Graph Search

Nunzio Lopardo (600005)

University of Pisa

Master’s degree in computer science, ICT Solution Architect

Final project presented for the

Parallel and distributed systems: paradigms and models course

Academic Year 2020/2021

Contents

1 Introduction

This report describes the work to create an implementation of parallel *Breadth-FirstSearch* in order to find in a graph all the occurrences of an input value. In particular, has been implemented three solutions, two using the C++ STL and one using the Fast Flow library.

2 Problem Analysis

2.1 Project Track

A graph is described by a set of nodes N (with an associated value) and a set of arcs (oriented pairs of nodes). 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 node X found in the graph during a bread first parallel search starting from the node S. The graph to be searched is assumed to be acyclic.

2.2 Graph Generation

For the creation of the graphs an algorithm has been developed taking into account the objectives of the project and also the available storage and computational capabilities. The algorithm is based on the Erdős-Rényi model and has been implemented in C++.

To manage the graph is used a $map < key, value >$, in which the key is the node *id* and the values are instances of the class $Node$. This class is provided of all the properties of a node:

- *int id*;
- *atomic<bool> visited*;
- *atomic<bool> discovered*;
- *vector<Node∗> neighbors*.

The generation procedure requires in input the desired number of nodes *no_nodes* and a *density*, that indicates the probability to have an edge between each distinct couple of nodes. The generation starts filling a $map\{int, Node*\}$ with all the nodes of the graph. The next step is the connection between the nodes taking into account the direct and acyclic nature of the graph. So for each node the algorithm iterate on other nodes with larger ids and generates a random value between 0 and 1, if the number is below the density values a link is established.

For testing the solutions has been created 3 graphs:

- 10000 nodes and 0.02 of density;
- 10000 nodes and 0.5 of density;
- 10000 nodes and 0.8 of density.

2.3 Problem Analysis

The *Breadth-First Search* (BFS) is a graph visit algorithm, that takes in input a graph $G(V, E)$ and a starter node $v \in V$. The BFS starts initializing the frontier vector F with the starter's neighbors v and an empty one called next frontier F'. In the next phase start the visit of the nodes in F, and for each node three operations are performed: 1) marking the node as visited, 2) check if has the same value as the input one, and 3) his list of neighbors is scanned to update the next frontier F'. Every time a new node is discovered is marked as *discovered* and added to F'. When the visit of the frontier F ends, it gets cleared and swapped with F'. The frontier visit and the swapping step are repeated until the next frontier F' is empty.

