All-Pairs Shortest Paths in Spark

Charles Y. Zheng and Jinshu 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@stanford.edu

Abstract

We propose an algorithm for the All-Pairs-Shortest-Paths (APSP) problem suitable for implementation in Spark, and analyze its performance. We begin by considering distributed Floyd-Warshall, as proposed by Kumar and Singh (1991). Distributed Floyd-Warshall has asymptotically optimal scaling and can be implemented in Spark by using BlockMatrix to represent the APSP distance matrix. However, we observe that its implementation in Spark suffers from poor performance for medium-sized problems due the large number of global updates of the APSP distance matrix required for the algorithm. Since the lineage of the algorithm grows with the number of vertices n, it becomes necessary to use a proportional number of checkpoints which further impacts the efficiency of the algorithm. This motivates the consideration of an algorithm for APSP which requires fewer global update steps. We adapt an approach by Solomonik et al. (2013) based on the "divide and conquer" algorithm for APSP. Our algorithm reduces the number of global updates by a factor of b, where the block size b determines the amount of computation done in each iteration. By adjusting the block size b we obtain a favorable tradeoff between checkpointing costs and computation cost per iteration, resulting in far improved performance compared to Distributed Floyd-Warshall.

1 Summary

For the convenience of the grader we present an overview of our approach and our results. The rest of the paper gives a detailed explanation of the results in this section.

1.1 Problem Specification

Let $G = (V, E)$ be a graph with n vertices. Assume the input is given in the form of the adjacency matrix A of the graph stored as a `BlockMatrix` with equally sized square blocks. Specifically define the adjacency matrix A as a square matrix with dimension $n = |V|$, and entries

$$A_{ij} = \begin{cases} w_{i,j} & \text{if } (i \to j) \in E \\ 0 & \text{if } i = j \\ \infty & \text{if } (i \to j) \notin E \end{cases}$$

Let b be the size of the block, and let $n = b\ell$ so that ℓ^2 is the number of blocks. Write

$$A = \begin{pmatrix} A^{11} & A^{12} & \cdots & A^{1\ell} \\ A^{21} & A^{22} & \cdots & A^{2\ell} \\ \vdots & \vdots & \ddots & \ddots \\ A^{\ell 1} & A^{\ell 2} & \cdots & A^{\ell\ell} \end{pmatrix}$$

so that A^{ij} is the (i, j)th block in the `BlockMatrix`.

The output is given by the APSP distance matrix S, where

$$S_{ij} = \begin{cases} \text{weight of shortest path} & \text{if there exists a path } i \to j \\ 0 & \text{if } i = j \\ \infty & \text{if there is no path } i \to j \end{cases}$$

Let S be stored as a `BlockMatrix` with the same dimensions and block sizes as A, so that

$$S = \begin{pmatrix} S^{11} & S^{12} & \cdots & S^{1\ell} \\ S^{21} & S^{22} & \cdots & S^{2\ell} \\ \vdots & \vdots & \ddots & \vdots \\ S^{\ell 1} & S^{\ell 2} & \cdots & S^{\ell\ell} \end{pmatrix}$$

1.2 Scaling

We consider the scaling n possibly larger than the memory per worker. We do *not* assume a sparse graph G, so the number of edges can scale as $E \sim n^2$.

Let p be the number of workers. We assume the memory of each worker is fixed. Therefore b must be constant since we assume each block must fit in memory, and p must increase as n increases.

1.3 Notation

Given an $n \times m$ matrix A and an $n \times m$ matrix B, define the entrywise minimum $C = \min(A, B)$ by

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

Meanwhile, given an $n \times k$ matrix A and a $k \times m$ matrix B, define the *min-plus* product $C = A \otimes B$ by

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

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

Define APSP(A) as the all-pairs-shortest-distance matrix for adjacency matrix A. For example, APSP(A) is obtained by running the Floyd-Warshall algorithm on A.

