Minimizing communication in all-pairs shortest paths

Edgar Solomonik[1], Aydın Buluç[3], James Demmel[1,2]

[1] Department of EECS, UC Berkeley
[2] Department of Mathematics, UC Berkeley
[3] Computational Research Division, Lawrence Berkeley National Laboratory

May 21, 2013

Outline

All-pairs shortest-paths problem

Definition (All-pairs shortest-paths)

Given graph $G = (V, E)$ find the shortest-path distances d_{ij} between each pair of vertices $v_i, v_j \in V$.

We will treat the graph according to its matrix formulation

- A is the adjacency matrix corresponding to edges E
- D is the (path) distance matrix corresponding to G
- even if A is sparse, D may be completely dense

Floyd-Warshall algorithm

Compute shortest paths between each pair of vertices using intermediate nodes $\{1, 2, \ldots k\}$,

$D = \text{FLOYD-WARSHALL}(A, n)$

$D = A$
for $k = 1$ **to** n
  for $i = 1$ **to** n
    for $j = 1$ **to** n
      $d_{ij} = \min(d_{ij}, d_{ik} + d_{kj})$

Reordering the outer loop changes the semantics of the algorithm, but reordering the two inner loops does not.

The min-plus semiring

The all-pairs shortest-paths computation can be expressed via matrix multiplication on a different semiring, which we denote as

$$a \odot b \rightarrow \min(a + b)$$
$$c := a \odot b \rightarrow c = \min(c, a + b)$$
$$C := A \odot B \rightarrow c_{ij} = \min(c_{ij}, \min_k(a_{ik} + b_{kj}))$$

$D = \text{FLOYD-WARSHALL}(A, n)$

$D = A$

for $k = 1$ **to** n

$D := D[:, k] \odot D[k, :]$

Parallel Floyd-Warshall

We can parallelize Floyd-Warshall by distributing A and D in blocks on a 2D grid

$D = \text{2D-FLOYD-WARSHALL}(A, \Pi[1:\sqrt{p}, 1:\sqrt{p}], n, p)$

$D = A$
for $k = 1$ **to** n
Broadcast $D[:, k]$ along processor rows
Broadcast $D[k, :]$ along processor columns
$D := D[:, k] \odot D[k, :]$

Bandwidth cost is $W = O(n^2/\sqrt{p})$ bytes, latency cost is $S = O(n)$ messages.

Divide-and-conquer all-pairs shortest-paths algorithm

Kleene's algorithm [Aho, Hopcroft, Ullman 1974]

