Parallel BFS and DFS

Prepared for

Dr. J. Park

CSCI 176 - Parallel Processing

California State University, Fresno

Prepared by:

Nav Sanya Anand - (navsanya@mail.freesnostate.edu)

Github Link

Introduction

This report examines the solution of depth first search and breadth first search parallelly and compares the time taken. Depth-first search can be performed in parallel by partitioning the search space into many small, disjunct parts that can be explored concurrently. The breadth-first-search algorithm is a way to explore the vertices of a graph layer by layer. It is a basic algorithm in graph theory which can be used as a part of other graph algorithms.

Breadth First Search

In the conventional sequential BFS algorithm, two data structures are created to store the frontier and the next frontier. The frontier contains the vertices that have the same distance(it is also called "level") from the source vertex, these vertices need to be explored in BFS. Every neighbor of these vertices will be checked, some of these neighbors which are not explored yet will be discovered and put into the next frontier. At the beginning of the BFS algorithm, a given source vertex s is the only vertex in the frontier. All direct neighbors of s are visited in the first step, which form the next frontier. After each layer-traversal, the "next frontier" is switched to the frontier and new vertices will be stored in the new next frontier.

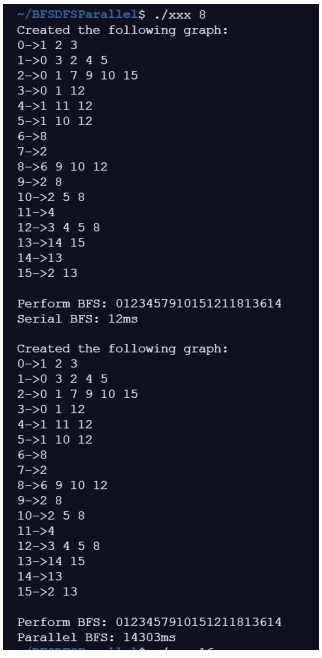
The following pseudo-code outlines the idea of it:

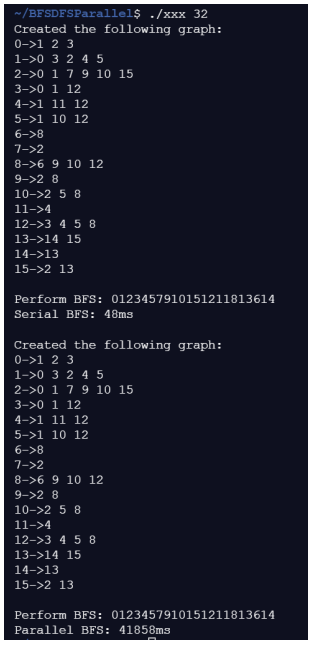
```
unmark all vertices
choose some starting vertex x
mark x
list L = x
tree T = x
while L nonempty
choose some vertex v from front of list
visit v
for each unmarked neighbor w
    mark w
    add it to end of list
    add edge vw to T
```

Time Complexity: O(V+E) where V is number of vertices in the graph and E is number of edges in the graph. The features of the BFS are space and time complexity, completeness, proof of completeness, and optimality. Space complexity refers to the proportion of the number of nodes at the deepest level of a search. Time complexity refers to the actual amount of 'time' used for considering every path a node will take in a search. Completeness is, essentially, a search that finds a solution in a graph regardless of what kind of graph it is. The proof of the completeness is the shallowest level at which a goal is found in a node at a definite depth. Finally, optimality refers to a BFS that is not weighted – that is a graph used for unit-step cost.

1D partitioning is the simplest way to combine the parallel BFS with distributed memory. It is based on a vertex partition. Load balancing is still an important issue for data partition, which determines how we can benefit from parallelization. In other words, each processor with distributed memory (e.g. processor) should be in charge of approximately the same number of vertices and their outgoing edges. The two for-loops can be executed in parallel. The update of the next frontier and the increase of distance need to be atomic. Atomic operations are program operations that can only run entirely without interruption and pause.

Code Output:





However, there are two problems in this simple parallelization. Firstly, the distance-checking and distance-updating operations introduce two benign races. The reason for

the race is that a neighbor of one vertex can also be the neighbor of another vertex in the frontier. As a result, the distance of this neighbor may be examined and updated more than one time. Although these races waste resources and lead to unnecessary overhead, with the help of synchronization, they don't influence the correctness of BFS, so these races are benign. Secondly, in spite of the speedup of each layer-traversal due to parallel processing, a barrier synchronization is needed after every layer in order to completely discover all neighbor vertices in the frontier.

Depth First Search

Depth first search is another way of traversing graphs, which is closely related to preorder traversal of a tree. Just like in BFS we can use marks to keep track of the vertices that have already been visited, and not visit them again. Also, just like in BFS, we can use this search to build a spanning tree with certain useful properties.

