Parallel and Distributed Systems

2020/2021 Project

`GraphSearch`

Marco Costa
m.costa22@studenti.unipi.it
545144

Contents

Abstract

The aim of the project is to develope and analyze theoretically a parallel version of a *breadth-first search* over *directed acyclic graphs* while counting the number of occurrences of a specific integer value associated to each node. Two versions must be realized, one in plain `C++` and one using the `FastFlow`[1] framework and both must be compared against the theoretically boundaries.

1 Analysis

The completion time of the application can be simply defined as the time needed to create the graph object (e.g. by random generation or by loading from file) and the time needed to compute the graph search. Formally we define it as follows:

$$CT = t_{graph\_create} + t_{graph\_search}$$

The first term represents the *serial fraction* of our application, so we must concentrate over the second one and see how we can effectively achieve a *speedup* over this computation.

Typically, a classical (i.e. sequential) BFS search is implemented using a queue (which at the beginning contains only the starting node) and a loop cycle; at each

[1] github.com/fastflow/fastflow

iteration the first element of the queue is *popped out*, *pushing* in the queue each one of its adjacency which hasn't been visited before. This repeats until the queue is empty. In order to translate this algorithm into a parallel context we adopt a slightly different approach: instead of using a single queue, we push the adjacencies of all the nodes belonging to the same queue inside a different one, which we denote as *next frontier*. When all the nodes in a frontier have been visited, we can move to the next one; the procedure completes when $f_{k+1} = \varnothing$ at a certain iteration k. Formally we can say that *each frontier f_i contains all and only the nodes visited at the level i of the BFS.*

This modification leads to obtain at each iteration a **finite size frontier** which can be efficiently split using a *Map* parallel pattern between nw workers. Since no communication nor ordering is needed between the workers this can be defined as an *embarrassingly parallel computation.*

Now, we are able to compute the theoretical speedup achievable within this procedure. For each iteration of the BFS we pay[2]:

$$\sum_{i \in f_i} t_i$$

where f_i is the frontier at the i-th iteration and t_i the cost of computing the adjacency list of the i-th node of the graph (the other operations to be performed, like checking if the node's associated value is an occurrence or not, are constant time operations). Defining $\bar{k}$ as the number of iterations needed to solve the problem[3], we obtain:

$$t_{graph\_search} = \sum_{i=0}^{\bar{k}} \sum_{j \in f_i} t_j \tag{1}$$

It finally follows that

$$sp(nw) = \frac{t_{graph\_search}}{\bar{k} \cdot nw \cdot (t_{split} + t_{merge}) + \frac{t_{graph\_search}}{nw}} \tag{2}$$

where $sp(nw) < nw$.

However, these results leave us quite unhappy, since (1) and (2) are strongly dependent from the factor $\bar{k}$, which can highly decrease the achievable *speedup*. For this reason we will now propose an analysis over the factor $\bar{k}$.

If we generate randomly the adjacency matrix of a graph with $P(i \to j \mid i < j) = p$, and supposing that, for simplicity, the node 0 is the starting node (i.e. $f_0 = \{0\}$), we obtain that at the second iteration

$$|f_1| \approx n \cdot p$$

[2]This analysis holds also for the classical approach described before, since the single queue at one instant of time can be seen as the union of two frontiers.

[3]More precisely $\bar{k} + 1$, since f_0 is the initial frontier containing only the starting node.

we can now compute the *minimum size of the frontier* for the next iteration as *the number of adjacencies expected for a single node extracted from the frontier* f_1:

$$\begin{aligned} |f_2| &> p \cdot (n - np) \\ &= np \cdot (1 - p) \end{aligned}$$

since an already visited node cannot be reinserted inside a new frontier. Likewise we have:

$$\begin{aligned} |f_3| &> p \cdot (n - np - np \cdot (1 - p)) \\ &= np \cdot (1 - p - p + p^2) \\ &= np \cdot (1 - 2p + p^2) \\ &= np \cdot (1 - p)^2 \end{aligned}$$

It would be now clear that we can generalize the size of a *generic* frontier in this way[4]

$$|f_{i+1}| > np \cdot (1 - p)^i$$

Always recalling the inequality $|f_i| \leq n - i \ \forall i$, which leads to the trivial upper bound $\bar{k} = O(n)$.

