Minimizing communication in all-pairs shortest-paths

Edgar Solomonik
Aydin Buluc
James Demmel

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2012-19
http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-19.html

February 1, 2012

Minimizing communication in all-pairs shortest-paths

(Regular Submission)

Edgar Solomonik
UC Berkeley
solomon@eecs.berkeley.edu

Aydın Buluç
Lawrence Berkeley Nat. Lab.
abuluc@lbl.gov

James Demmel
UC Berkeley
demmel@cs.berkeley.edu

Abstract

We consider distributed memory algorithms for the all-pairs shortest paths (APSP) problem. Scaling the APSP problem to high concurrencies requires both minimizing inter-processor communication as well as maximizing temporal data locality. Our 2.5D APSP algorithm, which is based on the divide-and-conquer paradigm, satisfies both of these requirements: it can utilize the extra available memory to perform asymptotically less communication, and it is rich in semiring matrix multiplications, which have high temporal locality. We start by introducing a block-cyclic 2D (minimal memory) APSP algorithm. With a careful choice of block-size, this algorithm achieves known communication lower-bounds on latency and bandwidth. We extend this 2D block-cyclic algorithm to a 2.5D algorithm, which can use c extra copies of data to reduce the bandwidth cost by a factor of $c^{1/2}$, compared to its 2D counterpart. However, the 2.5D algorithm increases the latency cost by $c^{1/2}$. We provide a tighter lower bound on latency, which dictates that the latency overhead is necessary to reduce bandwidth along the critical path of execution. Our implementation achieves impressive performance and scaling to 24,576 cores of a Cray XE6 supercomputer by utilizing well-tuned intra-node kernels within the distributed memory algorithm.

Keywords: semiring matrix multiplication, 2.5D algorithms, minimizing communication, all-pairs shortest-paths

1 Introduction

The all-pairs shortest paths (APSP) is a fundamental graph problem with many applications in urban planning and simulation [17], datacenter network design [9], metric nearness problem [7], and traffic routing. In fact, APSP and the decrease-only metric nearness problem are equivalent. APSP is also used as a subroutine in other graph algorithms, such as Ullman and Yannakakis's breadth-first search algorithm [25], which is suitable for high diameter graphs.

Given a directed graph $G = (V, E)$ with n vertices $V = \{v_1, v_2, ..., v_n\}$ and m edges $E = \{e_1, e_2, ..., e_m\}$, the distance version of the algorithm computes the length of the shortest path from v_i to v_j for all (v_i, v_j) pairs. The full version also returns the actual paths in the form of a predecessor matrix. Henceforth, we will call the distance-only version as all-pairs shortest distances (APSD) to avoid confusion.

The classical dynamic programming algorithm for APSP is due to Floyd [10] and Warshall [27]. Serial blocked versions of the Floyd-Warshall algorithm have been formulated [21] to increase data locality. The algorithm can also be recast into semiring algebra over vectors and matrices. This vectorized algorithm, attributed to Kleene, is rich in matrix multiplications over the (min,+) semiring. Several theoretical improvements have been made, resulting in subcubic algorithms for the APSD problem. In practice though, these algorithms are not competitive with simpler cubic algorithms.

Variants of the Floyd-Warshall algorithm are most suitable for dense graphs. Johnson's algorithm [15], which is based on repeated application of Dijkstra's single-source shortest path algorithm (SSSP), is theoretically faster than the Floyd-Warshall variants on sufficiently sparse graphs. However, the data dependency structure of this algorithm (and Dijkstra's algorithm in general) make scalable parallelization difficult. SSSP algorithms based on Δ-stepping [20] scale better in practice but their performance is input dependent and scales with $O(m + d \cdot L \cdot \log n)$, where d is the maximum vertex degree and L is the maximum shortest path weight from the source. Consequently, it is likely that a Floyd-Warshall based approach would be competitive even for sparse graphs, as realized on graphical processing units [8].

Given the $\Theta(n^2)$ output of the algorithm, large instances can not be solved on a single node due to memory limitations. Further, a distributed memory approach is favorable over an out-of-core method, because of the high computational complexity of the problem. In this paper, we are concerned with obtaining high performance in a practical implementation by reducing communication cost and increasing data locality through optimized matrix multiplication over semirings.

Communication-avoiding '2.5D' algorithms take advantage of the extra available memory and reduce the bandwidth cost of many algorithms in numerical linear algebra. Generally, 2.5D algorithms can use a factor of c more memory to reduce the bandwidth cost by a factor of $\sqrt{c}$ [24]. The theoretical communication reduction translates to a significant improvement in strong-scalability (scaling processor count with a constant total problem size) on large supercomputers [23].

Our main contributions in this work are:

1. A block-cyclic 2D version of the divide-and-conquer APSP algorithm, which minimizes latency and bandwidth given minimal memory.

2. A 2.5D generalization of the 2D APSP algorithm, which sends a minimal number of messages and words of data given any amount of available memory.

3. A distributed memory implementation with highly tuned intra-node kernels, achieving impressive performance in the highest concurrencies reported in literature (24,576 cores of the Hopper Cray XE6 [1]).

Our algorithms can simultaneously construct the paths themselves, at the expense of doubling the cost, by maintaining a predecessor matrix as classical iterative Floyd-Warshall does. Our divide-and-conquer algo-

rithm essentially performs the same path computation as Floyd-Warshall except with a different schedule. The experiments only report on the distance version to allow easier comparison with prior literature.

2 Previous work

Jenq and Sahni [14] were the first to give a 2D distributed memory algorithm for the APSP problem, based on the original Floyd-Warshall schedule. Since the algorithm does not employ blocking, it has to perform n global synchronizations, resulting in a latency lower bound of $\Omega(n)$. This SUMMA-like algorithm [26] is improved further by Kumar and Singh [16] by using pipelining to avoid global synchronizations. Although they reduced the synchronization costs, both of these algorithms have low data reuse: each processor performs n rank-1 updates on its local submatrix. Obtaining high-performance in practice requires increasing temporal locality and is achieved by the blocked divide-and-conquer algorithms we consider in this work.

The main idea behind the divide-and-conquer (DC) algorithm is based on a proof by Aho et al. [3] that shows that costs of semiring matrix multiplication and APSP are asymptotically equivalent in the random access machine (RAM) model of computation. Actual algorithms based on this proof are given by various researchers, with minor differences. Our decision to use the DC algorithm as our starting point is inspired by its demonstrated better cache reuse on CPUs [21], and its impressive performance attained on the many-core graphical processor units [8].

Previously known communication bounds [4, 12, 13] for 'classical' (triple-nested loop) matrix multiplication also apply to our algorithm, because Aho et al.'s proof shows how to get the semiring matrix product for free, given an algorithm to compute the APSP. These lower bounds, however, are not necessarily tight because the converse of their proof (to compute APSP given matrix multiplication) relies on the cost of matrix multiplication being $\Omega(n^2)$, which is true for its RAM complexity but not true for its bandwidth and latency costs. In Section 4, we show that a tighter bound exist for latency, one similar to the latency lower bound of LU decomposition [24].