The following pseudo-code outlines the idea of it:

```
dfs(vertex v)
 {
 visit(v);
 for each neighbor w of v
    if w is unvisited
    {
    dfs(w);
    add edge vw to tree T
    }
 }
```

The overall depth first search algorithm then simply initializes a set of markers so we can tell which vertices are visited, chooses a starting vertex x, initializes tree T to x, and calls dfs(x). Just like in breadth first search, if a vertex has several neighbors it would be equally correct to go through them in any order. I didn't simply say "for each unvisited neighbor of v" because it is very important to delay the test for whether a vertex is visited until the recursive calls for previous neighbors are finished.

Search methods are useful when a problem can be formulated in terms of finding a solution path in an (implicit) directed graph from an initial node to a goal node. The search begins

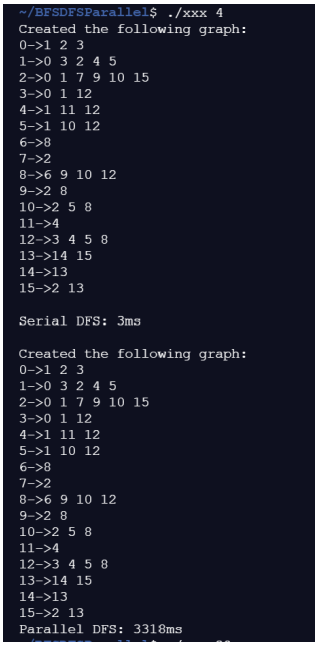
by expanding the initial node; i.e., by generating its successors. At each later step, one of the previously generated nodes is expanded until a goal node is found. (Now the solution path can be constructed by following backward pointers from the goal node to the initial node.) There are many ways in which a generated node can be chosen for expansion, each having its own advantages and disadvantages. In depth-first search, one of the most recently generated nodes is expanded first. Numbers on the right corners of the nodes show the order in which the nodes are expanded. Since deeper nodes are expanded first, the search is called depth-first.

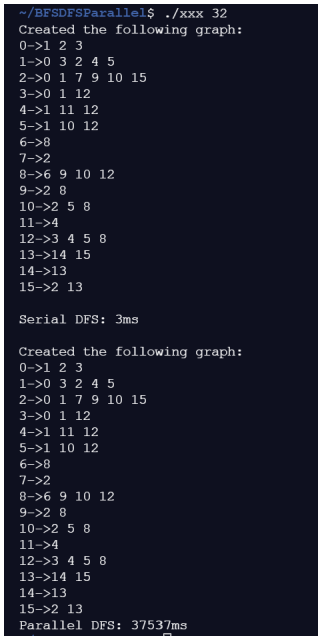
The main advantage of depth-first search over other search techniques is its low storage requirements. For most other search techniques the storage requirement is exponential in the length of the solution path, whereas for depth-first search, the storage requirement is linear in the depth of the space searched. But simple depth-first search has two major drawbacks.

1. If the search space to the left of the first goal node is infinite (or very large) then search would never terminate (or take a very long time).
2. It finds the left-most solution path,

We parallelize DFS by sharing the work to be done among a number of processors. Each processor searches a disjoint part of the search space in a depth-first fashion. When a processor has finished searching its part of the search space, it tries to get an unsearched part of the search space from the other processors. When a goal node is found, all of them quit. If the search space is finite and has no solutions, then eventually all the processors would run out of work, and the (parallel) search will terminate. Since each processor searches the space in a depth-first manner, the (part of) state space to be searched is efficiently represented by a stack. The depth of the stack is the depth of the node being currently explored; and each level of the stack keeps track of untried alternatives. Each processor maintains its own local stack on which it executes DFS. When the local stack is empty, it takes some of the untried alternatives of another processor's stack. In our implementation, at the start of each iteration, all the search space is given to one processor, and other processors are given null spaces (i.e., null stacks). From then on, the search space is divided and distributed among various processors.

Code Output:





Comparison and Conclusion

Comparing BFS and DFS, the big advantage of DFS is that it has much lower memory requirements than BFS, because it’s not necessary to store all of the child pointers at each level. Depending on the data and what you are looking for, either DFS or BFS could be advantageous. BFS is a procedure of propagating frontiers. 'Propagate' means push all their unvisited adjacent vertices into a queue, and they can be processed **independently**. DFS, the vertices are visited one by one along a direction such as A->B->C. Visiting B must occur before visiting C. It's a sequential procedure which can not be parallelized easily. But the truth is both BFS and DFS are hard to parallelize because all the processing nodes have to know the global information. Accessing global variables always requires synchronizations and communications between nodes.