
All-Pairs Shortest Paths in Spark

Charles Y. Zheng and Jingshu Wang

Department of Statistics
Stanford University
Stanford, CA 94305

{snarles, jinshuw}@stanford.edu

Arzav Jain

Department of Computer Science
Stanford University
Stanford, CA 94305

arzavj@cs.stanford.edu

Abstract

We consider the all-pairs-shortest-paths problem for a dense graph in a distributed setting, and study both the resource cost of computing the distance and midpoint matrices, and the resource cost of looking up the shortest path between any pair of nodes. We extend a previously proposed algorithm for distributed APSP to include computation of a midpoint matrix; our algorithm for storing midpoints guarantees that the lookup of a length L path requires at most $O(\log L)$ rounds of all-to-all communication, in contrast to the worst-case $O(L)$ sequential iterations required by existing midpoint storage approaches. In distributed applications where latency between the driver and worker nodes creates a substantial overhead per round of all-to-all communication, our scheme results in much faster lookup times for long paths.

1 Path finding in distributed APSP

The output of the original algorithm is a matrix of shortest distances $S \in \mathbb{R}^{n \times n}$ where each S_{ij} is the shortest distance from node i to j . Here we want to add a path lookup function $\text{FindPath}(i, j)$ which returns for a pair of node (i, j) the shortest path itself from node i to node j .

There are three approaches to consider:

1. Calculate $\text{FindPath}(i, j)$ directly from the distance matrix S
2. Store one midpoint for each (i, j) pair as a matrix $M \in \mathbb{R}^{n \times n}$ in the distributed block APSP algorithm, and then calculate $\text{FindPath}(i, j)$ from M
3. For each (i, j) pair, store two midpoints m_1, m_2 in a three-dimensional array $M \in \mathbb{R}^{n \times n \times 2}$ in the distributed block APSP algorithm, and then calculate $\text{FindPath}(i, j)$ from M .
4. A generalization of approach 3, except storing up to k midpoints m_1, \dots, m_k for each path.

In all four approaches, the lookup function must make a total of $O(L)$ queries, where L is the length of the shortest paths. However, in a distributed setting, we assume the possibility of making up to n queries in parallel, comprising a single *round* of all-to-all communication. For all approaches considered, we therefore define an *iteration* to include at most one round of all-to-all communication. The ordered sequence of nodes in the path is built up iteration by iteration. We begin with the endpoints of the path $\{i, j\}$. In the first iteration, one runs a single query $\text{Lookup}(i, j)$, and the result is either an empty set $\{\}$, or up to k midpoints m_1, \dots, m_k . If the empty set is returned, then we know that the path $\{i, j\}$ is of length one, and the algorithm is terminated. Otherwise, the current set of midpoints is $\{i, m_1, \dots, m_k, j\}$, and the next iteration makes the $k + 1$ queries *in parallel* to $\text{Lookup}(i, m_1)$, $\text{Lookup}(i, m_2)$, etc. Since each query may result in the addition of up to k new midpoints, the number of nodes in the working set may increase by $(k + 1)k$ by the conclusion of the second iteration, hence the working set may be of size $(k + 2)k + 2$ by the conclusion of the second iteration. The maximum number of nodes by the end of the m th iteration given by the recursion

$M_m = M_{m+1} + k(M_m - 1)$, where $M_1 = 2$, hence the maximum number of queries in the m th iteration is given by $Q_m = M_{m-1} + 1$. Of course, both M_m and Q_m are bounded by the length of the path.

Algorithm 1 has to potentially search the entire i th row and j th column per query, and hence has a computational cost of $O(n)$ per query, since $O(L)$ queries have to be made, the total computational cost is $O(nL)$. Algorithms 2-4 all have computational cost $O(k)$ per query, and since $O(L/k)$ queries have to be made, the total computational cost is $O(L)$ for all three algorithms.

If we consider the cost of computing the distance matrix S and matrix of midpoints M , then Algorithm 1 requires the minimal computation and space, Algorithm 2 can be implemented to require about twice the computation and 1.5 times the space of Algorithm 1, and Algorithm 3 and 4 require at least $(k/2 - 1)$ times the space as Algorithm 1. It is clear that Algorithm 2 allows one to invest extra time into the distance matrix calculation stage in order to gain a speedup in the path lookup stage. However, what is the benefit of considering Algorithms 3 and 4? How can we possibly improve the lookup time if the total computational cost per lookup is about the same?

In distributed settings we observe that the clock-time required for a job is dominated by communication and latency rather than computation. Hence, the advantage of Algorithms type 3 and 4 over Algorithm 2 is that the number of parallel iterations may potentially be reduced. We can prove that under the constraint that only one midpoint be stored per path, that the worst-case number of iterations is n . In contrast, we will show in this section that using the third approach, we can guarantee the number of iterations to be at most $\lceil \log_2 n \rceil$ by properly choosing midpoints.