We can use this formula to give a *better* **upper bound** over the number of iteration k needed to solve the problem, introducing the variable n_k as the *number of nodes visited after* k *iterations*

$$\begin{aligned} n_k &= 1 + \sum_{i=1}^{k} |f_i| \\ &> 1 + \sum_{i=0}^{k-1} np \cdot (1 - p)^i \end{aligned} \tag{3}$$

So, it should be obvious that posing $n_k = n$ and solving for k would produce the requested upper bound.

Unfortunately, since I'm currently lacking on time (and probably skills) we will do something a bit easier by posing $k = \log_2 n$ and $p = 1/2$. By substituting in (3) we obtain[5]:

$$\begin{aligned} n_{\log_2 n} &> 1 + \frac{n}{2} \cdot \sum_{i=0}^{\log_2 n - 1} \left(\frac{1}{2}\right)^i \\ &= 1 + \frac{n}{2} \cdot \left(2 - \frac{2}{n}\right) \\ &> n \end{aligned}$$

[4]Note that the formula is decreasing

[5]Try it to believe! Recalling that $\sum_{k=m}^{n} x^k = \frac{x^{n+1} - x^m}{x-1}$ with $x \neq 1$

for $n \to \infty$.

So we can conclude that under these, reasonable, assumptions we have termination of the procedure in $\bar{k} = O(\log n)$ iterations, which puts under a different light the analysis computed in (1) and (2).

2 Implementation

The final implementation follows the ideas proposed in the previous section, with some algorithmic choices made to optimize the executables. Probably the most relevant algorithmic choice made concerns the *merging phase* of the worker threads. In particular, each worker j works on a subset of the current frontier f_i, producing a new *partial* frontier f_{i+1}^j which is finally merged into the next frontier f_{i+1} (since there could be repeated elements) as

$$f_{i+1} = \bigcup_{j=1}^{nw} f_{i+1}^j$$

by the master thread. The most efficient way (in algorithmic terms) would be to insert all the element of the nw partial frontiers into an `std::unordered_set` and then, once finished, the `std::vector` representing the new frontier can be reconstructed from the set elements. Each insertion has an algorithmic cost of $O(1)$.

However, this approach does not take into account the effective structure of a directed acyclic graph.

Theorem 2.1. *A graph is a directed acyclic graph $\iff$ its adjacency matrix is upper triangular*[6]*.*

From the previous theorem is easy to prove the following corollary

Corollary 2.1.1. *If a DAG has uniformly random edges then*

$$\mathbb{E}[adj(n)] < \mathbb{E}[adj(n+1)]$$

where $adj(n)$ is the number of adjacencies of the n-th node.

Thus, we can exploit the fact that if the frontier is *sorted* then we can expect an *higher load at the beginning of the frontier rather than at its end.* It seems obvious that keeping the frontier sorted *while* using a static partitioning strategy will perform worse than applying a static partitioning strategy over an unsorted frontier, which can exploit randomness on its side.

A most effective approach about the sorted frontier is achieved by splitting the frontier in chunks and applying a round robin approach between the workers. In the same way, it would have no sense to apply this approach to the unsorted frontier. In order to do so we must either use an `std::set` or, better, sort the new frontier produced by the

[6]More or less, we are deliberately ignoring the topological sorting of the graph since the graph generated is already topologically sorted.

`std::unordered_set`[7]. This idea adds a $O(\log |f_i|)$ factor at each iteration; however, from the obtained measurements (Figure 1, but will be described in detail later) this idea leads to a gain in terms of achieved speedup, despite of the logarithmic factor added.

As far as it concerns the graph generation, in order to maintain the property of the Corollary 2.1.1 a randomly distributed upper triangular matrix of n nodes is generated. In the same random way, the value associated to the node is generated between 0 and the *max_value* provided as a parameter.

