Graph Search

Davide Montagno Bozzone (535910)

January 25, 2022

Contents

Abstract

This is the final report for the Parallel and Distributed Systems: Paradigms and Models course. The overall objective is to find how many occurrences, of a pre-selected value, are present within the graph. The algorithm requested for this project is the Breadth-First Search. In this report, I will show the difference between parallel and sequential execution using both the Standard Library offered by CPP and FastFlow.

1 Introduction

Breath-First Search (from now on I will refer to it as BFS) is a well-known algorithm used to process graphs. The computation maintains two frontiers: the first one is used to process the current level of nodes, where the latter is used to keep track of the children's nodes of the previous frontier. The algorithm repeats until the next frontier is empty. Since each nodes is independent from the others we focus on a particular kind of Data-Parallel Problem: the embarrassingly-parallel one.

2 Implementation design

Starting from the constraints provided by the project requests a graph is described as a set of N nodes, where each of those has an associated value; moreover, each node has a different number of edges connected to other nodes.

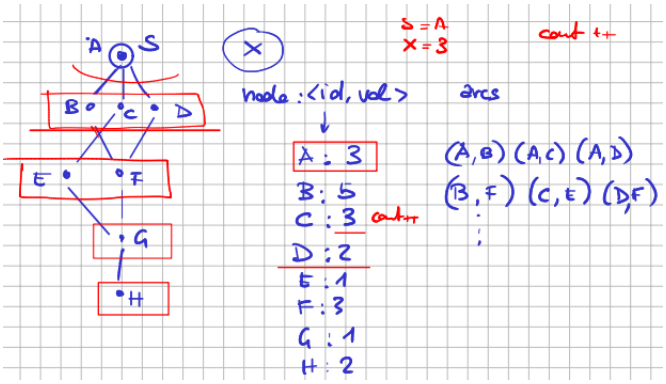

Figure 1: Example of Graph

2.1 Graph Generation

The file *gen_graph.cpp* provides an implementation of graph generation. It takes as input the *number of nodes*, the *minimum* and *maximum number of out-going connections/edges* and the *maximum value* assigned at each node as

corresponding value. Finally, a *seed* is used to generate random values.
The implementation starts with the creation of the corresponding graph file; then, it writes the number of nodes (used for parse purposes) before starting the nodes generation. All the nodes are represented as an integer increasing sequence where for each increasing value (*node*) is associated a random value (*node_value*). After that, the algorithm proceeds by adding at each generated node a random number of outgoing edges, by adding $nodeIDs$ for each node, in the range defined previously.

2.2 Sequential Implementation

As said at the beginning, the simplest way to perform this kind of computation is to maintain two frontiers and a visited queue. For the sequential implementation nothing special is done; in fact, the algorithm starts by filling the initial frontier (the current one), formed by all the out-going edges of the starting node. After visiting all the nodes, the current frontier is swapped with the next one until the next frontier is empty. As result the time for the sequential computation can be defined as:

$$T_{seq} = (\forall_{node \in Nodes}.T_{node}) + (\forall_{edge \in Edges}.T_{edge}) \quad (1)$$

where T_{node} is the cost time for visiting one node and T_{edges} is the cost time for visiting one edge. Since no concurrency is used in this kind of computation we omit two other times: $T_{counter}$ the cost time for updating the global counter and T_{swap} the cost time necessary to swap the frontier.

2.3 Parallel/FF Implementation

As before the computation maintains two frontiers. The first difference with respect to the sequential implementation is relative to the mutexes. In fact, if before only 1 worker performs the BFS, here a variable number of workers perform the same kind of computation using multiple shared data structures. Furthermore, before proceeding with the next frontier, each workers must await the termination of the other threads until they have finished processing all the nodes present in their current frontiers. Finally, if some thread finishes before the others it tries to steal jobs from other workers causing more overhead. As result the time for the parallel computation can be defined as:

$$T_{par} = [T_{sync} + (\forall_{node \in Nodes}.T_{pop} + T_{steal} + \forall_{edge \in Edges}.T_{edge}+)]/nw \quad (2)$$

where T_{sync} is the cost time relative to the overhead caused by thread synchronization, the T_{pop} is the cost time relative to the overhead caused by accessing shared data structures.

3 Performances Evaluation

The performances of each algorithm are tested by using the Xeon PHI machine with a number of threads varying from 2 to 256 (by using power of 2).

