Distributed-Memory Algorithms for Cardinality Matching using Matrix Algebra

Ariful Azad, Lawrence Berkeley National Laboratory
Joint work with Aydın Buluç (LBNL)

Support: DOE Office of Science

SIAM PP 2016, Paris
Matching: A subset of independent edges, i.e., at most one edge in the matching is incident on each vertex.

Maximal cardinality matching: A matching where if another edge is added it is not a matching anymore.

Maximum cardinality matching (MCM) has the maximum possible cardinality

Maximal Cardinality Matching
Cardinality = 2

Maximum Cardinality Matching
Cardinality = 3
Application of matching in scientific computing

Bipartite Cardinality Matching
- Maximal Cardinality Matching (1/2 approx.)
- Maximum Cardinality Matching (MCM)
- Block Triangular Form (BTF)
- Sparse QR
- KLU
- Least square on HBS matrices

Bipartite Weighted Matching
- Maximum-Weight Perfect Matching (MWPM) exact/approx.
- Maximum Weighted Matching (MWM) exact/approx.
- Sparse LU
- Pre-conditioning
- HSS-based multifrontal solver
- Graph Partitioning
- AMG

use relationship
Scope of this talk

- **Problem**: Cardinality matching in a bipartite graph
  - Maximum cardinality matching (**MCM**)
  - Maximal cardinality matching (used to initialize MCM)

- **Algorithm**: Distributed-memory parallel algorithms

- **Approach**: Matrix-algebraic formulations of graph primitives. Inspired by Graph BLAS ([http://graphblas.org/](http://graphblas.org/)).
  - More discussion on Friday (MS68): The GraphBLAS Effort: Kernels, API, and Parallel Implementations by Aydin Buluc.

- **Covers two recent papers**:
  - Maximal matching: Azad and Buluç, IEEE CLUSTER 2015
  - Maximum matching: Azad and Buluç, IPDPS 2016
MCM algorithm based on augmenting-path searches

- **Augmenting path**: A path that alternates between matched and unmatched edges with unmatched end points.

- **Algorithm**: Search for augmenting paths and flip edges across the paths to increase cardinality of the matching.
  - **Algorithmic options**: single source or multi-source, breadth-first search (BFS) or depth-first search (DFS)
## Algorithmic landscape of cardinality matching

*Duff, Kaya and Ucar (ACM TOMS 2011), Azad, Buluç, Pothen (TPDS 2016)*

<table>
<thead>
<tr>
<th>Class</th>
<th>Search strategy</th>
<th>Serial Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Maximum cardinality matching</strong></td>
<td>DFS or BFS</td>
<td>$O(nm)$</td>
</tr>
<tr>
<td>Single-source augmenting path search</td>
<td>DFS w lookahead (Pothen-Fan)</td>
<td>$O(nm)$</td>
</tr>
<tr>
<td>Multi-source augmenting path search</td>
<td>BFS (MS-BFS)</td>
<td>$O(nm)$</td>
</tr>
<tr>
<td></td>
<td>DFS &amp; BFS (Hopcroft-Karp)</td>
<td>$O(vnm)$</td>
</tr>
<tr>
<td>Push relabel</td>
<td>Label guided FIFO search</td>
<td>$O(nm)$</td>
</tr>
<tr>
<td><strong>Maximal cardinality matching</strong></td>
<td>Greedy Karp-Sipser Dynamic mindegree</td>
<td>Local</td>
</tr>
</tbody>
</table>

**Hopcroft-Karp:** best asymptotic complexity

**MS-BFS:** exposes more parallelism

**Our focus:**

- Initializes MCM
The need for distributed-memory algorithms

- When a graph does not fit in the memory of a node
- The graph is already distributed
  - Example: static pivoting in SuperLU_DIST (Li and Demmel, 2003)
  - The graph is gathered on a single node and MC64 is used to compute the matching, which is unscalable and expensive

Time to gather a graph and scatter the matching on 2048 cores of NERSC/Edison (Cray XC30)

Distributed algorithms are cheaper and scalable
Distributed-memory cardinality matching

- Prior work: **Push-relabel** by Langguth *et al.* (2011) and **Karp-Sipser** on general graph by Patwary *et al.* (2010).
  - does not scale beyond 64 processors

- Challenge
  - long paths passing through multiple processors
  - lots of fine-grained asynchronous communication

- Here we use **graph-matrix duality** and design matching algorithms using scalable matrix and vector operations.
  - A handful of standard operations
  - Offers bulk-synchronous parallelism
  - Jumping among algorithms is easier
Two required primitives