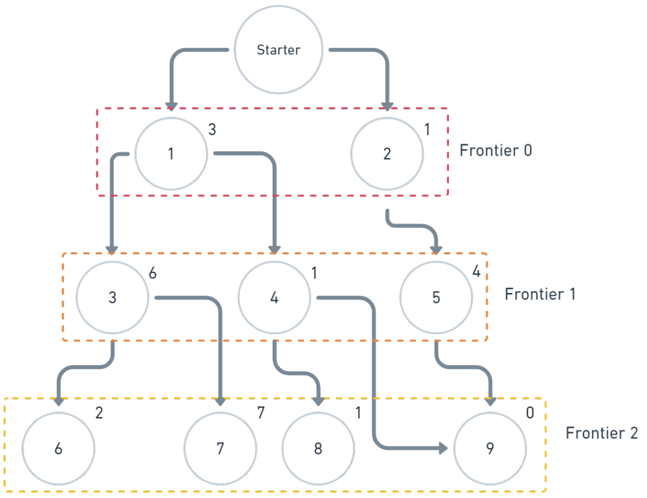

Figure 1: Example of the BFS visit

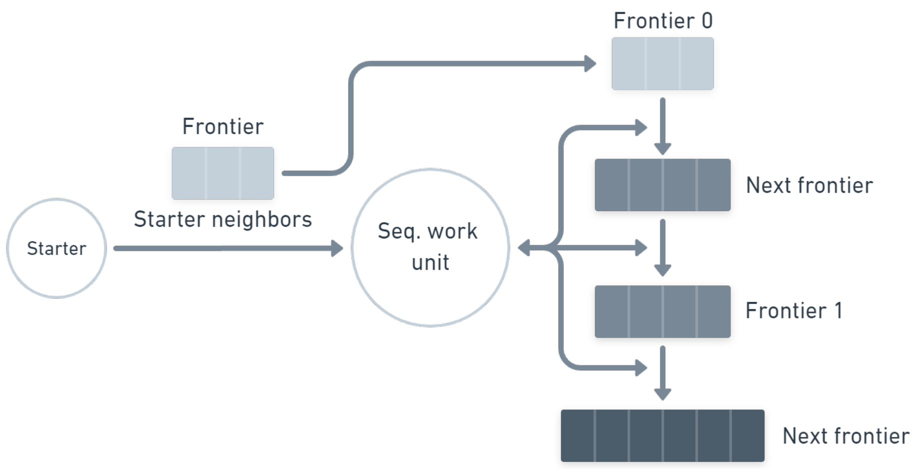

Figure 2: Schema of the sequential algorithm

So in time terms the BFS is performed as the sum (1) of this two terms:

- $T_{visit}(i)$: as the time to visit a new i th frontier;
- T_{swap}: as the time to exchange the old with the new frontier.

$$T_{seq} = \sum_{i=0}^{n} T_{visit_i} + T_{swap} \tag{1}$$

With n as the number of frontiers and $T_{visit_i} >> T_{swap}$. Due to this last consideration the phase that lends itself best to be parallelized is frontier analysis. Considering to chose a farm-based approach and use nw workers for visiting the frontier, we will obtain a visit time equal to T_{visit}/nw. However, to this, it is necessary to add the farm initialization time T_{init}, the time to divide and assign frontier chunks T_e and the collecting time T_c. These last two terms replace the T_{swap} of the sequential version. Thus the completion time of the parallel version will be:

$$T_{par} = T_{init} + \sum_{i=0}^{n} max\{T_e, T_{visit}/nw, T_c\} \tag{2}$$

It is necessary to say that everything depends on the properties of the graph, it makes sense to go for parallelization if you work with large frontiers in order to have $T_{visit}/nw >> T_{init} + T_e + T_c$. In fact, for graphs in which the degree of each node is very small, the disadvantages of parallelization will be greater than the benefits, this is due to the management of workers and synchronization in the merge phase of new frontiers.

3 Proposed Solutions

As mentioned in the 2.3 the best spot of the execution to parallelize is the frontier scanning, this due to its huge size in most of graphs. To achieve this has been used a **farm pattern** in which the frontier is splitted among all the workers by an **emitter** and the results of worker computation are merged in a new frontier by a **collector**. In all the following solutions the *nw* is intended only as the number of workers in the farm, so are not considered the emitter and collector threads.

3.1 C++ STL Solutions

3.1.1 Static Scheduling

In this section will be described in details the solution with the static scheduling of the frontier's chunks. The execution starts with the initialization of the farm: an emitter, a vector of workers and a collector are created. In addition to this, are initialized all data structures for level analysis and synchronization. In particular, for communication, each worker has a queue where the emitter distributes chunk ranges. On the other hand, the data are retrieved from the farm exploiting shared references of the workers' new frontier with the collector.