For what concerns the main algorithmic routine, the `C++` version implements a loop cycle working as follows:

- Compute the following frontier
- Start the workers and pause the master thread
- Each worker waits on the barrier once completed its task
- When all the workers reached the barrier resume the master thread

until a computed frontier is empty. Finally, a local reduction is performed to sum the number of occurrences found by each worker.

The `FastFlow` version is just a rewriting of the `C++` one, using a `ff::ParallelFor` which automatically acts as a barrier and manages itself the round robin scheduling.

As chunk size value, $ch = 2$ has be chosen for both the versions, since it seems to give the most benefits regarding the tradeoff between locality and cache misses.

3 Test

> All the tests were executed on a XEON PHI KNL with 64 cores and 4 hardware threads per core. Each measurement has been taken after 10 warmup iterations and averaged between as many iterations. All the graphs are generated randomly with $p = 0.35$ and starting from the node 0.
> **Important:** the executions with graph with less than 2000 nodes do not produce any benefit in terms of speedup.

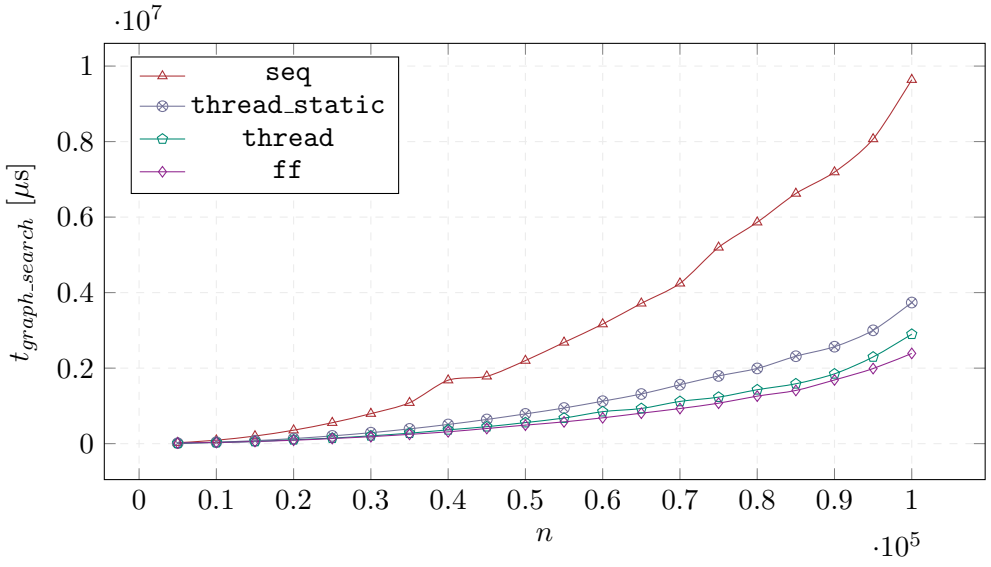
The first test shows the t_{graph_search} time over the same graph for the four different versions implemented (sequential, `C++` threads with static partitioning, `C++` threads with chunks and `FastFlow` version) when increasing the number of nodes up to 100k, executed with $nw = 4$.

The version with unsorted frontier and static partitioning (`thread_static`) *is surpassed* in every measurement made against the version with chunks, and confirms that, even adding a logarithmic factor, practically we have a benefit. From now on, we will discard the static partitioning version.

