Distributed Bipartite Matching

Brian A. Page
bpage1nd.edu
December 10, 2018
Bipartite Matching

- Matching (M) is set of edges such that E(u,v)
- Vertices incident to only one edge in M

A Maximum Matching
Alternating/Augmenting Paths

• Many flow networks rely on augmenting path algorithms
• Can be used to find a more optimal matching
• Runtime influenced by number of potential augmenting paths needing verification
MBM: Multithreaded

- Utilizes Hopcroft-Karp algorithm
  - OpenMP for alternating/augmenting path checks
  - Runs in $O(|E|\sqrt{|V|})$
- Requires atomic operations for matching reversal
- Slower than sequential
Hopcroft-Karp

1. BFS for alternate path frontier
2. DFS for augmenting verification
3. Check if matching increases if augmenting path chosen
4. stop when there are no more augmenting paths possible.
BFS

- Breadth First Search checks for vertices adjacent to a matched vertex
  - Adjacent vertices are candidates for alternating paths
- Parts of BFS can be parallelized using OpenMP
  - matched vertices can be checked simultaneously for adjacencies
- Initial experiments have used: #pragma omp for

```cpp
// Returns true if there is an augmenting path, else returns false
bool Bipartite::bfs()
{
    queue<int> Q; // an integer queue

    // First layer of vertices (set distance to 0)
    for (int u = 1; u <= uVertices.size(); u++) {
        // If this is a free vertex, add it to queue
        if (pairU[u] == NIL)
            // u is not matched
            dist[u] = 0;
            Q.push(u);
    }

    // Else set distance as infinite so that this vertex is considered next time
    else dist[u] = INF;

    // Initialize distance to NIL as infinite
    dist[NIL] = INF;

    // Q is going to contain vertices of left side only.
    while (!Q.empty()) {
        // Dequeue a vertex
        int u = Q.front();
        Q.pop();

        // If this node is not NIL and can provide a shorter path to NIL
        if (dist[u] < dist[NIL])
            // Get all adjacent vertices of the dequeued vertex u
            while(uList[u].end) { // iterator
                int v = uVertices[u].edgelist[i].v,

                // If pair of v is not considered so far
                // (v, pairV[v]) is not yet explored edge.
                if (dist[pairV[v]] == INF) {
                    // Consider the pair and add it to queue
                    dist[pairV[v]] = dist[u] + 1;
                    Q.push(pairV[v]);
                }
            }

        // If we could come back to NIL using alternating path of distinct
        // vertices then there is an augmenting path
        return (dist[NIL] != INF);
    }
}
DFS

- Depth First Search checks to see if the alternating paths are augmenting paths

- Like BFS parts are easily parallelizable using OpenMP
  - check each path(edge)
## Benchmark Graphs

<table>
<thead>
<tr>
<th>Graph Name</th>
<th>Rows</th>
<th>Columns</th>
<th>Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>divorce</td>
<td>50</td>
<td>9</td>
<td>225</td>
</tr>
<tr>
<td>Cities</td>
<td>50</td>
<td>46</td>
<td>1342</td>
</tr>
<tr>
<td>World Cities</td>
<td>315</td>
<td>100</td>
<td>7518</td>
</tr>
<tr>
<td>Notredame_actors</td>
<td>392400</td>
<td>127823</td>
<td>1470404</td>
</tr>
<tr>
<td>12month1</td>
<td>12471</td>
<td>872622</td>
<td>22624727</td>
</tr>
</tbody>
</table>

- Suite sparse matrix/graph collection
Multithreaded Results

MCM on Bipartite Graphs

- **sequential**
- **multithreaded**

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Time in microseconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>divorce</td>
<td>1000</td>
</tr>
<tr>
<td>Cities</td>
<td>1000</td>
</tr>
<tr>
<td>World Cities</td>
<td>10,000</td>
</tr>
<tr>
<td>Notre Dame</td>
<td>100,000</td>
</tr>
<tr>
<td>12month1</td>
<td>100,000</td>
</tr>
</tbody>
</table>

*Died!*
Distributed MBM: Scaling

- Use MPI for interprocess communication
- Hope to decrease run-time as process count scales
- Allow for streaming of edge/vertex changes

Challenges

- Optimal workload distribution necessary for reduced communication
- Dynamic Graph optimization
Partitioning

- Vertices sorted by degree
- Vertices assigned based on optimal packing of edge count
- Vertices have associated edge list
- Outing edges “owned” by source vertex
- Two copies of each edge
  - Aids alternating path search
BFS and DFS

- Process performs BFS with regard to locally owned vertices
- If adjacent vertices are not “owned” notify owner of traversal
- Similar behavior for DFS phase
Distributed MBM

Pros

- Partitioning possible and working as designed
  - Near uniform edge distribution

Cons

- Off node/process notifications don't work properly
Lessons Learned

- Scaling Bipartite Matching is hard!
  - race condition potential
  - sequential nature in native form
- Multithreading works, but due to some required atomic had worse performance
- I believe the use of futures might alleviate much of the off node communication issues (not necessarily the overhead)
Can This Be Done?!

- Yes. However it is non-trivial!

- Best methods to date still require heavy communication volumes
References


• https://en.wikipedia.org/wiki/Bipartite_graph

• https://en.wikipedia.org/wiki/Matching_(graph_theory)#Bipartite_matching

• https://www.geeksforgeeks.org/ford-fulkerson-algorithm-for-maximum-flow-problem/

• https://en.wikipedia.org/wiki/Hungarian_algorithm#The_algorithm_in_terms_of_bipartite_graphs