With this approach, the emitter divides the frontier F size by the number of workers (nw), then creates the pairs in which are present start and the end position of the chunk. These pairs are pushed in the worker's queue that extracts it and starts to work on his chunk. To avoid that the emitter execution restart a condition variable has been used to put the emitter thread in a wait state. Popped the chunk from the queue, the worker starts to iterate on the frontier in its range. For each node, firstly checks if the vertex has already been visited exploiting the *compare_and_exchange* that allows to perform both fetch and update operations in an atomic fashion. The method returns true if it was able to change the state of the node to visit, otherwise, it is skipped. In the case of a positive outcome, the analysis of the node continues by checking the value associated with it and visiting its list of neighbors. During this list iterations, is verified if the neighbor has been already added in the next frontier vector by another worker. To synchronize the farm with the collector the workers perform a $fetch\_add$ operation on an atomic variable end_of_task, when it reaches nw the collector operations start. In this phase all the workers' next frontiers are merged in F', the old frontier F gets cleared and swapped with F'. In addition, the condition variable state is changed to allow emitter restart. These three steps are repeated until the collector's next frontier is empty, in this case in the frontier is pushed a special node *Stopper* received by the emitter that blocks the worker using *Stop_exe* pairs. When the worker receives this pair atomically fetch and add the local counter of the found occurrences to a shared counter.

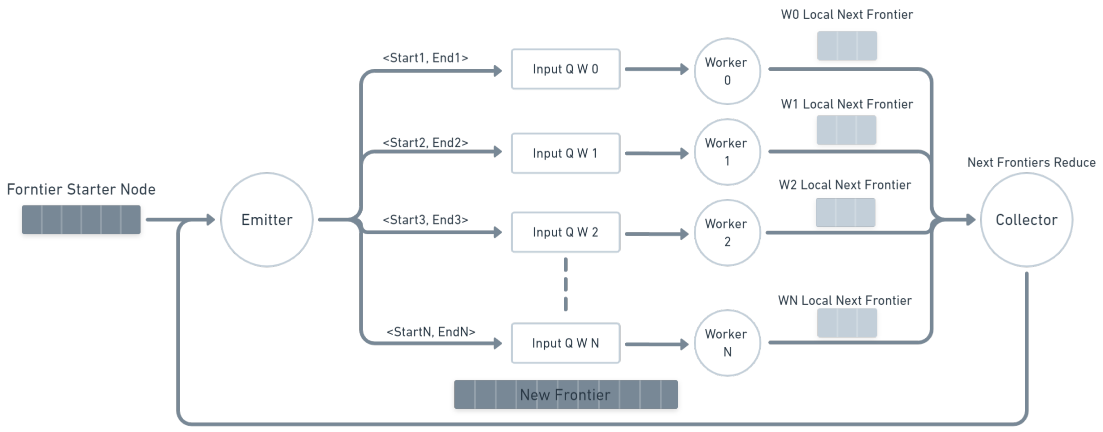

Figure 3: Schema of the static scheduling solution.

The main problem of this approach is the load balancing mostly in the cases in which the graph is highly connected. In this scenario, the worker that gets the first chunk has the possibility to visit a large section of the graph and so makes the visit of the other workers faster and creates a bottleneck due to the necessity of waiting for the first worker to finish the visit its portion making the service time of the farm becomes $T_{farm} = max\{T_{w_1}, T_{w_2}, ..., T_{w_n}\}$. This can be seen in the tables 1 and 2.

	Worker 1	Worker 2	Worker 3
Time (μsec)	103925	62256	14293
Discovered nodes	2048	1429	1514

Table 1: Load balancing of analysis, with static scheduling, of the first frontier, size 5013, graph 10K 0.5 density

	Worker 1	Worker 2	Worker 3
Time (μsec)	6685	5137	2971
Discovered nodes	1579	461	109

Table 2: Load balancing of analysis, with static scheduling, of the second frontier, size 7121 nodes, graph 10K 0.02 density

3.1.2 Dynamic Scheduling

To better manage the load balancing between threads and to avoid the possibility that the first worker explores most of the graph a solution can be to divide the frontier into smaller chunks to reduce the visibility of the graph to workers. Due to these changes, the scheduling policy has been changed in favor of a dynamic load balancing handled by a shared task queue Q. Another consequence is the addition of an additional input parameter, $chunk_size$. This value is used by the emitter to create the new range pairs. By increasing the number of chunks to be generated, increases the work of the emitter and consequently its time T_e, introducing more overhead. To preserve the solution performance, the workers extract the new chunks as soon as they are available in the shared queue. To monitor the execution, also in this case, is used the end_of_task variable that keeps track of all generated chunks: the emitter increments it for each new pair pushed in the Q and the workers decrements it for each popped one when it reaches 0 the collector is triggered.