1.4 Algorithm

The algorithm consists of an *outer loop* with $\ell = n/b$ iterations. Each iteration culminates in the global update of the `BlockMatrix` S containing the intermediate values of the APSP distance matrix. Each outer loop iteration involves the execution of three distributed subroutines in sequence, called the A-step, the B-step and C-step. In addition, after every q iterations, the `BlockMatrix` S is checkpointed.

We first give an shorthand description of the algorithm without explicitly specifying the Spark operations used in each step or what data needs to be communicated at each step. In the analysis, we expand each step to describe the specific Spark operations needed, including the broadcasts, joins, etc. needed to transfer the necessary data across workers. Do note that $S^{(0)}, S^{(1)}, \ldots, S^{(\ell)}$ refer to the sequence of `BlockMatrix` objects storing the results of each iteration.

Algorithm 1 Distributed Block APSP (shorthand)

```
function BlockAPSP(Adjacency matrix $A$ given as a BlockMatrix with $\ell$ row blocks and $\ell$ column blocks)
    $S^{(0)} \leftarrow A$
    for $k = 1, \dots, \ell$ do
        [A-step]
        $S^{kk(k)} \leftarrow \text{APSP}(S^{kk(k-1)})$
        [B-step]
        for $i = 1, \dots, \ell,\ j = 1, \dots, \ell$ do in parallel
            if $i = k$ and $j \neq k$ then
                $S^{kj(k)} \leftarrow \min(S^{kj(k-1)}, S^{kk(k)} \otimes S^{kj(k-1)})$
            end if
            if $i \neq k$ and $j = k$ then
                $S^{ik(k)} \leftarrow \min(S^{ik(k-1)}, S^{ik(k)} \otimes S^{kk(k)})$
            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
                $S^{ij(k)} \leftarrow \min(S^{ik(k-1)}, S^{ik(k)} \otimes S^{kj(k)})$
            end if
        end for
        [D-step]
        if $k \equiv 0 \mod q$ then
            Checkpoint $S^{(k)}$
        end if
    end for
    Return $S = S^{(\ell)}$, the APSP matrix in BlockMatrix form
end function
```

Correctness is proved in section 4.

1.5 Optimality

The single-core cost of Floyd-Warshall, the best known single-core algorithm for APSP, is $O(n^3)$. A perfectly distributed form of Floyd-Warshall therefore has a total runtime of $O(n^3/p)$, in the asymptotic regime $n \to \infty$ and $p = O(n)$. Our algorithm achieves the same asymptotic runtime of

$$O\left(\frac{n^3}{p} + \frac{n^2 b}{\sqrt{p}} + n^2 + nb^2 + nb\log(p)\right)$$

for details see section 3.

One can also consider the *communication cost* scaling in terms of the amount of data tranferred over the network. The paper by Solomonik et al. (2013) derived a theoretical lower bound on the communication cost of APSP as $\Omega(\frac{n^2}{p^{2/3}})$ words. In comparison, our algorithm communicates a total of $O(n^2\sqrt{p} + \frac{n}{b}\sqrt{p})$ words, which is worse by a power of p. Note however that distributed Floyd-Warshall has the same $O(n^2\sqrt{p})$ bandwidth.

1.6 Communication Cost and Type

We analyze the *bandwidth* (total words sent) and the type of communication in each step of the algorithm. The A-step involves a one-to-one communication of a matrix of size $b \times b$ from a worker to the driver, involving a bandwidth of $O(b^2)$ words. The B-step involves a one-to-all broadcast of a matrix of size $b \times b$ from the driver to p workers, hence a bandwidth of $O(b^2 p)$ words. The C-step involves an all-to-all communication (a map-side join) where each worker recieves two $n/\sqrt{p} \times b$ matrices, and therefore entails a bandwidth of $O(nb\sqrt{p})$. Therefore the per-iteration bandwidth is

$O(pb^2+nb\sqrt{p})$. There are n/b iterations, so the total bandwidth for the algorithm is $O(n^2\sqrt{p}+pnb)$. See section 3 for the derivation of the bandwidth per step.

