IJARCCE

ISSN (Online) 2278-1021
ISSN (Print) 2319 5940

International Journal of Advanced Research in Computer and Communication Engineering
Vol. 5, Issue 5, May 2016

A Naïve Breadth First Search Approach Incorporating Parallel Processing Technique For Optimal Network Traversal

Laxmikant Revdikar[1], Ayush Mittal[2], Anuj Sharma[3], Dr. Sunanda Gupta[4]

Shri Mata Vaishno Devi University, Katra, Jammu & Kashmir, India[1,2,3]

Dept. of Computer Science & Engineering, Shri Mata Vaishno Devi University, Katra, India[4]

Abstract: The authors have modied the existing breadth first search (BFS) technique by incorporating a parallel processing feature to it. A multithreaded implementation of breadth-first search (BFS) of a graph using Open MP. the results of our research reveal that implementing BFS using multiprocessor runs much faster than the standard BFS.

Keywords: Breadth-First Search, Parallel Programming, optimal Network Traversal Open MP.

I. INTRODUCTION

As we know that a graph can be used to represent any network so using graph theory approaches we can traverse the network, there are many graph traversal algorithms [1] such as

1) Depth First Search
2) Breadth First-Search
3) A*
4) Dijkstra
5) Prim
6) Kruskal
7) Floyd Warshall
8) Bellman Ford

Here we will use BFS and change it run for multi-processor systems.

A. BREADTH FIRST-SEARCH

BFS explores the vertices and edges of a graph, beginning from a specified "starting vertex" that we'll call s. It assigns each vertex a "level" number, which is the smallest number of hops in the graph it takes to reach that vertex from s. BFS begins by assigning s itself level 0. It first visits all the "neighbors" of s, which are the vertices that can be reached from s by following one edge, and assigns them level 1.

Then it visits the neighbors of the level-1 vertices: some of those neighbors might already be on level 0 or 1, but any that haven't already been assigned a level get level 2. And so on -- the so-far-unreached neighbors of level-2 vertices get level 3, then 4, and so forth until there are no more unreached[2].

BFS uses FIFO queue to decide what vertices to visit next. The queue starts out with only s on it, with level[s] =0. Then the general step is to take the front vertex v from the queue and visit all its neighbors. Any neighbor that hasn't yet been visited is added to the back of the queue and assigned a level one larger than LEVEL [v].

IT IS USEFUL IN
1) Social Media[3]
2) Logistic
3) E-Commerce[4]
4) Counter Terrorism
5) Fraud Detection[5]

For example, in the following graph (Fig. 1.), we start traversal from vertex 2. When we come to vertex 0, we look for all adjacent vertices of it. 2 is also an adjacent vertex of 0. If we don’t mark visited vertices, then 2 will be processed again and it will become a non-terminating process. A Breadth First Traversal of the following graph is 2, 0, 3, 1.

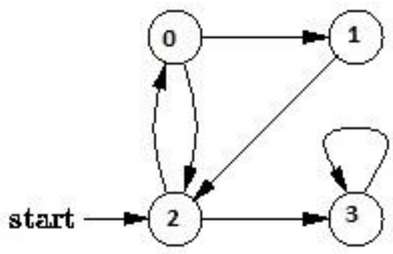


Fig. 1. Example of Breadth First-Search

B. PARALLEL BREADTH FIRST-SEARCH

The idea of doing BFS in parallel is that, in principal, you can process all the vertices on a single level at the same time. That is, once you've found all the level-1 vertices, you can do a parallel loop that explores from each of them to find level-2 vertices. Thus, the parallel code will have an important sequential loop over levels, starting at 0.

In the parallel code, it's possible that when you're processing[6] level i, two vertices v and w will both find the same level-i+1 vertex x as a neighbor. This will cause a data race when they both try to set level[x] =i+1, and also when they each try to set parent[x] to themselves. But if you're careful this is a "benign data race" -- it doesn't actually cause any problem, because there's no disagreement about what level[x] should be, and it doesn't matter in the end whether parent[x] turns out to be v or w.

C. OPENMP

OpenMp[7] is a set of compiler directives and library routines for parallel application programs. It is a parallel programming system that aims to be powerful and easy to use, while at the same time allowing the programmer to write high performance programs. Its initial focus was on numerical applications involving loops written in Fortran or C/C++, but it includes the necessary constructs to deal with more kinds of parallel algorithms.