$D = \text{DC-APSP}(A, n)$

$D = A$

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

$D_{11} = \text{DC-DPSP}(D_{11}, n/2)$

$D_{12} = D_{11} \odot D_{12}$

$D_{21} := D_{21} \odot D_{11}$

$D_{22} := D_{21} \odot D_{12}$

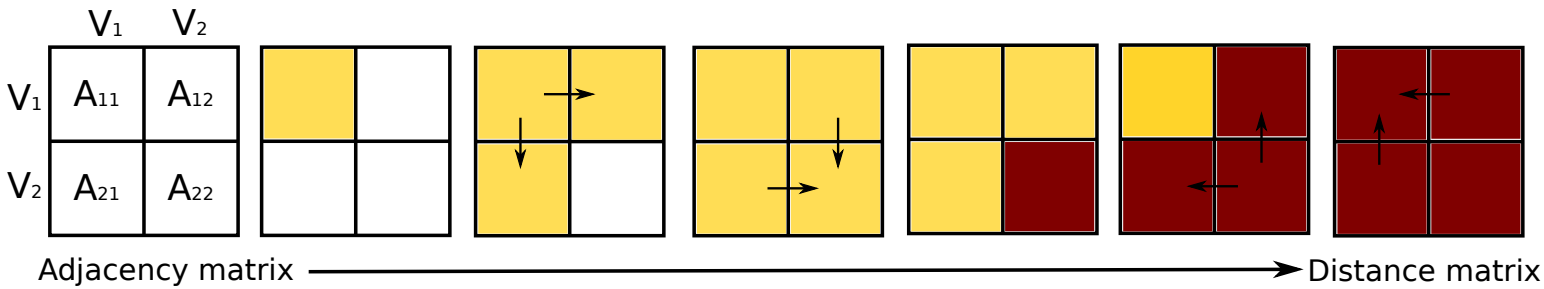
$D_{22} = \text{DC-DPSP}(D_{22}, n/2)$

$D_{21} := D_{22} \odot D_{21}$

$D_{12} := D_{12} \odot D_{22}$

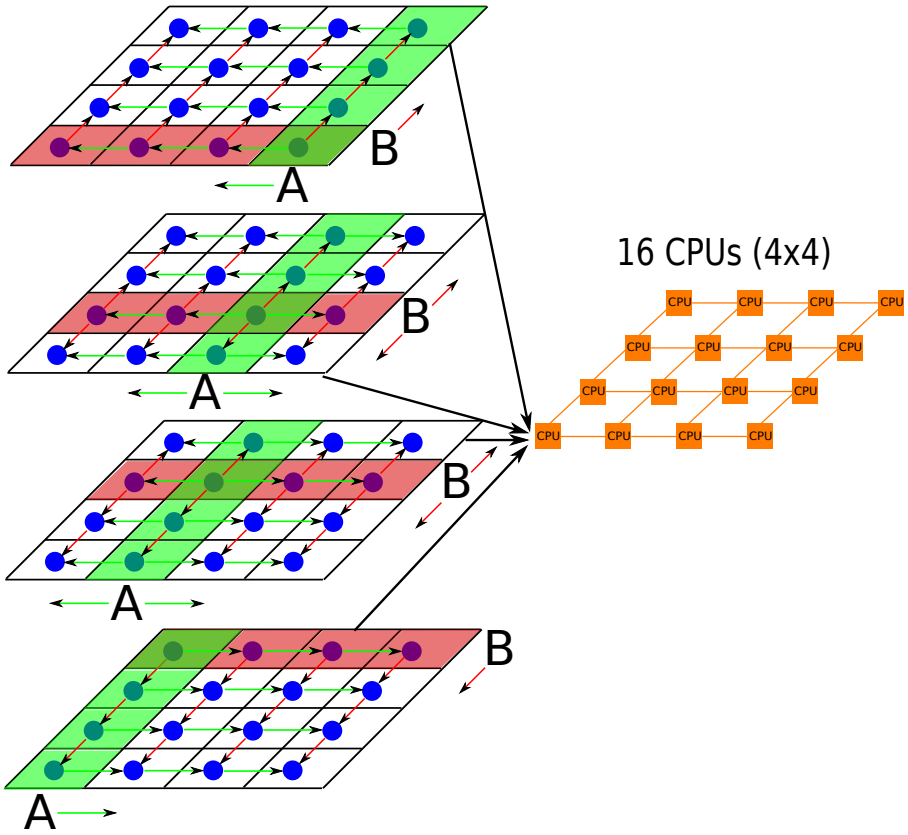
$D_{11} := D_{12} \odot D_{21}$

Divide-and-conquer all-pairs shortest-paths algorithm



2D matrix multiplication

[Cannon 69], [Agarwal et al 95], [Van De Geijn and Watts 97]



