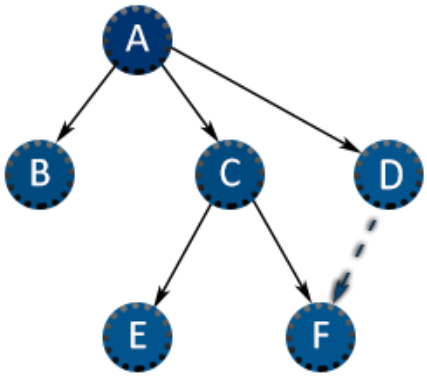
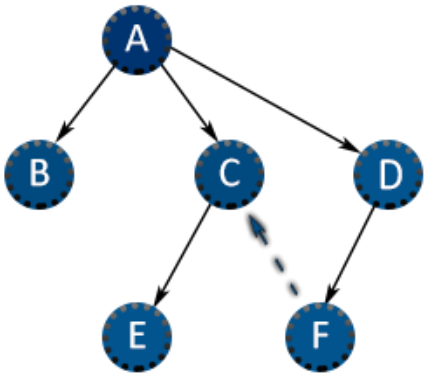
BFS



A B C D E F

DFS



A D F C E B

Concurrent Depth-first search

Made by:

Zdravko Hvarlingov

Content

Task description

Let’s have the graph G(V, E), where V is the set of vertices and E is the set of edges. The graph can be directed or undirected, connect or not. We are going to use adjacency matrix for the representation.

We must implement a program which performs a depth-first search. We have all the nodes as a starting step, or all the adjacency nodes of a given one. In this way we construct a spanning tree(if the graph is connected) or a spanning tree forest(if the graph is not connected and directed). The work of the algorithms should be divided into several tasks(two or more threads).

Serial algorithms solving the task

There are two main serial approaches which solve the task no matter the type of graph – recursive and iterative:

1. **Recursive**

```
4   #include <iostream>
5   #include <vector>
6
7   std::vector<std::vector<bool>> adjMatrix;
8   std::vector<bool> visited;
9   int n;
10
11  void DFS_visit(int node)
12  {
13      visited[node] = true;
14      //Do something with node
15      for (int i = 0; i < n; i++)
16      {
17          if (adjMatrix[node][i] && !visited[i])
18          {
19              DFS_visit(i);
20          }
21      }
22  }
23
24  void DFS()
25  {
26      for (int i = 0; i < n; i++)
27      {
28          visited.push_back(false);
29      }
30
31      for (int i = 0; i < n; i++)
32      {
33          if (!visited[i])
34          {
35              DFS_visit(i);
36          }
37      }
38  }
```

2. Iterative

```
4   #include <iostream>
5   #include <vector>
6   #include <stack>
7
8   std::vector<std::vector<bool>> adjMatrix;
9   std::vector<bool> visited;
10  int n;
11
12  void DFS_iterative()
13  {
14      std::stack<int> dfsStack;
15      for (int i = n - 1; i >= 0; --i)
16      {
17          visited.push_back(false);
18          dfsStack.push(i);
19      }
20
21      while (!dfsStack.empty())
22      {
23          int node = dfsStack.top(); dfsStack.pop();
24
25          if (!visited[node])
26          {
27              visited[node] = true;
28              //Do something with node
29              for (int i = n - 1; i >= 0; --i)
30              {
31                  if (adjMatrix[node][i])
32                  {
33                      dfsStack.push(i);
34                  }
35              }
36          }
37      }
38  }
```

Recursive algorithms and solutions are often not easily made into concurrent ones. Because of this we are going to use the iterative solutions as the first step of construction our concurrent DFS solution. The iterative solution is working the same way as the recursive one with the following characteristics:

- One node can be pushed into the stack more than once;
- We are checking if node is visited and set it to visited NOT before the stack push. In this way we simulate the work of the recursive approach(stack is LIFO data structure);

Implemented concurrent algorithms architecture

As we already said some lines ago, we are going to use the iterative approach as the first step of constructing our concurrent solution. There are two implementations of the algorithm. The first one gets all the nodes as the first step of traversing the graph(all nodes are pushed into the stack). The second one uses just one node(or its adjacency nodes) as the starting step. So, considering the above we have the following key points to discuss:

1. **Work distribution and balancing**

There is simply no way of achieving good static work distribution between the different threads considering that there are no constraints about the graph. It is highly possible that some of the threads will finish their work much earlier while others will continue executing till the very end.