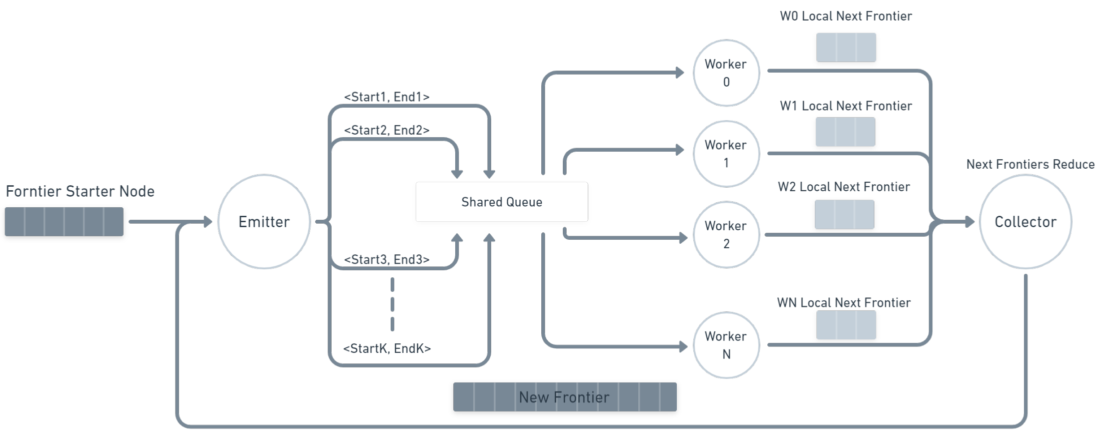

Figure 4: Schema of the dynamic scheduling solution.

Using a dynamic approach for task management the load balancing is distributed evenly across threads as can be seen in the table 3.1.2.

	Worker 1	Worker 2	Worker 3
Time (μsec)	23121	23376	22797
Discovered nodes	2543	3376	578

Table 3: Load balancing of analysis, with dynamic scheduling, of the first frontier graph 10K 0.5 density

The considerable gain in terms of load balancing removing the bottlenecks in the execution of the farm is counterbalanced by a increase time of the emitter T_e, as can been seen in the following tables 4 and 5.

NW	10K 0.02D	10K 0.5D	10K 0.8D	10K 0.02D	10K 0.5D	10K 0.8D
1	398	411	552	149	126	38
2	905	325	47	238	149	898
4	1299	665	466	270	330	77
8	1091	1284	1738	290	171	77
16	1524	712	1524	359	230	359
32	1631	1460	1464	461	335	52

Table 4: Total time (μsec) spend by Emitter and Collector in μsec, static solution.

NW	10K 0.02D	10K 0.5D	10K 0.8D	10K 0.02D	10K 0.5D	10K 0.8D
1	1137	565	602	111	78	24
2	1524	1091	899	244	103	26
4	1465	2014	1131	211	102	49
8	2398	905	2120	214	278	85
16	2614	2211	1047	162	119	47
32	7947	6751	3645	188	165	75

Table 5: Total time (μsec) spend by Emitter and Collector in μsec, dynamic solution with 32 of chunk size.

3.2 Fast Flow Solution

The Fast Flow solution is based on the same reasoning of the first C++ STL solution (3.1), in particular, it has been implemented using a farm in which has been removed the collector and added for each worker feedback channel to the emitter. For data communication has been used a personalized task struct in which is present a pair that indicates the start and the end position of the chunk and a next frontier vector. Similar to the first C++ version, the load balancing is static but in this case, the scheduling exploits the Round Robin algorithm embedded in Fast Flow.

