A Parallel $\frac{1}{2}$-approx Weighted Matching Algorithm

Mahantesh Halappanavar, Florin Dobrian and Alex Pothen

CSCAPES Seminar. 16 September, 2008.
Outline

1. Introduction
2. Brief Survey of Parallel Matching Algorithms
3. A ½-approx Parallel Matching Algorithm
4. Computational Results
5. Conclusions and Future Work
A graph $G$ is a pair $(V, E)$

- $V$ is a set of vertices
- $E$ is a set of edges that represent a binary relation on $V$.

- Nonbipartite / Bipartite
- Weighted / Unweighted
Matching

Given a graph, a matching $M$ is a subset of edges such that no two edges in $M$ are incident on the same vertex.

Types:
- Maximum Cardinality Matching (no weights)
- Maximum Weight Matching (sum of weights)
Applications of Matchings

- Sparse matrix computations
  - Matrix preconditioning
  - Block Triangular Form
- Multilevel Graph Algorithms
  - Graph partitioners
  - Graph clustering
- Scheduling Problem
  - High speed network switching
  - Facility scheduling problem
- Bioinformatics
  - Homology detection
  - Structural alignment
Outline

1. Introduction
2. Brief Survey of Parallel Matching Algorithms
3. A $\frac{1}{2}$-approx Parallel Matching Algorithm
4. Computational Results
5. Conclusions and Future work
A Brief Survey of Parallel Matching Algorithms

• Bipartite Graphs:
  – Auction-based algorithms
  – Augmentation-based algorithms

• Nonbipartite Graphs:
  – Augmentation-based algorithms

Note: Non-exhaustive survey
Auction-based Algorithms

• Primary work:
  – Dimitri P. Bertsekas, MIT

• Basic idea:
  – Buyers bid for objects
  – Iterative process
  – Two basic approaches:
    • Gauss-Seidel: one buyer at a time
    • Jacobi: all buyers bid concurrently
  – Reverse auctions for asymmetric problems
  – Combined forward/reverse (hybrid) approaches for performance
Auction-based Algorithms

• Parallel work:
  – 1979: Bertsekas
  – 1989: Bertsekas and Castanon
  – 1989: Kempka, Kennington and Zaki (Alliant FX/8)
  – 1990: Wein and Zenios : (Connection Machine, CM2)
  – 1992: Goldberg, Plotkin, Shmoys and Tardos (interior point methods)
  – 2003: Reidy and Demmel (In the context of sparse direct solvers – SuperLU)
Augmentation

- An alternating path:

- An augmenting path:

- Augmentation by Symmetric Difference $\oplus$:
Augmentation-based algorithms

• 1993: Goldberg, Plotkin and Vaidya
• 1997: Storøy and Sørevik (MasPar MP1 and MP2)
• 1998: Haglin
• 1999: Gupta and Ying (vertex separators)
• 2006: Hougardy and Vinkemeier (path growing, \( \frac{1}{2} \)-approx)
• 2008: Chan, Dehne, Bose, Latzel (coarse grained algorithms for convex bipartite graphs and trees)

Theoretical in nature.
Outline

1. Introduction
2. Brief Survey of Parallel Matching Algorithms
3. A $\frac{1}{2}$-approx Parallel Matching Algorithm
   - Introduction
   - Implementation Details
4. Computational Results
5. Conclusions and Future work
A Serial $\frac{1}{2}$-approx Algorithm: **Global**

- **Sort-based (Avis):** $O(|E| \log |E|)$

---

**Algorithm 6**

**Input:** A graph $G$. **Output:** a matching $M$. **Effect:** computes a $\frac{1}{2}$-approx matching $M$ in $G$.

1: procedure GLOBAL-HEAVY($G = (V, E), w : E \to \mathbb{R}^+, M$)
2:    $M \leftarrow \phi$;
3:    repeat
4:        Pick a globally heaviest edge $e_{uv} \in E$;
5:        $M \leftarrow M \cup e_{uv}$;
6:        Delete all edges incident on $u$ and $v$ from $E$;
7:    until $E = \phi$;
8: end procedure
A Serial $\frac{1}{2}$-approx Algorithm: Global