That's why we are going to use dynamic balancing. There will be a shared resource – the set of nodes still not visited or the current state of the traversal. Considering the iterative algorithm, it is quite obvious that this resource is going to be the stack of vertices. In addition to it a Boolean array is going to be maintained to store if a node is already visited or not. Of course, that array is going to be a shared resource also.

2. **Synchronization techniques**

We are going to use three different ways of synchronization between the threads – mutex, condition variables and semaphores.

In order to synchronize all the operations (push and pop) from the stack of nodes, we will use a single mutex and semaphore, which will lock the data structure on push and pop operations and will keep the number of nodes inside the stack. In this way the threads will be blocked if there is no work to be done.

The Boolean array, storing information if a given vertex is already visited or not, will be also locked with mutex. The problem is that if we use just one single mutex for the whole array, there are going to be a lot of race conditions. It is clear that the probability of multiple threads accessing a single index is quite small. That's why we are going to use multiple locks(mutexes) in order maintain a valid state of the array. In the current implementation they are 2 * (number of threads). If we want to lock a given index **ind**, we are going to use the mutex at index **ind % (number of mutexes)**.

3. **Signalling threads when all the work is done**

In the first approach, having all the nodes as the first step, the signalling is accomplished using a shared variable storing the number of nodes already visited. Of course, that variable is going to be change with mutex, so we are going to be sure its value is the correct one.

When a certain thread increases the counter and its value is n(the number of nodes), it will notify the main, controlling thread, of the event. That will be achieved with condition variables. Once the main thread is woken up, it will wake up all the sleeping threads setting an end flag before that. After that the execution is over.

In the alternative approach, for the purpose of the explanation, let's consider that the starting step of the algorithm consists of a single node. The main, controlling thread iterates through all the nodes. If a given one is not visited yet, it pushes it into the stack and wakes up the threads, so they can start traversing to that component and construct the spanning tree. Unfortunately, we don't know the exact number of vertices before the end of the traversal of that component, so the signaling for end should be different. In that case if all the threads, except the main one, fall asleep that means that the component is already traversed. If this event occurs the main thread is woken up. If there are more nodes to be visited, another component traversal is started. Otherwise the execution is over. All that event handling is maintained within the semaphore. It stores the number of threads currently waiting on it. Once they meet a certain number (the number of threads except the main one), the main thread is notified.

4. Terminating the threads when all the work is done

In both approaches in most of the cases, at the end of the algorithm, all the threads, except the main one, are going to be blocked. In order to stop their execution, the controlling thread sets a Boolean flag for end and wakes them all using the value in the semaphore. The new value of the semaphore is the number of threads. In that way we are sure that all the threads are going to be woken up. It is important to tell that each thread checks for the Boolean flag once it wakes up. So, once it is set and the thread is unblocked – its execution is over.

5. Efficiency and speed-up

Both implementations achieve a significant speed-up when the graph has a lot of nodes(somewhere above 10 000) and edges. If there are less edges, close or less than the number of nodes, the spanning three concurrent DFS is less efficient. On the other hand, the alternative approach shows the same significant speed-up.