Seidel [22] showed a way to use fast matrix multiplication algorithms, such as Strassen's algorithm, for the solution of the APSP problem by embedding the (min,+) semiring into a ring. However, his method only works for undirected and unweighted graphs. We cannot, therefore, utilize the recently discovered communication-optimal Strassen based algorithms [4] directly for the general problem.

Habbal et al. [11] gave a parallel APSP algorithm for the Connection Machine CM-2 that proceeds in three stages. Given a decomposition of the graph, the first step constructs SSSP trees from all the 'cutset' (separator) vertices, the second step runs the classical Floyd-Warshall algorithm for each partition independently, and the last step combines these results using 'minisummation' operations that is essentially semiring matrix multiplication. The algorithm's performance depends on the size of separators for balanced partitions. Without good sublinear (say, $O(\sqrt{n})$) separators, the algorithm degenerates into Johnson's algorithm. Good separators do not exist in almost all graphs [18], including those from social networks. Note that the number of partitions are independent (and generally much less) from the number of active processors. The algorithm sends $\Theta(n)$ messages and moves $\Theta(n^2)$ words for the 5-point stencil (2-D grid).

Brickell et al. [7] came up with a linear programming formulation for the APSP problem, by exploiting its equivalence to the decrease-only version of the metric nearness problem (DOMN). Their algorithm runs in $O(n^3)$ time using a Fibonacci heap, and the dual problem can be used to obtain the actual paths. Unfortunately, heaps are inherently sequential data structures that limit parallelism. Since the equivalence between APSP and DOMN goes both ways, our algorithm provides a highly parallel solution to the DOMN problem as well.

3 Divide-and-Conquer APSP

The all-pairs shortest-paths problem corresponds to finding the matrix closure on the tropical (min,+) semiring. Algorithm 1 gives the high-level structure of the divide-and-conquer all-pairs-shortest-path algorithm (DC-APSP). The correctness of this algorithm has been proved by many researchers [3, 8, 21] using various methods. Edge weights can be arbitrary, including negative numbers, but we assume that the graph is free of negative cycles. Compared to the classical matrix multiplication over the ring of real numbers, in our semiring-matrix-matrix-multiplication, each multiply operation is replaced with an addition (to calculate the length of a larger path from smaller paths or edges) and each add operation is replaced with a minimum operation (to get the minimum in the presence of multiple paths).

Algorithm 1: **DC-APSP(A, n)**

Input: $n \times n$ matrix A, representing the adjacency matrix of a graph G.
Output: $n \times n$ matrix A, representing the APSP distance matrix of graph G.
if $n = 1$ **then**
  return.
end
Partition $A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$, where all A_{ij} are $n/2$-by-$n/2$
$A_{11} \leftarrow$ `DC-APSP(`A_{11}`,` $n/2$`)`
`/* Operation` $\cdot$ `denotes Semiring-Matrix-Matrix-Multiply */`
$A_{12} = A_{11} \cdot A_{12}$
$A_{21} = A_{21} \cdot A_{11}$
$A_{22} = \min(A_{22}, A_{21} \cdot A_{12})$
$A_{22} \leftarrow$ `DC-APSP(`A_{22}`,` $n/2$`)`
$A_{21} = A_{22} \cdot A_{21}$
$A_{12} = A_{12} \cdot A_{22}$
$A_{11} = \min(A_{11}, A_{12} \cdot A_{21})$

For simplicity, we formulate our algorithms and give results only for adjacency matrices of power-of-two dimension. Extending the algorithms and analysis to general adjacency matrices is straight-forward.

Each semiring-matrix-matrix-multiplication performs $O(n^3)$ additions and $O(n^2)$ minimum (min) operations. If we count each addition and min operation as $O(1)$ flops, the total computation cost of DC-APSP, F, is given by a recurrence

$$F(n) = 2 \cdot F(n/2) + O(n^3)$$
$$F(n) = O(n^3).$$

4 Communication lower bounds

A good parallel algorithm has as little inter-processor communication as possible. In this section, we prove lower bounds on the inter-processor communication required to compute DC-APSP in parallel. All of our lower bounds are extensions of dense linear algebra communication lower bounds.

4.1 Bandwidth lower bound

We measure the bandwidth cost as the number of words (bytes) sent or received by any processor along the critical path of execution. Semiring matrix multiplication has the same computational dependency structure

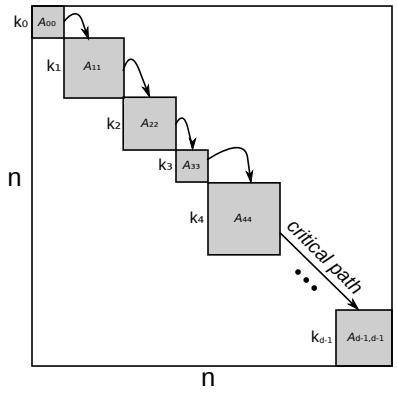


Figure 1: DC-APSP diagonal block dependency path. These blocks must be computed in order and communication is required between each block.

as classical matrix multiplication. The same communication cost analysis applies because only the scalar multiply and add operations are different. Our analysis will assume no data is replicated at the start and that the computational work is load-balanced.

The lower bound on bandwidth cost of matrix multiplication is due to Hong and Kung [12]. Ballard et al. [5] extended those lower bounds to other traditional numerical linear algebra algorithms. For a local memory of size M, matrix multiplication requires

$$W(M) = \Omega\left(\frac{n^3}{p\sqrt{M}}\right) \tag{1}$$

words to be sent by some processor. Further, for a memory of any size, matrix multiplication requires

$$W = \Omega\left(\frac{n^2}{p^{2/3}}\right)$$

words to be sent [2, 24]. These bounds apply directly to semiring matrix multiplication and consequently to DC-APSP, which performs many semiring matrix multiplications.

4.2 Latency lower bound

The first bandwidth lower-bound in the previous section (Equation 1), provides a latency lower-bound on semiring-matrix multiplication. Since no message can be larger than the local memory on a given processor,

$$S(M) = \Omega\left(\frac{n^2}{p \cdot M^{3/2}}\right)$$

messages must be sent by some processor. This latency lower-bound applies for classical and semiring matrix multiplication, as well as DC-APSP.

However, we can obtain a tighter lower-bound for DC-APSP by considering the dependency structure of the algorithm. As it turns out, we can use the same argument as presented in [24] for 2.5D LU factorization. Figure 1 considers how the distance matrix A is blocked along its diagonal. Some blocking along the diagonal always exists, since some processor must compute the diagonal element of the distance matrix. Each

block of dimension b requires $\Omega(1)$ message to be sent, $\Omega(b^2)$ words to be sent and $\Omega(b^3)$ computational operations. The requirements are the same as 2.5D LU and yield a lower bound on the latency cost. If we desire a bandwidth cost of

$$W = O\left(n^2/\sqrt{cp}\right),$$

for some c, the blocking must have a latency cost of

$$S = \Omega(\sqrt{cp}).$$

5 Parallelization of DC-APSP

In this section, we introduce techniques for parallelization of the divide-and-conquer all-pairs-shortest-path algorithm (DC-APSP). Our first approach uses a 2D block-cyclic parallelization. We demonstrate that a careful choice of block-size can minimize both latency and bandwidth costs simultaneously. Our second approach utilizes a 2.5D decomposition [23, 24]. Our cost analysis shows that the 2.5D algorithm reduces the bandwidth cost and improves strong scalability.