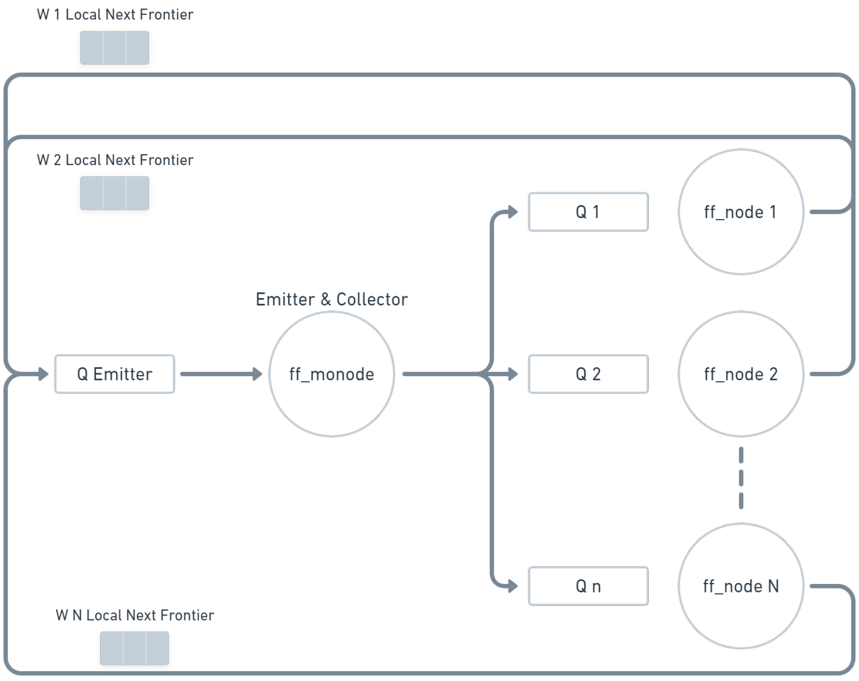

Figure 5: Schema of the dynamic scheduling solution.

4 Results Analysis

In this section, the results obtained will be shown and compared. Each solution was tested on the proposed graphs in 2.2 to verify their behavior in different contexts. The test has been executed on **Xeon PHI**, with 64 cores and 4 hardware thread per core.

In the table (6) below, are included the sequential times used in the plots and to compute the speedup of the various solutions.

Sequential	10K 0.02D	10K 0.5D	10K 0.8D
Time (μsec)	21964	162399	359583

Table 6: Time results of the sequential executions.

The following plots show the performance trend of the proposed solutions, starting from the low dense graph. In this first graph, with density equal to 0.02, all the four solution have the same behavior also for the performance drop around the 10 workers. Moreover, the dynamic solution with chunk size of 128 the knee point comes before due the smaller size of the frontiers.

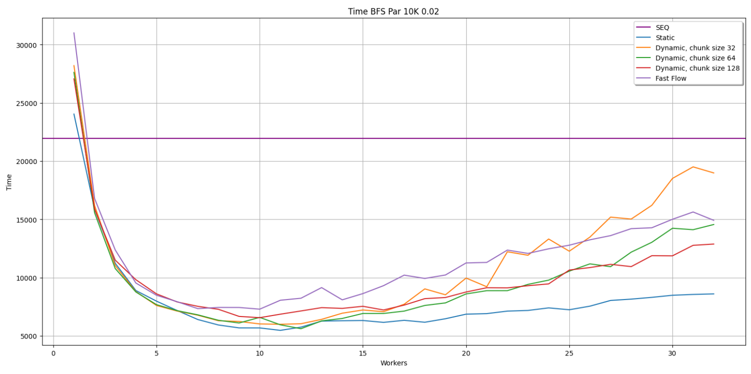

Figure 6: Time plot graph 0.02 density.