1.1 Criteria for choosing the midpoints

For an (i, j) pair with its shortest path $i \rightarrow k_1 \rightarrow k_2 \rightarrow \dots \rightarrow k_{L-1} \rightarrow j$, define its path length as $l_{ij} = L$. We require the midpoints $m_1 = M_{ij1}$ and $m_2 = M_{ij2}$ to satisfy

1. $m_1, m_2 \in \{i, k_1, k_2, \dots, k_{L-1}, j\}$
2. $l_{ij} = l_{im_1} + l_{m_1m_2} + l_{m_2j}$ (implied by criteria 1)
3. $\max(l_{im_1}, l_{m_1m_2}, l_{m_2j}) \leq \max(l_{ij}/2, 1)$

If M satisfies the above criteria, then the number of iterations in the lookup function `path(i, j)` will be at most $\lceil \log_2 n \rceil$. More details can be found in 1.3.

1.2 Algorithm for updating the midpoints in distributed APSP

The initialization of M is

$$M_{ij1}^{(0)} = M_{ij2}^{(0)} = \begin{cases} i & \text{if } (i \rightarrow j) \in E \text{ or } i = j \\ \star & \text{if } (i \rightarrow j) \notin E \end{cases}$$

where $\star \notin V$ is some symbol to denote an invalid midpoint. To properly update midpoints in our distributed block APSP algorithm, we need to store and update another three-dimensional array $W \in \mathbb{R}^{n \times n \times 3}$ which stores for each (i, j) pair and midpoints (m_1, m_2) the current path lengths l_{im_1} , $l_{m_1m_2}$ and l_{m_2j} . The initialization of W is

$$W_{ij1}^{(0)} = W_{ij2}^{(0)} = \begin{cases} 0 & \text{if } (i \rightarrow j) \in E \text{ or } i = j \\ \infty & \text{if } (i \rightarrow j) \notin E \end{cases}$$

$$W_{ij3}^{(0)} = \begin{cases} 1 & \text{if } (i \rightarrow j) \in E \text{ or } i = j \\ \infty & \text{if } (i \rightarrow j) \notin E \end{cases}$$

For a path $i \rightarrow \dots \rightarrow j$, denote $v_{ij} = (m_1, m_2, l_{im_1}, l_{m_1m_2}, l_{m_2j})$. Then for joining two paths $i \rightarrow \dots \rightarrow k$ and $k \rightarrow \dots \rightarrow j$, we define the following function `MERGE(v_{ik}, v_{kj}, k)` to get v_{ij} for the joint path $i \rightarrow \dots \rightarrow k \rightarrow \dots \rightarrow j$:

Algorithm 1 Merge midpoints of two adjacent paths

```

function MERGE( $v_1 = (m_1, m_2, l_1, l_2, l_3)$ ,  $v_2 = (m_4, m_5, l_4, l_5, l_6)$ ,  $m_3$ )
     $l = \sum_{i=1}^6 l_i$ 
    for  $t = 1, 2, 3, 4$  do
        if  $\sum_{i=1}^t l_i \leq l/2$  &  $\sum_{i=1}^{t+1} l_i \geq l/2$  then
            Break
        end if
    end for
    Return  $v = (m_j, m_{t+1}, \sum_{i=1}^t l_i, l_{t+1}, \sum_{i=t+2}^6 l_i)$ 
end function

```

We call $v = (m_1, m_2, l_1, l_2, l_3)$ as *flat* if

$$\max(l_1, l_2, l_3) \leq \max(1, (l_1 + l_2 + l_3)/2) < \infty$$

Lemma 1.1. *If v_{ik} and v_{kj} are flat and $i \neq k \neq j$, then $v = \text{MERGE}(v_{ik}, v_{kj}, k)$ is also flat.*

Proof. Let $v_{ik} = (m_1, m_2, l_1, l_2, l_3)$, $v_{kj} = (m_4, m_5, l_4, l_5, l_6)$, $k = m_3$ and $l = \sum_{s=1}^6 l_s$. From $i \neq k \neq j$, we have $l \geq 2$.

As v_{ik} and v_{kj} are *flat*, we have $l_1 \leq \max(1, (l_1 + l_2 + l_3)/2) \leq l/2$ and similarly $l_6 \leq l/2$. Thus, there exists $t \in \{1, 2, 3, 4\}$ that both $\sum_{s=1}^t l_s \leq l/2$ and $\sum_{s=1}^{t+1} l_s \geq l/2$ holds. Also $l_{t+1} \leq \max(1, (l_1 + l_2 + l_3)/2, (l_4 + l_5 + l_6)/2) \leq l/2$, thus $v = \text{MERGE}(v_{ik}, v_{kj}, k)$ is also *flat*. \square

We can now modify the original distributed block APSP algorithm to include updating W and M .