$O(n^3/p)$ flops

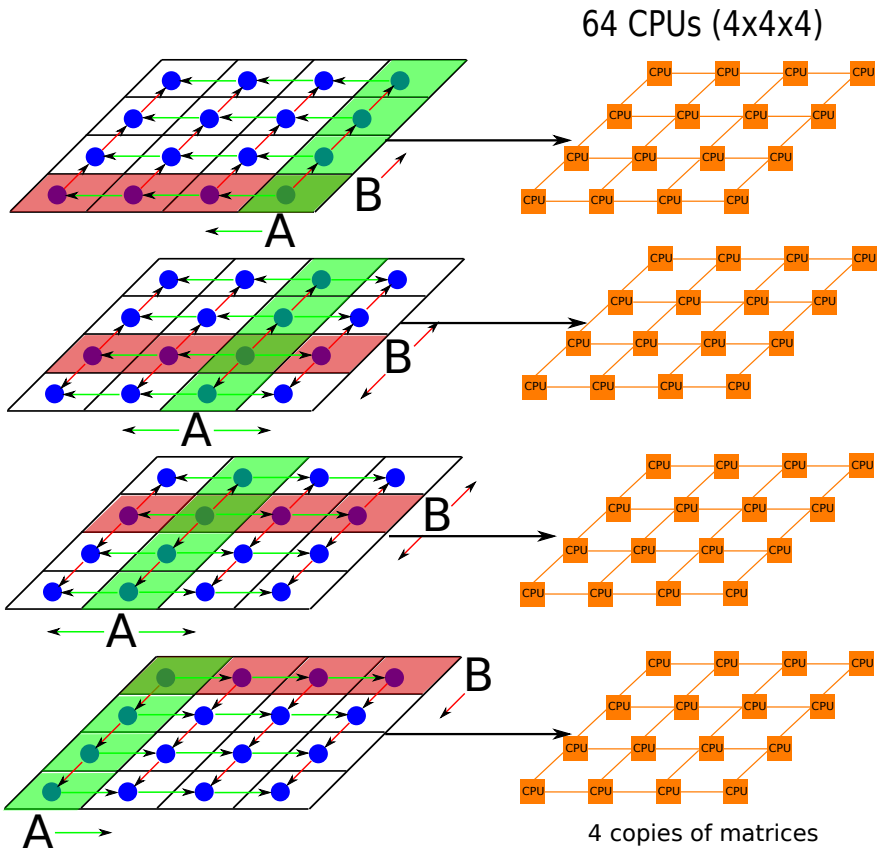
$O(n^2/\sqrt{p})$ words moved

$O(\sqrt{p})$ messages

$O(n^2/p)$ bytes of memory

3D matrix multiplication

[Agarwal et al 95],
[Aggarwal, Chandra, and Snir 90],
[Bernsten 89], [McColl and Tiskin 99]



$O(n^3/p)$ flops

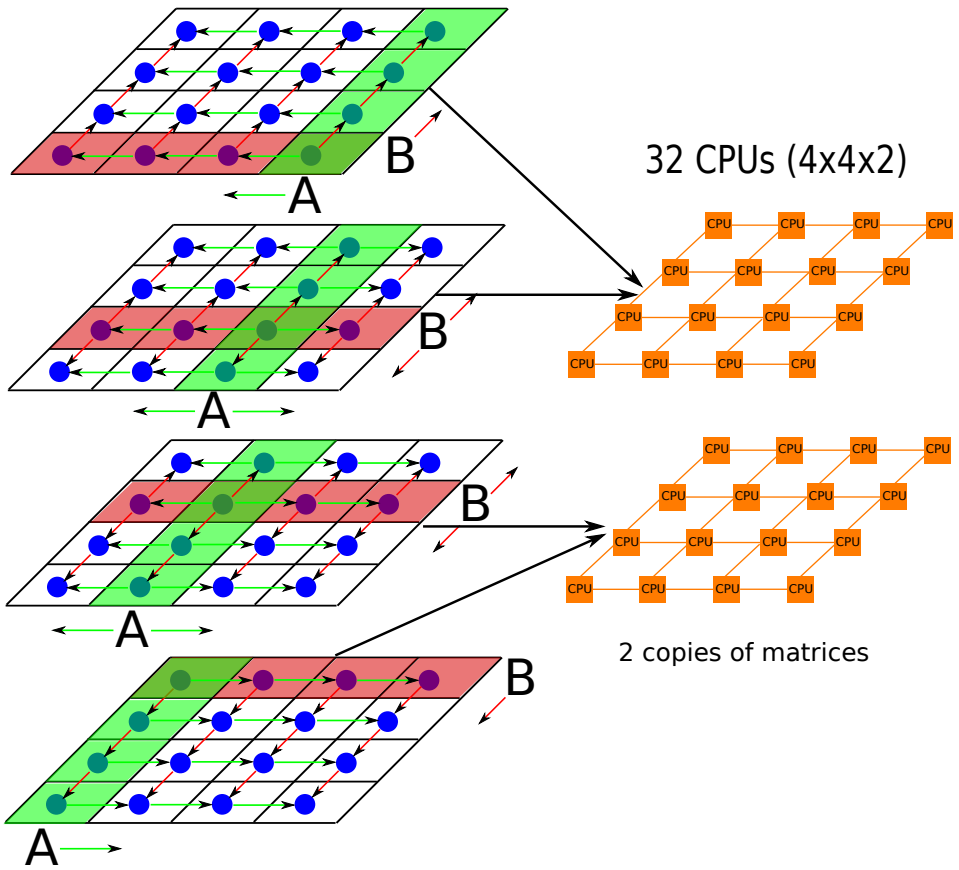
$O(n^2/p^{2/3})$ words moved

$O(1)$ messages

$O(n^2/p^{2/3})$ bytes of memory

2.5D matrix multiplication

[McColl and Tiskin 99]



$O(n^3/p)$ flops

$O(n^2/\sqrt{c \cdot p})$ words moved

$O(\sqrt{p/c^3})$ messages

$O(c \cdot n^2/p)$ bytes of memory

2.5D divide-and-conquer blocked APSP algorithm

$D = \text{2.5D-B-DC-APSP}(A, \Pi, n, p, c)$

Let Π_{ijk} for $i, j, k \in \{1, 2\}$ denote an octant of the processor grid

$D = A$

Π_{111} computes $D_{11} = \text{2.5D-B-DC-APSP}(D_{11}, \Pi_{111}, n/2, p/8)$

Π_{111} sends D_{11} to Π_{121} and Π_{211}.