1. Sparse matrix-sparse vector multiply (SpMSpV)

Semiring Option: (multiply, add)
(select2nd, min)

Unmatched columns: \( f_c \)

Select columns

Matrix \( A \)

In each row, retain the minimum product from the selected columns

Graph Operation
Traverse unvisited neighbors

Matrix Operation
SpMSpV

A matching

Bipartite graph \( G(R,C,E) \)
Two required primitives

2. Inverted index in a sparse vector

Graph Operation
1. Keep unique child
2. Swap matched and unmatched edges

Vector Operation
Inverted index in a sparse vector

Swap parents and children
Duplicates removed

Invert

Index: child
Value: parent

Index: parent
Value: child
Multi-source BFS (MS-BFS) algorithm using matrix and vector operations

Step-1: Discover vertex-disjoint augmenting paths

(a) A maximal matching in a Bipartite Graph

- Not explored to maintain vertex-disjoint trees

(b) Alternating BFS Forest

- Roots of BFS trees
- Sparse matrix-sparse vector multiply (SpMSpV)
- Inverted index using matching vector
- Sparse matrix-sparse vector multiply (SpMSpV)
MS-BFS algorithm using matrix and vector operations

Step-2: Augment matching by flipping matched and unmatched edges along the augmenting paths

Augment matching
ALGORITHM:
1. Gather vertices in *processor column* [communication]
2. Local multiplication [computation]
3. Find owners of the current frontier’s adjacency and exchange adjacencies in *processor row* [communication]
Shared-memory parallelization (SpMSSpV)

- Explicitly split local submatrices to $t$ (#threads) pieces along the rows.
## Computation and communication time of discovering vertex-disjoint augmenting paths (a phase)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Per processor Computation (lower bound)</th>
<th>Per processor Comm (latency)</th>
<th>Per processor Comm (bandwidth)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SpMSpV</td>
<td>$\frac{m}{p}$</td>
<td>$height \cdot \alpha \sqrt{p}$</td>
<td>$\beta \left( \frac{m}{p} + \frac{n}{\sqrt{p}} \right)$</td>
</tr>
<tr>
<td>Invert</td>
<td>$\frac{n}{p}$</td>
<td>$height \cdot \alpha p$</td>
<td>$\beta \frac{n}{p}$</td>
</tr>
</tbody>
</table>

$n$: number of vertices, $m$: number of edges

$height$: maximum height of the BFS forest

$\alpha$: latency (0.25 μs to 3.7 μs MPI latency on Edison)

$\beta$: inverse bandwidth (~8GB/sec MPI bandwidth on Edison)

$p$: number of processors
Special treatments for long augmenting paths

Level synchronous: BFS Style

One path per process
Using one-sided communication via MPI Remote Memory Access (RMA)
Results: experimental Setup

- **Platform: Edison (NERSC)**
  - 2.4 GHz Intel Ivy Bridge processor, 24 cores (2 sockets) and 64 GB RAM per node
  - Cray Aries network using a Dragonfly topology (0.25 µs to 3.7 µs MPI latency, ~8GB/sec MPI bandwidth)
  - Programming environment: C++ and Cray MPI, *Combinatorial BLAS* library (Buluc and Gilbert, 2011)

- **Input graphs**
  - **Real matrices** from Florida sparse matrix collection and randomly generated matrices.
  - Matrix- bipartite graph conversion
    - rows: vertices in one part, **columns**: vertices in another part, nonzeros: edges.
Karp-Sipser obtains the **highest cardinality** for many practical problems, but it runs the **slowest** on high concurrency.

We found that dynamic mindegree + MCM often runs the fastest on high concurrency.
MCM strong scaling (real matrices)

1 node
(24 cores of Edison)

12x-18x speedups

\~80x increase of cores

To appear: Azad and Buluç, IPDPS 2016
MCM strong scaling (G500 RMAT matrices)

Scale-30 RMAT: 2 billion vertices, 32 billion edges
Scaling continues beyond 10K core on Large matrices
MCM: Breakdown of runtime

GL7d19

V1 = 1.91M, V2 = 1.96M, #edges = 37M
Ideas for weighted matching

- Similar graph-matrix transformation applies to weighted matching algorithms.