5.1 2D Divide-and-Conquer APSP

We start by deriving a parallel DC-APSP algorithm that operates on a square 2D processor grid and consider cyclic and blocked variants.

5.1.1 2D Semiring-Matrix-Matrix-Multiply

Algorithm 2: **[C] = 2D-SMMM(**A**,** B**,** C**,** Λ**,** n**)**

Input: n-by-n matrix A, n-by-n matrix B, each spread over a square processor grid Λ
Output: n-by-n matrix C, such that $C = \min(C, A \cdot B)$, and C is spread over a square processor grid Λ

```
/* These updates can be blocked or pipelined                    */
pipelined for t = 1 to t = n do
    Replicate A[:, t] on columns of Λ[:, :]        /* Broadcast a column of A */
    Replicate B[t, :] on rows of Λ[:, :]           /* Broadcast a row of B    */
    /* Perform Semiring-Matrix-Matrix-Multiply                  */
    C[:, :] := min(C[:, :], A[:, t] + B[t, :])
end
```

Algorithm 2 describes an algorithm for performing Semiring-Matrix-Matrix-Multiply (SMMM) on a 2D processor grid. Since the data dependency structure of SMMM is identical to traditional matrix multiply, we employ the popular SUMMA algorithm [26]. The algorithm is formulated in terms of distributed rank-1 updates. These updates are associative and commutative so they can be pipelined or blocked. To achieve optimal communication performance, the matrices should be laid out in a blocked fashion, and each row and column of processors should broadcast its block-row and block-column in turn. Given p processors, each processor would then receive $O(\sqrt{p})$ messages of size $O(n^2/p)$, giving a bandwidth cost of $O(n^2/\sqrt{p})$. We note that any different classical distributed matrix multiplication algorithm (e.g. Cannon's algorithm) can be used here in place of SUMMA.

Algorithm 3: **BLOCKED-DC-APSP**(A, Λ, n, p)

Input: $n \times n$ matrix A, spread over $\sqrt{p}$-by-$\sqrt{p}$ processor grid Λ, representing the adjacency matrix of a graph G.

Output: $n \times n$ matrix A, spread over $\sqrt{p}$-by-$\sqrt{p}$ processor grid Λ, representing the APSP distance matrix of graph G.

if $p = 1$ **then**

$A \leftarrow$`DC-APSP(`A`,` n`)`

else

Partition $A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$, where all A_{ij} are $n/2$-by-$n/2$

Partition $\Lambda = \begin{bmatrix} \Lambda_{11} & \Lambda_{12} \\ \Lambda_{21} & \Lambda_{22} \end{bmatrix}$, where all Λ_{ij} are $\sqrt{p}/2$-by-$\sqrt{p}/2$

$A_{11} \leftarrow$`BLOCKED-DC-APSP(`A_{11}`,` Λ_{11}`,` $n/2$`,` $p/4$`)`

$\Lambda_{11}[:,:]$ sends A_{11} to $\Lambda_{12}[:,:]$.

$A_{12} \leftarrow$`2D-SMMM(`A_{11}`,` A_{12}`,` A_{12}`,` Λ_{12}`,` $n/2$`)`

$\Lambda_{11}[:,:]$ sends A_{11} to $\Lambda_{21}[:,:]$.

$A_{21} \leftarrow$`2D-SMMM(`A_{21}`,` A_{11}`,` A_{21}`,` Λ_{21}`,` $n/2$`)`

$\Lambda_{12}[:,:]$ sends A_{12} to $\Lambda_{22}[:,:]$.

$\Lambda_{21}[:,:]$ sends A_{21} to $\Lambda_{22}[:,:]$.

$A_{22} \leftarrow$`2D-SMMM(`A_{21}`,` A_{12}`,` A_{22}`,` Λ_{22}`,` $n/2$`)`

$A_{11} \leftarrow$`BLOCKED-DC-APSP(`A_{22}`,` Λ_{22} $n/2$`,` $p/4$`)`

$\Lambda_{22}[:,:]$ sends A_{22} to $\Lambda_{21}[:,:]$.

$A_{21} \leftarrow$`2D-SMMM(`A_{22}`,` A_{21}`,` A_{21}`,` Λ_{21}`,` $n/2$`)`

$\Lambda_{22}[:,:]$ sends A_{22} to $\Lambda_{12}[:,:]$.

$A_{12} \leftarrow$`2D-SMMM(`A_{12}`,` A_{22}`,` A_{12}`,` Λ_{12}`,` $n/2$`)`

$\Lambda_{21}[:,:]$ sends A_{21} to $\Lambda_{11}[:,:]$.

$\Lambda_{12}[:,:]$ sends A_{12} to $\Lambda_{11}[:,:]$.

$A_{22} \leftarrow$`2D-SMMM(`A_{12}`,` A_{21}`,` A_{11}`,` Λ_{11}`,` $n/2$`)`

end

5.1.2 2D blocked Divide-and-Conquer APSP

Algorithm 3 gives a parallel 2D blocked version of the DC-APSP algorithm. In this algorithm, each SMMM is computed on the quadrant of the processor grid on which the result belongs. The operands, A and B, must be sent to the processor grid quadrant on which C is distributed. At each recursive step, the algorithm proceeds with one quadrant of the processor grid.

This blocked algorithm has a clear flaw, in that at most a quarter of the processors are active at any point in the algorithm. We will alleviate this load-imbalance by introducing a block-cyclic version of the algorithm.

5.1.3 2D block-cyclic Divide-and-Conquer APSP

Algorithm 4 gives the full 2D block-cyclic DC-APSP algorithm. This block-cyclic algorithm operates by performing *cyclic-steps* until a given block-size, then proceeding with *blocked-steps* by calling the blocked algorithm as a subroutine. At each cyclic-step, each processor operates on sub-blocks of its local matrix block, while at each blocked-step a sub-grid of processors operate on their full matrix blocks. These two steps are demonstrated in sequence in Figure 2 on a 4x4 processor grid.

