Graph Search
(Application Project)

Project for the Parallel and Distributed Systems: paradigms and model course.

Raffaele Villani (608837)

September 2021

University of Pisa
Supervisors: Prof Marco Danelutto & Massimo Torquati

Contents

1 Introduction **1**

1.1 Graph 1

2 Algorithms Implementation **2**

2.1 Sequential implementation 2

2.2 CPP STL parallel implementation 3

2.3 Fast Flow parallel implementation 4

3 Comparison and Results **4**

3.1 10k nodes 5

3.2 50k nodes 6

3.3 100k nodes 8

4 Conclusions **9**

5 How to use **10**

1 Introduction

In this section, I am going to give you an overview of the task that has to be solved in order to have an idea of what we will talk about in the next chapters.
The task consist in visit a *Directed Acyclic Graph (DAG)* that is described as a set of nodes N and a set of directed arcs. The application should take a node value X, a starting node S, and must return the number of occurrences in the graph of the input value X found in the graph during a *Breadth-first search (BFS)* visit starting from the node S.

1.1 Graph

To execute the experiments, a Simple *Graph class* has been implemented. The Graph class contains different methods, the most relevant are:

- **AddNode**: That's allows to add a node into the graph.
- **Addedge**: That's allows to create an edge between 2 nodes.
- **generateRandomDAG**: That's allows to generate a random *Direct Acyclic Graph*
- **createGraphFomFile**: That's allows to read a file and generate the graph from it.

A script for the graph generation has been implemented, that script allows to generate a random DAG and write it on a file. in the section 5 is explained how to run the script.
The graph generation is a really simple one, there is defined a *percentage* to manage the density of the graph. The algorithm 1 show the pseudo-code.

Algorithm 1 Graph Generation

```
1: Inputs: Graph g, int N
2: Output: Generated Graph
3:
4: for i=0; i <N; i++ do
5:     for j=i+1; j <N; j++ do
6:         if (rand() % 100) <PERCENT then
7:             g.addEdge(i,j)
```

2 Algorithms Implementation

2.1 Sequential implementation

The way the BFS algorithm works is by exploring the vertexes of the graph level by level. So the algorithm first perform a visit of all the nodes in the frontier at *level k* and then when all the nodes at this level are visited it could start to visit all the nodes in the frontier at *level k+1*. The conventional version of this algorithm uses two data structures to store the frontier and the next frontier. The first one contains all the vertex at the current level, so each vertex in the frontier has the same distance from the source vertex, instead, the next frontier is filled up during the visit with all the children's nodes of the nodes in the first frontier. At each iteration the next frontier and the current one are switched. In order to adapt this algorithm to the task assigned we have only to add a condition where we increment a counter when the value of a node is equal to the value specified in input. The pseudo-code is presented in Algorithm 2.

Algorithm 2 Sequential BFS

```
1: Inputs: Graph g, Source Node s, Value v
2: Output: Number of occurrences of Node value v in the Graph
3:
4: counter = 0;
5: fs = {}
6: ns = {}
7: visited = [ ]
8: fs.push(s, fs)
9: visited[s] = true;
10:
11: while fs !empty do
12:     for u in fs do
13:         nodeID = pop(fs)
14:         if nodeID.value == v then
15:             counter++
16:         for each neighbor v of u do
17:             if visited[v] == false then
18:                 push(v, ns)
19:                 visited[v] = true
20: fs = ns
21: ns = {}
22: return counter
```

2.2 CPP STL parallel implementation

To parallelize the Algorithm 2 we have to take into account some particular details. The algorithm as we said works visiting the graph level by level, so if we split the work among different thread we have to ensure that before pass at the next level all the threads have to finish the visit of the current one, in order to do this we have to take care of some kind of synchronization like a *barrier* among the threads. Another problem that occurs in the parallelization is related to the update of visited nodes in line 17 and 19, in fact there could happen a race between the thread. The reason of the race is that a child of a node in the frontier can be a child of another node too. In order to avoid this problem, has been decided to

let all the thread access at the *visited nodes* data structure in reading mode but without update the data structure, and so in the next frontier could happen that a node can appear more than one time. Only at the end of the level iteration, another thread update the frontier eliminating the duplicates merges all the partial frontiers and updates the *visited nodes* data structure in sequential mode.
In the same way, only one thread split the work among all the workers in a sequential way, the split phase is a *static partitioning*, in fact, once the thread knows the size of the frontier he split in equal size this frontier among the workers.