1.7 Summary of Results

For details see Section 5.

2 Background

The fastest known single-core algorithm for APSP is the Floyd-Warshall (FW) algorithm which takes $O(n^3)$ operations. The kth iteration can be expressed in matrix notation as

$$S^{(k)} \leftarrow \min(S^{(k-1)}, S_{\cdot k}^{(k-1)} \otimes S_{k\cdot}^{(k-1)})$$

Hence FW can be parallelized given a parallel algorithm for min-plus multiplication, as Kumar and Singh proposed in 1991 using Cannon's algorithm. However, since FW takes n sequential updates, its speed is limited for GPU computing and distributed computing. Buduc et al. (2010) proposed using a recursive formulation of APSP, known as the Divide-and-Conquer algorithm, to formulate a block-based recursive algorithm for APSP. Solomonik et al. (2013) extended Buduc's recursive approach to the distributed setting and analyzed the communication cost.

Both approaches by Buduc and Solomonik involve recursive algorithms which are difficult to implement in Spark. We therefore propose a *non-recursive* algorithm for APSP which can be considered a block-based generalization of the original Floyd-Warshall algorithm.

3 Complexity Analysis

For each step of the algorithm (A, B, C) we give the *computational cost* (computation done on each worker), the *bandwidth* (number of words sent) and the total *runtime* of the entire step. Here we define runtime as the time between the start of the algorithm and its termination as measured by an absolute clock. As such, the runtime incorporates both the time spent communcating and the time spent computing. To model communication times, we first consider a *latency-free* analysis which assumes a zero-latency network. Finally we give an analysis adjusting for latency.

All of our analysis is non-asymptotic. As such, we make the following assumptions on the computational costs and runtimes of atomic operations:

1. The time it takes to run Floyd-Warshall on a local matrix of size $b \times b$ is given by $\kappa_F b^3$
2. The time it takes to locally perform min-plus multiplication on matrices of size $a \times b$ and $b \times c$ be given by $\kappa_M ab^2c$. The time to update $C \leftarrow \min(C, A \otimes B)$ is the same as the time to compute $A \otimes B$ since C can be modifed in-place.
3. Separate the cost of communication and computation, so that sending messages and receiving messages is not included in the computational cost.
4. Time taken to send one message consisting of m words from one machine (whether a worker or driver) to another machine is $\kappa_T m$, where κ_T describes the rate of transmission
5. Time taken to *broadcast* m words from the driver to all workers is given by

   $$\log(p)\kappa_T m$$

   due to the usage of bittorrent broadcast.
6. Costs of checkpointing are considered in the latency-added analysis

3.1 A-step

In the kth iteration, the A-step is written in shorthand as $S^{kk(k)} \leftarrow \text{APSP}(S^{kk(k-1)})$. In fact, the updated `BlockMatrix` $S^{(k)}$ is not formed until the end of the C-step, and in the A-step $S^{kk(k)}$ only exists on the driver. $S^{ik(k)} \leftarrow \min(S^{ik(k-1)}, S^{ik(k)} \otimes S^{kk(k)})$ The detailed A-step is as follows:

1. The block $S^{kk(k-1)}$ is copied to the driver via invoking the `lookup` method with key (k, k) on the `BlockMatrix` $S^{(k-1)}$
2. The driver locally computes $S^{kk(k)} \leftarrow$ APSP($S^{kk(k-1)}$)

Computation. Only the driver performs computation, running Floyd-Warshall on $S^{kk(k-1)}$. This costs $\kappa_F b^3$.

Bandwidth. The lookup involves a one-to-one communication between the worker holding S^{kk} and the driver. Exactly b^2 words are transmitted.

