HYBRID PARALLEL BFS USING MPI AND OPEN MP

Author – Aditya Nongmeikapam
UB Number - 50418825
Course – SEM708
Instructor – Russ Miller
CONTENTS

• Introduction to BFS (Breadth-First Search)
• Applications of BFS
• Parallel BFS Algorithm using vertex partitioning
• Results
Breath-First Search

- It is a graph traversing algorithm
- It traverses a tree/graph by exploring all of the nodes at the current level before moving on to the next level.
- A queue is used to keep track of unexplored child nodes.
- Runtime is $O(N^2)$ where $N$ is no. of vertices. (For adjacency matrix based approach).
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
Parallel BFS Algorithm Using Vertex Partition (1-D Partitioning)

- We do a level synchronous traversal.
- A set of vertices are assigned to a single processor. This processor handles all the computation involved for exploring the neighbors.
- All the processors calculate the neighbors using OMP Threads.
- All to All communication between all the processor to know the next set of frontier vertices.
- MPI is executed in one processor in a node.
- OpenMP threads are executed in cores inside that processor.
1-D(Vertex) Partition Parallel BFS Steps

- Chunk Size is calculated as vertices/processors. Each chunk of vertices are assigned to one processor.
- All the calculations for next neighbors are done by the same processor.
- The calculations done by a processor is multithreaded using Open MP.
- All to All communications between all the processor to sync the next frontier vertices using MPI.
**1-D Partition of Adjacency matrix**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></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>0</td>
<td></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>1</td>
<td></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>2</td>
<td></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>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

MPI Processor - P1

Shaded region of the adjacency matrix is the chunk which is available to MPI Processor P1.

The calculation of next frontier elements are done parallelly using OMP threads.
OMP Loop Level Parallelization

Master Thread

FORK

Team

DO/for loop

JOIN

Master Thread

Source – UB CCR Handouts
Algorithm 2 Hybrid parallel BFS with vertex partitioning.

Input: \(G(V,E)\), source vertex \(s\).

Output: \(d[1..n]\), where \(d[v]\) gives the length of the shortest path from \(s\) to \(v \in V\).

1: for all \(v \in V\) do
2: \(d[v] \leftarrow \infty\)
3: \(level \leftarrow 1\), \(FS \leftarrow \phi\), \(NS \leftarrow \phi\)
4: \(op_i \leftarrow \text{find_owner}(s)\)
5: if \(op_i = \text{rank}\) then
6: push \(s \rightarrow FS\)
7: \(d[s] \leftarrow 0\)
8: for \(0 \leq j < p\) do
9: \(SendBuf_j \leftarrow \phi\) \(\triangleright p\) shared message buffers
10: \(RecvBuf_j \leftarrow \phi\) \(\triangleright\) for MPI communication
11: \(tBuf_{ij} \leftarrow \phi\) \(\triangleright\) thread-local stack for thread \(i\)
12: while \(FS \neq \phi\) do
13: for each \(u\) in \(FS\) in parallel do
14: for each neighbor \(v\) of \(u\) do
15: \(p_v \leftarrow \text{find_owner}(v)\)
16: push \(v \rightarrow tBuf_{ip_v}\)
17: Thread Barrier
18: for \(0 \leq j < p\) do
19: Merge thread-local \(tBuf_{ij}\)'s in parallel, form \(SendBuf_i\)
20: Thread Barrier
21: All-to-all collective step with the master thread:
Send data in \(SendBuf\), aggregate
newly-visited vertices into \(RecvBuf\)
22: Thread Barrier
23: for each \(u\) in \(RecvBuf\) in parallel do
24: if \(d[u] = \infty\) then
25: \(d[u] \leftarrow level\)
26: push \(u \rightarrow NS_i\)
27: Thread Barrier
28: \(FS \leftarrow \bigcup NS_i\) \(\triangleright\) thread-parallel
29: Thread Barrier

Source - https://people.eecs.berkeley.edu/~aydin/sc11_bfs.pdf
# Results Pure MPI

<table>
<thead>
<tr>
<th>Vertices</th>
<th>Approx. Edges</th>
<th>Seq</th>
<th>MPI 2</th>
<th>MPI 4</th>
<th>MPI 8</th>
<th>MPI 16</th>
<th>MPI 32</th>
<th>MPI 64</th>
</tr>
</thead>
<tbody>
<tr>
<td>20000</td>
<td>132,000,000</td>
<td>0.812</td>
<td>0.503</td>
<td>0.301</td>
<td>0.155</td>
<td>0.081</td>
<td>0.082</td>
<td>0.085</td>
</tr>
<tr>
<td>40000</td>
<td>528,000,000</td>
<td>3.977</td>
<td>2.02</td>
<td>1.12</td>
<td>0.502</td>
<td>0.312</td>
<td>0.165</td>
<td>0.155</td>
</tr>
<tr>
<td>80000</td>
<td>2,122,000,000</td>
<td>16.65</td>
<td>8.56</td>
<td>4.15</td>
<td>2.08</td>
<td>1.05</td>
<td>0.55</td>
<td>0.28</td>
</tr>
<tr>
<td>160000</td>
<td>8,448,000,000</td>
<td>69.9</td>
<td>36.67</td>
<td>18.47</td>
<td>8.95</td>
<td>4.65</td>
<td>2.36</td>
<td>1.2</td>
</tr>
</tbody>
</table>

Runtime in secs
Density of the graph is 33%
Pure MPI Results

![Graph 1](#)

- **10^8 Edges**

![Graph 2](#)

- **80^9 Edges**

Number of processors vs. runtime in sec.
## (MPI + OMP) Hybrid Results

<table>
<thead>
<tr>
<th>Vertices</th>
<th>Approx. Edges</th>
<th>MPI Processor</th>
<th>Pure MPI</th>
<th>OMP 2</th>
<th>OMP 4</th>
<th>OMP 8</th>
<th>OMP 16</th>
<th>OMP 32</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
<td>330000000</td>
<td>2</td>
<td>0.187</td>
<td>0.103</td>
<td>0.1</td>
<td>0.11</td>
<td>0.12</td>
<td>1.7</td>
</tr>
<tr>
<td>160000</td>
<td>8,448,000,000</td>
<td>2</td>
<td>36.67</td>
<td>18.81</td>
<td>8.7</td>
<td>4.4</td>
<td>3.9</td>
<td>3.9</td>
</tr>
<tr>
<td>160000</td>
<td>8,448,000,000,000</td>
<td>4</td>
<td>18.57</td>
<td>9.31</td>
<td>4.73</td>
<td>2.28</td>
<td>2.1</td>
<td>2.1</td>
</tr>
</tbody>
</table>

Runtime in secs
Hybrid (OMP + MPI) vs Pure MPI
Observations

- With Pure MPI we observe that for smaller input data after 16 MPI nodes we don’t see any gain in speed up. Whereas for larger input data we see that there is consistent gain in speedup even for 64 MPI processor.
- For Hybrid (OMP + MPI ) solution, There is a minor improvement in runtime when compared to Pure MPI if we use less then 16 OMP cores.
References

- https://people.eecs.berkeley.edu/~aydin/sc11_bfs.pdf [Parallel Breadth-First Search on Distributed Memory Systems]
- https://en.wikipedia.org/wiki/Parallel_breadth-first_search
- https://janth.home.xs4all.nl/MPIcourse/PDF/05_MPI_Collectives.pdf
- https://slurm.schedmd.com/