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) x N/sqrt(p).
- Vertex are partitioned into N/P size groups.
- N -¿ number of processors
- p -¿ 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

```
/*
* BFS Parallelization for undirected graphs
* Uses 4 threads to parallelize the traversal
* Uses 2 threads to populate frontier queues
*/

//Necessary Libraries
#include <bits/stdc++.h>
#include <omp.h>
using namespace std;

//Defining Threads used
#define NUM_THREADS 4
#define POP_THREADS 2

//discovering for a specific queue and is shared among the threads
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){
    int l, r;
    if(type % 2 == 0){
        l = 1;
        r = N;
        r /= 2;
    }
    else{
        l = N;
        l /= 2;
        l++;
        r = N;
    }
    while(!q[type].empty()){
        pair<string,int> node = q[type].front();
```

```
        q[type].pop();
        if(node.second == goal) p.push_back(node.first);
        else{
            int nodeidx = node.second;
            string path = node.first;
            for(int nextnode=l; nextnode<=r; nextnode++){
                if(G[nodeidx][nextnode] == 1){
                    string newpath = path + "->" + to_string(
    nextnode);
                    G[nodeidx][nextnode] = -1;
                    G[nextnode][nodeidx] = -1;
                    if(nextnode == goal){
                        p.push_back(newpath);
                        continue;
                    }
                    tq[type].push({newpath,nextnode});
                }
            }
        }
    }
    return;
}

//random graph generator
// void RandomGraph(int &N,vector<vector<int>> &G){
//     srand(time(NULL));
//     for(int i=1;i<=N;i++){
//         for(int j=i;j<=N;j++){
//             int edge = rand() % 2;
//             G[i][j] = edge;
//             G[j][i] = edge;
//         }
//     }
//     return;
// }
void RandomGraph(int &N, vector<vector<int>> &G) {
    srand(time(NULL));
    for (int i = 1; i <= N; i++) {
        for (int j = i; j <= N; j++) {
            int edge = rand() % 2;
            G[i][j] = edge;
            G[j][i] = edge;
        }
    }
    // Add additional edges to ensure there are at least N/2 edges
    int edgesCount = 0;
    for (int i = 1; i <= N; i++) {
        for (int j = i + 1; j <= N; j++) {
            if (G[i][j] == 1) edgesCount++;
        }
    }
    while (edgesCount < N / 2) {
        int i = rand() % N + 1;
        int j = rand() % N + 1;
        if (i != j && G[i][j] == 0) {
            G[i][j] = 1;
            G[j][i] = 1;
```

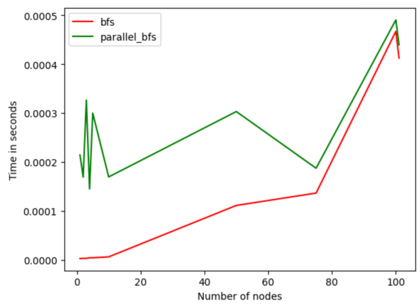
```
            edgesCount++;
        }
    }
    return;
}


//custom graph generator
void CustomGraph(vector<vector<int>> &G){
    int u, v, edges;
    cout << "Enter the number of edges: ";
    cin >> edges;
    for(int i=0;i<edges;i++){
        cout << "Enter u & v for edge connection: ";
        cin >> u;
        cin >> v;
        G[u][v] = 1;
        G[v][u] = 1;
    }
    return;
}

int main(){
    //Initializing parameters
    int N, op, s, g;
    cout << "Define the number of nodes in your Graph: ";
    cin >> N;
    cout << "Enter 1 to generate a random graph, 2 for your own
    graph: ";
    cin >> op;

    //Graph Generation
    vector<vector<int>> G(N+1,vector<int>(N+1,0));
    if(op == 1) RandomGraph(N,G);
    else CustomGraph(G);
    vector<vector<int>> GCopy = G;

    //Queues for the parallelism
    vector<queue<pair<string,int>>> q(NUM_THREADS);
    vector<queue<pair<string,int>>> tq(NUM_THREADS);
    cout << "Enter root node: ";
    cin >> s;
    cout << "Enter goal node: ";
    cin >> g;
    if(s <= (N / 2)){
        q[0].push({to_string(s),s});
        q[1].push({to_string(s),s});
    }
    else{
        q[2].push({to_string(s),s});
        q[3].push({to_string(s),s});
    }

    //Running BFS in parallel
    double start_time = omp_get_wtime();
    vector<string> paths;
    while((paths.size() == 0) && (!q[0].empty() || !q[1].empty() ||
```

```
     !q[2].empty() || !q[3].empty())){

        //Running BFS Traversal in 4 parallel threads
        omp_set_num_threads(NUM_THREADS);
        #pragma omp parallel
        {
            int thread_id = omp_get_thread_num();
            discoverLevel(N,thread_id,q,tq,g,paths,G);
        }

        //Populating frontier queues in 2 parallel threads
        omp_set_num_threads(POP_THREADS);
        #pragma omp parallel
        {

            int thread_id = omp_get_thread_num();
            #pragma omp critical
            {
                if(thread_id == 0){
                while(!tq[0].empty()){
                    q[0].push(tq[0].front());
                    q[1].push(tq[0].front());
                    tq[0].pop();
                }
                while(!tq[2].empty()){
                    q[0].push(tq[2].front());
                    q[1].push(tq[2].front());
                    tq[2].pop();
                }
            }
                else{
                    while(!tq[1].empty()){
                        q[2].push(tq[1].front());
                        q[3].push(tq[1].front());
                        tq[1].pop();
                    }
                    while(!tq[3].empty()){
                        q[2].push(tq[3].front());
                        q[3].push(tq[3].front());
                        tq[3].pop();
                    }
                }
            }

        }
    }
    double end_time = omp_get_wtime();

    //Printing Solutions and execution time of algorithm
    cout << "----Parallelized BFS----\n";
    for(auto sol : paths) cout << sol << "\n";
    cout << "Computed in " << end_time - start_time << " units of time\n";

    //Non-Parallelized BFS
    start_time = omp_get_wtime();
    queue<pair<int,pair<string,int>>> Q;
```