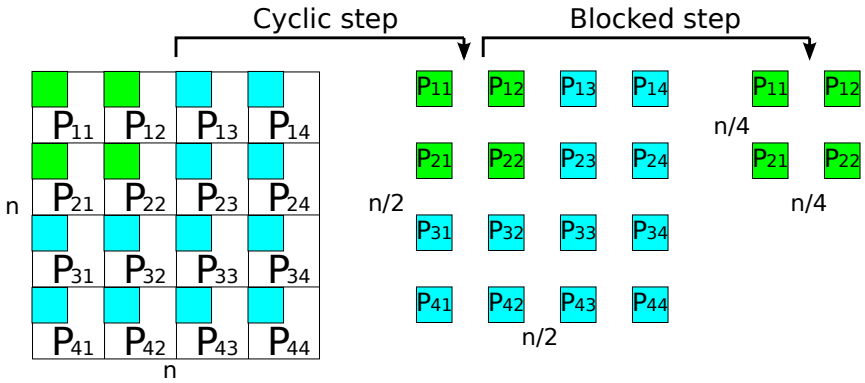


Figure 2: Our block-cyclic 2D APSP algorithm performs cyclic-steps until a given block-size, then performs blocked-steps as shown in this diagram.

We note that no redistribution of data is required to use a block-cyclic layout. Traditionally, (e.g. in ScaLAPACK [6]) using a block-cyclic layout requires that each processor own a block-cyclic portion of the starting matrix. However, the APSP problem is invariant to permutation (permuting the numbering of the node labels does not change the answer). We exploit permutation invariance by assigning each process the same sub-block of the adjacency and distance matrices, no matter how many blocked or cyclic steps are taken.

As derived in Appendix A, if the block size is picked as $b = O(n/\log(p))$ (execute $O(\log\log(p))$ cyclic recursive steps), the bandwidth cost is

$$W_{\text{bc-2D}}(n,p) = O(n^2/\sqrt{p}),$$

and the latency cost is

$$S_{\text{bc-2D}}(p) = O(\sqrt{p}\log^2(p)).$$

These costs are optimal (modulo the polylog latency term) when the memory size is $M = O(n^2/p)$. The costs are measured along the critical path of the algorithm, showing that both the computation and communication are load balanced throughout execution.

5.2 2.5D DC-APSP

In order to construct a communication-optimal DC-APSP algorithm, we utilize 2.5D-SMMM. Transforming 2D SUMMA (Algorithm 2) to a 2.5D algorithm can be done simply by performing a different subset of updates on each one of c processor layers. Algorithm 5 details 2.5D SUMMA, modified to perform SMMM. Giving a replication factor $c \in [1, p^{1/3}]$, the blocked 2.5D-SMMM algorithm moves $O(n^2/\sqrt{cp})$ words and sends $O(\sqrt{p/c^3})$ messages.

Algorithm 6 gives the blocked version of the 2.5D DC-APSP algorithm. The blocked algorithm executes multiplies and recurses on octants of the processor grid (rather than quadrants in the 2D version). The algorithm recurses until $c = 1$, which must occur while $p \geq 1$, since $c \leq p^{1/3}$. The algorithm then calls the 2D block-cyclic algorithm on the remaining 2D sub-partition.

The 2.5D blocked algorithm suffers from load-imbalance. In fact, the top half of the processor grid does no work. We can fix this by constructing a block-cyclic version of the algorithm, which performs cyclic steps with the entire 3D processor grid, until the block-size is small enough to switch to the blocked version. Algorithm 7 gives the 2.5D block-cyclic DC-APSP algorithm.

Algorithm 4: **BLOCK-CYCLIC-DC-APSP**(A, Λ, n, p, b)

Input: $n \times n$ matrix A, spread over square processor grid Λ, representing the adjacency matrix of a graph G.

Output: $n \times n$ matrix A, spread over square processor grid Λ, representing the APSP distance matrix of graph G.

Partition $A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$, where all A_{ij} are $n/2$-by-$n/2$

if $n > b$ **then**

$A_{11} \leftarrow$BLOCK-CYCLIC-DC-APSP(A_{11}, Λ, $n/2$, p, b)

$A_{12} \leftarrow$2D-SMMM(A_{11}, A_{12}, A_{12}, Λ, $n/2$)

$A_{21} \leftarrow$2D-SMMM(A_{21}, A_{11}, A_{21}, Λ, $n/2$)

$A_{22} \leftarrow$2D-SMMM(A_{21}, A_{12}, A_{22}, Λ, $n/2$)

$A_{11} \leftarrow$BLOCK-CYCLIC-DC-APSP(A_{22}, Λ, $n/2$, p, b)

$A_{21} \leftarrow$2D-SMMM(A_{22}, A_{21}, A_{21}, Λ, $n/2$)

$A_{12} \leftarrow$2D-SMMM(A_{12}, A_{22}, A_{12}, Λ, $n/2$)

$A_{22} \leftarrow$2D-SMMM(A_{12}, A_{21}, A_{11}, Λ, $n/2$)

else

$A \leftarrow$BLOCKED-DC-APSP(A, Λ, n, p)

end

As derived in Appendix B, if the 2.5D block size is picked as $b_1 = O(n/c)$ (execute $O(\log(c))$ 2.5D cyclic recursive steps), the bandwidth cost is

$$W_{\text{bc-2.5D}}(n,p) = O(n^2/\sqrt{cp}),$$

and the latency cost is

$$S_{\text{bc-2.5D}}(p) = O(\sqrt{cp}\log^2(p)).$$

These costs are optimal for any memory size (modulo the polylog latency term).

6 Experiments

In this section, we show that the distributed APSP algorithms do not just lower the theoretical communication cost, but actually improve performance on large supercomputers. We implement the 2D and 2.5D variants of DC-APSP recursively, as described in the previous section.

6.1 Implementation

The dominant sequential computational work of the DC-APSP algorithm is the Semiring-Matrix-Matrix-Multiplies (SMMM) called at every step of recursion. Our implementation of SMMM uses two-level cache-blocking (L1, L3), register blocking, explicit SIMD intrinsics, and loop unrolling. We implement threading by assigning L1-cache blocks of C to different threads.

Our 2.5D DC-APSP implementation generalizes the following algorithms: 2D cyclic, 2D blocked, 2D block-cyclic, 2.5D blocked, 2.5D cyclic, and 2.5D block-cyclic. Block sizes b_1 and b_2 control how many 2.5D and 2D cyclic and blocked steps are taken. These block-sizes are set at run-time and require no modification to the algorithm input or distribution.

