

IMPLEMENTING PARALLELISM IN BFS

Kaushik Naraynan Saaketh Balachendil

March 28, 2024

Language used

C++

Libraries used

OpenMP

Algorithm

- 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.
- The adj matrix is divided into P blocks of size $N/\sqrt{p} \times N/\sqrt{p}$.
- Vertex are partitioned into N/P size groups.
- $N - i$ number of processors
- $p - i$ number of threads (for our case it is 4).
- Do a transpose of the frontier vector between the processors.
- After this all the columns processor s will have matching frontier with their local adjacency matrix.
- We then do a column wise all gather for the frontier vertices.
- This will broadcast the required frontier vertices for each column.
- 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.
- 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.
- 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.

Code

```

1  /*
2  * BFS Parallelization for undirected graphs
3  * Uses 4 threads to parallelize the traversal
4  * Uses 2 threads to populate frontier queues
5  */
6
7  //Necessary Libraries
8  #include <bits/stdc++.h>
9  #include <omp.h>
10 using namespace std;
11
12 //Defining Threads used
13 #define NUM_THREADS 4
14 #define POP_THREADS 2
15
16 //discovering for a specific queue and is shared among the threads
17 void discoverLevel(int &N,int type,vector<queue<pair<string,int>>>
    &q,vector<queue<pair<string,int>>> &tq,int &goal,vector<string>
    &p,vector<vector<int>> &G){
18     int l, r;
19     if(type % 2 == 0){
20         l = 1;
21         r = N;
22         r /= 2;
23     }
24     else{
25         l = N;
26         l /= 2;
27         l++;
28         r = N;
29     }
30     while(!q[type].empty()){
31         pair<string,int> node = q[type].front();

```

```

32     q[type].pop();
33     if(node.second == goal) p.push_back(node.first);
34     else{
35         int nodeidx = node.second;
36         string path = node.first;
37         for(int nextnode=1; nextnode<=r; nextnode++){
38             if(G[nodeidx][nextnode] == 1){
39                 string newpath = path + "->" + to_string(
nextnode);
40                 G[nodeidx][nextnode] = -1;
41                 G[nextnode][nodeidx] = -1;
42                 if(nextnode == goal){
43                     p.push_back(newpath);
44                     continue;
45                 }
46                 tq[type].push({newpath,nextnode});
47             }
48         }
49     }
50 }
51 return;
52 }
53
54 //random graph generator
55 // void RandomGraph(int &N,vector<vector<int>> &G){
56 //     srand(time(NULL));
57 //     for(int i=1;i<=N;i++){
58 //         for(int j=i;j<=N;j++){
59 //             int edge = rand() % 2;
60 //             G[i][j] = edge;
61 //             G[j][i] = edge;
62 //         }
63 //     }
64 //     return;
65 // }
66 void RandomGraph(int &N, vector<vector<int>> &G) {
67     srand(time(NULL));
68     for (int i = 1; i <= N; i++) {
69         for (int j = i; j <= N; j++) {
70             int edge = rand() % 2;
71             G[i][j] = edge;
72             G[j][i] = edge;
73         }
74     }
75     // Add additional edges to ensure there are at least N/2 edges
76     int edgesCount = 0;
77     for (int i = 1; i <= N; i++) {
78         for (int j = i + 1; j <= N; j++) {
79             if (G[i][j] == 1) edgesCount++;
80         }
81     }
82     while (edgesCount < N / 2) {
83         int i = rand() % N + 1;
84         int j = rand() % N + 1;
85         if (i != j && G[i][j] == 0) {
86             G[i][j] = 1;
87             G[j][i] = 1;