- Sample execution of sort-based algorithm:

Sequential in nature.
A Serial $\frac{1}{2}$-approx Algorithm: Local

- Robert Preis’s LAM algorithm: $O(|E|)$

---

**Algorithm 7**

**Input:** A graph $G$. **Output:** a matching $M$. **Effect:** computes a $\frac{1}{2}$-approx matching $M$ in $G$.

1: procedure LAM($G = (V, E), w : E \rightarrow \mathbb{R}^+, M$)
2: $M \leftarrow \emptyset$;
3: repeat
4: Pick a locally-heaviest edge $e_{uv} \in E$;
5: $M \leftarrow M \cup e_{uv}$;
6: Delete all edges incident on $u$ and $v$ from $E$;
7: until $E = \emptyset$;
8: end procedure
A Serial $\frac{1}{2}$-approx Algorithm: **Local**

- Sample execution of LAM:

  Sequential in nature.
Assumptions for Parallelization

• **Vertex-oriented** data structures for graph representation
• Graph distributed among processors via vertex partitioning
• Owner-computes Model: each processor owns a set of vertices that it is responsible for
Towards Parallelization

**Pointer-based algorithm:**

1. For each vertex, set a pointer to the heaviest adjacent vertex.
2. If two vertices point to each other, then add these (locally dominating) edges to the matching.
3. Remove all edges incident on the matched edges, reset the pointers, and repeat.
Towards Parallelization

• Sample execution of the pointer-based approach:

Parallel in nature.
A Worst-case Scenario

Forced sequentialness
Related Work (Pointer-based algorithm)

• 2004: Jaap-Henk Hoepman
  – Show parallel algorithm as a variant of Preis’s algorithm
  – One vertex per processor (theoretical)
  – Algorithm converges in $\Theta(2.|E|)$ messages

• 2007: Fredrik Manne and Rob Bisseling:
  – Extend Hoepman’s work
  – Show parallel algorithm as a variant of Luby’s algorithm
  – Complexity: $O(|V|d^2+|E|)$
  – No clear description of the parallel algorithm
  – BSP style

Note: Fredrik Manne independently developed the pointer-based algorithm that he presented at SIAM Parallel Processing 2006.
Outline

1. Introduction
2. Brief Survey of Parallel Matching Algorithms
3. A $\frac{1}{2}$-approx Parallel Matching Algorithm
   - Introduction
   - Implementation Details
4. Computational Results
5. Conclusions and Future work
Data Distribution

Ghost vertices
Distributed Graph Data structure

<table>
<thead>
<tr>
<th>VtxPointer</th>
<th>0, 3, 6, 8, 10, 13, 16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adjacency</td>
<td>(3, 4, 5)(2, 4, 5)(1, 5)(0, 4)(0, 1, 3)(0, 1, 2)</td>
</tr>
<tr>
<td>EdgeWt</td>
<td>w1, w2, w3, …</td>
</tr>
</tbody>
</table>

Compressed Storage Format

<table>
<thead>
<tr>
<th>Vertex Indices</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Owner Processor</td>
<td>P0</td>
<td>P1</td>
<td>P1</td>
<td>P0</td>
<td>P0</td>
<td>P1</td>
</tr>
<tr>
<td>New Indices</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>5</td>
</tr>
</tbody>
</table>

Vertex distribution and renumbering
Distributed Graph Data structure

Processor 0:
- Processor Pointer: 0, 3, 6
- VtxPointer: 0, 3, 5, 8
- Adjacency (m): (1, 2, 5)(0, 2)(0, 1, 3)
- EdgeWt: e1, e2, e3, ...
- VtxWt: v1, v2, v3, ...

Processor 1:
- Processor Pointer: 0, 3, 6
- VtxPointer: 0, 3, 5, 8
- Adjacency: (2, 4, 5)(3, 5)(0, 3, 4)
- EdgeWt: e1, e2, e3, ...
- VtxWt: v1, v2, v3, ...

