Parallelized Breadth-First-Search on Graphs: implementations and benchmarking

Lusiani Federico *

Parallel and Distributed Systems - Paradigms and Principles, Academic Year: 2020/2021

21 July 2021

1 Abstract

The project explores the improvements on execution times that can be achieved for a graph Breadth-First-Search application, by use of parallelization (specifically, multi-threading). Starting from a sequential implementation of the application, a theoretical study of the achievable speed-up is performed. This is subsequentially benchmarked against the measured performance of different parallelized implementations.

For the parallelized implementations, different approaches have been explored, both in the use of libraries (std thread[1] and fastflow[2]) and in the actual algorithms implemented. The code developed includes utility code for the generation of benchmarking graphs and performances measurement. All the code has been developed in the C++17 language (with some python code for the processing and plotting of the results) and is avaiable as a public repository on GitHub[3].

2 Introduction

2.1 The Breadth-First-Search algorithm

The Breadth-First-Search (BFS) algorithm is a policy for exploring nodes in a graph, starting from a root node (located at the depth level 0 of the graph). At each iteration, the BFS explores the nodes at the next depth level. In this project, only directed acyclic graphs are considered (DAGs), so the algorithm is assured to terminate in a finite number of steps (equal to the depth of the graph), at the end of which all the nodes have been explored.

*Università di Pisa, Master Degree in Computer Science, f.lusiani@studenti.unipi.it

[1]https://www.cplusplus.com/reference/thread/thread/

[2]http://calvados.di.unipi.it/dokuwiki/doku.php/ffnamespace:about

[3]https://github.com/FeLusiani/Parallel_BFS

Algorithm 1 Breadth-First-Search algorithm

```
1: Given input graph G and starting node root.
2: Initialize F = {root} and F_next = {}
3: Initialize explored[n] = False for each node n ∈ G
4: while not_empty(F) do
5:    for all n ∈ F do
6:       if ¬explored[n] then
7:          explored[n] = True
8:          explore(n)
9:          F_next = F_next ∪ n.children
10:      end if
11:   end for
12:   F = F_next
13:   F_next = {}
14: end while
```

Algorithm 1 shows the pseudo-code of the BFS. The project application uses the BFS algorithm to count the number of occurrences of a chosen node value x over the whole graph. For this reason, the **explore**(*node*) function consists in incrementing a counter if the value of *node* is x. Note that each node in the graph is assumed to have both a value and a (numeric) ID it can be indexed with.

3 Analysis of the parallelization problem

Since each iteration of the BFS algorithm requires the results of the previous, it is unfeasible to execute the iterations in parallel. Parellelization can be instead leveraged in the processing of each single BFS iteration Since no parellization occurs at the BFS iteration level, the total execution time T_{tot} is given by the sum of the execution times of each iteration:

$$T = \sum_{i=1}^{D} T_i$$

where D is the depth of the graph, starting from the root node. This is true both for the sequential and parallel implementation. Therefore the speed-up S as a function of the number of workers n can be written as:

$$S(n) = \frac{T_{seq}}{T_{par}(n)} = \frac{\sum_{i=1}^{D} T^i_{seq}}{\sum_{i=1}^{D} T^i_{par}(n)} = \frac{\sum_{i=1}^{D} T^i_{seq}}{\sum_{i=1}^{D} T^i_{seq}/S_i(n)}$$

Where $S_i(n) = T^i_{seq}/T^i_{par}$ is the speed-up on the iteration i. The result is that the speed-up $S(n)$ is equal to the harmonic mean of the speed-ups $S_i(n)$, weighted by the sequential times T^i_{seq}.

Therefore, if we can estimate a value $S^*(n)$ that approximates well the values $S_i(n)$, at least for the iterations with larger sequential times T^i_{seq}, we can write:

$$S(n) = \frac{\sum_{i=1}^{D} T^i_{seq}}{\sum_{i=1}^{D} T^i_{seq}/S_i(n)} \approx S^*(n)$$

For this reason, estimating the total speed-up $S(n)$ means estimating the speed-up $S^*(n)$ for a single generic iteration.