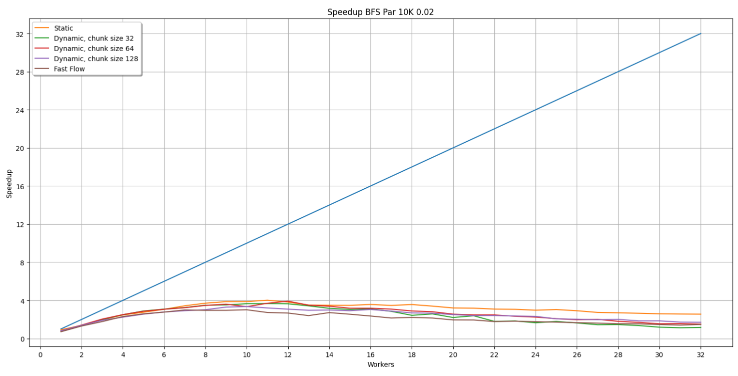

Figure 7: Speedup plot graph 0.02 density.

Growing density to 0.5, and so the frontier size, the static version starts to suffer of load balancing problems mainly in with lower *nw*. Indeed, static approaches, the speedup growth is limited by the large size of the chunks that give the possibility to the first workers to visit most of the graph nodes creating a bottleneck. Meanwhile, dividing the frontier in smaller chunks provides the possibility to handle efficiently the free workers.

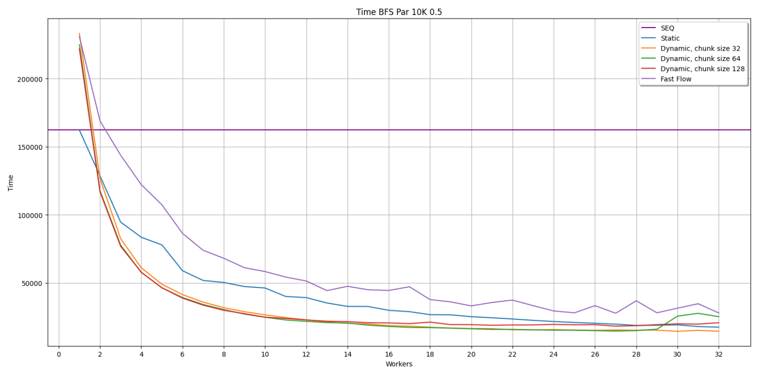

Figure 8: Time plot graph 0.5 density.

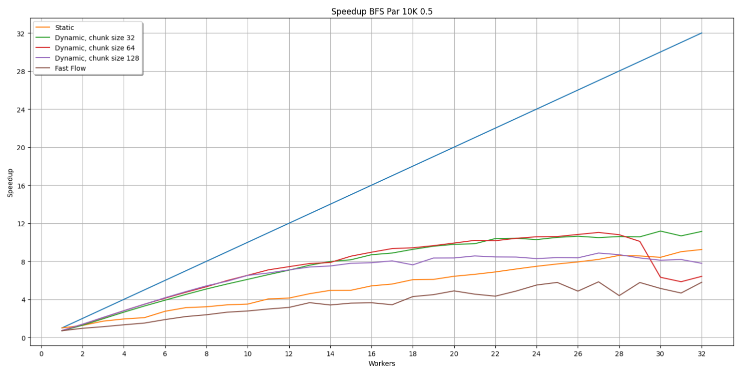

Figure 9: Speedup plot graph 0.5 density.

The analysis of the last graph is almost the same and also in this case the Fast Flow version maintains a speedup margin about one point less than the version with static scheduling. This could be due to the task management by fast flow and for runs with many threads the need to re-execute *nw* times the *svc* method of the emitter with related checks. Also, for each new level of the graph the tasks are not reused but new ones are created, for very large frontiers this is a considerable overhead compared to the emitter in other versions (see table 4).