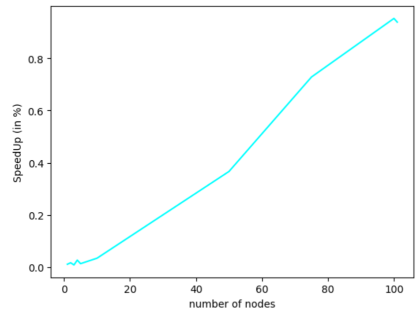
```
    Q.push({0,{to_string(s),s}});
    vector<string> ans;
    while(!Q.empty()){
        pair<int,pair<string,int>> cur = Q.front();
        Q.pop();
        int nodeidx = cur.second.second, level = cur.first;
        string path = cur.second.first;
        if(nodeidx == g){
            ans.push_back(path);
            while(!Q.empty() && Q.front().first == level){
                if(Q.front().second.second == g) ans.push_back(Q.
    front().second.first);
                Q.pop();
            }
            break;
        }
        else{
            for(int i=1;i<=N;i++){
                if(GCopy[nodeidx][i] == 1){
                    GCopy[nodeidx][i] = -1;
                    GCopy[i][nodeidx] = -1;
                    string newpath = path + "->" + to_string(i);
                    Q.push({level+1,{newpath,i}});
                }
            }
        }
    }
    end_time = omp_get_wtime();

    //Printing solutions and time taken for the non parallelized
    BFS
    cout << "\n----Normal BFS----\n";
    for(auto p : ans) cout << p << "\n";
    cout << "Computed in " << end_time - start_time << " units of
    time\n";
    return 0;
}
```

Results

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

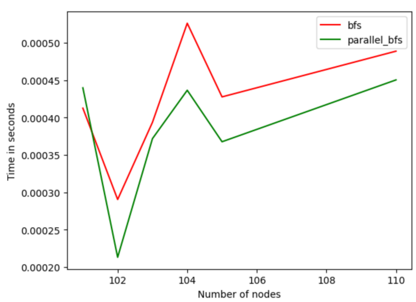


(a) Performance in seconds(upto 99 nodes)

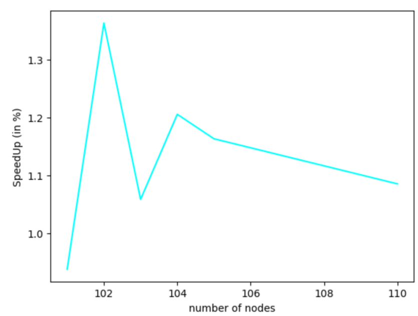


(b) Speed Up %(upto 99 nodes)

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

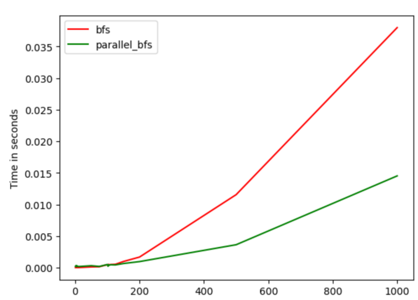


(c) Performance in seconds (100-110 nodes)

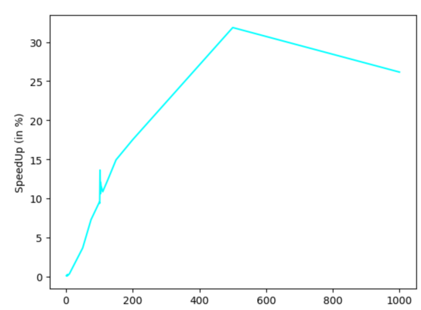


(d) Speed Up %

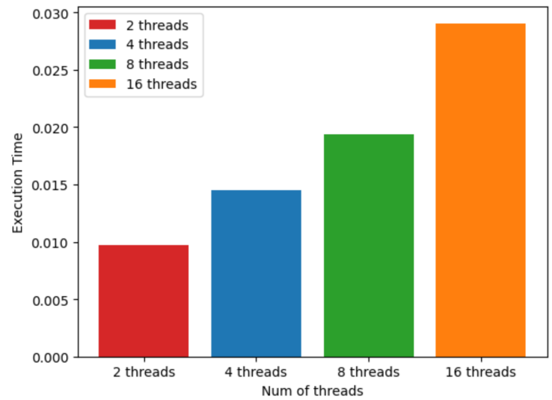
- The code is then run for a graph with 200,500 and 1000 nodes and here are the results:



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



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



(g) Comparison based on threads (for 1000 nodes)