *IMA Journal of Applied Mathematics*, 15(2):161–186, 1975.
[4] Guy E Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3):85–97, 1996.
[5] Aydın Buluç, John R Gilbert, and Ceren Budak. Solving path problems on the gpu. *Parallel Computing*, 36(5-6):241–253, 2010.
[6] Bernard A Carré. An algebra for network routing problems. *IMA Journal of Applied Mathematics*, 7(3):273–294, 1971.
[7] Timothy M Chan. All-pairs shortest paths for unweighted undirected graphs in o (mn) time. In *Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, pages 514–523. Society for Industrial and Applied Mathematics, 2006.
[8] Hristo Djidjev, Guillaume Chapuis, Rumen Andonov, Sunil Thulasidasan, and Dominique Lavenier. All-pairs shortest path algorithms for planar graph for gpu-accelerated clusters. *Journal of Parallel and Distributed Computing*, 85:91–103, 2015.
[9] Iain S. Duff, A.M. Erisman, and John K. Ried. *Direct methods for sparse matrices.* Oxford University Press, 2nd edition, 2017.
[10] Paul Erdös, Frank Harary, and William T Tutte. On the dimension of a graph. *Mathematika*, 12(2):118–122, 1965.
[11] Greg N Federickson. Fast algorithms for shortest paths in planar graphs, with applications. *SIAM Journal on Computing*, 16(6):1004–1022, 1987.
[12] Jeremy T. Fineman and Eric Robinson. Fundamental graph algorithms. In Jeremy Kepner and John Gilbert, editors, *Graph Algorithms in*

the Language of Linear Algebra, chapter 5, pages 45–58. Society of Industrial and Applied Mathematics, Philadelphia, PA, USA, 2011.

[13] A. George. Nested dissection of a regular finite element mesh. *SIAM Journal on Numerical Analysis*, 10(2):345–363, 1973.

[14] Alan George. Nested dissection of a regular finite element mesh. *SIAM Journal on Numerical Analysis*, 10(2), 1973.

[15] T. George, V. Saxena, A. Gupta, A. Singh, and A. Choudjury. Multifrontal factorization of sparse SPD matrices on GPUs. In *Proc. of IEEE International Parallel and Distributed Processing Symposium (IPDPS 2011)*, Anchorage, Alaska, May 16-20 2011.

[16] Michel Gondran and Michel Minoux. *Graphs, dioids and semirings: new models and algorithms*, volume 41. Springer Science & Business Media, 2008.

[17] Kazushige Goto and Robert A Geijn. Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software (TOMS)*, 34(3):12, 2008.

[18] Anshul Gupta, George Karypis, and Vipin Kumar. Highly scalable parallel algorithms for sparse matrix factorization. *IEEE Transactions on Parallel and Distributed Systems*, 8(5):502–520, 1997.

[19] Michael T Heath, Esmond Ng, and Barry W Peyton. Parallel algorithms for sparse linear systems. *SIAM review*, 33(3):420–460, 1991.

[20] Shlomo Hoory, Nathan Linial, and Avi Wigderson. Expander graphs and their applications. *Bulletin of the American Mathematical Society*, 43(4):439–561, 2006.

[21] Donald B. Johnson. Efficient algorithms for shortest paths in sparse networks. *Journal of the ACM (JACM)*, 24(1):1–13, 1 1977.

[22] George Karypis and Vipin Kumar. A fast and high-quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing (SISC)*, 20(1), 1998.

[23] Kyungjoo Kim and Victor Eijkhout. Scheduling a parallel sparse direct solver to multiple GPUs. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 1401–1408. IEEE, 2013.

[24] G. Krawezik and G. Poole. Accelerating the ANSYS direct sparse solver with GPUs. In *Proc. Symposium on Application Accelerators in High Performance Computing (SAAHPC)*, Urbana-Champaign, IL, NCSA, 2009.

[25] Xavier Lacoste, Mathieu Faverge, George Bosilca, Pierre Ramet, and Samuel Thibault. Taking advantage of hybrid systems for sparse direct solvers via task-based runtimes. In *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, pages 29–38. IEEE, 2014.

