PARALLEL BREADTH-FIRST SEARCH USING MPI

CSE 633: Parallel Algorithms
Guide: Dr. Russ Miller
Presenter: Sumanth Thota
• Breadth-First Search
• Application of BFS
• Sequential BFS Algorithm
• Communication
• Parallel BFS Algorithm & implementation
• Results
• Conclusion
• References
Breadth-First Search

- BFS is a graph traversal algorithm that visits all the vertices of a graph in breadth-first order.
- It starts at the root node and visits all the nodes at the same level before moving on to the next level.
- BFS is typically used to find the shortest path between two nodes.
- One drawback of BFS is that it requires more memory as it needs to keep track of all the nodes in the queue.
Applications of BFS

• BFS is used by search engines like Google to crawl the web and index web pages.

• BFS can be used to find the shortest path between two users on a social networking site like Facebook or LinkedIn.

• BFS can be used to find the shortest path between two locations on a map as routing algorithms for navigation systems.

• BFS can be used in AI applications, such as pathfinding and decision-making.
Sequential BFS Algorithm

BreadthFirstSearch(G, A):

// G is a graph and A is the source node
Q = Queue()
// Create an empty queue
Q.enqueue(A)
// Enqueue the source node A
visited = set()
// Create an empty set of visited nodes
visited.add(A)
// Mark A as visited
while not Q.isEmpty:
    // While the queue is not empty
    B = Q.dequeue()
    // Dequeue the next node B from the queue
    for C in G.neighbors(B):
        // Process all the neighbors of B
        if C not in visited:
            // If C is not visited
            Q.enqueue(C)
            // Enqueue C
            visited.add(C)
            // Mark C as visited
Parallel BFS Algorithm

1. define 1D_distributed_memory_BFS(graph(V,E), source s):
2.     // normal initialization
3.     for all v in V do
4.         d[v] = -1;  
5.     d[s] = 0; level = 0; FS = {}; NS = {};
6.     // begin BFS traversal
7.     while True do:
8.         FS = {the set of local vertices with level}
9.         // all vertices traversed
10.        if FS = {} for all processors then:
11.            terminate the while loop
12.        // construct the NS based on local vertices in current frontier
13.        NS = {neighbors of vertices in FS, both local and not local vertices}
14.        // synchronization: all-to-all communication
15.        for 0 <= j < p do:
16.            N_j = {vertices in NS owned by processor j}
17.            send N_j to processor j
18.        receive N_j_rcv from processor j
19.        // combine the received message to form local next vertex frontier then update the level for them
20.        NS_rcv = Union(N_j_rcv)
21.        for v in NS_rcv and d[v] == -1 do:
22.            d[v] = level + 1

Source: https://en.wikipedia.org/wiki/Parallel_breadth-first_search
Communication
Parallel BFS Implementation

- Creating the data offset and sending it to all processors.

```c
// parallel processing
void parallel(int n, int* adjacency_matrix, int rank, int size, int save)
{
    int level = 0;
    bool alive = true;
    std::queue<int> fs, ns; // "frontier" queue and queue for the next level
    std::vector<bool> used(n);
    std::vector<int> d(n); // distance to vertices
    int* sendcounts = (int*)malloc(sizeof(int) * size);
    int* displs = (int*)malloc(sizeof(int) * size);

    // calculate the number of vertices and offset in the adjacency matrix
    int count = n;
    for (int i = 0; i < size - 1; i++)
    {
        sendcounts[i] = (n / size) * n;
        displs[i] = (n - count) * n;
        count -= (n / size);
    }
    sendcounts[size - 1] = count * n;
    displs[size - 1] = (n - count) * n;

    // adjacency matrix distribution for each rank
    int* adjacency_thread = (int*)malloc(sizeof(int) * n * n);
    MPI_Scatter(adjacency_matrix, sendcounts, displs, MPI_INT, adjacency_thread, n * n, MPI_INT, 0, MPI_COMM_WORLD);
}```
Parallel BFS Implementation

- Pop and push operations on FS and NS for all processors.