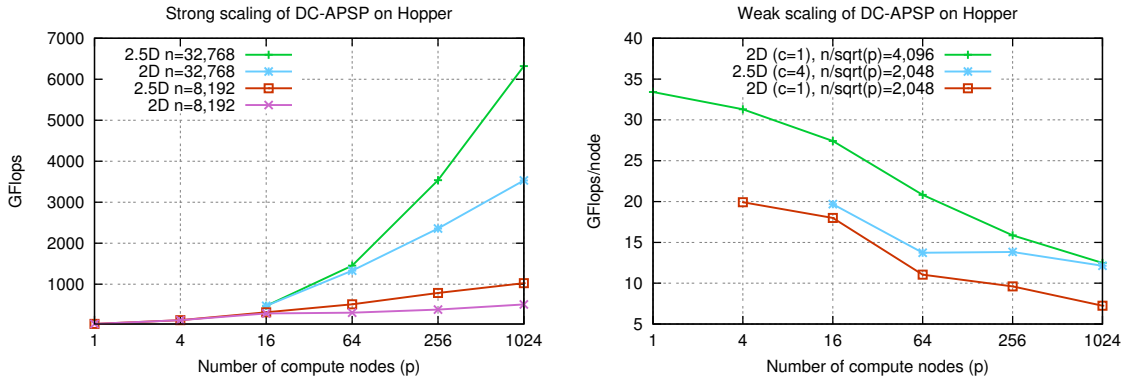
Algorithm 5: **[C] = 2.5D-SMMM(A, B, C, Π, n, p, c)**

Input: n-by-n matrices A, B, C, each spread over $\sqrt{p/c}$-by-$\sqrt{p/c}$ processor grid $\Pi[:,:,1]$.
Output: n-by-n matrix C, such that $C = \min(C, A \cdot B)$, and C is spread over $\sqrt{p/c}$-by-$\sqrt{p/c}$ processor grid $\Pi[:,:,1]$.

```
/* Do with each processor in parallel                                    */
```
for all $i, j \in [1, \sqrt{p/c}], k \in [1, c]$ **do**
  Replicate $A[i,j]$, $B[i,j]$ on all layers $\Pi[i,j,:]$
  if $k > 1$ **then**
    Initialize $C[:,:,k] = \infty$
  end
  pipelined for $t = (k-1) \cdot n/c$ **to** $t = k \cdot n/c$ **do**
    Replicate $A[:,t]$ on columns of $\Lambda[:,:]$
    Replicate $B[t,:]$ on rows of $\Lambda[:,:]$
```
                                  /* Perform Semiring-Matrix-Matrix-Multiply */
```
    $C[:,:,k] := \min(C[:,:], A[:,t] + B[t,:])$
  end
  $C[:,:,1] := \min(C[:,:,:])$ `/* min-reduce C across layers */`
end

We compiled our codes with the GNU C/C++ compilers (v4.6) with the -O3 flag. We use Cray's MPI implementation, which is based on MPICH2. We run 4 MPI processes per node, and use 6-way intra-node threading with the GNU OpenMP library. The input is an adjacency matrix with entries representing edge-weights in double-precision floating-point numbers.

6.2 Performance



(a) DC-APSP strong scaling

(b) DC-APSP weak scaling

Figure 3: Scaling of 2D and 2.5D block-cyclic DC-APSP on Hopper (Cray XE6)

Our experimental platform is 'Hopper', which is a Cray XE6 supercomputer, built from dual-socket 12-core "Magny-Cours" Opteron compute nodes. Each node can be viewed as a four-chip compute configuration due to NUMA domains. Each of these four chips have six super-scalar, out-of-order cores running at 2.1 GHz with private 64 KB L1 and 512 KB L2 caches. The six cores on a chip share a 6 MB L3 cache and dual DDR3-1333 memory controllers capable of providing an average stream [19] bandwidth of 12 GB/s

Algorithm 6: **2.5D-BLOCKED-DC-APSP**(A, Π, n, p, c, b)

Input: $n \times n$ matrix A, spread over $\sqrt{p/c}$-by-$\sqrt{p/c}$ processor grid $\Pi[:, :, 1]$, representing the adjacency matrix of a graph G.

Output: $n \times n$ matrix A, spread over $\sqrt{p/c}$-by-$\sqrt{p/c}$ processor grid $\Pi[:, :, 1]$, representing the APSP distance matrix of graph G.

if $c = 1$ **then**

$A \leftarrow$`BLOCK-CYCLIC-DC-APSP(`A`,` Π`,` n`,` p`,` b`)`

else

Partition $A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$, where all A_{ij} are $n/2$-by-$n/2$

Partition Π into 8 cubic block Π_{ijk}, for $i, j, k \in 1, 2$, where all Π_{ijk} are $\sqrt{p/c}/2$-by-$\sqrt{p/c}/2$-by-$c/2$.

$A_{11} \leftarrow$`2.5D-BLOCKED-DC-APSP(`A_{11}`,` Π_{111}`,` $n/2$`,` $p/8$`,` $c/2$`)`

$\Pi_{111}[:, :, 1]$ sends A_{11} to $\Pi_{121}[:, :, 1]$.

$A_{12} \leftarrow$`2.5D-SMMM(`A_{11}`,` A_{12}`,` A_{12}`,` Π_{121}`,` $n/2$`,` $p/8$`,` $c/2$`)`

$\Pi_{111}[:, :, 1]$ sends A_{11} to $\Pi_{211}[:, :, 1]$.

$A_{21} \leftarrow$`2.5D-SMMM(`A_{21}`,` A_{11}`,` A_{21}`,` Π_{211}`,` $n/2$`,` $p/8$`,` $c/2$`)`

$\Pi_{121}[:, :, 1]$ sends A_{12} to $\Pi_{221}[:, :, 1]$.

$\Pi_{211}[:, :, 1]$ sends A_{21} to $\Pi_{221}[:, :, 1]$.

$A_{22} \leftarrow$`2.5D-SMMM(`A_{21}`,` A_{12}`,` A_{22}`,` Π_{221}`,` $n/2$`,` $p/8$`,` $c/2$`)`

$A_{11} \leftarrow$`2.5D-BLOCKED-DC-APSP(`A_{22}`,` Π_{221} $n/2$`,` $p/8$`,` $c/2$`)`

$\Pi_{221}[:, :, 1]$ sends A_{22} to $\Pi_{211}[:, :, 1]$.

$A_{21} \leftarrow$`2.5D-SMMM(`A_{22}`,` A_{21}`,` A_{21}`,` Π_{211}`,` $n/2$`,` $p/8$`,` $c/2$`)`

$\Pi_{221}[:, :, 1]$ sends A_{22} to $\Pi_{121}[:, :, 1]$.

$A_{12} \leftarrow$`2.5D-SMMM(`A_{12}`,` A_{22}`,` A_{12}`,` Π_{121}`,` $n/2$`,` $p/8$`,` $c/2$`)`

$\Pi_{211}[:, :, 1]$ sends A_{21} to $\Pi_{111}[:, :, 1]$.

$\Pi_{121}[:, :, 1]$ sends A_{12} to $\Pi_{111}[:, :, 1]$.

$A_{22} \leftarrow$`2.5D-SMMM(`A_{12}`,` A_{21}`,` A_{11}`,` Π_{111}`,` $n/2$`,` $p/8$`,` $c/2$`)`

end

per chip. Nodes are connected through Cray's 'Gemini' network, which has a 3D torus topology. Each Gemini chip, which is shared by two Hopper nodes, is capable of 9.8 GB/s bandwidth.

Our threaded Semiring-Matrix-Matrix-Multiply achieves up to 13.6 GF on 6-cores of Hopper, which is roughly 25% of theoretical floating-point peak. This is a fairly good fraction in the absence of an equivalent fused multiply-add operation for our semiring. Our implementation of DC-APSP uses this subroutine to perform APSP at 17% of peak computational performance on 1 node (24 cores, 4 processes, 6 threads per process).