Looking at Algorithm 1 we can see that each iteration is made of two tasks: the for loop at line 5 (exploring the frontier) and the final updates of F and F_{next}. The for loop can be parallelized since each loop iteration is independent from the others, therefore:

$$T^*_{par}(n) = \frac{T^E_{par}}{n} + T^U_{par}$$

where T^E_{par} is the time required to explore the frontier F and the T^U_{par} is the time required for the final updates of F and F_{next}. In the sequential case, we have $T^*_{seq} = T^E_{seq} + T^U_{seq}$, we can write:

$$S^*(n) = \frac{T^E_{seq} + T^U_{seq}}{T^E_{par}/n + T^U_{par}}$$

Note that T^E and T^U have different values for the parallel and sequential implementation, because of the measures against race conditions that must be implemented in the parallel case (see Section 4.2).
Since in the sequential case the final update of F and F_{next} only requires clearing F and swapping F with F_{next}, we can approximate $T^U_{seq} \approx 0$:

$$S^*(n) = \frac{T^E_{seq}}{T^E_{par}/n + T^U_{par}}$$

T^E is the time needed to explore the frontier F, therefore we have $T^E = \text{size}(F) \cdot T^{E1}$, where T^{E1} is the time to explore a single node (which can be approximated to constant if the number of children of each node has low variance). As for T^U_{par}, this also grows linearly with the size of F_{next} (at least in the two implementations tested in this project), with a factor T^{U1}_{par}. Updating the equation gives:

$$S^*(n) = \frac{\text{size}(F) \cdot T^{E1}_{seq}}{\text{size}(F) \cdot T^{E1}_{par}/n + \text{size}(F_{next}) \cdot T^{U1}_{par}}$$

If on average the size of the frontier grows with a factor C (see below Section 4.1), we can consider

$$\text{size}(F_{next}) = \text{size}(F) \cdot C$$

so that

$$S^*(n) = \frac{\text{size}(F) \cdot T_{seq}^{E1}}{\text{size}(F) \cdot T_{par}^{E1}/n + \text{size}(F) \cdot T_{par}^{U1} \cdot C}$$

Simplifying:

$$S^*(n) = \frac{T_{seq}^{E1}}{T_{par}^{E1}/n + T_{par}^{U1} \cdot C} \tag{1}$$

As stated by the Amdhal Law, given f the sequential portion of a task, the maximum obtainable speedup is $\frac{1}{f}$. In our case:

$$\lim_{n \to \infty} S^*(n) \leq \frac{T_{seq}^{E1}}{T_{par}^{U1} \cdot C}$$

4 Implementations and hardware performances

The application has been tested on the following machine: Server Intel XEON-PHI, with 256 cores at 1.70 GHz.

4.1 Graph construction

When choosing a data-structure to represent a graph in memory, different implementations are available. In this project, the fact that every node can be indexed has been leveraged to store the whole graph as an array of `Node` objects, where the ID of each node corresponds to its index in the array.
To benchmark the application, a random graph generator has been implemented. The generator creates a random tree containing N nodes, starting with the node at index 0 as the root. After generating the whole tree, N extra connections are added randomly (but in such a way that cycles are not created). In the end, the resulting graph is a DAG (acyclic directed graph), where every node can be reached starting from the node 0.
The tests are run on graphs with $N = 10^7$ nodes, and on average, the frontier grows by a factor ≈ 1.7 at each BFS iteration, with a total graph depth of ≈ 8. Each node is initialized with a random value between 0 and 10.

4.2 Implementations

4.2.1 Sequential implementation

The sequential implementation uses `std::vector` for both `F` and `F_next`, since `std::set` gives much worse performances (slower insertion). By not enforcing the elements in `F_next` to be unique (set property), the same node can be inserted more than once in the frontier. However, because of the `explored[n]` being set, this does not make the algorithm inexact.