Π_{121} computes $D_{12} := D_{11} \odot D_{12}$

Π_{211} computes $D_{21} := D_{21} \odot D_{11}$

Π_{121} sends D_{12} to Π_{221}.

Π_{211} sends D_{21} to Π_{221}.

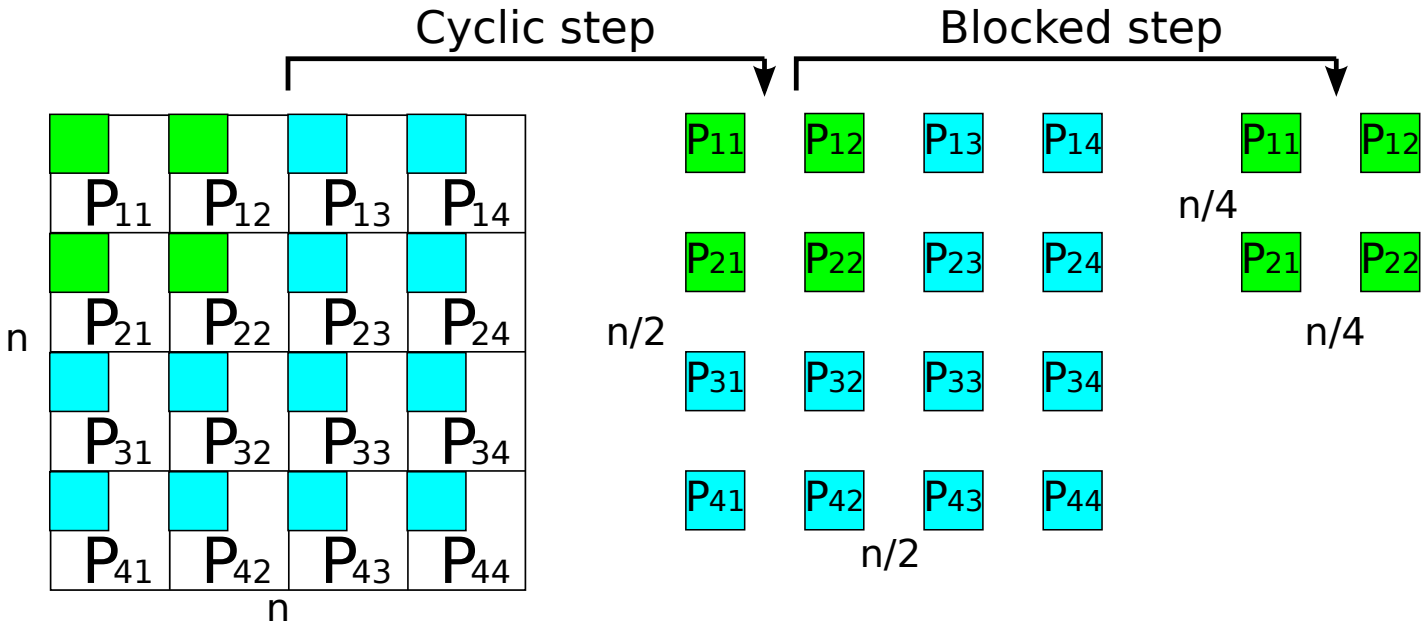
Π_{221} computes $D_{22} := D_{12} \odot D_{21}$

Π_{221} computes $D_{22} = \text{2.5D-B-DC-APSP}(D_{22}, \Pi_{221}, n/2, p/8)$

...

Problem: not load balanced

Blocked and cyclic steps



2.5D divide-and-conquer block-cyclic APSP algorithm

$D = \text{2.5D-BC-DC-APSP}(A, \Pi, n, p, c, b)$

if $n \leq b$

$D = \text{2.5D-B-DC-APSP}(A, \Pi, n, p, c)$

else

$D = A$

Let D_{ij} for $i, j \in [1, 2]$ denote a local sub-block of D

Π computes $D_{11} = \text{2.5D-BC-DC-APSP}(D_{11}, \Pi, n/2, p)$

Π computes $D_{12} := D_{11} \odot D_{12}$

Π computes $D_{21} := D_{21} \odot D_{11}$

Π computes $D_{22} := D_{12} \odot D_{21}$

Π computes $D_{22} = \text{2.5D-BC-DC-APSP}(D_{22}, \Pi, n/2, p)$

...

