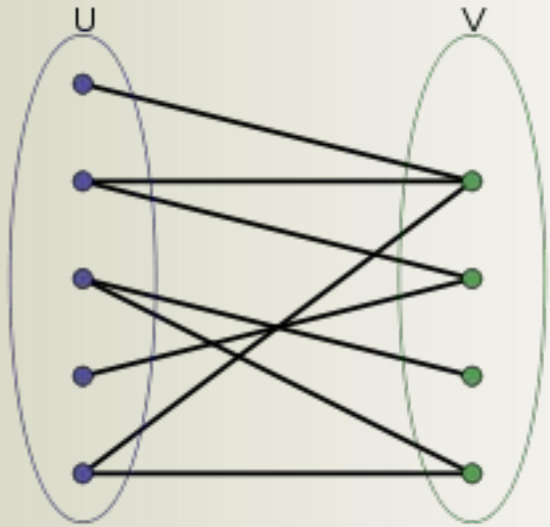
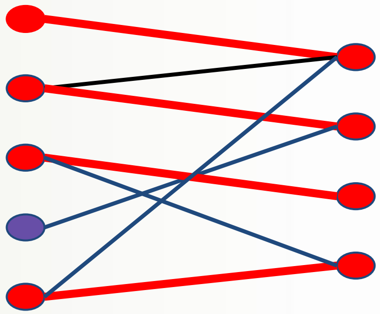
# Distributed Bipartite Matching