Runtime. The runtime consists of the time taken to transmit S^{kk} and the time to compute APSPS^{kk}. Therefore the total runtime is

$$\kappa_T b^2 + \kappa_F b^3$$

3.2 B-step

In the kth iteration, the B-step is written in shorthand as $S^{kj(k)} \leftarrow \min(S^{kj(k-1)}, S^{kk(k)} \otimes S^{kj(k-1)})$ and $S^{ik(k)} \leftarrow \min(S^{ik(k-1)}, S^{ik(k)} \otimes S^{kk(k)})$ in parallel. This is accomplished by the following:

1. Form the RDD `rows` and the RDD `columns` by invoking the `filter` method on $S^{(k-1)}$ to keep only the blocks $S^{ik(k)}$ and $S^{kj(k)}$ for $i = 1, \ldots, \ell$ and $j = 1, \ldots, \ell$.
2. Broadcast $S^{kk(k)}$ from the driver to each worker holding a block in `rows` or `columns`. There should be $\sqrt{p}$ workers storing blocks in `rows` and another $\sqrt{p}$ workers storing blocks in `columns`.
3. Invoke `mapValues` on `rows` to perform the update $S^{kj(k)} \leftarrow \min(S^{kj(k)}, S^{kk(k)} \otimes S^{kj(k-1)})$ and similarly on `cols` to perform the update $S^{ik(k)} \leftarrow \min(S^{ik(k)}, S^{ik(k)} \otimes S^{kk(k)})$

Computation. Only the workers holding values in `rows` or `columns` perform computation, when `mapValues` is invoked. Each of those workers holds $\frac{n}{\sqrt{p}b}$ blocks. The update for each block entails a matrix multiply costing $\kappa_M b^3$, so the cost per worker is $\kappa_m \frac{nb^2}{\sqrt{p}}$, and the total cost is $2\kappa_M b^2 n$.

Bandwidth. The `broadcast` step entails sending a $b \times b$ matrix from the driver to $2\sqrt{p}$ workers. Therefore the bandwidth is $2\sqrt{p}b^2$

Runtime. The runtime cost of the broadcast is

$$\log(p)(\kappa_T b^2)$$

due to bittorrent broadcast. The runtime cost of computation is

$$\kappa_M \frac{b^2 n}{\sqrt{p}}$$

since each worker computes in parallel. Hence the total runtime is

$$\log(p)\kappa_T b^2 + \kappa_M \frac{b^2 n}{\sqrt{p}}$$

3.3 C-step

In shorthand, the C-step is written as $S^{ij(k)} \leftarrow \min(S^{ik(k-1)}, S^{ik(k)} \otimes S^{kj(k)})$. Here it becomes important to note the partitioner used for the various RDDs. We assume that the same partitioner is used for all $S^{(0)}, \ldots, S^{(\ell)}$ and for the RDDs `dupRows`, `dupCols`, and `temp` to be defined in the following.

1. Create RDD `dupRows` by invoking `flatMap` on `rows` to flatMap the key-value pair
$$(i, j, S^{ij})$$
to the key-value pairs
$$(1, j, S^{ij}), \ldots, (\ell, j, S^{ij})$$

2. Create RDD `dupCols` in an analogous way
3. Create RDD `temp` by joining `dupCols`, `dupRows`, and $S^{(k-1)}$. The order of joining does not matter.
4. Create `Blockmatrix` $S^{(k)}$ by invoking `mapValues` to `temp`. The (i, j) entry of `temp` contains the blocks $S^{ij(k-1)}, S^{ik(k)}, S^{kj(k)}$ produced in the A-step and B-step. The effect of `mapValues` is to produce

$$S^{ij(k)} \leftarrow \min(S^{ik(k-1)}, S^{ik(k)} \otimes S^{kj(k)})$$

as (i, j) block in $S^{(k)}$.

Computation. The computation occurs when `mapValues` is called. There, each worker computes $S^{ij(k)} \leftarrow \min(S^{ik(k-1)}, S^{ik(k)} \otimes S^{kj(k)})$ for each of its $n^2/(b^2p)$ blocks. The computation per block is $\kappa_M b^3$, and hence the computation per worker is $\kappa_M \frac{n^2b}{p}$. The total computational cost is $\kappa_M n^2 b$.