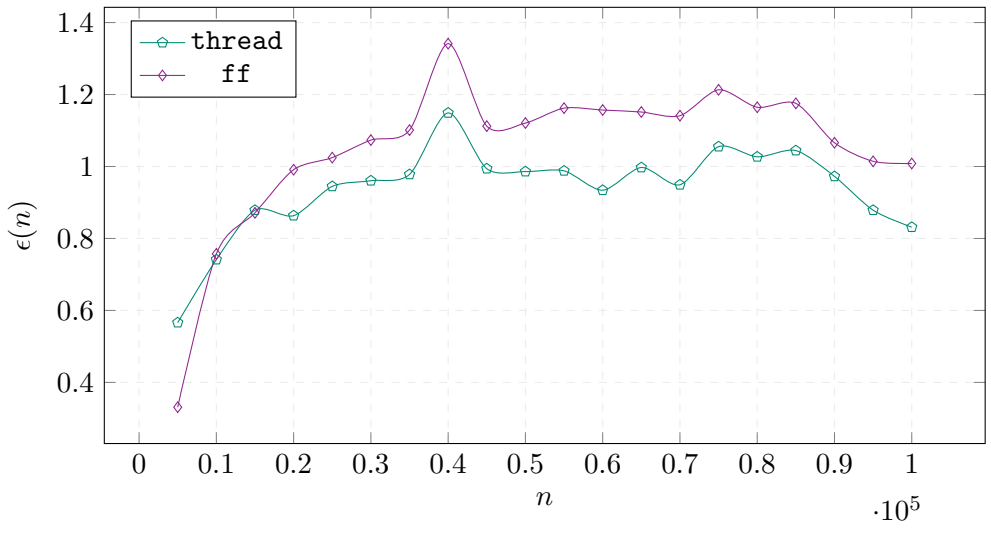
[7] This because using an `std::set` we pay $O(\log |f_{i+1}|)$ for each element in all the partial frontiers, which may include repeated ones

Looking forward with the analysis, we can see that from a number of nodes $n \geq 20000$ the computation maintains a good level of efficiency, stabilizing near 1 when $n \in [25000, 90000]$, even surpassing it in some points (which may indicate noise in the measurements and/or that $\frac{T_{seq}}{n} \neq T_{id}(n)$[8]).

It is worth to note that, even if the `FastFlow` version is faster, both the efficiency curves show the same trend.



(a) Execution time



(b) Efficiency

Figure 1: Execution time and efficiency as the number of nodes varies fixing p

[8] In other terms that T_{seq} is not the optimal one.

3.1 50k nodes

Next, in Figure 2 the results of the analysis fixing the number of nodes at $n = 50000$. As we can see, the *FastFlow* version still overtakes the plain `C++` when increasing the number of workers, even almost doubling the gained speedup with $nw = 32$, which results in a maximum point for the function itself, decreasing thereafter; conversely, the plain `C++` version maintains a more linear increasing of the speedup.

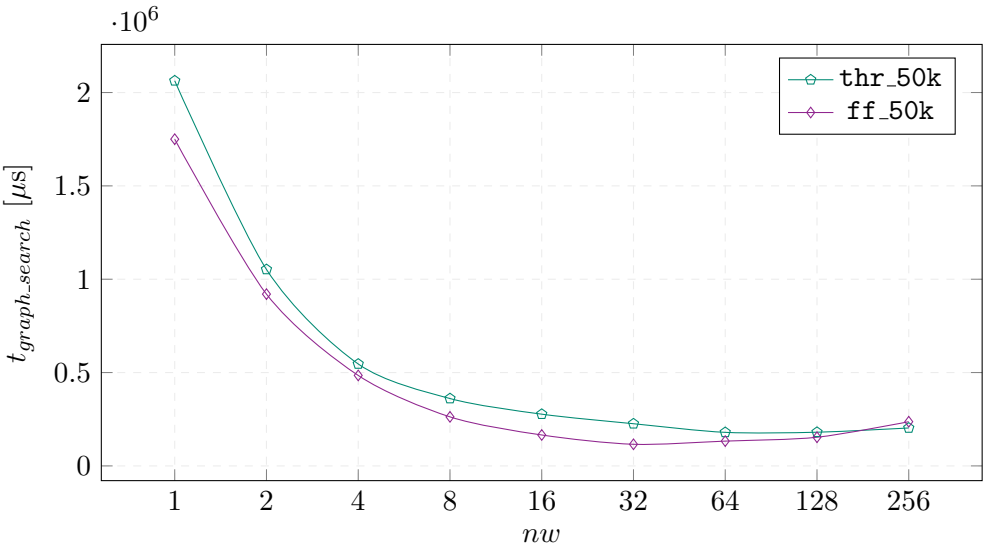
Since the two versions are nearly the same in what concerns the algorithmic routine and the data structures used, the differences between the two curves are probably related to the *finer low-level* optimization and better synchronization mechanisms exploited by the *FastFlow* framework. However, with $nw \leq 8$ they both reach an *almost-optimal* speedup value.

