PARALLEL BREADTH-FIRST SEARCH USING MPI

Author – Aditya Nongmeikapam
Course – 633 Parallel Algorithms
Instructor – Russ Miller
CONTENTS

• Introduction to BFS (Breadth-First Search)
• Applications of BFS
• Sequential BFS Algorithm
• Parallel BFS Algorithm
• Results
• Conclusion & Challenges
• Future Work
Breadth-First Search

- It is a graph traversing algorithm
- Starts with a given start node and traverse the graph layer wise. We then move towards the next level neighbors.
- Extra memory required, usually a queue.
  - To keep track of unexplored child nodes.
Applications of BFS

- Used to solve many graph theory problems like shortest path between two nodes for an unweighted graph.
- For computing the maximum flow in a flow network.
- In Social networking websites (e.g. LinkedIn), we can find the ith connection of a source person.
- Detect cycles in an undirected graph
Sequential BFS Algorithm

- Set all the vertices to not visited.
- Create a queue and add the start node or nodes.
- While the queue becomes not empty -
  - Take the first node from queue and remove it
  - If not visited already
    - Make the node visited
    - Add all the neighbors of the node into the queue.

- Time Complexity will be $O(N^2)$
- Space Complexity will be $O(N^2)$
- $N$ is total number of vertices and my implementation is based on adjacency matrix.
Parallel BFS Algorithm

• Similar algorithm as the sequential BFS.
• Instead of popping out one vertex at a time, pop out all the nodes in the same level. (These nodes are known as **frontier nodes**)
• **Level synchronous** traversal. Each the processor will take a set of frontier vertices and calculate their next frontier vertices in parallel.
• For the above step we will need to partition the adjacency matrix and the vertices and allocate them to the processors.
2-D Partition of Adjacency Matrix

- The adj matrix is divided into $P$ blocks of size $\frac{N}{\sqrt{P}} \times \frac{N}{\sqrt{P}}$
- Vertex are partitioned into $N/P$ size groups.
- $N$ – Number of Vertices
- $P$ – Number of Processors (In my case it is always a perfect square)
Steps of Parallel BFS Algorithm

• Do a transpose of the frontier vector between the processors.
• After this all the columns processors will have matching frontier with their local adjacency matrix.
Steps of Parallel BFS Algorithm

- We then do a column wise all gather for the frontier vertices.
- This will broadcast the required frontier vertices for each column.
Steps of Parallel BFS Algorithm

- Calculation of next frontier vertices is based on the current frontier vector that the processor has.
- Using the local adj matrix the next frontier vector is calculated.
- Note that each processor row now has the full information of the next frontier vertices.
Steps of Parallel BFS Algorithm

• Now we do a all to all gather row wise so that all the next frontier vectors are merged. *(union)*

• All the processors now know if they have any frontier element (Next frontier now becomes current local frontier) that they own.

• We mark the node as visited and store its parent node.
Steps of Parallel BFS Algorithm

• We do a row wise all gather and then column wise all gather to broadcast the local frontiers globally.
• We continue the process till there is no vertices left in the global frontier vertices.
• Note- The communication cost here is $O(\sqrt{P})$
Results

• For small number of processors the graph is linear but as the number of processor increases the speed up goes down.

• But due to parallel communication overhead, we get a point (“Sweet spot”) from where the speed up starts decreasing with increasing processors.
Results

• For large graphs (i.e., size of adjacency matrix > $10^8$). The speed up remains keeps increasing linearly with increasing processors.

• As we increase the size of our problem input size, putting more processors makes more sense as it leads to more speed up.
Execution Time Vs Processor

[Graph showing the relationship between execution time and the number of processors for different numbers of vertices.]
How Diameter affects the runtime

- The runtime of the algorithm depends on the diameter of the graph.
- As we increase the diameter of 6 to 20, the runtime is also increased by a factor of around 3.3x.
- This is in fact expected as the number of rounds of the algorithm is also increased by the same factor.
- PRAM asymptotic time complexity for a level-synchronous parallel BFS is $O(D)$ where $D$ is the diameter of the graph.
How Density affects runtime

- The runtime of the algorithm is not depending on the density of the graph.
- As we double the density of the graph from 33% to 66%, there is no significant change in the runtime.
- This is because we use a adjacency matrix based approach and do not take advantage of the sparseness of the matrix or the frontier vectors.
## Average Execution times