Data structure on each processor

FindOwner(ghost-vtx): O(lg P); Storage: O(P)
A parallel algorithm:
Hoepman’s algorithm with one vertex per processor
Our algorithm: many vertices per processor

1. Initialization: //(local computation)
   – Identify locally dominant edges
   – Send requests if needed

2. Computation: //(communication/computation)
   – Receive messages
   – Computation based on the received messages
   – Send messages is needed
   – Repeat until no more edges can be matched

Note: SPMD model; Distributed memory; Explicit messages
PART-1: Initialization

• For each vertex $v_i$ set the pointer to the heaviest neighbor
  – If the heaviest neighbor is a ghost vertex, send a REQUEST message to its owner; //Non-blocking
  – If $v_i$ has at least one cross-edge incident on it:
    • $S \leftarrow S \cup \{v_i\}$
    • $\text{Counter}[v_i] = \#\text{cross-edges incident on } v_i$
• Repeat:
  – For all vertex pairs that point to each other, add the corresponding edges to the matching
  – Remove edges incident on the matched edges (send SUCCESS messages)
  – Reset the pointers (send messages if needed)
  – Repeat until no more edges can be added to the matching
PART-2: Computation

• WHILE (S ≠ NULL) DO
  – Receive a Message //Blocking; from any source
  – Process the Message based on type
    • Request, Success, or Failure
    • Add to matching, and remove edges incident (send SUCCESS messages)
    • Reset pointers for vertices that were pointing to the matched vertices (Send messages if needed)
  – Update:
    • Counter[v_i]: Decrement the counter
    • S (remove v_i from S when Counter[v_i]=0)
    • Send FAILURE messages if some vertex cannot be matched

MPI standard requires that every SEND be matched with a RECEIVE. Therefore, we need set S and Counter[v] to keep track of all the messages that need to be received.
Our scheme needs $\leq 3|\text{EdgeCut}|$ messages.

Can be optimized to $2|\text{EdgeCut}|$ messages.
MPI: Buffered Sends

We also have an implementation with MPI_Isend() with similar performance.

Source: Dr. Gerhard Wellein (RRZE) et al.
Graph algorithms: Issues & Challenges

• Load balancing:
  – Pre-distributed data; 1D V/s 2D; performance of partitioners

• Locality:
  – Cache-aware V/s Cache-oblivious

• Ghost vertices:
  – Memory V/s Performance

“... computation done by 32,768 processors on BlueGene/L could be done by five to 10 processors of an MTA-2 with sufficient memory.”

Outline

1. Introduction
2. Brief Survey of Parallel Matching Algorithms
3. A ½-approx Parallel Matching Algorithm
4. Computational Results
   - Performance of serial ½-approx algorithm
   - Performance of parallel ½-approx algorithm
5. Conclusions and Future work
Platform Details

• Zorka Compute Cluster:
  • *Compute Node*: Two dual core 3.0 GHz Intel Xeon (4 CPUs); 8 GB RAM
  • *Total Nodes*: 40 (160 cores)
• Network: Infiniband 4X (20 Gbits/s)
• Software:
  • Intel C++ compilers (–O2 –axT)
  • MVAPICH2, with 4 processes per node (wrap around if #processes > #cores)