Brian A. Page
bpage1nd.edu
December 10, 2018

# Bipartite Matching

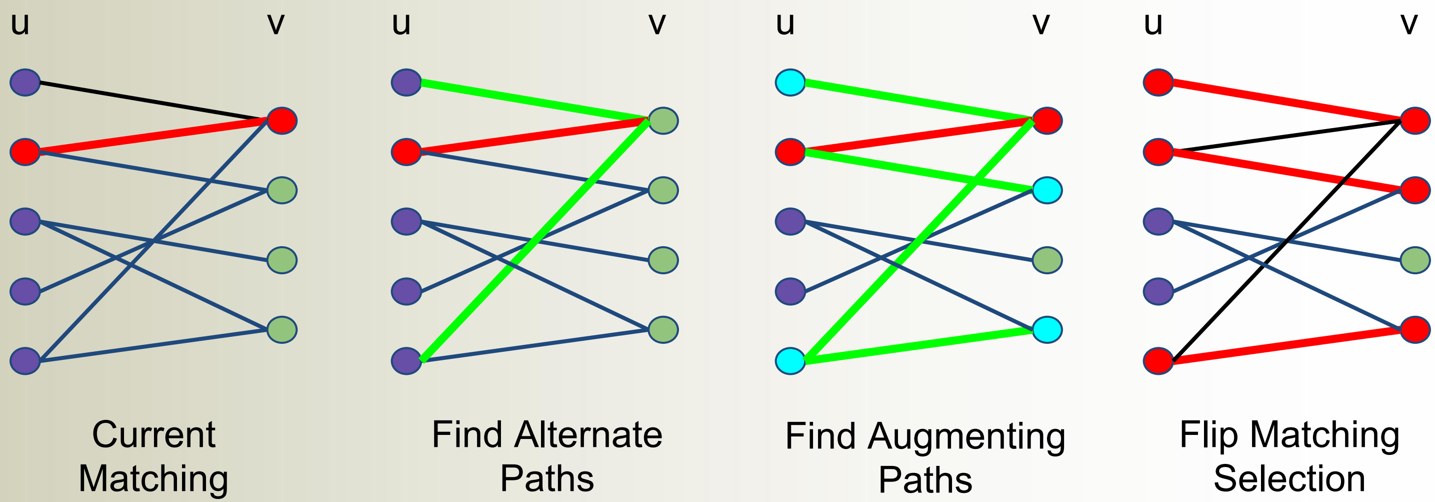


A Maximum Matching

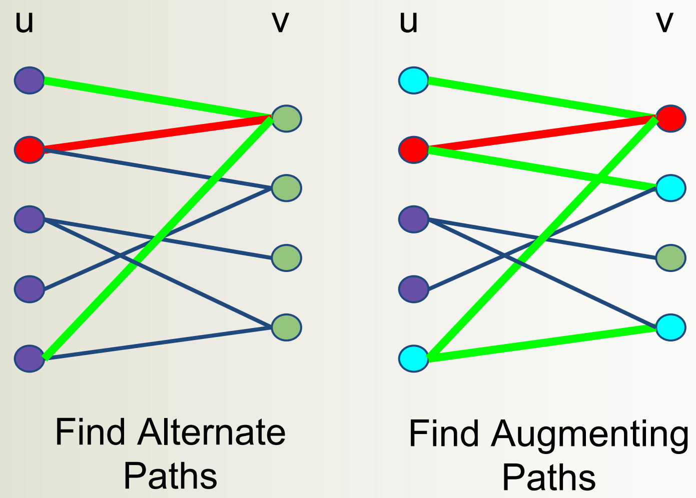
- Matching (M) is set of edges such that E(u,v)
- Vertices incident to only one edge in M

# 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

```
// Returns size of maximum matching
int Bipartite::hopcroftKarp() {
    pairU = new int[uVertices.size() + 1];
    pairV = new int[vVertices.size() + 1];
    dist = new int[uVertices.size() + 1];

    // Initialize NIL as pair of all vertices
    for (int u = 0; u < uVertices.size(); u++) {
        pairU[u] = NIL;
    }
    for (int v = 0; v < vVertices.size(); v++) {
        pairV[v] = NIL;
    }

    // Initialize result
    int result = 0;

    // Keep updating the result while there is an augmenting path.
    while (bfs())    {
        // Find a free vertex
        for (int u=1; u<=uVertices.size(); u++) {
            // If current vertex is free and there is
            // an augmenting path from current vertex
            if (pairU[u] == NIL && dfs(u)) {
                result++;
            }
        }
    }
    return result;
}
```

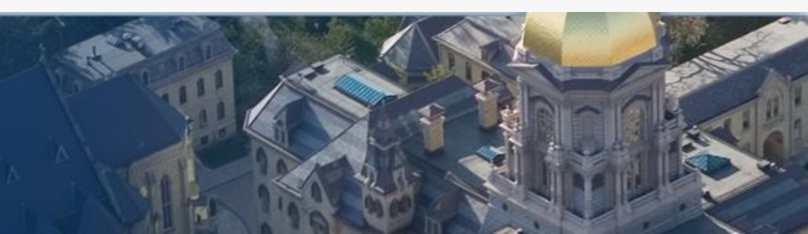
1. BFS for alternate path frontier
2. DFS for augmenting verification
3. Check if matching increases if augmenting path chosen
4. stop when there are no more augmenting paths possible.

# 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 as 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
            //list<int>::iterator i;
            for (int i = 0; i < uVertices[u].edgeList.size(); ++i){
                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

```cpp
// Returns true if there is an augmenting path beginning with free vertex u
bool Bipartite::dfs(int u){
    if (u != NIL){
        //list<int>::iterator i;
        for (int i = 0; i < uVertices[u].edgeList.size(); ++i){
            int v = uVertices[u].edgeList[i].v;

            // Follow the distances set by BFS
            if (dist[pairV[v]] == dist[u]+1)            {
                // If dfs for pair of v also returns
                // true
                if (dfs(pairV[v]) == true){
                    pairV[v] = u;
                    pairU[u] = v;
                    return true;
                }
            }
        }

        // If there is no augmenting path beginning with u.
        dist[u] = INF;
        return false;
    }
    return true;
}
```