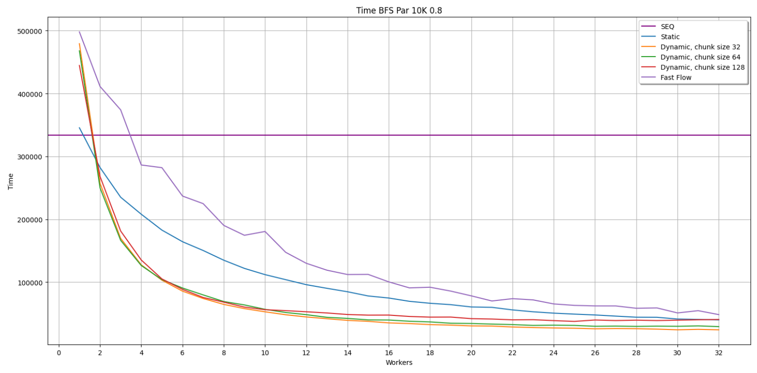

Figure 10: Time plot graph 0.8 density.

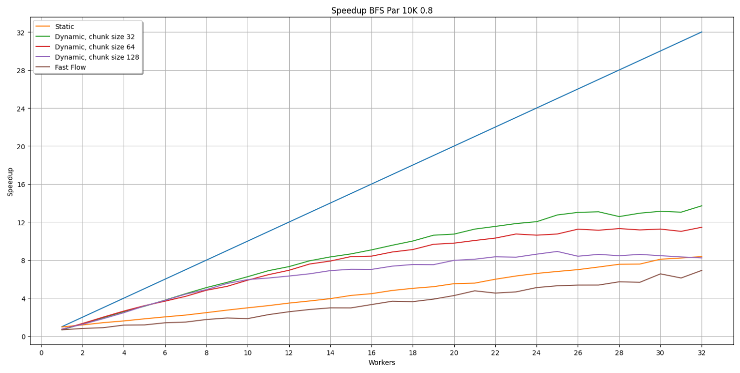

Figure 11: Speedup plot graph 0.8 density.

In the following tables you can see the speedup trend from a numerical point of view for all versions.

Workers	10K 0.02D	10K 0.5D	10K 0.8D
1	0,913606	1,001233	1,040779
2	1,383123	1,262705	1,273857
4	2,471753	1,943502	1,729321
8	3,705753	3,223546	2,667787
16	3,569641	5,421432	4,819437
32	2,554548	9,240867	9,029531

Table 7: Speedup static scheduling.

Workers	10K 0.02D	10K 0.5D	10K 0.8D
1	0,708265	0,703803	0,791419
2	1,308471	0,961453	0,943141
4	2,303997	1,328917	1,39952
8	2,953739	2,382475	2,059774
16	2,360451	3,646711	3,770874
32	1,472611	5,801207	7,44725

Table 8: Speedup Fast Flow solution.

Workers	10K 0.02D	10K 0.5D	10K 0.8D
1	0,778865	0,696421	0,750639
2	1,357478	1,284081	1,378061
4	2,497896	2,655661	2,922916
8	3,490782	5,093752	5,879286
16	3,107527	8,691875	10,69519
32	1,156913	11,14306	15,42083

Table 9: Speedup dynamic scheduling solution with chunk size of 32 nodes.

5 Conclusion

In the report, the Breadth-First Search algorithm has been analyzed and some possible parallel solutions to improve the performance of the sequential implementation have been proposed. The problem analysis leads to determine that the frontier visit is the best spot for the parallelization. To achieve it, a farm-based approach has been used, in particular, three solutions were developed: static and dynamic scheduling using pthread and static scheduling with Fast Flow version.

The evaluation of the results shows that with a static partitioning, that splits the frontier based on the number of workers in the farm, presents a worse load balancing. This scenario occurs especially with a low number of workers and with very dense graphs. To solve this problem, a dynamic scheduling version has been proposed that allows to divide the level into smaller chunks and using a shared data structure between workers to manage the workload in a more balanced way. In Fast Flow the obtained results are very similar to the static pthread version, as they share the same workload management, with a drop in performance for numerous farms that carry a greater overhead due to the communication mechanisms. The behavior of the solutions on the tested graphs leads to believe that with larger number of nodes it could be possible, in principle, to obtain remarkable speedups, therefore they could be preferred to a sequential version.