```

```

88         edgesCount++;
89     }
90 }
91     return;
92 }
93
94
95 //custom graph generator
96 void CustomGraph(vector<vector<int>> &G){
97     int u, v, edges;
98     cout << "Enter the number of edges: ";
99     cin >> edges;
100     for(int i=0;i<edges;i++){
101         cout << "Enter u & v for edge connection: ";
102         cin >> u;
103         cin >> v;
104         G[u][v] = 1;
105         G[v][u] = 1;
106     }
107     return;
108 }
109
110 int main(){
111     //Initializing parameters
112     int N, op, s, g;
113     cout << "Define the number of nodes in your Graph: ";
114     cin >> N;
115     cout << "Enter 1 to generate a random graph, 2 for your own
graph: ";
116     cin >> op;
117
118     //Graph Generation
119     vector<vector<int>> G(N+1,vector<int>(N+1,0));
120     if(op == 1) RandomGraph(N,G);
121     else CustomGraph(G);
122     vector<vector<int>> GCopy = G;
123
124     //Queues for the parallelism
125     vector<queue<pair<string,int>>> q(NUM_THREADS);
126     vector<queue<pair<string,int>>> tq(NUM_THREADS);
127     cout << "Enter root node: ";
128     cin >> s;
129     cout << "Enter goal node: ";
130     cin >> g;
131     if(s <= (N / 2)){
132         q[0].push({to_string(s),s});
133         q[1].push({to_string(s),s});
134     }
135     else{
136         q[2].push({to_string(s),s});
137         q[3].push({to_string(s),s});
138     }
139
140     //Running BFS in parallel
141     double start_time = omp_get_wtime();
142     vector<string> paths;
143     while((paths.size() == 0) && (!q[0].empty() || !q[1].empty() ||

```

```

144     !q[2].empty() || !q[3].empty())){
145         //Running BFS Traversal in 4 parallel threads
146         omp_set_num_threads(NUM_THREADS);
147         #pragma omp parallel
148         {
149             int thread_id = omp_get_thread_num();
150             discoverLevel(N,thread_id,q,tq,g,paths,G);
151         }
152
153         //Populating frontier queues in 2 parallel threads
154         omp_set_num_threads(POP_THREADS);
155         #pragma omp parallel
156         {
157
158             int thread_id = omp_get_thread_num();
159             #pragma omp critical
160             {
161                 if(thread_id == 0){
162                     while(!tq[0].empty()){
163                         q[0].push(tq[0].front());
164                         q[1].push(tq[0].front());
165                         tq[0].pop();
166                     }
167                     while(!tq[2].empty()){
168                         q[0].push(tq[2].front());
169                         q[1].push(tq[2].front());
170                         tq[2].pop();
171                     }
172                 }
173                 else{
174                     while(!tq[1].empty()){
175                         q[2].push(tq[1].front());
176                         q[3].push(tq[1].front());
177                         tq[1].pop();
178                     }
179                     while(!tq[3].empty()){
180                         q[2].push(tq[3].front());
181                         q[3].push(tq[3].front());
182                         tq[3].pop();
183                     }
184                 }
185             }
186         }
187     }
188 }
189 double end_time = omp_get_wtime();
190
191 //Printing Solutions and execution time of algorithm
192 cout << "----Parallelized BFS----\n";
193 for(auto sol : paths) cout << sol << "\n";
194 cout << "Computed in " << end_time - start_time << " units of
time\n";
195
196 //Non-Parallelized BFS
197 start_time = omp_get_wtime();
198 queue<pair<int,pair<string,int>>> Q;