Communication costs of 2.5D-BC-DC-APSP

Given local available memory of size $M = \Omega(c \cdot n^2/p)$, our algorithm achieves the following complexity (F is flops, W is bandwidth cost, S is latency cost)

$$F_{\text{bc-2.5D}}(n,p) = O(n^3/p)$$
$$W_{\text{bc-2.5D}}(n,p) = O(n^2/\sqrt{c \cdot p})$$
$$S_{\text{bc-2.5D}}(p) = O(\sqrt{c \cdot p}\log^2(p))$$

Modulo polylog factors this 2.5D algorithm is optimal to the best of our knowledge.

Intra-node optimizations

For the divide-and-conquer algorithm, we implemented our own matrix multiplication for the min-plus semiring

- two-level cache blocking
- register blocking
- explicit SIMD intrinsics
- OpenMP threading based on L1-level blocking

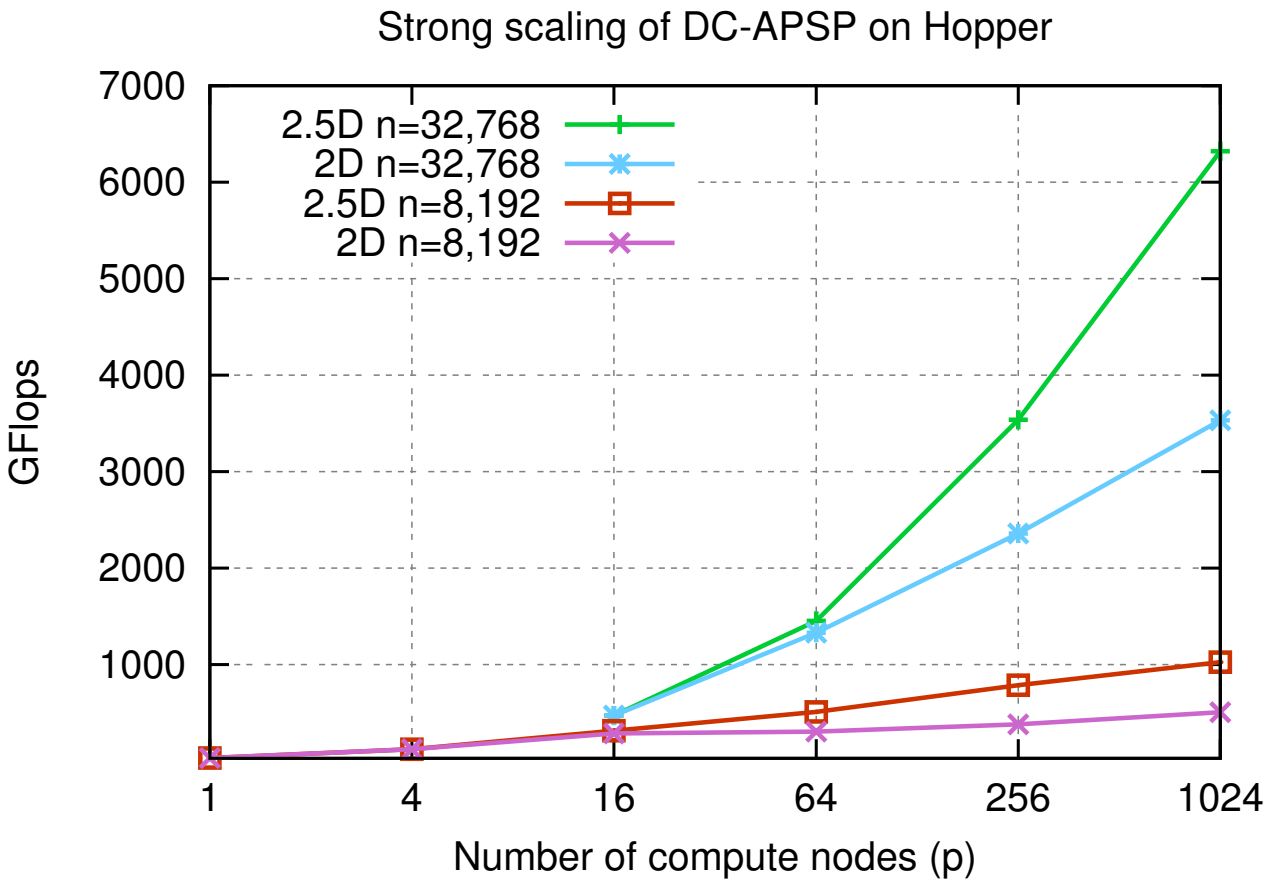
Achieves 25% of theoretical peak on 6-cores of Hopper (peak based on fused multiply-add potential)