[26] Richard J Lipton, Donald J Rose, and Robert Endre Tarjan. Generalized nested dissection. *SIAM journal on numerical analysis*, 16(2):346–358, 1979.

[27] Richard J Lipton and Robert Endre Tarjan. A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics*, 36(2):177–189, 1979.

[28] Grigory L Litvinov. Idempotent/tropical analysis, the hamilton–jacobi and bellman equations. In *Hamilton-Jacobi equations: approximations, numerical analysis and applications*, pages 251–301. Springer, 2013.

[29] R. Lucas, G. Wagenbreth, D. Davis, and R. Grimes. Multifrontal computations on GPUs and their multi-core hosts. In *VECPAR'10: Proc. 9th Intl. Meeting for High Performance Computing for Computational Science*, Berkeley, CA, 2010.

[30] Ulrich Meyer and Peter Sanders. δ-stepping: A parallel single source shortest path algorithm. In *European symposium on algorithms*, pages 393–404. Springer, 1998.

[31] François Pellegrini and Jean Roman. Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In *Proceedings of High-Performance Computing and Networking (HPCN-Europe)*, volume LNCS 1067, 1996.

[32] Keshav Pingali. High-speed graph analytics with the galois system. In *Proceedings of the first workshop on Parallel programming for analytics applications*, pages 41–42. ACM, 2014.

[33] Leon R Planken, Mathijs M de Weerdt, and Roman PJ van der Krogt. Computing all-pairs shortest paths by leveraging low treewidth. *Journal of Artificial Intelligence Research*, 43:353–388, 2012.

[34] Edward Rothberg. Exploiting the memory hierarchy in sequential and parallel sparse Cholesky factorization. Technical report, Stanford University, Department of Computer Science, 1992.

[35] Piyush Sao, Xiaoye S. Li, and Richard Vuduc. A communication-avoiding 3D LU factorization algorithm for sparse matrices. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Vancouver, BC, Canada, May 2018.

[36] Piyush Sao, Richard Vuduc, and Xiaoye Sherry Li. A distributed CPU-GPU sparse direct solver. In *Euro-Par 2014 Parallel Processing*, pages 487–498. Springer International Publishing, 2014.

[37] Jeremy Siek, Andrew Lumsdaine, and Lie-Quan Lee. *The boost graph library: user guide and reference manual.* Addison-Wesley, 2002.

[38] Edgar Solomonik, Aydın Buluç, and James Demmel. Minimizing communication in all-pairs shortest paths. In *Proceedings of the 27th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Boston, MA, USA, 5 2013.

[39] Daniel A Spielmat and Shang-Hua Teng. Spectral partitioning works: Planar graphs and finite element meshes. In *Proceedings of 37th Conference on Foundations of Computer Science*, pages 96–105. IEEE, 1996.

[40] Alexandre Tiskin. All-pairs shortest paths computation in the bsp model. In *International Colloquium on Automata, Languages, and Programming*, pages 178–189. Springer, 2001.

[41] R. Vuduc, A. Chandramowlishwaran, J. Choi, M. Guney, and A. Shringarpure. On the limits of GPU acceleration. In *Proc. of the 2nd USENIX conference on Hot topics in parallelism, HotPar'10*, Berkeley, CA, 2010.

[42] Oren Weimann and Raphael Yuster. Computing the girth of a planar graph in o(n\logn) time. *SIAM Journal on Discrete Mathematics*, 24(2):609–616, 2010.

[43] R Ryan Williams. Faster all-pairs shortest paths via circuit complexity. *SIAM Journal on Computing*, 47(5):1965–1985, 2018.

[44] Virginia Vassilevska Williams and Ryan Williams. Subcubic equivalences between path, matrix and triangle problems. In *2010 IEEE 51st Annual Symposium on Foundations of Computer Science*, pages 645–654. IEEE, 2010.

[45] Ichitaro Yamazaki and Xiaoye S Li. New scheduling strategies and hybrid programming for a parallel right-looking sparse LU factorization algorithm on multicore cluster systems. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 619–630. IEEE, 2012.

[46] C.D. Yu, W. Wang, and D. Pierce. A CPU-GPU hybrid approach for the unsymmetric multifrontal method. *Parallel Computing*, 37:759–770, 2011.