2.3 Fast Flow parallel implementation

The logic behind the application implemented with the *Fast Flow Library* is the same of the *CPP STL*. I have opted to use a Farm pattern, and as before there is an *Emitter* node that computes a *Static partitioning* of the frontier among the workers. Then there is the *Worker* node that works exactly in the same way as CPP. As last one the *Collector* node is being implemented, and this one merges the local frontiers computed by the workers, computes the global sum of the counters and updates as before the *visited nodes* data structure.

3 Comparison and Results

In the following section I am going to compare the different algorithms using different type of graphs. The comparison will be between the different implementation and the expected performances. In order to get good results, i executed at least 5 times the applications and then the average of the results was computed. The metrics used for this analysis are:

- **Completion Time**: the time in $\mu secs$ to execute the application.
- **Speedup**: The ratio between the *state-of-art* sequential algorithm and the parallel implemented algorithm. $sp(nw) = \frac{T_{seq}}{T_{par(nw)}}$

- **Scalability**: The ratio between the parallel execution with a parallelism degree of 1 and the parallel execution with a parallelism degree of nw. $scalab(nw) = \frac{T_{par(1)}}{T_{par(nw)}}$
- **Efficiency**: The ratio between the speedup(nw) and nw $\epsilon(nw) = \frac{T_{seq}}{nw * T_{par(nw)}}$

The experiments are executed on graphs of different size, in particular there were tested graph with 10k, 50k and 100k nodes, as we will see in the experiments the size of graph affects the max speedup achieved by the application. All the experiments are performed on the Xeon PHI machine with starting node 0 and value to search 5.

3.1 10k nodes

This is the first experiment that I have performed. The Table 1 shows how the parallelization of the algorithm goes well until is reached a parallel degree of 8, then the overhead grows too much and so the parallel algorithm becomes more expensive than the sequential one. As we can see we have a similar behaviour in both versions the only difference is that in the *FastFlow* version we have smaller time from a parallel degree of 8 and so on. in Figure 1 is represented the speedup, the scalability and the efficiency.

Graph 10k nodes		
Par Degree	**CPP STL** ($\mu secs$)	**FastFlow** ($\mu secs$)
Seq	67537	67537
1	84991	92086
2	52159	52019
4	31045	34490
8	29126	25988
16	42813	28868
32	67612	37467
64	111223	64198
128	163744	110624
256	266742	218753