Figure 3(a) demonstrates the strong scaling performance of 2D and 2.5D APSP. Strong scaling performance is collected by keeping the adjacency matrix size constant and computing APSP with more processors. The 2.5D performance is given as the best performing variant for any replication factor c (in almost all cases, $c = 4$). Strong scaling a problem to a higher core-count lowers the memory usage per processor, allowing increased replication (increased c). Performing 2.5D style replication improves efficiency significantly, especially at large scale. On 24,576 cores of Hopper, the 2.5D algorithm improves on the performance of the 2D APSP algorithm by a factor of 1.8x for $n = 8,192$ and 2.0x for $n = 32,768$.

Figure 3(b) shows the weak scaling performance of the 2D and 2.5D DC-APSP algorithms. To collect weak scaling data, we keep the problem size per processor $(n/\sqrt{p})$ constant and grow the number of proces-

Algorithm 7: **2.5D-BLOCK-CYCLIC-DC-APSP**(A, Π, n, p, c, b_1, b_2)

Input: $n \times n$ matrix A, spread over $\sqrt{p/c}$-by-$\sqrt{p/c}$ processor grid $\Pi[:, :, 1]$, representing the adjacency matrix of a graph G.

Output: $n \times n$ matrix A, spread over $\sqrt{p/c}$-by-$\sqrt{p/c}$ processor grid $\Pi[:, :, 1]$, representing the APSP distance matrix of graph G.

Partition $A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$, where all A_{ij} are $n/2$-by-$n/2$

if $n > b_1$ **then**

$A_{11} \leftarrow$ `2.5D-BLOCK-CYCLIC-DC-APSP(`A_{11}, Π, $n/2$, p, c, b_1, b_2`)`

$A_{12} \leftarrow$ `2.5D-SMMM(`A_{11}, A_{12}, A_{12}, Π, $n/2$, p, c`)`

$A_{21} \leftarrow$ `2.5D-SMMM(`A_{21}, A_{11}, A_{21}, Π, $n/2$, p, c`)`

$A_{22} \leftarrow$ `2.5D-SMMM(`A_{21}, A_{12}, A_{22}, Π, $n/2$, p, c`)`

$A_{11} \leftarrow$ `2-5D-BLOCK-CYCLIC-DC-APSP(`A_{22}, Π, $n/2$, p, c, b_1, b_2`)`

$A_{21} \leftarrow$ `2.5D-SMMM(`A_{22}, A_{21}, A_{21}, Π, $n/2$, p, c`)`

$A_{12} \leftarrow$ `2.5D-SMMM(`A_{12}, A_{22}, A_{12}, Π, $n/2$, p, c`)`

$A_{22} \leftarrow$ `2.5D-SMMM(`A_{12}, A_{21}, A_{11}, Π, $n/2$, p, c`)`

else

$A \leftarrow$ `2.5D-BLOCKED-DC-APSP(`A, Λ, n, p, c, b_2`)`

end

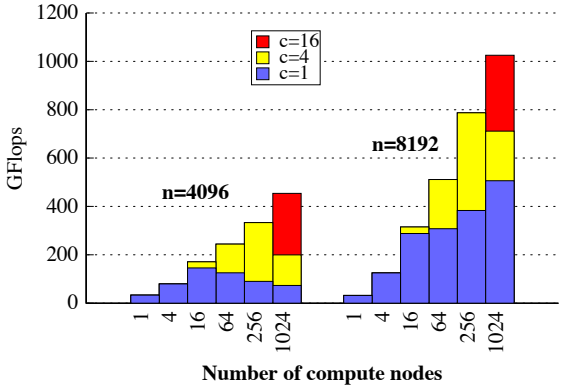
sors. Since the memory usage per processor does not decrease with the number of processors during weak scaling, the replication factor cannot increase. We compare data with $n/\sqrt{p} = 2048, 4096$ for 2D ($c = 1$) and with $n/\sqrt{p} = 2048$ for 2.5D ($c = 4$). The 2.5D DC-APSP algorithm performs almost as well as the 2D algorithm with a larger problem size and significantly better than the 2D algorithm with the same problem size.

The overall weak-scaling efficiency is good all the way up to the 24,576 cores (as far as we tested), where the code achieves an impressive aggregate performance over 12 Teraflops. At this scale, our 2.5D implementation solves the all-pairs shortest-paths problem for 65,536 vertices in roughly 2 minutes. With respect to 1-node performance, strong scaling allows us to solve a problem with 8,192 vertices over 30x faster on 1024 compute nodes. Weak scaling gives us a performance rate up to 380x higher on 1024 compute nodes than on one node.

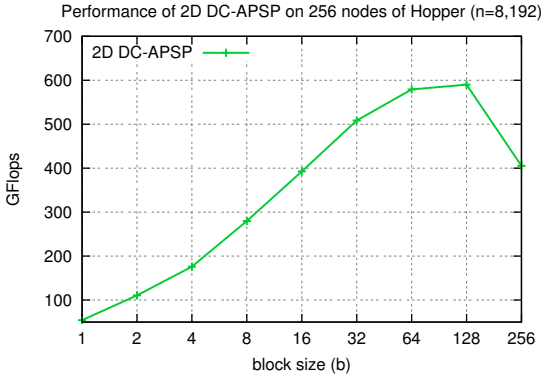
Figure 4(a) shows the performance of 2.5D DC-APSP on small matrices. The bars are stacked so the $c = 4$ case shows the added performance over the $c = 1$ case, while the $c = 16$ case shows the added performance over the $c = 4$ case. A replication factor of $c = 16$ results in a speed-up of 6.2x for the smallest matrix size $n = 4,096$. Overall, we see that 2.5D algorithm hits the scalability limit much later than the 2D counterpart. Tuning over the block sizes (Figure 4(b)), we also see the benefit of the block-cyclic layout for the 2D algorithm. The best performance over all block sizes is significantly higher than either the cyclic ($b = 1$) or blocked ($b = n/\sqrt{p}$) performance.

7 Conclusion

The divide-and-conquer APSP algorithm is well suited for parallelization in a distributed memory environment. The algorithm resembles well-studied linear algebra algorithms (e.g. matrix multiply, LU factorization). We exploit this resemblance to transfer implementation and optimization techniques from the linear algebra domain to the graph-theoretic APSP problem. In particular, we use a block-cyclic layout to load-balance the computation and data movement, while simultaneously minimizing message latency overhead.



(a) DC-APSP small matrix performance



(b) Performance of DC-APSP with respect to block size

Figure 4: Performance of DC-APSP on small matrices

Further, we formulate a 2.5D DC-APSP algorithm, which lowers the bandwidth cost and improves parallel scalability. Our implementations of these algorithms achieve good scalability at very high concurrency and confirm the practicality of our analysis.