<table>
<thead>
<tr>
<th>Processors</th>
<th>1 (Seq)</th>
<th>4</th>
<th>9</th>
<th>16</th>
<th>36</th>
<th>49</th>
<th>64</th>
<th>81</th>
<th>100</th>
<th>121</th>
<th>144*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertices</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>800</td>
<td>2.55</td>
<td>2.2</td>
<td>0.96</td>
<td>0.55</td>
<td>0.3</td>
<td>0.24</td>
<td>0.18</td>
<td>0.17</td>
<td>0.13</td>
<td>0.15</td>
<td>0.38</td>
</tr>
<tr>
<td>1600</td>
<td>10.05</td>
<td>8.69</td>
<td>3.91</td>
<td>2.18</td>
<td>1.05</td>
<td>0.76</td>
<td>0.59</td>
<td>0.5</td>
<td>0.4</td>
<td>0.49</td>
<td>0.55</td>
</tr>
<tr>
<td>3200</td>
<td>39.77</td>
<td>35</td>
<td>15.66</td>
<td>9.11</td>
<td>4.14</td>
<td>3.03</td>
<td>2.31</td>
<td>1.9</td>
<td>1.49</td>
<td>1.3</td>
<td>1.15</td>
</tr>
<tr>
<td>6400</td>
<td>132</td>
<td>142</td>
<td>64.19</td>
<td>36.88</td>
<td>16.39</td>
<td>11.75</td>
<td>9.03</td>
<td>7.41</td>
<td>5.82</td>
<td>5.03</td>
<td>4.16</td>
</tr>
<tr>
<td>12800</td>
<td>545.54</td>
<td>547</td>
<td>254.53</td>
<td>142.17</td>
<td>62.02</td>
<td>46.46</td>
<td>36.47</td>
<td>29.1</td>
<td>23.3</td>
<td>20.07</td>
<td>16.54</td>
</tr>
<tr>
<td>25600</td>
<td>2566.99</td>
<td>2152.41</td>
<td>1006.26</td>
<td>584.11</td>
<td>258.76</td>
<td>189.89</td>
<td>142.79</td>
<td>119.56</td>
<td>92</td>
<td>78.87</td>
<td>67.29</td>
</tr>
<tr>
<td>51200</td>
<td>11052.59</td>
<td>8545.06</td>
<td>4080.1</td>
<td>2265.75</td>
<td>1050.67</td>
<td>753</td>
<td>572.11</td>
<td>453.41</td>
<td>379.88</td>
<td>311.34</td>
<td>256.66</td>
</tr>
</tbody>
</table>

* - Executed in 142 nodes with 1 processor and 1 node with 2 processor
Conclusion & Challenges

• We see that for smaller input sizes after 100 processors the speedup is decreasing.
• We have access to only 143 nodes in the HPC cluster
• Was able to run with $10^5$ Vertices ( 6 billion edges ) (320 GB of memory used) but had problems running $10^6$ vertices.

Benefits of using HPC

<table>
<thead>
<tr>
<th>Processor</th>
<th>Vertices</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>128x2 = 256</td>
<td>$10^5$</td>
<td>0.16 hrs</td>
</tr>
<tr>
<td>1 --&gt; sequential</td>
<td>$10^5$</td>
<td>12.3 hrs (estimated)</td>
</tr>
</tbody>
</table>
Future Work

- Optimisation of the algorithm using sparse representation of the matrix.
- Use space efficient bitmaps for storing the data/vector.
- Inter-processor collective communication optimisation.
References

• [Parallel Breadth-First Search on Distributed Memory Systems](https://people.eecs.berkeley.edu/~aydin/sc11_bfs.pdf)
• [A Scalable Distributed Parallel Breadth-First Search Algorithm on BlueGene/L](https://ieeexplore.ieee.org/document/1559977)
• [Parallel Breadth-first Search](https://en.wikipedia.org/wiki/Parallel_breadth-first_search)
• [MPI Scatter-Gather and Allgather](https://mpitutorial.com/tutorials/mpi-scatter-gather-and-allgather/)
• [Collective Operation](https://en.wikipedia.org/wiki/Collective_operation)
• [Support Home](https://ubccr.freshdesk.com/support/home)
• [Slurm](https://slurm.schedmd.com/)
Thank You