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

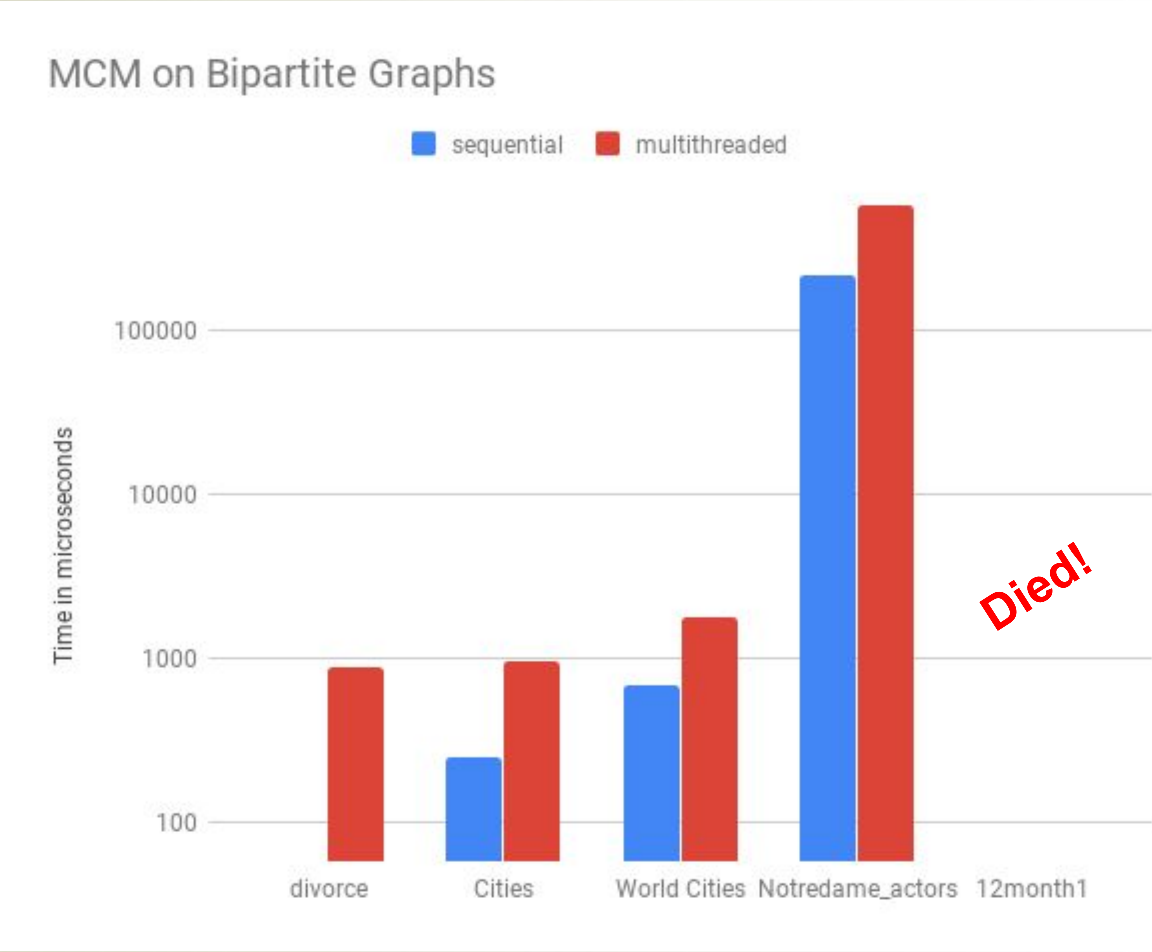
- Like BFS  parts are easily parallelizable using OpenMP
    - check each path(edge)

# Benchmark Graphs

| Graph Name | Rows | Columns | Edges |
|---|---|---|---|
| divorce | 50 | 9 | 225 |
| Cities | 50 | 46 | 1342 |
| World Cities | 315 | 100 | 7518 |
| Notredame_actors | 392400 | 127823 | 1470404 |
| 12month1 | 12471 | 872622 | 22624727 |