- Auction algorithm ideas [Ongoing work]
  - Bidders bid for most profitable objects: $\text{SpMSpV with (select2nd, max) semiring}$
  - An object selects the best bidder from which it received bid: $\text{Inverted index}$
  - Dual updates can be done using vector operations
Summary of contributions

- **Methods**: distributed memory matching algorithms based on matrix algebra
- **Performance**: scales up to 10K cores on large graphs.
- Easy to implement an algorithm using matrix-algebraic primitives.
- Source code publicly available at:
  

Future work

- Distributed weighted matching using matrix algebra

A. Azad and A. Buluç, CLUSTER 2015, Distributed-memory algorithms for maximal cardinality matching using matrix algebra.

Langguth et al., Parallel Computing 2011, Parallel algorithms for bipartite matching problems on distributed memory computers.

M. Patwary, R. Bisseling, F. Manne, HPPA 2010, Parallel greedy graph matching using an edge partitioning approach.


Thanks for your attention
Supporting slides
Maximal matching algorithms using matrix and vector operations

- Used to initialize MCM
- Example: dynamic mindegree algorithm
  - Greedy and Karp-Sipser are similar (Azad and Buluc, 2015)

Matrix

<table>
<thead>
<tr>
<th>Matrix Op</th>
<th>SpMSpv Addion = min (degree)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Graph Op</td>
<td>neighbor with mindegree</td>
</tr>
<tr>
<td></td>
<td>d=2 x_1 → y_1 d=3</td>
</tr>
<tr>
<td></td>
<td>d=2 x_2 → y_2 d=2</td>
</tr>
<tr>
<td></td>
<td>d=3 x_3 → y_3 d=2</td>
</tr>
</tbody>
</table>

Inverted Index

<table>
<thead>
<tr>
<th>Inverted Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>Match</td>
</tr>
<tr>
<td>d=2 x_1 → y_1 d=3</td>
</tr>
<tr>
<td>d=2 x_2 → y_2 d=2</td>
</tr>
<tr>
<td>d=3 x_3 → y_3 d=2</td>
</tr>
</tbody>
</table>

Update degree

<table>
<thead>
<tr>
<th>Update degree</th>
</tr>
</thead>
<tbody>
<tr>
<td>d=2 x_1 → y_1 d=2</td>
</tr>
<tr>
<td>d=2 x_2 → y_2 d=2</td>
</tr>
<tr>
<td>d=2 x_3 → y_3 d=2</td>
</tr>
</tbody>
</table>
Maximal matching strong Scaling
Randomly generated RMAT graphs

For 16x increase of cores: 1,024 – 16,384

<table>
<thead>
<tr>
<th>Graph</th>
<th>#vertices</th>
<th>#edges</th>
<th>Greedy</th>
<th>Karp-Sipser</th>
<th>Dynamic Mindegree</th>
</tr>
</thead>
<tbody>
<tr>
<td>RMAT-26</td>
<td>128 million</td>
<td>2 billion</td>
<td>3x</td>
<td>no</td>
<td>6x</td>
</tr>
<tr>
<td>RMAT-28</td>
<td>512 million</td>
<td>8 billion</td>
<td>7x</td>
<td>3x</td>
<td>10x</td>
</tr>
<tr>
<td>RMAT-30</td>
<td>2 billion</td>
<td>32 billion</td>
<td>12x</td>
<td>8x</td>
<td>15x</td>
</tr>
</tbody>
</table>

Larger graphs
Higher speedups
Strong Scaling

Why does dynamic mindegree scale better?

For 16x increase of cores: 1,024 – 16,384

<table>
<thead>
<tr>
<th>Graph</th>
<th>#vertices</th>
<th>#edges</th>
<th>Greedy</th>
<th>Karp-Sipser</th>
<th>Dynamic Mindegree</th>
</tr>
</thead>
<tbody>
<tr>
<td>RMAT-26</td>
<td>128 million</td>
<td>2 billion</td>
<td>3x</td>
<td>0x</td>
<td>6x</td>
</tr>
<tr>
<td>RMAT-28</td>
<td>512 million</td>
<td>8 billion</td>
<td>7x</td>
<td>3x</td>
<td>10x</td>
</tr>
<tr>
<td>RMAT-30</td>
<td>2 billion</td>
<td>32 billion</td>
<td>12x</td>
<td>8x</td>
<td>15x</td>
</tr>
</tbody>
</table>

(b) Karp–Sipser

(c) Dynamic Mindegree

% time or % matched

% of maximal matching

% of total runtime

Iteration

Iteration
For graph-based algorithms, matching quality decreases with increased concurrency.