Our techniques for avoiding communication allow for a scalable implementation of the divide-and-conquer APSP algorithm. The benefit of such optimizations grows with machine size and level of concurrency. The performance of our implementation can be further improved upon by exploiting locality via topology-aware mapping. The current Hopper job scheduler does not allocate contiguous partitions but other supercomputers (e.g. IBM BlueGene) allocate toroidal partitions, well-suited for mapping of 2D and 2.5D algorithms.

References

[1] Hopper, NERSC's Cray XE6 system. `http://www.nersc.gov/users/computational-systems/hopper/`.

[2] A. Aggarwal, A. K. Chandra, and M. Snir. Communication complexity of PRAMs. *Theoretical Computer Science*, 71(1):3 – 28, 1990.

[3] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Longman, Boston, MA, USA, 1974.

[4] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz. Graph expansion and communication costs of fast matrix multiplication: regular submission. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, SPAA '11, pages 1–12, New York, NY, USA, 2011. ACM.

[5] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz. Minimizing communication in numerical linear algebra. *SIAM J. Matrix Analysis Applications*, 32(3):866–901, 2011.

[6] L. S. Blackford, J. Choi, A. Cleary, E. D'Azeuedo, J. Demmel, I. Dhillon, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK user's guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1997.

[7] J. Brickell, I. S. Dhillon, S. Sra, and J. A. Tropp. The metric nearness problem. *SIAM J. Matrix Anal. Appl.*, 30:375–396, 2008.

[8] A. Buluç, J. R. Gilbert, and C. Budak. Solving path problems on the GPU. *Parallel Computing*, 36(5-6):241 – 253, 2010.

[9] A. R. Curtis, T. Carpenter, M. Elsheikh, A. Lpez-Ortiz, and S. Keshav. REWIRE: an optimization-based framework for data center network design. In *INFOCOM*, 2012.

[10] R. W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5:345–, June 1962.

[11] M. B. Habbal, H. N. Koutsopoulos, and S. R. Lerman. A decomposition algorithm for the all-pairs shortest path problem on massively parallel computer architectures. *Transportation Science*, 28(4):292–308, 1994.

[12] J.-W. Hong and H. T. Kung. I/O complexity: The red-blue pebble game. In *Proceedings of the thirteenth annual ACM symposium on Theory of computing*, STOC '81, pages 326–333, New York, NY, USA, 1981. ACM.

[13] D. Irony, S. Toledo, and A. Tiskin. Communication lower bounds for distributed-memory matrix multiplication. *Journal of Parallel and Distributed Computing*, 64(9):1017–1026, 2004.

[14] J. Jenq and S. Sahni. All pairs shortest paths on a hypercube multiprocessor. In *ICPP '87: Proc. of the Intl. Conf. on Parallel Processing*, pages 713–716, 1987.

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

[16] V. Kumar and V. Singh. Scalability of parallel algorithms for the all-pairs shortest-path problem. *J. Parallel Distrib. Comput.*, 13:124–138, 1991.

[17] R. C. Larson and A. R. Odoni. *Urban operations research*. Prentice-Hall, Englewood Cliffs, NJ, USA, 1981.

[18] R. J. Lipton, D. J. Rose, and R. E. Tarjan. Generalized nested dissection. *SIAM J. Numer. Analysis*, 16:346–358, 1979.

[19] J. D. McCalpin. STREAM: Sustainable Memory Bandwidth in High Performance Computers. `http://www.cs.virginia.edu/stream/`.

[20] U. Meyer and P. Sanders. Δ-stepping: a parallelizable shortest path algorithm. *J. Algorithms*, 49(1):114–152, 2003.

[21] J.-S. Park, M. Penner, and V. K. Prasanna. Optimizing graph algorithms for improved cache performance. *IEEE Transactions on Parallel and Distributed Systems*, 15(9):769–782, 2004.

[22] R. Seidel. On the all-pairs-shortest-path problem in unweighted undirected graphs. *Journal of Computer and System Sciences*, 51(3):400–403, 1995.

[23] E. Solomonik, A. Bhatele, and J. Demmel. Improving communication performance in dense linear algebra via topology aware collectives. In *Supercomputing, Seattle, WA, USA*, Nov 2011.

[24] E. Solomonik and J. Demmel. Communication-optimal 2.5D matrix multiplication and LU factorization algorithms. In *Lecture Notes in Computer Science, Euro-Par, Bordeaux, France*, Aug 2011.

[25] J. D. Ullman and M. Yannakakis. High probability parallel transitive-closure algorithms. *SIAM Journal of Computing*, 20:100–125, February 1991.

[26] R. A. Van De Geijn and J. Watts. SUMMA: scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience*, 9(4):255–274, 1997.

[27] S. Warshall. A theorem on boolean matrices. *J. ACM*, 9:11–12, January 1962.

8 Appendix A: Derivation of 2D DC-APSP communication costs

We use all processors at every recursive step (up to $n = \sqrt{p}$), in a cyclic layout. We will not consider the costs for $n > \sqrt{p}$ since, typically $n^2 >> p$[1]. Every processor will do an equal amount of flops in such a cyclic layout, and no extra flops are done. Therefore, the flops cost is

$$F_{\text{c-2D}} = O(n^3/p)$$

At each recursive level of the algorithm six SMMM operations are performed. Therefore, if all p processors are used at every step (cyclic layout), the bandwidth cost of 2D DC-APSP, $W_{\text{c-2D}}$, is given by the recurrence

$$\begin{aligned} W_{\text{c-2D}}(n,p) &= 2 \cdot W_{\text{c-2D}}(n/2,p) + 6 \cdot W(\text{SMMM-2D}) \\ W_{\text{c-2D}}(n,p) &= 2 \cdot W_{\text{c-2D}}(n/2,p) + O(n^2/\sqrt{p}) \\ &= O(n^2/\sqrt{p}). \end{aligned}$$

The latency cost of cyclic 2D DC-APSP, $S_{\text{c-2D}}$, is given by

$$\begin{aligned} S_{\text{c-2D}}(n,p) &= 2 \cdot S_{\text{c-2D}}(n/2,p) + O(\sqrt{p}) \\ &= O(n \cdot \sqrt{p}). \end{aligned}$$

If we use a blocked layout, we perform work with fewer processors at each level of recursion. The flops are no longer balanced among processors. Since $n^2 >> p$, the recursion tree has height $\log_4(p)$, making the base case $F_{\text{b-2D}}(n/\sqrt{p}, 1) = O(n/\sqrt{p})^3$. We calculate the computational cost of this blocked layout via the recurrence

$$\begin{aligned} F_{\text{b-2D}}(n,p) &= 2 \cdot F_{\text{b-2D}}(n/2,p/4) + O(n^3/p) \\ &= O(\frac{n^3}{p}\log(p)) \end{aligned}$$

The bandwidth cost of the blocked algorithm is,

$$\begin{aligned} W_{\text{b-2D}}(n,p) &= 2 \cdot W_{\text{b-2D}}(n/2,p/4) + O(n^2/\sqrt{p}) \\ &= O(\frac{n^2}{\sqrt{p}}\log(p)). \end{aligned}$$