- Suite sparse matrix/graph collection

# Multithreaded Results



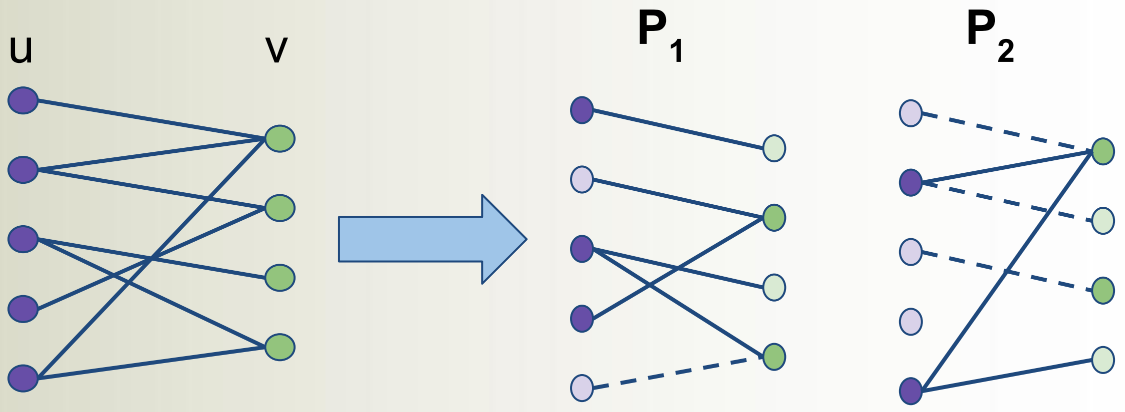
# Distributed MBM: Scaling

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

Challenges

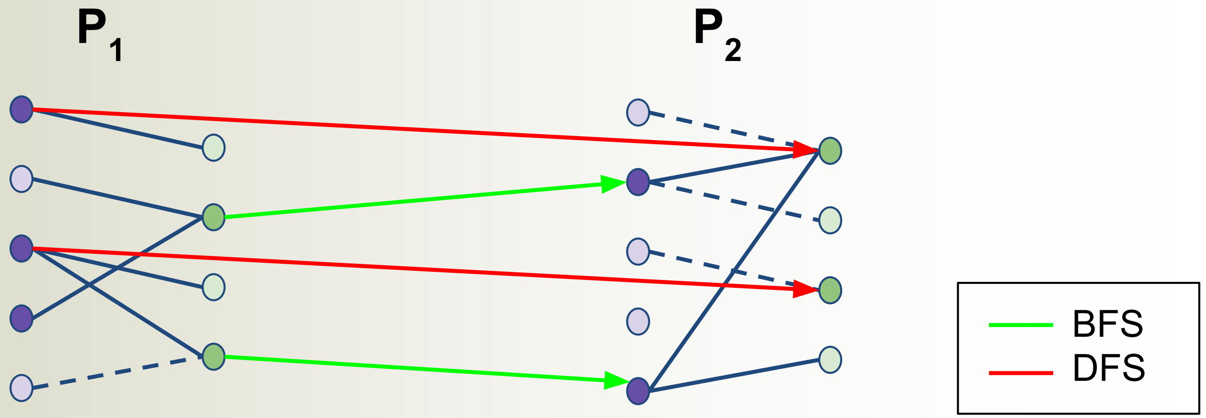
- Optimal workload distribution necessary for reduced communication
- Dynamic Graph optimization

# 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 dont work properly

```
|mbm(G)| = [cswarmfe.crc.nd.edu:mpi_rank_0][error_sighandler] Caught error: Segmentation fault (signal 11)

===================================================================================
=   BAD TERMINATION OF ONE OF YOUR APPLICATION PROCESSES
=   PID 5706 RUNNING AT cswarmfe.crc.nd.edu
=   EXIT CODE: 11
=   CLEANING UP REMAINING PROCESSES
=   YOU CAN IGNORE THE BELOW CLEANUP MESSAGES
===================================================================================
YOUR APPLICATION TERMINATED WITH THE EXIT STRING: Segmentation fault (signal 11)
```





# 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


- Jeremy Kepner and John Gilbert. 2011. Graph Algorithms in the Language of Linear Algebra. Soc. for Industrial and Applied Math., Philadelphia, PA, USA.
- 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
- Ariful Azad and Aydin Buluc. 2016. Distributed-Memory Algorithms for Maximum Cardinality Matching in Bipartite Graphs. Lawrence Berkeley National Labs.