Given an $n \times m$ distance matrix A and an $n \times m$ distance matrix B together with the midpoints matrices (W^A, M^A) and (W^B, M^B) , define a minimum operation as $(C, W^C, M^C) = \min_P((A, W^A, M^A), (B, W^B, M^B))$ by

$$C_{ij} = \min(A_{ij}, B_{ij})$$

$$(M_{ij}^C, W_{ij}^C) = \begin{cases} (M_{ij}^A, W_{ij}^A) & \text{if } C_{ij} = A_{ij} \\ (M_{ij}^B, W_{ij}^B) & \text{if } C_{ij} = B_{ij} \end{cases}$$

Given an $n \times k$ distance matrix A and a $k \times m$ distance matrix B together with the midpoints matrices (W^A, M^A) and (W^B, M^B) , define a *min-plus* product $(C, W^C, M^C) = (A, W^A, M^A) \otimes_P (B, W^B, M^B)$ as

$$C_{ij} = \min_{l=1}^k A_{il} + B_{lj}$$

$$(M_{ij}^C, W_{ij}^C) = \begin{cases} (M_{ij}^A, W_{ij}^A) & \text{if } \operatorname{argmin}_l (A_{il} + B_{lj}) = j \\ (M_{ij}^B, W_{ij}^B) & \text{if } \operatorname{argmin}_l (A_{il} + B_{lj}) = i \\ \text{MERGE}((M_{il^*}^A, W_{il^*}^A), (M_{l^*j}^B, W_{l^*j}^B), l^*) & \text{if } l^* = \operatorname{argmin}_l (A_{il} + B_{lj}) \neq i \text{ or } j \end{cases}$$

for $i = 1, \dots, n$ and $j = 1, \dots, m$.

Also, $\text{APSP}_P(A, M^A, W^A)$ is defined as a modified local APSP method for finding the shortest distance matrix together with the desired midpoints and path lengths matrices.

Here, we give a shorthand description of the modified distributed block APSP including updating W and M , without explicitly specifying the Spark operations.

Algorithm 2 Path-Finding Distributed Block APSP (shorthand)

```
function BLOCKAPSPATH(Adjacency matrix  $A$  given as a BlockMatrix with  $\ell$  row blocks
and  $\ell$  column blocks,  $M^{(0)}, W^{(0)}$ )
   $H^{(0)} \leftarrow (A, M^{(0)}, W^{(0)})$ 
  for  $k = 1, \dots, \ell$  do
    [A-step]
     $H^{kk(k)} \leftarrow \text{APSP}_P(H^{kk(k-1)})$ 
    [B-step]
    for  $i = 1, \dots, \ell, j = 1, \dots, \ell$  do in parallel
      if  $i = k$  and  $j \neq k$  then
         $H^{kj(k)} \leftarrow \min_P(H^{kj(k-1)}, H^{kk(k)} \otimes_P H^{kj(k-1)})$ 
      end if
      if  $i \neq k$  and  $j = k$  then
         $H^{ik(k)} \leftarrow \min_P(H^{ik(k-1)}, H^{ik(k-1)} \otimes_P H^{kk(k)})$ 
      end if
    end for
    [C-step]
    for  $i = 1, \dots, \ell, j = 1, \dots, \ell$  do in parallel
      if  $i \neq k$  and  $j \neq k$  then
         $H^{ij(k)} \leftarrow \min_P(H^{ij(k-1)}, H^{ik(k)} \otimes_P H^{kj(k)})$ 
      end if
    end for
    [D-step]
    if  $k \equiv 0 \pmod q$  then
      Checkpoint  $H^{(k)}$ 
    end if
  end for
  Return  $(S, M, W) = H^{(\ell)}$ , the APSP result tuple
end function
```

1.3 The path lookup function

After obtaining the the midpoints three-dimensional array M , we can efficiently lookup the shortest path of an (i, j) pair of nodes. The lookup function returns a vector of all the other nodes in the path in order except for the starting node i . Note that if there are multiple shortest paths, the algorithm is only able to find one of them.

Algorithm 3 Lookup the path from one node to another

```
function FINDPATH( $i, j$ )
  if  $i == j$  then
    Return NULL
  end if
  if  $M_{ij1} == M_{ij2}$  then
    Return  $j$ 
  end if
  Return (FindPath( $i, M_{ij1}$ ), FindPath( $M_{ij1}, M_{ij2}$ ), FindPath( $M_{ij2}, j$ ))
end function
```

As $\max(l_{iM_{ij1}}, l_{M_{ij1}M_{ij2}}, l_{M_{ij2}j}) \leq \max(1, l_{ij}/2)$, the recursion depth of the above algorithm is upper bounded by $\lceil \log_2 l_{ij} \rceil$, which is at most $\lceil \log_2 n \rceil$ for any node pair in the graph.