The latency cost of the blocked algorithm is

$$\begin{aligned} S_{\text{b-2D}}(n,p) &= 2 \cdot S_{\text{b-2D}}(n/2,p/4) + O(\sqrt{p}) \\ &= O(\sqrt{p}\log(p)). \end{aligned}$$

[1] In a weak scaling study, n would scale with $\Theta(\sqrt{p})$ with a constant of $10,000$ (the biggest problem solvable by a typical single core with 1GB memory)

So, the cyclic approach achieves a desirable computational and bandwidth cost, while the blocked approach achieves a lower latency. We can balance these costs via a block-cyclic decomposition. We set the block-cyclic factor $r = O(\log\log(p))$ so that each processor owns a r-by-r grid of blocks of dimension $n/(\sqrt{p} \cdot r)$. This way, after r levels of recursion, all the processors are still going to be active The costs of this algorithm can be calculated by plugging in the cost of the blocked algorithm as the base-case of the cyclic algorithm. The computational cost is

$$\begin{aligned}
F_{\text{bc-2D}}(n,p) &= 2 \cdot F_{\text{bc-2D}}(n/2,p) + O(n^3/p) \\
F_{\text{bc-2D}}(n/2^{\log\log(p)},p) &= F_{\text{b-2D}}(n/2^{\log\log(p)},p) \\
F_{\text{bc-2D}}(n,p) &= O(n^3/p) + O(2^{\log\log(p)} \cdot ((n/2^{\log\log(p)})^3/p)\log(p)) \\
&= O(n^3/p) + O(\log(p) \cdot ((n/\log(p))^3/p)\log(p)) \\
&= O(n^3/p) + O((n^3/p)/\log(p)) \\
&= O(n^3/p)
\end{aligned}$$

We can calculate the bandwidth and latency costs of the block-cyclic algorithm in a similar fashion. The bandwidth cost is

$$\begin{aligned}
W_{\text{bc-2D}}(n,p) &= 2 \cdot W_{\text{bc-2D}}(n/2,p) + O(n^2/\sqrt{p}) \\
W_{\text{bc-2D}}(n/2^{\log\log(p)},p) &= W_{\text{b-2D}}(n/2^{\log\log(p)},p) \\
W_{\text{bc-2D}}(n,p) &= O(n^2/\sqrt{p}) + O(2^{\log\log(p)} \cdot ((n/2^{\log\log(p)})^2/\sqrt{p})\log(p)) \\
&= O(n^2/\sqrt{p}) + O(\log(p) \cdot ((n/\log(p))^2/\sqrt{p})\log(p)) \\
&= O(n^2/\sqrt{p}) + O(n^2/\sqrt{p}) \\
&= O(n^2/\sqrt{p})
\end{aligned}$$

And the latency cost is

$$\begin{aligned}
S_{\text{bc-2D}}(n,p) &= 2 \cdot S_{\text{bc-2D}}(n/2,p) + O(\sqrt{p}) \\
S_{\text{bc-2D}}(n/2^{\log\log(p)},p) &= S_{\text{b-2D}}(n/2^{\log\log(p)},p) \\
S_{\text{bc-2D}}(n,p) &= O(\sqrt{p}\log(p)) + O(\sqrt{p}\log^2(p)) \\
&= O(\sqrt{p}\log^2(p))
\end{aligned}$$

9 Appendix B: Derivation of 2.5D DC-APSP communication costs

The 2.5D block-cyclic DC-APSP algorithm (Algorithm 7) consists of 4-nested DC-APSP algorithms. At the top level is a block-cyclic 2.5D algorithm, which recursively calls a blocked 2.5D algorithm, which calls the block-cyclic 2D algorithm. We define two blocking parameters b_1 and b_2 to determine when to switch between blocked and cyclic algorithms.

If we set $b_1 = n/c$, the bandwidth cost of 2.5D DC-APSP is the following recurrence,

$$\begin{aligned}
W_{\text{bc-2.5D}}(n > n/c, p, c) &= 2W_{\text{bc-2.5D}}(n/2, p, c) + O(n^2/\sqrt{pc}) \\
W_{\text{bc-2.5D}}(n < n/c, p, c) &= W_{\text{b-2.5D}}(n, p, c) \\
W_{\text{b-2.5D}}(n, p, c > 1) &= 2W_{\text{b-2.5D}}(n/2, p/8, c/2) + O(n^2/\sqrt{pc}) \\
W_{\text{b-2.5D}}(n, p, c = 1) &= W_{\text{bc-2D}}(n, p) \\
W_{\text{bc-2D}}(n, p) &= O(n^2/\sqrt{p}) \\
W_{\text{b-2.5D}}(n, p, c) &= c \cdot W_{\text{bc-2D}}(n/c, p/c^3) + O(\sqrt{c} \cdot n^2/\sqrt{p}) \\
&= O(c \cdot (n/c)^2/\sqrt{p/c^3}) + O(\sqrt{c} \cdot n^2/\sqrt{p}) \\
&= O(\sqrt{c} \cdot n^2/\sqrt{p}) \\
W_{\text{bc-2.5D}}(n, p, c) &= c \cdot W_{\text{b-2.5D}}(n/c, p, c) + O(n^2/\sqrt{pc}) \\
&= O(c \cdot \sqrt{c} \cdot (n/c)^2/\sqrt{p}) + O(n^2/\sqrt{pc}) \\
&= O(n^2/\sqrt{pc})
\end{aligned}$$

We can derive the latency cost in a similar fashion,

$$\begin{aligned}
S_{\text{bc-2.5D}}(n > n/c, p, c) &= 2S_{\text{bc-2.5D}}(n/2, p, c) + O(\sqrt{p/c^3}) \\
S_{\text{bc-2.5D}}(n < n/c, p, c) &= S_{\text{b-2.5D}}(n, p, c) \\
S_{\text{b-2.5D}}(n, p, c > 1) &= 2S_{\text{b-2.5D}}(n/2, p/8, c/2) + O(\sqrt{p/c^3}) \\
S_{\text{b-2.5D}}(n, p, c = 1) &= S_{\text{bc-2D}}(n, p) \\
S_{\text{bc-2D}}(n, p) &= O(\sqrt{p}\log^2(p)) \\
S_{\text{b-2.5D}}(n, p, c) &= c \cdot S_{\text{bc-2D}}(n/c, p/c^3, 1) + O(c \cdot \sqrt{p/c^3}) \\
&= O(c \cdot \sqrt{p/c^3}\log^2(p/c^3)) + O(\sqrt{p/c}) \\
&= O(\sqrt{p/c}\log^2(p/c^3)) \\
S_{\text{bc-2.5D}}(n, p, c) &= c \cdot S_{\text{b-2.5D}}(n/c, p, c) + O(c \cdot \sqrt{p/c^3}) \\
&= O(c \cdot \sqrt{p/c}\log^2 p/c^3) + O(\sqrt{p/c}) \\
&= O(\sqrt{pc}\log^2(p/c^3))
\end{aligned}$$