```c
if (rank == 0)
{
    fs.push(0);
    used[0] = true;
    d[0] = level;
}

// processing is in progress while the "frontier" queue of at least one of the ranks is not empty
while (alive)
{
    level++;
    // while the queue at the current level is not empty, we look through the vertices
    while (!fs.empty())
    {
        int v = fs.front();
        fs.pop();
        for (int i = 0; i < n; i++)
        {
            // not previously visited edge enqueued for the next level and mark visited
            int to = adjacency_thread[adjacent_vertex(n, size, v) * n + i];
            if (to == i && used[i])
            {
                used[i] = true;
                ns.push(i);
            }
        }
    }
}
MPI_Barrier(MPI_COMM_WORLD);
```
Parallel BFS Implementation

- Merging the received data and sending the updated data to all processors

```c
if (rank == 0)
{
    // if the rank is root, then we accept queue from other ranks, otherwise we send
    bool recv_q = (bool*)malloc(sizeof(bool));
    memcpy(recv_q, send_q, sizeof(bool) * n);
    for (int i = 1; i < size; i++)
    {
        MPI_Recv(send_q, n, MPI_C_BOOL, i, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        for (int j = 0; j < n; j++)
        {
            if (send_q[j] == true)
            {
                recv_q[j] = true;
                used[j] = true;
                if (d[j] == 0 && j != 0) d[j] = level;
            }
        }
        for (int i = 0; i < n / size; i++) if (recv_q[i]) fs.push(i);
        for (int i = 1; i < size; i++) MPI_Send(recv_q, n, MPI_C_BOOL, i, MPI_COMM_WORLD);
        free(recv_q);
    }
    else
    {
        MPI_Send(send_q, n, MPI_C_BOOL, 0, rank, MPI_COMM_WORLD); // send next level queue to rank 0
        MPI_Recv(send_q, n, MPI_C_BOOL, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE); // receive the next level common queue
        for (int i = 0; i < n / size; i++) if (recv_q[i]) fs.push(i);
        else for (int i = 0; i < n / size; i++) if (recv_q[i]) fs.push(i);
        MPI_Barrier(MPI_COMM_WORLD);
    }
}
Sample Output

Generating adjacency matrix...
Serial processing, please wait...
TIME: 2.629992 seconds
Parallel processing, please wait...
TIME: 1.182144 seconds
Slurm Script

```bash
#!/bin/sh
#SBATCH --nodes=8
#SBATCH --ntasks-per-node=1
#SBATCH --constraint=18|OPA
#SBATCH --time=00:10:00
#SBATCH --partition-general-compute
#SBATCH --qos=general-compute
#SBATCH --mail-type=END
#SBATCH --mail-user=shota@buffalo.edu
#SBATCH --job-name="test"
#SBATCH --output-pbfs.out
#SBATCH --exclusive
#SBATCH --mem=200G

module load intel
module list
export I_MPI_PMI_LIBRARY=/usr/lib64/libpmi.so
source /util/academic/intel/20.2/compilers_and_libraries_2020.2.254/Linux/mpi/intel64/bin/mpivars.sh
unset I_MPI_PMI_LIBRARY
mpicc -o pbfs pbfs.cpp
srun --n 8 ./pbfs
```
Execution time VS No. of vertices with constant 8 PE

Logarithmic graph

<table>
<thead>
<tr>
<th>No of vertices</th>
<th>Execution time serial</th>
<th>Execution time parallel w 8 PE</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>0.000049</td>
<td>0.001647</td>
</tr>
<tr>
<td>16</td>
<td>0.000051</td>
<td>0.004066</td>
</tr>
<tr>
<td>32</td>
<td>0.000061</td>
<td>0.001355</td>
</tr>
<tr>
<td>64</td>
<td>0.000152</td>
<td>0.001477</td>
</tr>
<tr>
<td>128</td>
<td>0.000483</td>
<td>0.001041</td>
</tr>
<tr>
<td>256</td>
<td>0.001925</td>
<td>0.001688</td>
</tr>
<tr>
<td>512</td>
<td>0.007164</td>
<td>0.005234</td>
</tr>
<tr>
<td>1024</td>
<td>0.027783</td>
<td>0.012878</td>
</tr>
<tr>
<td>2048</td>
<td>0.071938</td>
<td>0.03344</td>
</tr>
<tr>
<td>4096</td>
<td>0.26456</td>
<td>0.120046</td>
</tr>
<tr>
<td>8192</td>
<td>1.038307</td>
<td>0.462638</td>
</tr>
<tr>
<td>16384</td>
<td>4.149131</td>
<td>1.825428</td>
</tr>
<tr>
<td>32768</td>
<td>16.324268</td>
<td>6.438635</td>
</tr>
</tbody>
</table>
Execution time VS No. of processors

- The Execution time reduces linearly with increase in the number of processors
- The decrease in execution time gets more signification in more vertices