We see about 20% performance difference between GigE and Infiniband.
<table>
<thead>
<tr>
<th>Name</th>
<th>#Rows</th>
<th>#Cols</th>
<th>#nnz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Circuit Simulation</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rajat16</td>
<td>94,294</td>
<td>94,294</td>
<td>476,766</td>
</tr>
<tr>
<td>Rajat21</td>
<td>411,676</td>
<td>411,676</td>
<td>1,876,011</td>
</tr>
<tr>
<td>Rajat29</td>
<td>643,994</td>
<td>643,994</td>
<td>3,760,246</td>
</tr>
<tr>
<td>Rajat30</td>
<td>643,994</td>
<td>643,994</td>
<td>6,175,244</td>
</tr>
<tr>
<td>Rajat31</td>
<td>4,690,002</td>
<td>4,690,002</td>
<td>20,316,253</td>
</tr>
<tr>
<td>ASIC_320ks</td>
<td>321,671</td>
<td>321,671</td>
<td>1,316,085</td>
</tr>
<tr>
<td>ASIC_680k</td>
<td>682,862</td>
<td>682,862</td>
<td>2,638,997</td>
</tr>
<tr>
<td>G3_circuit</td>
<td>1,585,478</td>
<td>1,585,478</td>
<td>7,660,826</td>
</tr>
<tr>
<td>Structural Engineering</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bodyy6</td>
<td>19,366</td>
<td>19,366</td>
<td>134,208</td>
</tr>
<tr>
<td>bcsstk38</td>
<td>8,032</td>
<td>8,032</td>
<td>355,460</td>
</tr>
<tr>
<td>bcsstk35</td>
<td>30,237</td>
<td>30,237</td>
<td>1,450,163</td>
</tr>
<tr>
<td>bcsstk39</td>
<td>46,772</td>
<td>46,772</td>
<td>2,060,662</td>
</tr>
<tr>
<td>crystk03</td>
<td>24,696</td>
<td>24,696</td>
<td>1,751,178</td>
</tr>
<tr>
<td>ct20stif</td>
<td>52,329</td>
<td>52,329</td>
<td>2,600,295</td>
</tr>
<tr>
<td>ptwk</td>
<td>217,918</td>
<td>217,918</td>
<td>11,524,432</td>
</tr>
<tr>
<td>Fluid Dynamics</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pres_Poisson</td>
<td>14,822</td>
<td>14,822</td>
<td>715,804</td>
</tr>
<tr>
<td>af23560</td>
<td>23,560</td>
<td>23,560</td>
<td>460,598</td>
</tr>
<tr>
<td>Electrical Engineering</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>onetone2</td>
<td>36,057</td>
<td>36,057</td>
<td>222,596</td>
</tr>
<tr>
<td>twotone</td>
<td>120,750</td>
<td>120,750</td>
<td>1,206,265</td>
</tr>
<tr>
<td>pre2</td>
<td>659,033</td>
<td>659,033</td>
<td>5,834,044</td>
</tr>
<tr>
<td>DNA Electrophoresis</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cage10</td>
<td>11,397</td>
<td>11,397</td>
<td>150,645</td>
</tr>
<tr>
<td>cage11</td>
<td>39,082</td>
<td>39,082</td>
<td>559,722</td>
</tr>
<tr>
<td>cage12</td>
<td>130,228</td>
<td>130,228</td>
<td>2,032,536</td>
</tr>
<tr>
<td>cage13</td>
<td>445,315</td>
<td>445,315</td>
<td>7,479,343</td>
</tr>
<tr>
<td>cage14</td>
<td>1,505,785</td>
<td>1,505,785</td>
<td>27,130,349</td>
</tr>
<tr>
<td>cage15</td>
<td>5,154,859</td>
<td>5,154,859</td>
<td>99,199,551</td>
</tr>
<tr>
<td>Chemical Engineering</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>meg1</td>
<td>2,904</td>
<td>2,904</td>
<td>58,142</td>
</tr>
<tr>
<td>bayer04</td>
<td>20,545</td>
<td>20,545</td>
<td>85,537</td>
</tr>
<tr>
<td>bayer01</td>
<td>57,735</td>
<td>57,735</td>
<td>275,094</td>
</tr>
<tr>
<td>Economic Models</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>g7jac040</td>
<td>11,790</td>
<td>11,790</td>
<td>107,383</td>
</tr>
<tr>
<td>g7jac080</td>
<td>23,670</td>
<td>23,670</td>
<td>259,648</td>
</tr>
<tr>
<td>g7jac160</td>
<td>47,430</td>
<td>47,430</td>
<td>564,952</td>
</tr>
<tr>
<td>g7jac200</td>
<td>59,310</td>
<td>59,310</td>
<td>717,620</td>
</tr>
<tr>
<td>Symmetric Indefinite</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F2</td>
<td>71,505</td>
<td>71,505</td>
<td>5,294,285</td>
</tr>
<tr>
<td>F1</td>
<td>343,791</td>
<td>343,791</td>
<td>26,837,113</td>
</tr>
</tbody>
</table>
Performance of Sequential Algorithm