6. Technologies

Both approaches are implemented using C++, version >= 11. Implementation can be downloaded from the following link:
https://github.com/ZdravkoHvarlingov/Concurrent-DFS

Command line arguments

Using the algorithm, we can set the following parameters:

- **-t <number of threads>** - it sets the maximum number of threads used in the execution, the default value is 8;
- **-n <number of nodes>** - it sets the number of nodes in the randomly generated graph. All the edges are generated randomly and if their number is not set the default value is 10 * n(the number of nodes);
- **-e <number of edges>** - it sets the number of edges randomly generated;
- **-i <file name>** - it sets the name of an input file. It should contain the adjacency matrix of the graph with the first row containing a single number – the number of nodes. If that parameter is used, a random graph is not generated! The format of the file should be as shown below:

  === quote ===

  n
  0 1 1 0 1 1 0 … 0
  1 0 1 0 1 0 0 … 1
  …
  1 0 0 0 0 0 0 … 1

  === quote ===

- **-o <file name>** - it sets the name of the output file. If the algorithm generates the spanning tree, the format of the result is all the edges. Otherwise the result is all the nodes in the order of visit. If no output file is specified, the result is not saved;
- **-q** – the program executes in quite mode. No information about the execution time is given;

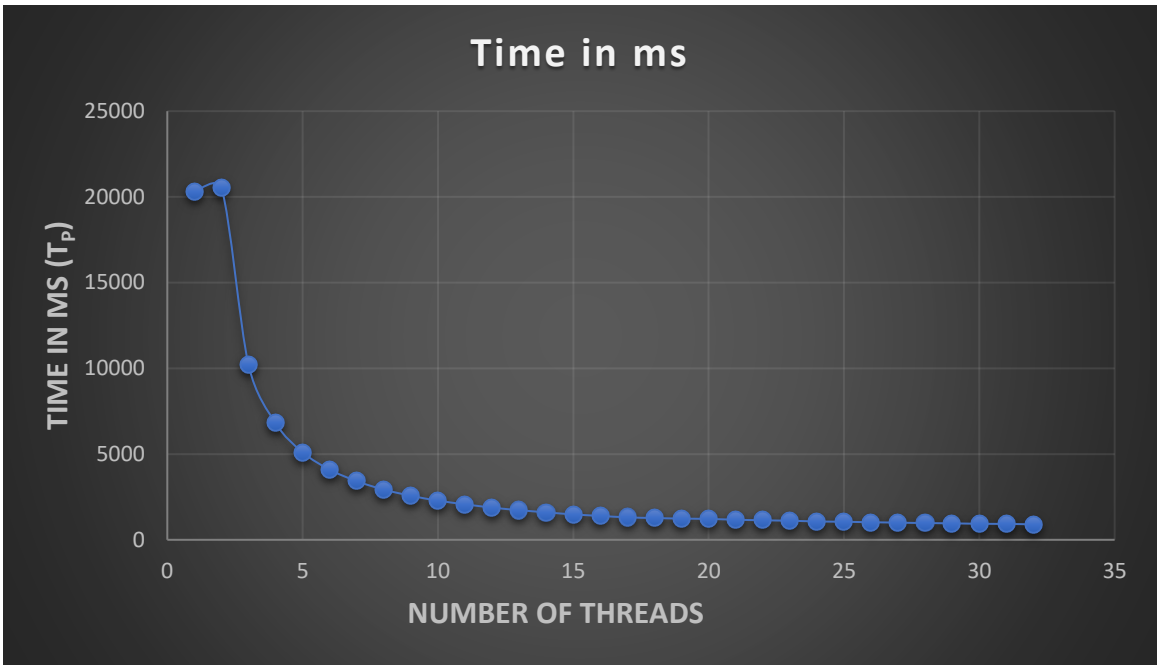
Tests and efficiency measurements

On the given machine the tests were with a graph with 20 000 nodes and 100 000 edges using the algorithm constructing a spanning tree forest and takes all the nodes as the first step of the traversal. The values with the alternative approach are close to these ones. The graph is randomly generated. You can see the results in the following table:

Number of cores	Time in ms	Speed-up	Efficiency
1	20315	1	1
2	20517	0.990154506	0.495077253
3	10204	1.990885927	0.663628642
4	6804	2.98574368	0.74643592
5	5105	3.979431929	0.795886386
6	4091	4.965778538	0.827629756
7	3418	5.943534231	0.849076319
8	2925	6.945299145	0.868162393
9	2569	7.907746205	0.878638467
10	2285	8.89059081	0.889059081
11	2070	9.814009662	0.892182697
12	1878	10.81735889	0.901446574
13	1728	11.75636574	0.904335826
14	1602	12.68102372	0.905787409
15	1481	13.71708305	0.914472203
16	1393	14.58363245	0.911477028
17	1309	15.51948052	0.912910619
18	1269	16.00866824	0.889370458
19	1237	16.42279709	0.864357742
20	1218	16.67898194	0.833949097
21	1171	17.34842015	0.826115245
22	1138	17.85149385	0.811431539
23	1112	18.26888489	0.794299343
24	1071	18.96825397	0.790343915
25	1051	19.32921028	0.773168411
26	1025	19.8195122	0.762288931
27	1001	20.29470529	0.751655752
28	976	20.81454918	0.743376756
29	960	21.16145833	0.72970546
30	934	21.75053533	0.725017844
31	926	21.93844492	0.707691772
32	901	22.54716981	0.704599057

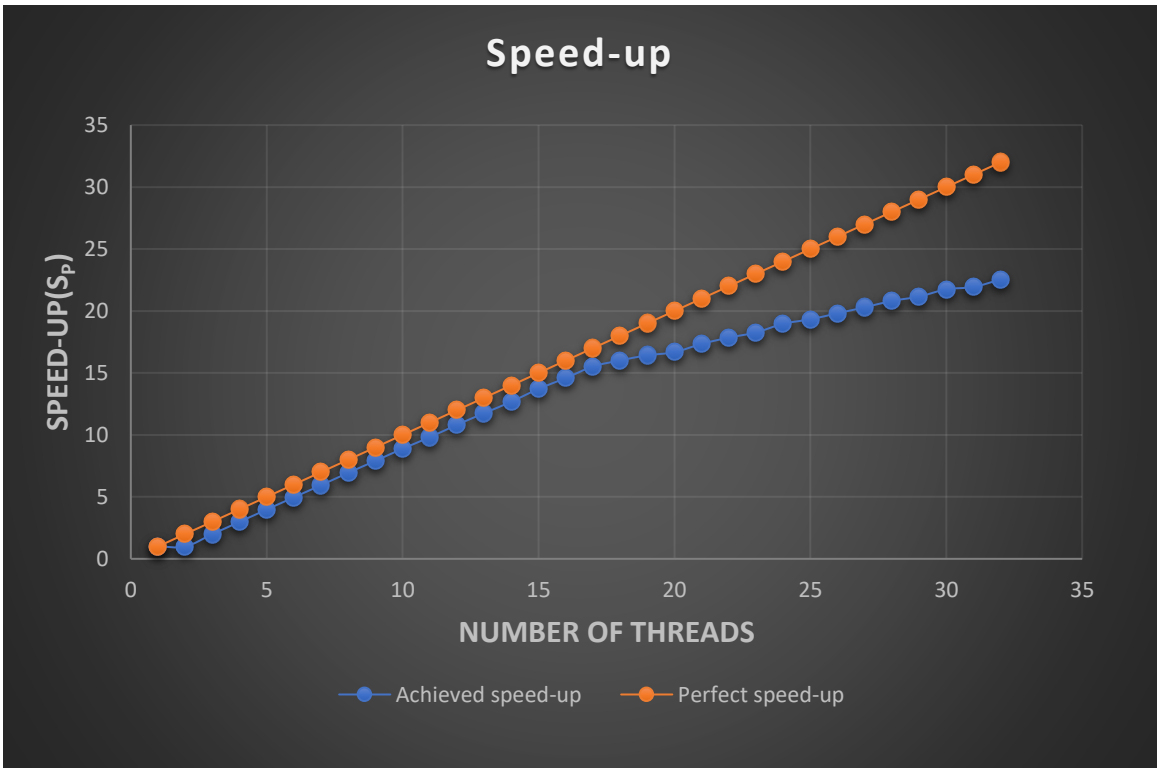
All the data has visual representation with x-y charts:

- The execution time needed for a fixed number of threads:



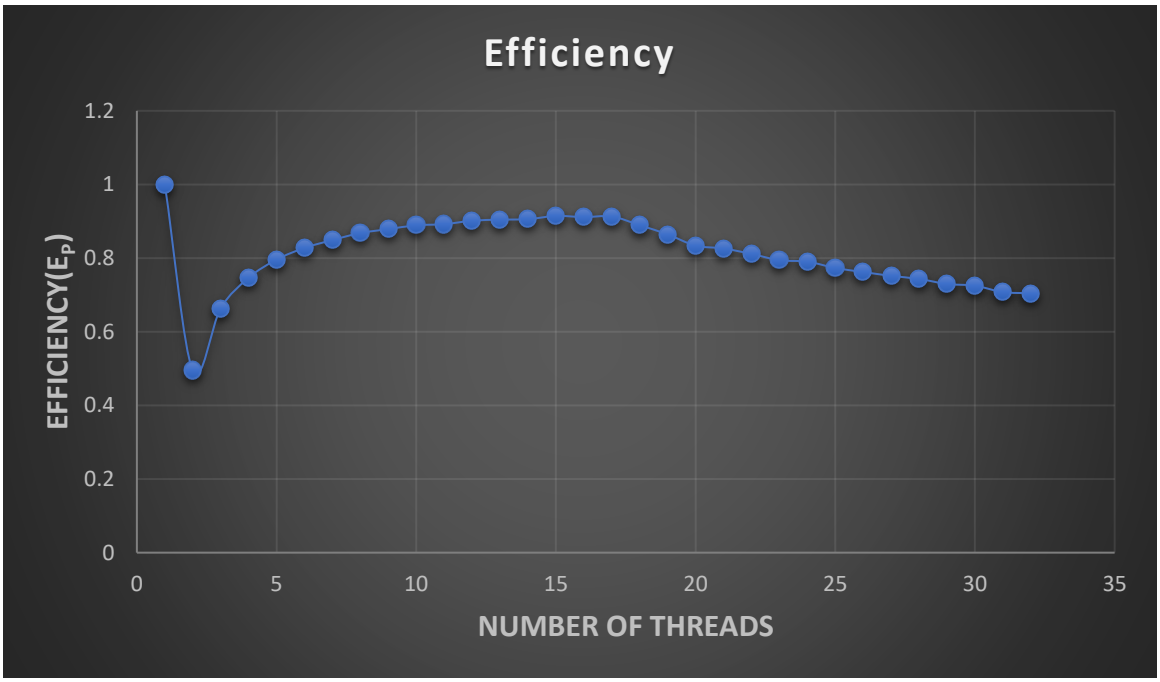
T_1 is the execution time using just one thread, T_p – the execution time using p threads.

- The achieved speed-up for a fixed number of threads:



The speed-up S_p is calculated using the following formula: $S_p = T_1 / T_p$.

- Achieved efficiency for a fixed number of threads:



The efficiency E_p is calculated using the following formula: $E_p = S_p / p$, where S_p is the speed-up with parallelization parameter(number of threads in our case) p.

Sources

- The Art of Concurrency: A Thread Monkey's Guide to Writing Parallel Applications, Clay Breashers
- https://en.wikipedia.org/wiki/Depth-first_search
- Introduction to Algorithms, 3rd Edition (The MIT Press), Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein
- https://www.geeksforgeeks.org/iterative-depth-first-traversal/