3.2 100k nodes

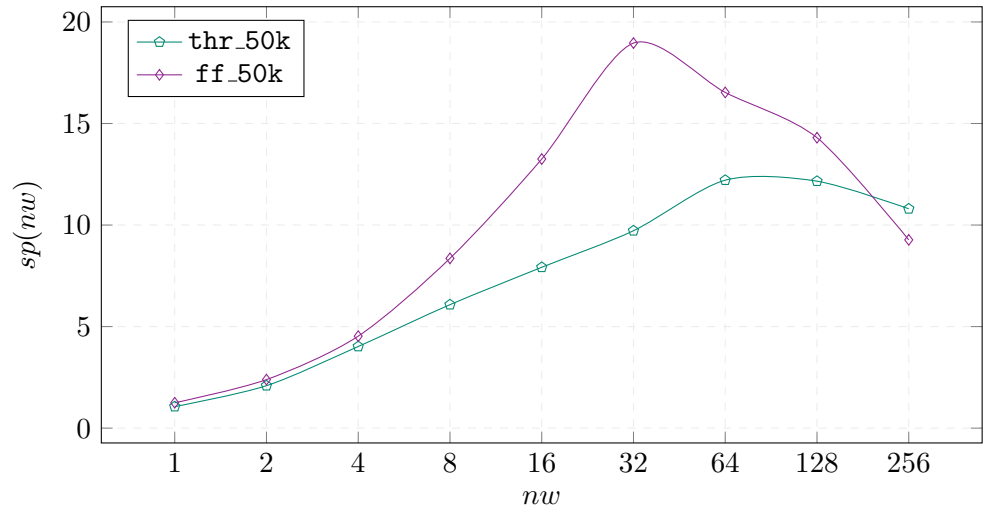
In the last test, we repeat the one performed above, increasing the number of nodes to $n = 100000$. The results can be examined in Figure 3 and show an excellent speedup, which compared to the one obtained in the previous test, reaches an *almost-optimal* value for $n \leq 16$. Furthermore, both the speedup functions keep the same trend as before: the `FastFlow` version increases in a "more-exponential" like trend, before reaching its maximum point, while the plain `C++` version tends to increase more linearly and, as before even if in a slighter way, this version surpasses the `FastFlow` one for $nw = 256$ (the maximum availability of the machine).

4 Conclusion

We have provided an analysis which examines in detail the achievable speedup giving an **upper bound** over the number of iterations (i.e. the number of frontiers) the algorithm can make. Following, the principal design choices have been described trying to give an algorithmic explanation on why they have been preferred to other possible options. Finally, the results obtained from both versions have been compared each other and against the theoretically results, showing an interesting gain in terms of speedup and efficiency with respect to the sequential computation.