```

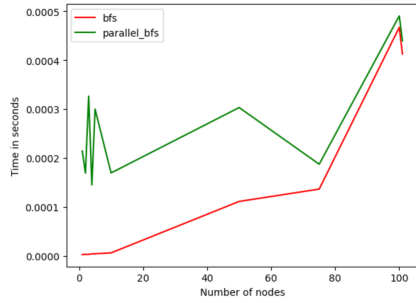
```

199     Q.push({0,{to_string(s),s}});
200     vector<string> ans;
201     while(!Q.empty()){
202         pair<int,pair<string,int>> cur = Q.front();
203         Q.pop();
204         int nodeidx = cur.second.second, level = cur.first;
205         string path = cur.second.first;
206         if(nodeidx == g){
207             ans.push_back(path);
208             while(!Q.empty() && Q.front().first == level){
209                 if(Q.front().second.second == g) ans.push_back(Q.
front().second.first);
210                 Q.pop();
211             }
212             break;
213         }
214         else{
215             for(int i=1;i<=N;i++){
216                 if(GCopy[nodeidx][i] == 1){
217                     GCopy[nodeidx][i] = -1;
218                     GCopy[i][nodeidx] = -1;
219                     string newpath = path + "->" + to_string(i);
220                     Q.push({level+1,{newpath,i}});
221                 }
222             }
223         }
224     }
225     end_time = omp_get_wtime();
226
227     //Printing solutions and time taken for the non parallelized
BFS
228     cout << "\n----Normal BFS----\n";
229     for(auto p : ans) cout << p << "\n";
230     cout << "Computed in " << end_time - start_time << " units of
time\n";
231     return 0;
232 }

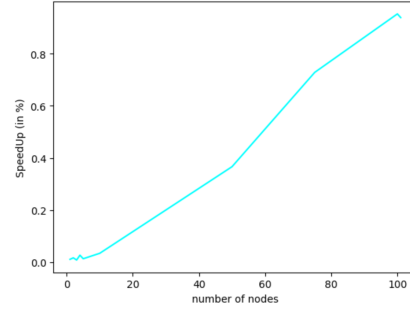
```

Results

- For less than or equal to 100 nodes sequential bfs outperforms parallelized bfs:

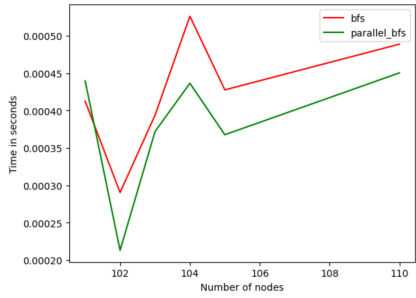


(a) Performance in seconds(upto 99 nodes)

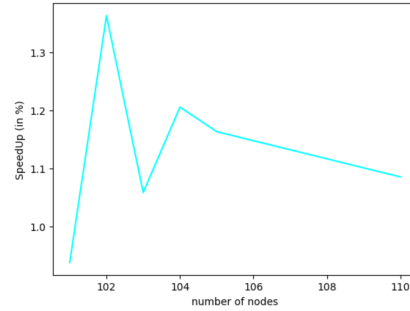


(b) Speed Up %(upto 99 nodes)

- When there are more than 100 nodes we can see parallelized bfs starts outperforming sequential bfs:

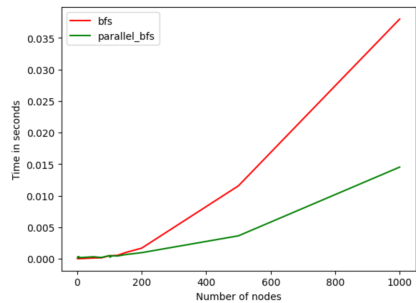


(c) Performance in seconds (100-110 nodes)

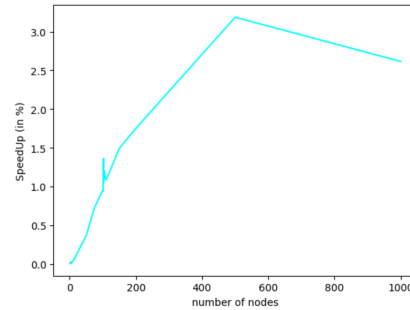


(d) Speed Up %

- The code is then run for a graph with 200,500 and 1000 nodes and here are the results:



(e) Performance in seconds (200,500 and 1000 nodes)



(f) Speed Up % (200,500 and 1000 nodes)