• Exact algorithm:
  – **Perfect** matching of maximum weight (similar to the algorithm implemented in MC64)
  – Binary heap data structure
  – Greedy initialization is critical for performance
  – \( O(|V||E| + |V|^2\log|V|) \)

• Approximation algorithm:
  – Pointer-based algorithm
  – \( O(|V|d^2 + |E|) \)

• Why?
  – Maximum weight matching is very slow
  – Context: Sparse matrix preconditioners
The approximation algorithm is very fast.
Outline

1. Introduction
2. Brief Survey of Parallel Matching Algorithms
3. A $\frac{1}{2}$-approx Parallel Matching Algorithm
4. Computational Results
   - Performance of serial $\frac{1}{2}$-approx algorithm
   - Performance of parallel $\frac{1}{2}$-approx algorithm
5. Conclusions and Future work
Min and Max times are the shortest and longest times on any given process (core). Avg is the average time of all the processes.
G3_Circuit:  # NVtx: 3,170,956;  #Edge: 4,623,152
Bayer01: #Vtx=115,470; #Edges= 277,774
ASIC_320ks: #Vtx=643,342; #Edges= 1,827,807
bcsstk39: #Vtx=38,732; #Edges= 77,057
g7jac200: #Vtx=118,620; #Edges= 837,936
meg1: \#Vtx=5,808; \#Edges= 58,142
crystk03: #Vtx=49,392; #Edges= 887,937
Synthetic Graph: SSCA#2

Graph: #Vtx: 2,097,152; #Edge: 63,148,387
Original graph generated with GT-Graph Generator. Graph modified (treat it as bipartite graph) and duplicates eliminated.

Super linear Speedup? Most probably due to cache effects rather than an inefficient serial implementation.

Visualizing SSCA#2 graphs using Fiedler coordinates; Source: ctwatch.org
Graph: #Vtx: 500,000; #Edge: 1,500,000
Original graph generated with GT-Graph Generator.
Graph modified (treated as a bipartite graph) and duplicates eliminated.
Synthetic Graph: Random Graph

Graph: #Vtx: 1,000,000; #Edge: 2,250,000
Original graph generated with GT-Graph Generator.
Graph modified (treated as a bipartite graph) and duplicates eliminated.
Jumpshot Pictures

• Input: Rajat31 (#Vtx: 9,380,004; #Edges: 20,316,253)
• Edgecut: 36,998; Transfer: 2.78 s; Weight: 6.25e+07; Cardinality: 4,688,751;
• Compute Time: Min: 2.79e-02; Max: 3.09e-02; Avg (32): 3.73e-02 seconds.
Entire Execution

Initialization and Graph partitioning with Metis
Data transfer
Matching
Communicate results
Close-up: Communication

Legend: Purple = B_Send; Green = Receive
Closeup: Communication

Legend: Purple = B_Send; Green = Receive
Close-up: Communication

Long green bars: Waiting to receive $\rightarrow$ scope for improvement (speculation algorithms)
Outline

1. Introduction
2. Brief Survey of Parallel Matching Algorithms
3. A $\frac{1}{2}$-approx Parallel Matching Algorithm
4. Computational Results
5. Conclusions and Future work
Contributions

• Extended the existing work
• Design of asynchronous communication scheme
• Efficient implementation for distributed memory system:
  – MatchBoxP
  – C++, STL, MPI
Conclusions

- Speedup is not a right goal for parallelization
- Graph structure and graph partitioning are critical for performance (but, probably, cannot be controlled)
- Memory limitations may change data structures, and therefore, performance
- One sided communications will probably help when used on systems with fast interconnects
Future Work

• Tests for performance on the DOE Leadership-class machines (NERSC)
• Massive graphs
• Software engineering: data structures, error handling, documentation, etc.

THANK YOU!

We would like to thank Assefaw Gebremedhin for his time and suggestions to improve this work.
Performance: Cardinality & Weight

Performance: Cardinality and Weight

%Goodness (Approx/Exact\times100)

Matrices sorted by name ➔