4.2.2 Parallel implementation

A Map-Reduce pattern can be applied to each BFS iteration: Map to process the nodes in `F`, and Reduce to generate `F_next`. To divide the Mapping work between the workers, a static round-robin scheduling is used, using a user-input *chunk_size*.

Since (as explained before) either the `F_next` set property or atomic access to each `explored[n]` must be guaranteed, two implementations have been developed:

1. The frontier `F` is represented by a boolean array. Node `n` is present in `F` if `F[n]==True`. The same holds for `F_next`. This ensures uniqueness on the elements contained (set property), and makes concurrent insertions (writes) admissible.

2. Each worker pushes the children nodes in its own `F_next`, which is then joined (concatenated) to the others. Each element `explored[n]` is a `std::atomic` variable.

Note that in Implementation 1, iterating through the nodes in the frontier is $O(N)$ (with N the number of nodes in the graph), compared to $O(size(F))$ in Implementation 2. However, Implementation 1 does not require to use protected variables or joining the `F_next` of each worker. Indeed, Implementation 1 showed much better performances, and for this reason it has been chosen as the leading parallel implementation.

4.2.3 Calculating the theoretical speed-up

We will now refer to Equation 1, and calculate the values for Implementation 1 (implemented using `std::thread`) to get a measure of the speed-up we can expect to achieve. The values have been measured taking the mean out of 3 similar-sized iterations (about $3 \cdot 10^5$ frontier size):

$$S^*(n) = \frac{[\text{size}(F) \cdot T_{seq}^{E1}]}{[\text{size}(F) \cdot T_{par}^{E1}]/n + [\text{size}(F) \cdot T_{par}^{U1}] \cdot C} = \frac{4308.47\mu s}{16035.34\mu s/n + 723.12\mu s \cdot 1.7} \quad (2)$$

For the C value, see end of Section 4.1.

5 Final application and experiments

5.1 Libraries and functions

Implementation 1 and 2 have been implemented both through the `std::thread` library and the `fastflow` library, for a total of 4 parallel BFS implementations:

1. `BFS_par_th`: Implementation 1 using `std::thread`

2. `BFS_par_th2`: Implementation 2 using `std::thread`

3. `BFS_par_ff`: Implementation 1 using `std::fastflow`

4. `BFS_par_ff2`: Implementation 2 using `std::fastflow`

These implementation have all the same interface: `BFS_par_XX(int x, const vector<Node> &nodes, int nw, int chunk)`, where x is the node value to search for. The sequential implementation `BFS_seq` takes the same arguments, except for the last two.

The `fastflow` implementations use a `ParallelForReduce` object to instantiate the workers once and launch them for each iteration. The `std::thread` implementations achieves the same functioning by launching the threads and having them synchronize on a barrier (using a custom implementation of a `Barrier` class).
When given a *chunk_size* of 0 or less, `std::thread` implementations will use the default value of 32 (which as been observed to work best on average).

5.2 User interface

The `execute.sh` bash script will run the `make` command and then launch the executable passing it the arguments it received. To launch the graph generator, the first argument must be `build_graph`, followed by the number of nodes, the maximum number of children per node (before extra connections are added), and the filepath for saving the generated graph:

```
./execute.sh build_graph N_NODES OUT_BOUND FILEPATH.
```

In the experiments, graphs generated with `N_NODES = 10E7` and `OUT_BOUND = 20` have been used. To test the BFS functions on a saved graph, the following command can be used:

```
./execute.sh FILEPATH NW CHUNK [UP_TO_NW STEP]
```

If the arguments `UP_TO_NW` and `STEP` are set, it will run multiple tests, with the number of workers going from `NW` to `UP_TO_NW` using `STEP`. The results of these tests can be saved in a `results_server.txt` file and then plotted using the Jupyter Notebook `results_plotter.ipynb`.