Table 1: Completion time on 10k nodes

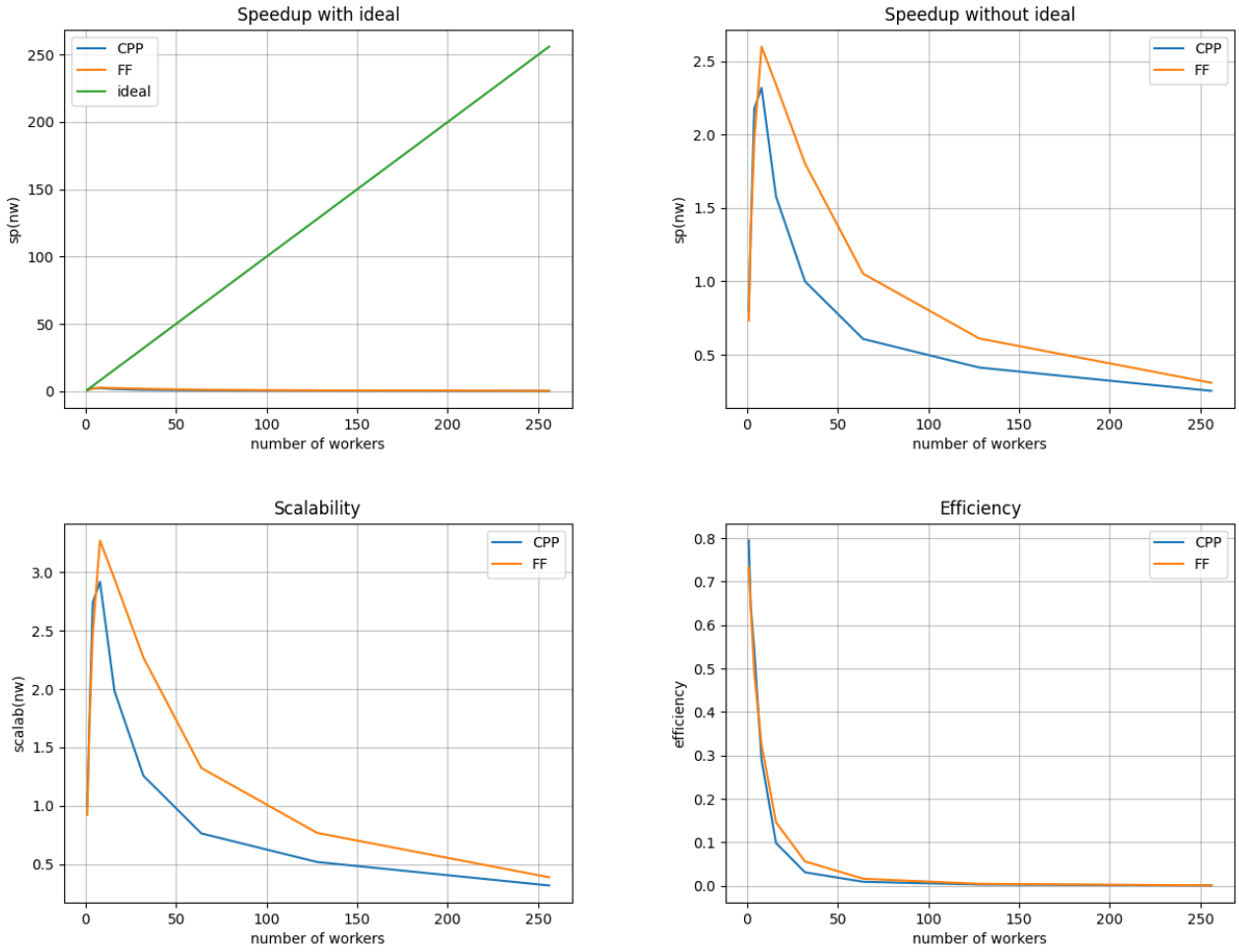

Figure 1: Mentrics on 10k Graph

3.2 50k nodes

From the last experiment emerged how the overhead affects too much after a certain parallel degree, so I experimented with a larger graph, in this case the workers have more work to do and could reduce the overhead effect. As we can see from the Table 2 this is true in fact the *CPP STL* algorithm has a good behaviour until reaches a parallel degree of 64, instead the *FastFlow* version seems to have the best results with a parallel degree of 32, then as before the overhead grows. As we can see in Figure 2 with this graph a better speedup, scalability and efficiency are achieved.

Graph 50k nodes		
Par Degree	**CPP STL** ($\mu secs$)	**FastFlow** ($\mu secs$)
Seq	3976504	3976504
1	4332620	4499632
2	2578070	2741522
4	1129120	1377130
8	593367	618890
16	356249	417254
32	295002	329696
64	281728	447396
128	342824	498421
256	460928	483520