Irregular parallel algorithms involve sub computations whose amount of work is not known in advance, and hence the work can only be distributed at runtime. Important subclasses include algorithms using task pools, as well as speculative algorithms. We are concentrating on the first type, although the problem and solutions we present apply to other types as well. Examples for irregular algorithms are search and sorting algorithms, graph algorithms, and more involved applications like volume rendering.

The expected graph (Fig. 2.) to be of sequential and Parallel BFS is given below:-

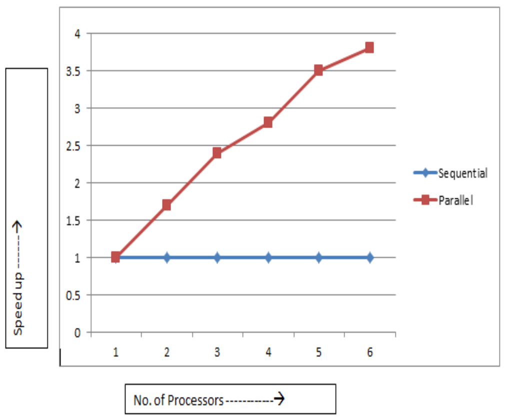


Fig. 2. Expected Speed up using parallel and sequential BFS

II. PURPOSE

Why BFS?
1) Least work/byte of the graph algorithms
2) Building blocks for many other graph algorithms

WHY PARALLEL BFS?

In past history when BFS was formed by scientist for traversing it was meant for single processor but we have opted for the parallel approach because nowadays its rarely practised and also there are systems which have multiple processor.

III. PROBLEM STATEMENT

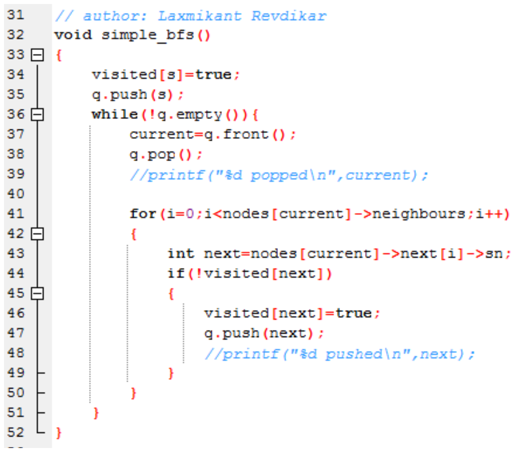
We Have To Convert Sequential BFS Code into Parallel BFS To Reduce Time Complexity In Traversing Of Graph

IV. GOAL & VISION

Our Goal Is To Reduce Time Complexity Of The Traversing Of The Graph. Our Vision Is To Use Openmp Api In Sequential BFS To Make It Parallel BFS.

V. SEQUENTIAL BFS

A. CODE SNIPPET



```
// author: Laxmikant Revdikar
void simple_bfs()
{
    visited[s]=true;
    q.push(s);
    while(!q.empty()){
        current=q.front();
        q.pop();
        //printf("%d popped\n",current);

        for(i=0;i<nodes[current]->neighbours;i++)
        {
            int next=nodes[current]->next[i]->sn;
            if(!visited[next])
            {
                visited[next]=true;
                q.push(next);
                //printf("%d pushed\n",next);
            }
        }
    }
}
```

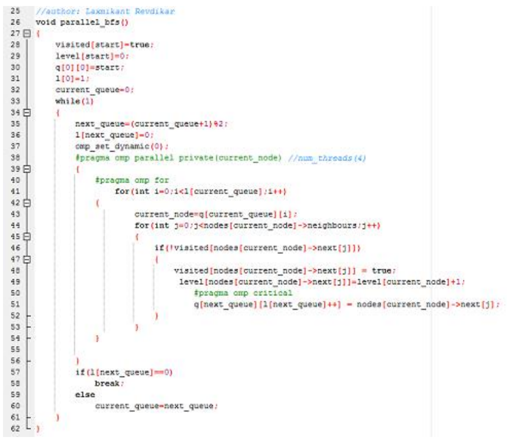
Fig. 3. Code Snippet of Sequential BFS in C++

B. EXPLANATION OF CODE

In the above code we have used a queue in which initially we enqueue start node and then change its visited mode and enqueue all its neighbour and then dequeue the start node and now we process the front node of queue and put all its neighbours into the queue and then dequeue it & change its visited mode, we repeat this process until the queue is empty. We created n input containing 99900 edges and then run the above code. The time taken to traverse all the edges was 0.036000.