5.3 Experiments setup

The BFS implementations have been tested on a randomly generated graph of $N = 10^7$ nodes (See Section 4.1), on which they look for nodes of value 0 (about $\frac{1}{11}$ of the nodes). The input chunk size is set to 0, meaning it has a value of 32 for the `thread` implementations and to a computed value for the `fastflow` ones (these chunk size settings showed the best performances on average). For each test, 5 "wall-clock" time measurements are taken (in microseconds).

5.4 Experiment results

The results are shown in Figures 2 and 1. As we can see in Figure 2, Implementation 1 made with `std::thread` gave by far the best performance. Surprinsingly, when using `fastflow`, Implementation 2 actually performs better than Implementation 1 (but still much worse than `BFS_par_th`).
Looking at Figure 1, we note that the Implementation 1 ideal speed-up corresponds to the one measured only for `nw` in the $[1 - 10]$ range. After that, it is apparent that the overheads introduced by the increasing number of workers start to weight on the observed speed-up, that reaches its maximum of ≈ 3 for $nw \approx 60$.
It is probable that the overheads are introduced by the increasing number of memory accesses performed by different workers on memory locations that are close to each other, which can cause a decline in the efficiency of the caching system.

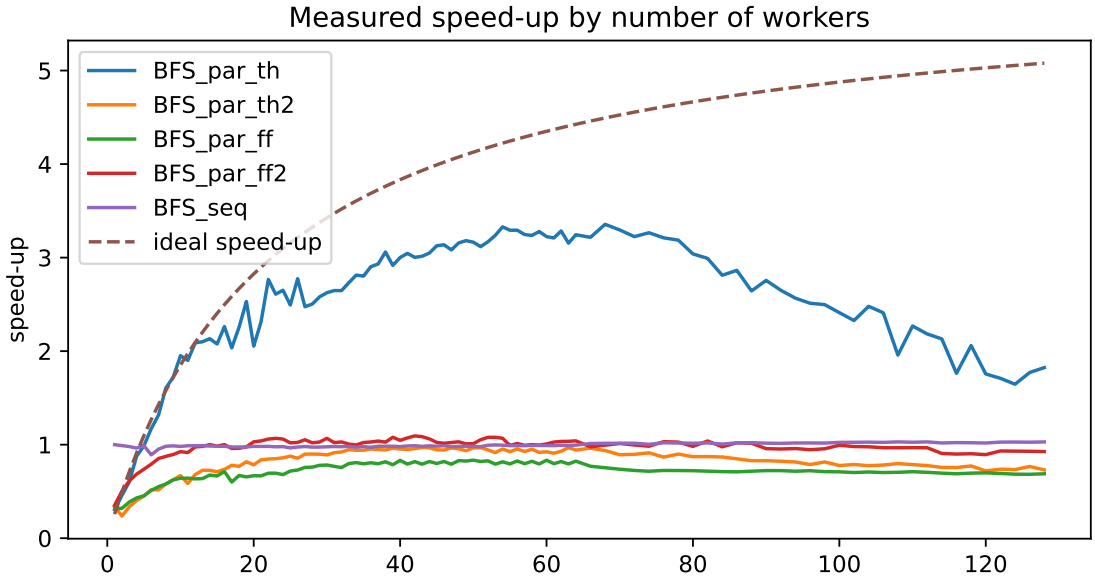

Figure 1: Mean speed-ups measured (5 runs per measure). Ideal (expected) speed-up is computed using Equation 2

6 Conclusions

Although the overall efficiency achieved by the best of the four BFS implementations is quite low overall, the speed-ups observed are in line with our starting expectations. The project has also shown how, in some cases, just applying a parallel pattern (such as Map Reduce) might not be sufficient to achieve a well performing implementation. Adapting

Figure 2: Mean and std execution times (μs) measured (5 runs per measure).

the target algorithm and the data-structures used can be crucial in order to actually gain a performance boost from the parallelization.