Table 2: Completion time on 50k nodes

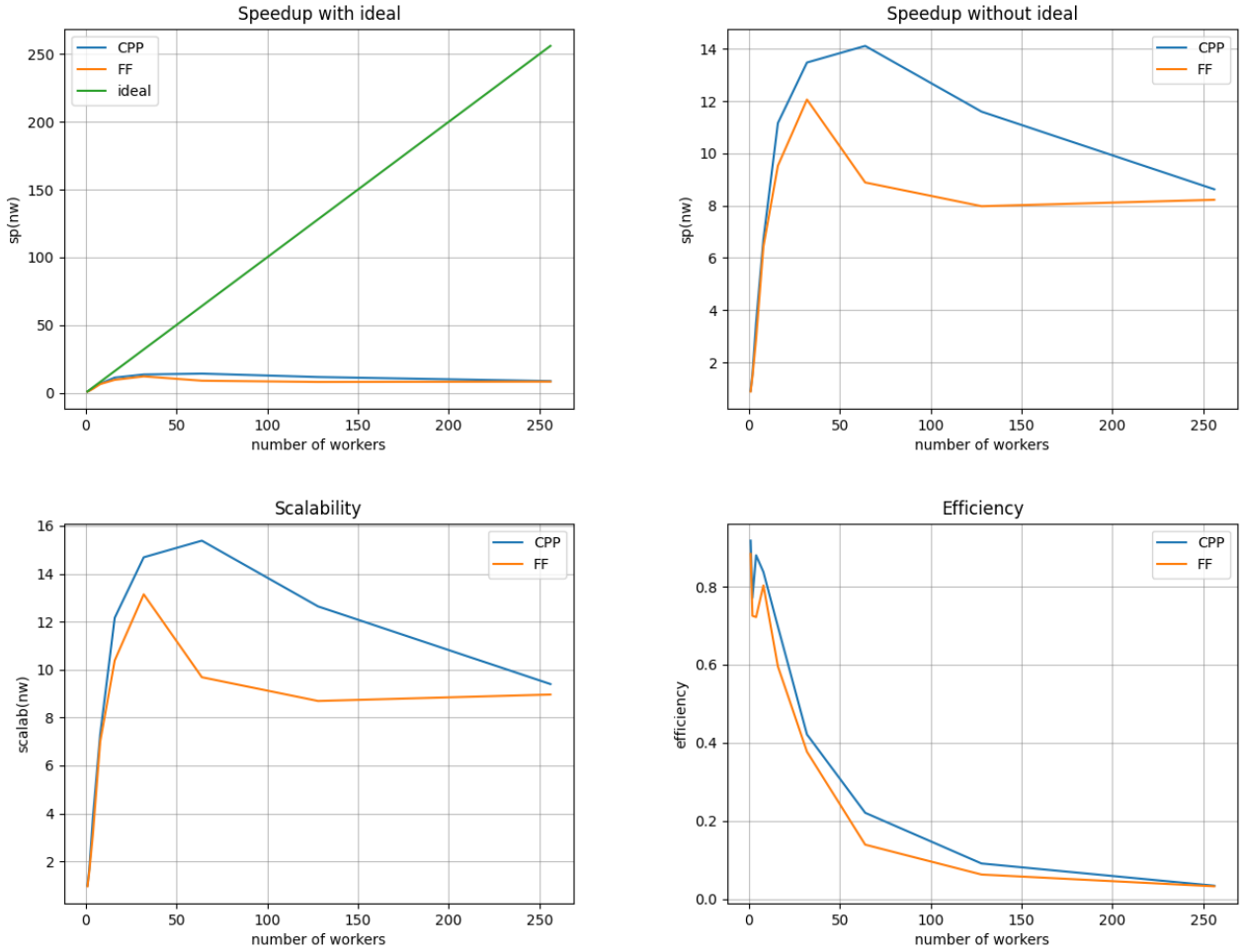

Figure 2: Metrics on 50k Graph

3.3 100k nodes

As last experiment I have used a graph with 100k nodes to see if it was possible to achieve a better speedup and a better behavior of the algorithm with a large parallel degree. As the Table 3 shows the best completion time from the *CPP STL* version is always with a parallel degree of 64 instead for the *FastFlow* one the best one is achieved with a parallel degree of 128 but with the same completion time of the *CPP STL* version, in any case doubling the size of the graph have not mitigated the overhead of creating threads when we overtake a certain parallel degree, a better explanation of that is shown in section 4. In Figure 3 are presented as before the metrics, and here as we can see it happen a strange thing, in fact until a parallel degree of 16 in the *CPP STL version* we achieve a *super linear speedup* and so we achieve an efficiency >1.

Graph 100k nodes		
Par Degree	**CPP STL** ($\mu secs$)	**FastFlow** ($\mu secs$)
Seq	11387200	11387200
1	11414800	13663900
2	6398610	6789442
4	2776840	3316594
8	1305330	1669896
16	658028	759623
32	436111	475954
64	405796	486981
128	483022	397810
256	649708	493711