Bandwidth. Creation of `dupRows` and `dupCols` require bandwidth. Creation of `temp` and $S^{(k)}$ do not require bandwidth because of the map-side join. Each block in `rows` and `cols` must be sent to $\sqrt{p}$ workers. Hence the bandwidth cost is $2nb\sqrt{p}$.

Runtime. Creation of `dupRows` requires each worker holding values in `rows` to transmit a message of size $\frac{nb}{\sqrt{p}}$ to $\sqrt{p}$ other workers. This must be done sequentially under the current version of Spark (1.3.1) and hence takes time

$$\kappa_T nb$$

In practice, the RDD `dupRows` may be realized simultaneously. However, since we do not analyze scheduling, we make the simplifying assumtion that `dupRows` is created after `dupCols` is realized. Therefore, the runtime due to communcation-related issues is

$$2\kappa_T nb$$

Adding the computation per worker, the total runtime is

$$2\kappa_T nb + \kappa_M \frac{n^2b}{p}$$

3.4 Latency-Free Totals

We add up the costs derived for each step, and multiply by the number of iterations $\ell = n/b$.

Computational cost:

$$\kappa_M n^3 + 2\kappa_M n^2 b + \kappa_F nb^2$$

Bandwidth:

$$2n^2\sqrt{p} + 2n\sqrt{p}b + nb$$

Runtime:

$$\kappa_M \frac{n^3}{p} + \kappa_M \frac{bn^2}{\sqrt{p}} + 2\kappa_T n^2 + \kappa_F nb^2 + (\log(p) + 1)\kappa_T nb$$

3.5 Latency-Added Analysis

Here we *uniform* network topology so the latency of transmission is the same for every pair of workers. This is in contrast to the *grid* or *hypercube* topologies considered in most of the existing literature on distributed APSP.

Here we will consider large-p asymptotics so that that analysis is not sensitive to the distribution of the lag, as long as has exponential tails.

We assume the following model for latency:

- Whenever a machine transmits a message to another machine, it must wait a random amount of time T to make contact with the recipient. After contact is made, data is transferred at the rate $\frac{1}{\kappa_T}$.
- The lags T are independent and identically distributed. For iid lags T_i we have

  $$\mathrm{E}\sup_{i=1}^{p} T_i = \kappa_L \log p$$

  where κ_L is some constant.
- *Checkpointing:* Writing m bytes to disk takes $\kappa_C m$ runtime and does not cost bandwidth nor CPU.

As a consequence of these assumptions, one-to-one communications incur zero additional runtime due to lag, one-to-all (broadcast) communications incur $\kappa_L \log^2 p$ additional runtime due to lag and all-to-all communications incur $p\kappa_L \log(p)$ additional runtime due to lag.

Given that the algorithm involves one one-to-one communication, one one-to-all communcation, four all-to-all communications, and $1/q$ checkpointing operations per iteration, the added runtime from latency and checkpointing is

$$L = \kappa_L(\log^2(p) + 4p\log(p)) + \kappa_C \frac{n^2}{pq}$$

The runtime of the algorithm is given by

$$L\frac{n}{b} + \kappa_M \frac{n^3}{p} + \kappa_M \frac{bn^2}{\sqrt{p}} + 2\kappa_T n^2 + \kappa_F nb^2 + (\log(p)+1)\kappa_T nb$$

4 Correctness

4.1 Notation

We assume a complete weighted and directed graph (with all possible arcs). Any incomplete graph is equivalent to the complete graph where missing edges are represented by infinite-weight edges.