Distributed-memory implementation

Divide and conquer APSP algorithm

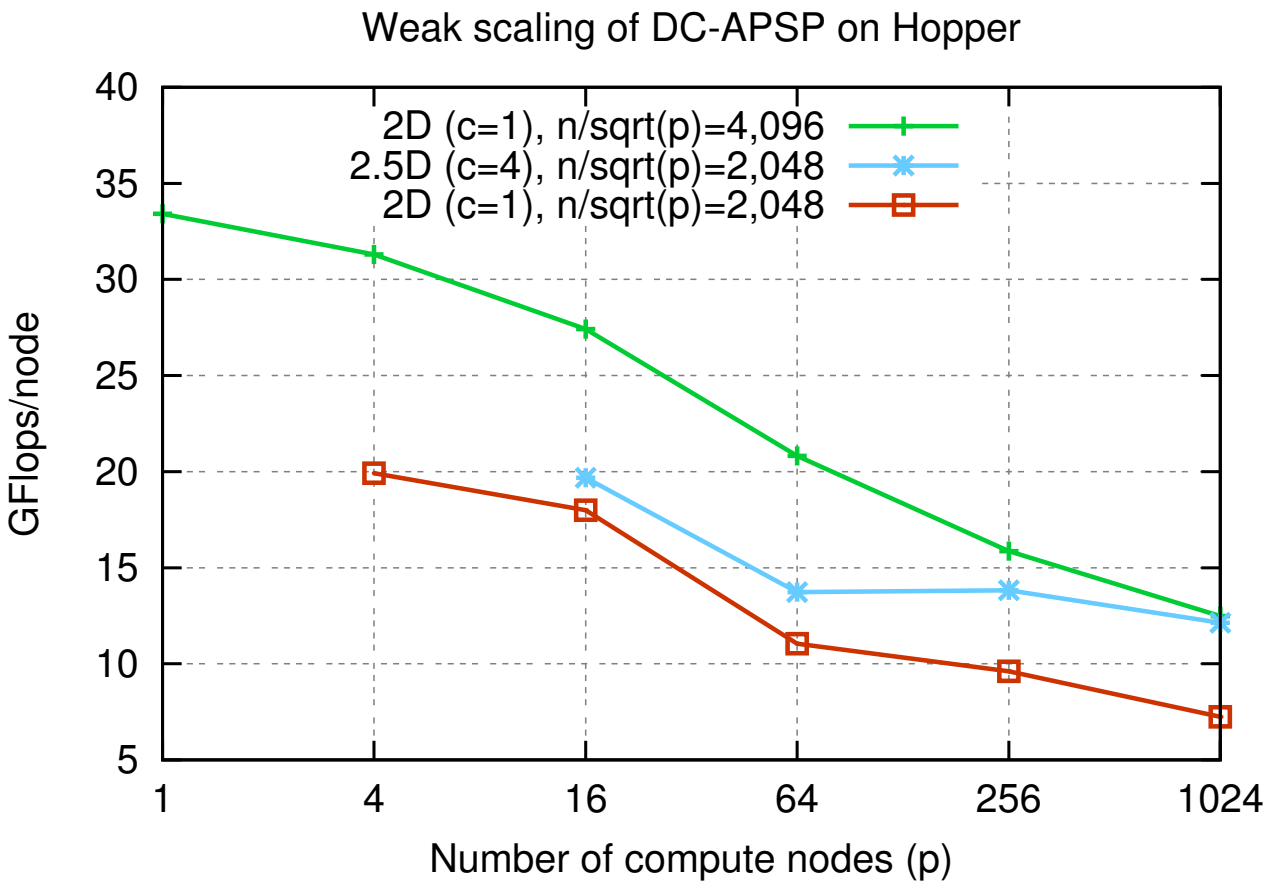
- two-level MPI/OpenMP parallization (4 nodes / 6 threads on Hopper)
- MPI point-to-point used for DC-APSP sends
- MPI broadcasts used for (min-plus) matrix multiplication
- parameterized control for block sizes and replication