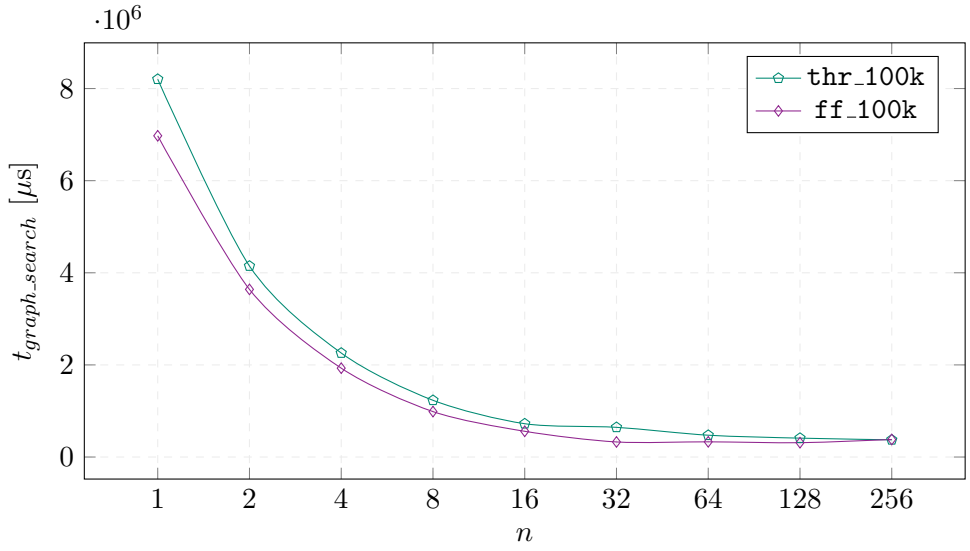


(a) Execution time

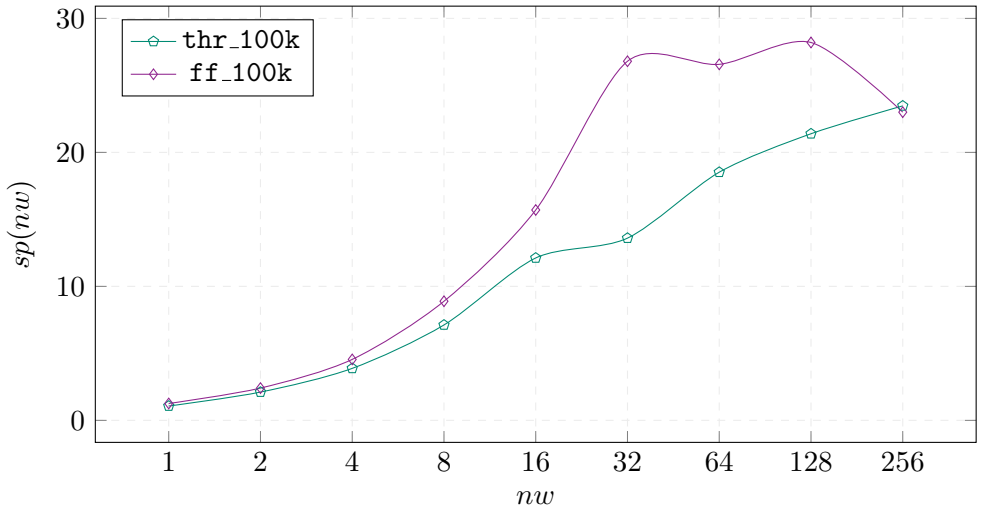


(b) Speedup

Figure 2: Analysis of a BFS on a graph with 50k nodes, increasing the number of workers



(a) Execution time



(b) Speedup

Figure 3: Analysis of a BFS on a graph with 100k nodes, increasing the number of workers

Appendix: Compilation and Execution

The sources produce three different executables:

- `bfs_seq`: the sequential version
- `bfs_thr`: the `C++` thread version
- `bfs_ff`: the `FastFlow` version

each executable can be obtained either by manually compiling the *"executable_name.cpp"* file or by executing **make executable_name**. With the command **make**, instead, all the three executables will be compiled.

The usage of the executables is the following:

```
$ executable n_nodes n_workers --start [start_node] --search [search_value]
--max [max_value] --percent [percent] --seed [seed]
```

where `bfs_seq` **does not** accept the number of workers parameter.
The default behavior of the application without the optional parameters is:

- `seed`: the seed for the graph generation. *Default*: 1234
- `start_node`: the starting node. *Default*: 0
- `search_value`: the value to search. *Default*: 5
- `max_value`: the $[1, max]$ value to assign to a node. *Default*: 6
- `percent`: the value of p in percentage. *Default*: 35 (%)

Note: the `FF_PATH` line in the *Makefile* must be updated pointing to the current path of the `FastFlow` library.