- If $i, j \in V$, let $i \to j$ denote the arc from i to j. An arc is also a path (of length 1).
- If $v_1, \ldots, v_m \in V$, then $p = v_1 \to \cdots \to v_m$ denotes a path composed of arcs $v_1 \to v_2, \ldots, v_{m-1} \to v_m$.
- Let $w(p)$ denote the weight of the path, i.e. the weight of the arcs in the path
- If P is a set of paths, and $v \in V$, then

  $$v \to P = \{v \to p : p \in P\}$$

  and

  $$P \to v = \{p \to v : p \in P\}$$
- If $v, w \in V$ and $S \subset V$, then

  $$v \to S \to w = \{v \to s \to w : s \in S\}$$

  and

  $$v \to S^* \to w = \{v \to w\} \cup \{v \to s_1 \to w : s \in S\} \cup \{v \to s_1 \to s_2 \to w : s_1, s_2 \in S\} \cup \cdots$$
- If P is a set of paths,

  $$w(P) = \min_{p \in P} w(p)$$
- If $v, w \in V$, and $S, U \subset V$, then

  $$\begin{aligned} v \to (S \to U \to S)^* \to w &= \{v \to w\} \cup \{v \to s_1 \to u \to s_2 \to w\} \cup \cdots \\ &= v \to (S \cup U)^* \to w \end{aligned}$$

It follows that the APSP matrix S has the property

$$S_{ij} = w(i \to V^* \to j)$$

4.2 Proof

Let $V = V_1 \cup \cdots \cup V_\ell$ where $V_i = \{(b(i-1)+1, \cdots, bi\}$. Let $W_i = \bigcup_{k=1}^{i} V_i$. Correctness follows if we can show that for $k = 1, \ldots, \ell$, we have

$$S_{ij}^{(k)} = w(i \to W_k \to j)$$

Fix ℓ. We proceed by finite induction on k.

Base case.

We will show the case $k = 1$. From the A-step, we have for $a, b \in V_1$

$$S_{ab}^{11(1)} = w(a \to V_1^* \to b)$$

Therefore, from the B-step, we have for $a \in V_1$, $b \in V_i$

$$S_{ab}^{1i(1)} = w(a \to V_1^* \to b)$$

$$S_{ba}^{i1(1)} = w(b \to V_1^* \to a)$$

Then from the C-step, we have for $a \in V_i$, $b \in V_j$ that

$$S_{ab}^{ij(1)} = w(a \to V_1^* \to b)$$

Of course $W_1 = V_1$ so the base case is established.

Induction.

Assume

$$S_{ab}^{(k-1)} = w(a \to W_{k-1}^* \to b)$$

Then in the kth iteration, from the A-step we have

$$S_{ab}^{kk(k)} = w(a \to (W_{k-1}^* \to V_k^* \to W_{w-1}^*)^* \to b) = w(a \to W_k^* \to b)$$

From the B-step we have

$$\begin{aligned} S_{ab}^{ik(k)} &= w((a \to W_{k-1}^* \to b) \cup (a \to W_k^* \to V_k \to W_{k-1}^* \to b)) \\ &= w(a \to W_k^* \to b) \end{aligned}$$

and similarly

$$S_{ab}^{ki(k)} = w(a \to W_k^* \to b)$$

Therefore in the C-step

$$\begin{aligned} S_{ab}^{ij(k)} &= w((a \to W_{k-1}^* \to b) \cup (a \to W_k^* \to V_k \to W_k^* \to b)) \\ &= w(a \to W_k^* \to b) \end{aligned}$$

Hence we have shown

$$S_{ab}^{(k)} = w(a \to W_k^* \to b)$$

as needed.

5 Results

6 References

- Buluc, Aydn, John R. Gilbert, and Ceren Budak. ”Solving path problems on the GPU.” Parallel Computing 36.5 (2010): 241-253.
- Zaharia, M., & Chowdhury, M. (2010). Spark: cluster computing with working sets.
- Kumar, V., & Singh, V. (1991). Scalability of parallel algorithms for the all-pairs shortest-path problem. Journal of Parallel and Distributed Computing, 13(2), 124138. doi:10.1016/0743-7315(91)90083-L
- Solomonik, E., Buluc, A., & Demmel, J. (2013). Minimizing communication in all-pairs shortest paths. Proceedings - IEEE 27th International Parallel and Distributed Processing Symposium, IPDPS 2013, 548559. doi:10.1109/IPDPS.2013.111