Divide-and-Conquer APSP strong scaling

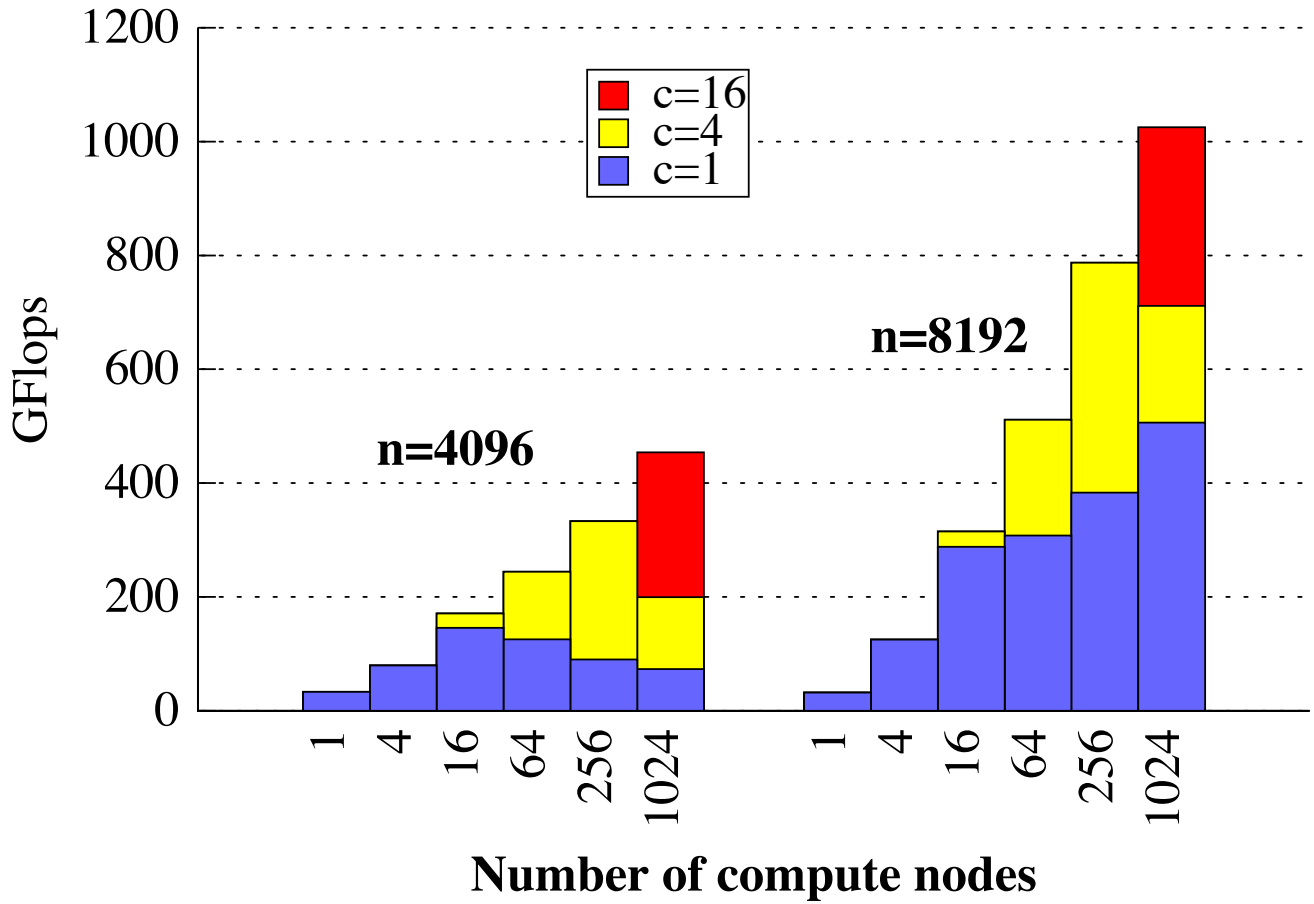


For $n = 32,768$ a 1.8X speed-up is obtained on 1024 nodes

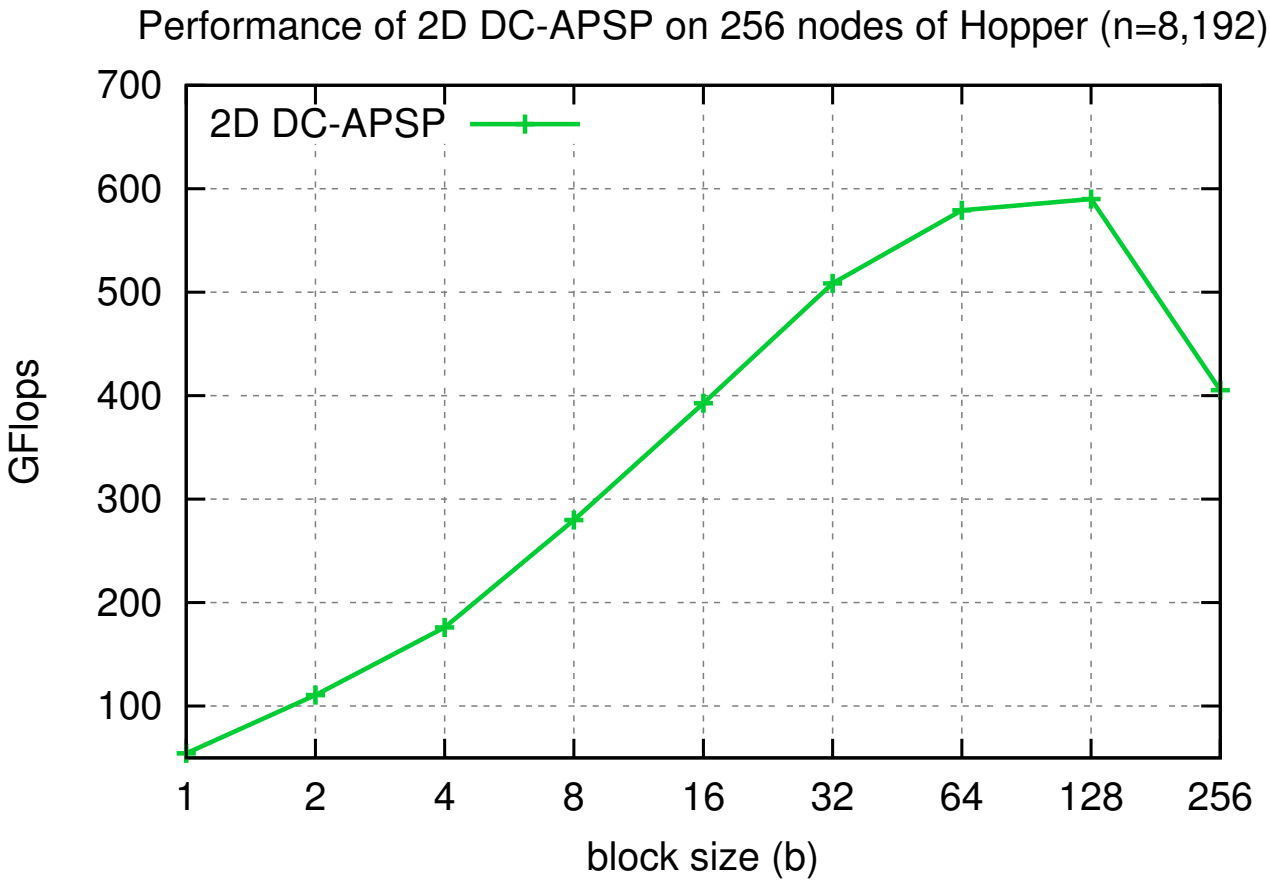
Divide-and-Conquer APSP weak scaling



Divide-and-Conquer APSP on small matrices



Divide-and-Conquer APSP block size sensitivity



Alternative algorithms

Other algorithms can exploit sparsity more efficiently. For instance, we can

1. Replicate the graph
2. Run Johnson's (Dijkstra's) algorithm from a different source node in parallel

For a 16K vertex 5% dense graph, on 384 cores replicated Johnson's algorithm is 10% faster than ours.

Accelerator potential

The main intra-node workload for the divide-and-conquer APSP algorithm is min-plus matrix multiplication

- easy to implement and achieve high performance on vector-based machines/accelerators
- low latency overhead ($O(\sqrt{pc})$ latency)

On 4 nodes of Hopper for a 8,192 vertex graph, our performance matched a single-Fermi APSP implementation. A many-GPU implementation should be promising and simple.

Summary

APSP algorithms:

- Floyd-Warshall: simplest with minimum computational complexity
- Divide-and-Conquer APSP: easier to parallelize
- better algorithms may be specifically designed for particular sparse graphs

Distributed-memory APSP implementation

- reduces latency cost by using recursive algorithm
- reduces bandwidth cost by using data replication
- two-level parallelization scales on Hopper (Cray XE6)
- suitable for GPU/accelerator parallelization

Backup slides