Table 3: Completion time on 100k nodes

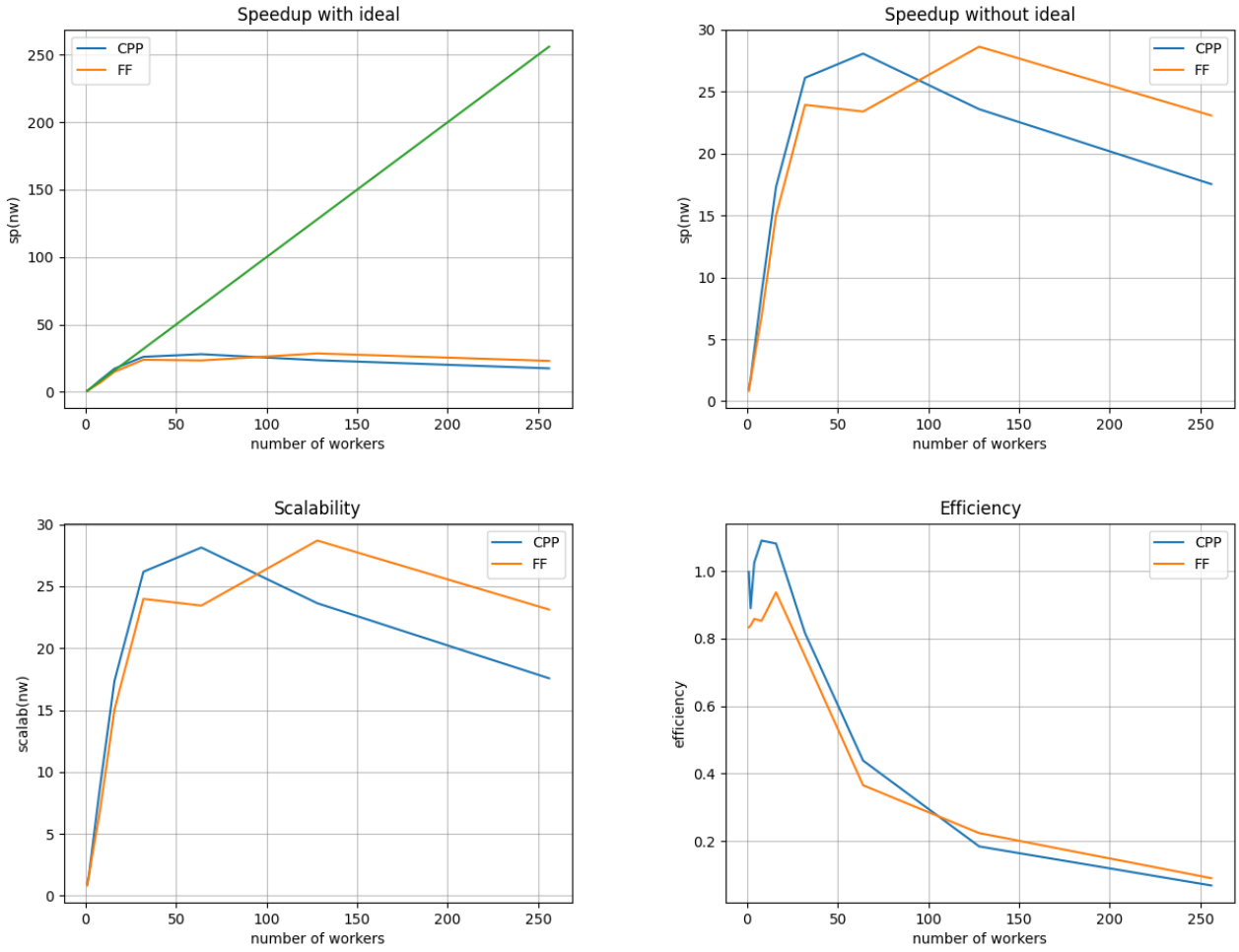

Figure 3: Metrics on 100k Graph

4 Conclusions

In this project, are being proposed two solutions using the same logic but different frameworks, and then has been calculated some performance, and compared the results. As seen both the application have similar behavior. The performance achieves are quite good, except for a high parallel degree, and the cause of it is clearly visible with the Table 4 that reports the time to compute the frontiers by the workers. The problem is about the graph visiting, as we can see when at the second level the frontier is really big the algorithm perform quite well and continue to achieve better and better results, but the time of creating a large number of threads for the frontiers of level 3,4,5,6,8 and 9 (that have a really small size)

in this case add a quite large amount of overhead and so the algorithm starts to perform worse.

Overhead Table ($\mu secs$)								
Frontier Size	**2**	**4**	**8**	**16**	**32**	**64**	**128**	**256**
2005	164119	87639	70734	57133	107304	66566	93236	89886
95860	5735031	2642597	1123729	531723	268484	171620	151697	140548
1808	276521	120861	51426	26408	16927	19929	34614	55810
93	15562	7457	4694	3778	12524	13833	34102	53047
36	6613	3307	2237	4599	7954	12462	32171	52499
15	3328	1888	2688	3259	9177	13421	32348	52247
10	2228	1642	1806	4221	7425	12519	31209	51269
5	1351	1344	2095	2989	8594	13319	32247	50753
1	853	899	1086	4104	6529	12485	31468	51348

Table 4: Table to show how the overhead effects on performances

5 How to use

In order to compile the different algorithms it is possible to run the following commands

```
make
#or to compile only 1 vesion at time
make bin/graph_generator
make bin/sequential_graph_search
make bin/cpp_graph_search
make bin/ff_graph_search
```

To run the executable files, go in folder **bin** and run:

```
./graph_generator [seed] [number of nodes]
./sequential_graph_search [file_path] [starting node] [value]
./cpp_graph_search [file_path] [starting node] [value] [nw]
./ff_graph_search [file_path] [starting node] [value] [nw]
```

For all the graphs has been used a *percentage* of 5, instead for the **100k graph** has been used a *percentage* of 2, because the size of the file grows too much.