<table>
<thead>
<tr>
<th>No of Processors</th>
<th>4000 vertices</th>
<th>8000 vertices</th>
<th>16000 vertices</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.1963</td>
<td>0.7855</td>
<td>3.1379</td>
</tr>
<tr>
<td>2</td>
<td>0.1513</td>
<td>0.6113</td>
<td>2.4469</td>
</tr>
<tr>
<td>4</td>
<td>0.0830</td>
<td>0.3272</td>
<td>1.3138</td>
</tr>
<tr>
<td>8</td>
<td>0.0650</td>
<td>0.3235</td>
<td>1.1504</td>
</tr>
<tr>
<td>16</td>
<td>0.0883</td>
<td>0.3146</td>
<td>1.0477</td>
</tr>
<tr>
<td>32</td>
<td>0.1482</td>
<td>0.3023</td>
<td>0.8340</td>
</tr>
<tr>
<td>64</td>
<td>0.2119</td>
<td>0.3136</td>
<td>0.7888</td>
</tr>
<tr>
<td>128</td>
<td>0.3081</td>
<td>0.4961</td>
<td>0.9360</td>
</tr>
</tbody>
</table>

1 task per node
Execution time VS No. of processors

- The experiment shows a distinct increase in the execution time as the number of processors exceeds 32.

<table>
<thead>
<tr>
<th>No of Processors</th>
<th>4000 vertices</th>
<th>8000 vertices</th>
<th>16000 vertices</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>0.0831</td>
<td>0.3301</td>
<td>1.2770</td>
</tr>
<tr>
<td>16</td>
<td>0.0647</td>
<td>0.2643</td>
<td>1.1593</td>
</tr>
<tr>
<td>32</td>
<td>0.0733</td>
<td>0.2440</td>
<td>1.0018</td>
</tr>
<tr>
<td>64</td>
<td>0.2130</td>
<td>0.3789</td>
<td>1.0333</td>
</tr>
<tr>
<td>128</td>
<td>0.3319</td>
<td>0.5918</td>
<td>1.3178</td>
</tr>
</tbody>
</table>
Speedup VS No. of processors

\[ \text{Speed up} = \frac{T_{\text{serial}}}{T_{\text{Parallel}}} \]

- For smaller number of graph there isn’t much improvement in the speedup.
- As we increase the number of vertices the speedup increases with processors.
- After one point the trend starts going down even for large number of vertices.
- Likely because of the parallel communication overhead.

<table>
<thead>
<tr>
<th>No of Processors</th>
<th>speedup for 4000</th>
<th>speedup for 8000</th>
<th>speedup for 16000</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.3114</td>
<td>0.9461</td>
<td>0.8867</td>
</tr>
<tr>
<td>2</td>
<td>0.3976</td>
<td>1.2157</td>
<td>1.1397</td>
</tr>
<tr>
<td>4</td>
<td>0.7099</td>
<td>2.1888</td>
<td>2.0592</td>
</tr>
<tr>
<td>8</td>
<td>1.0136</td>
<td>3.1585</td>
<td>3.0337</td>
</tr>
<tr>
<td>16</td>
<td>1.2496</td>
<td>3.9910</td>
<td>3.7524</td>
</tr>
<tr>
<td>32</td>
<td>1.4026</td>
<td>4.5546</td>
<td>4.2779</td>
</tr>
<tr>
<td>64</td>
<td>0.6290</td>
<td>4.0648</td>
<td>4.5996</td>
</tr>
<tr>
<td>128</td>
<td>0.3900</td>
<td>2.6419</td>
<td>3.3467</td>
</tr>
</tbody>
</table>
Bar graph comparison for all three types of execution

Logarithmic Bar Graph
Conclusion

• As can be interpreted form the graphs that the algorithm for parallel BFS is working effectively.

• The trends in the graphs reflect that parallelizing the process make it more efficient in terms of execution time but only up to a certain number of processors.

• For the input size of around 30K vertices, making the adjacency matrix size (30K * 30k), we can see the algorithm works effectively until 32 processors. From there we can observe the increase in the execution time with increase in processors due to communication over head.
References

- [https://en.wikipedia.org/wiki/Parallel_breadth-first_search](https://en.wikipedia.org/wiki/Parallel_breadth-first_search)
- [https://people.eecs.berkeley.edu/~aydin/sc11_bfs.pdf](https://people.eecs.berkeley.edu/~aydin/sc11_bfs.pdf)
- [https://docs.ccr.buffalo.edu/en/latest/hpc/data-transfer/#data-transfer](https://docs.ccr.buffalo.edu/en/latest/hpc/data-transfer/#data-transfer)
Thank You