VI. PARALLEL BFS

A. CODE SNIPPET



```
//author: Laxmikant Revdikar
void parallel_bfs()
{
    visited[start]=true;
    level[start]=0;
    q[0][0]=start;
    l[0]=1;
    current_queue=0;
    while(1)
    {
        next_queue=(current_queue+1)%2;
        l[next_queue]=0;
        omp_set_dynamic(0);
        #pragma omp parallel private(current_node) //num_threads(4)
        {
            #pragma omp for
                for(int i=0;i<l[current_queue];i++)
            {
                    current_node=q[current_queue][i];
                    for(int j=0;j<nodes[current_node]->neighbours;j++)
                    {
                        if(!visited[nodes[current_node]->next[j]])
                        {
                            visited[nodes[current_node]->next[j]] = true;
                             level[nodes[current_node]->next[j]]=level[current_node]+1;
                                #pragma omp critical
                                q[next_queue][l[next_queue]++] = nodes[current_node]->next[j];
                        }
                    }
            }

        }
        if(l[next_queue]==0)
            break;
        else
            current_queue=next_queue;
    }
}
```

Fig. 4. Code Snippet of Parallel BFS in C++

B. EXPLANATION OF CODE

In the parallel version of the BFS we have taken two array, current array and next array. Current array stores the nodes which are processing and the neighbours of all those nodes which have not been visited are kept in next array when we have visited all nodes of current array then we swap current array and next array and this keeps going on until the no. Of nodes in next array becomes zero and we have kept the section critical where the threads insert the value of nodes into the next array in critical section only one thread at a time will be able to enter that region, so it will prevent overwriting of same values by different threads thus maintaining the synchronization.

We created n input containing 99900 edges and then run the above code and the time taken to traverse the network was 0.01900.

VII. TESTING

Now we have used a same parallel code with different no. Threads for same input. Following is the graph representing time taken vs. No. Of threads

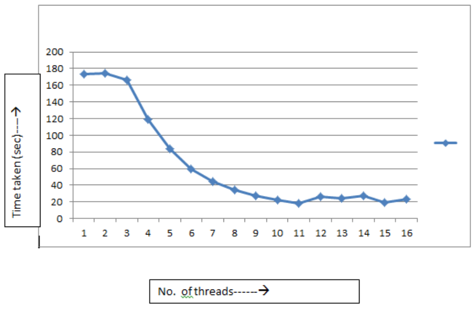


Fig. 5. Plotted Graph of Proposed Parallel BFS

VIII. CONCLUSION

We have studied about graph traversal algorithms one of them is BFS that is breadth first search algorithm and we have implemented a parallel approach to the sequential breadth first search. We experimented with some input graphs and then came to the result that on parallel processing with our proposed algorithm the graph is traversed much faster. We have also studied that by using different no. Of threads, different timings of traversing the graph is noted. To achieve the above we have used openmp with c++.

REFERENCES

[1]. Raynal, M. (2013). Basic Definitions and Network Traversal Algorithms. Distributed Algorithms for Message-Passing Systems, 3-34. doi:10.1007/978-3-642-38123-2_1
[2]. Bundy, A., & Wallen, L. (1984). Breadth-First Search. Catalogue of Artificial Intelligence Tools, 13-13. doi:10.1007/978-3-642-96868-6_25
[3]. Fay, D. (2016). Predictive Partitioning for Efficient BFS Traversal in Social Networks. Studies in Computational Intelligence Complex Networks VII, 11-26. doi:10.1007/978-3-319-30569-1_2
[4]. Bishop, M. (1999). A Breadth-First Strategy for Mating Search. Automated Deduction — CADE-16 Lecture Notes in Computer Science, 359-373. doi:10.1007/3-540-48660-7_32
[5]. Fraud Detection. (2014). Fraud and Fraud Detection A Data Analytics Approach, 7-15. doi:10.1002/9781118936764.ch2
[6]. Parallel Breadth-First Search on Distributed Memory Systems. (2011). doi:10.2172/1050644
[7]. A Quick Reference to OpenMP. (2001). Parallel Programming in OpenMP, 211-216. doi:10.1016/b978-155860671-5/50008-6.