Furthermore an active wait is added to compare the behavior with the models without. From now on all the times are expressed in *usec*
Finally, if we are interested in the speedup of the following computations we could use the following relationship:

$$Speedup(n) = \frac{T_{seq}}{T_{par(n)}} \tag{3}$$

3.1 Sequential

Section 2.2 introduced some cost times relative to the different different computations. In particular, the results relative to the sequential are stated in Table 1. As we can see, when we increase the number of nodes the time for

Graph	Time Frontier	Time Edges
10000_100_300_2	24.1727	7.0406
20000_100_300_2	24.1284	7.0396
20000_100_900_2	36.3117	16.1726
100000_400_1000_2	43.9912	22.511

Table 1: Timing operations relative to sequential code

computing a single loop (the current frontier is completely seen) also increases. Notice that, by adding just more nodes the different times not vary so much. Instead, the situation is different when we increase the minimum and maximum number of out-going edges per node; in fact, the different times start to increase faster than before (when we added just more nodes).
For the reported results, I decided to omit the times relative to swap and for update the counter since they are negligible.

3.2 Parallel

The results obtained by the Parallel implementation are stated in Table 2. The latter considers the timing for Pop, for calculating the Edges and finally the same values are shown for the times related to job stealing.
The first thing to notice is that the time for computing the Edges increases, so we have more overhead, when we increase the number of threads. This behavior is due to the fact that each thread try to access the *visited data-structure* in order to mark the node as visited. As consequence, the main influential effect for the overhead is the number of edges. In fact, by increasing just the number of nodes the values are not too much different; but if instead we increment the maximum number of out-going edges the time starts to increase a lot.
If we considered the parallel version with job stealing the effect is to add more overhead for popping each node, since the threads can access other queues to steal something; in particular, if the number of threads increases the time will also increase due to the competitions. Recalling that we are introducing this

kind of overhead to reduce the time spent in waiting uselessly other threads. Let's see now what happened to the time spent for waiting in the barrier. As result, the effect of adding the *steal behavior* reduces the time for each thread spent in the barrier as the Table 2 shows since each thread doesn't waste time because of is occupied to steal other jobs. As conclusion we can use this kind of tool whenever the number of workers is not too much high since the overhead spent for waiting to access other queues becomes larger. Furthermore if the graph has a low number of nodes and low number of edges the overhead generated by the steal behavior surpass the time for the computation itself, so it is useless to add it.

3.3 Comparisons

In Figures 2, 3, 4, 5 are stated the different behaviours of the algorithm. The first thing to notice is that the parallel (both with standard and *Fastflow* library) computation is always the worst with respect to the sequential one. Instead, this behavior is not shown when we consider the version with adding of an active wait, that simulates somehow an heavier computation, since this time is spread over the threads.
Furthermore, the graphs show a slightly benefit in the initial part (in particular when we use a number of threads not so much high); after that, the timing relative to all the graphs, without active wait, start to increase since the overhead caused by accessing shared data structures plus the addition of waiting in the barrier start to increase, as said in the previous section.

4 Usage

- To compile and use the gen_graph file use:
  g++ -std=c++17 gen_graph.cpp -o gen_graph
  ./gen_graph total_nodes min_edges max_edges max_value seed
- To compile and use the seq version use:
  g++ -std=c++17 -O3 seq.cpp -o seq
  ./seq num_nodes min_edges max_edges start_node value active_wait(0/1) debug(0/1)
- To compile and use the par version use:
  g++ -std=c++17 -O3 -pthread par.cpp -o par
  ./par num_nodes min_edges max_edges start_node value num_workers
- To compile and use the FastFlow version use:
  g++ -std=c++17 -O3 -pthread -I fastflow ff.cpp -o ff
  ./ff num_nodes min_edges max_edges start_node value num_workers active_wait(0/1) debug(0/1)

In order to get all the stats for each graph, please save all the outputs in the *stat* folder which is divided into 3 parts: **ff,par,seq**. Inside each folder, there is another one named **active**; it is used for the execution of models with active wait. Moreover a **steal** folder is added to compare the two behaviors (with and without the steal job). Notice that the latter contains only **ff, par** with their relative subfolders for the active wait.
After executed and saved all the stats file, please use the *parse.py* file in order to parse correctly all the files. It is used for removing multiple strings, printed by different threads, which are in the same line. After this process, you will have all the parsed files ending with $-parsed.txt$.
In order to get all the stats for the sequential part please use the *grep* and *awk* commands by searching for *Sequential, Frontier and Edges (time is in position 5)*. For for the parallel parte instead please use the *grep* and *awk* commands by searching for *Parallel/Fastflow, Pop, Edges, Swap, counter*.
The *plot.py* file is used for printing all the stats or plot the different timing relative to the parallel (both with standard and *Fastflow* library) computations with respect to the sequential one.

5 Final Considerations

BFS computation as we saw is a very naive procedure. From the previous results I can state that the check if one node is visited or not takes, as shown in previous sections, a time which is much less with respect to the synchronization of different threads.
In general, we can use this kind of tool whenever the task computed by each thread is much heavier or if it is not, the number of workers necessary for this kind of computation must be not too much high (obviously it depends on how much the graph is large). Nevertheless, the BFS can be applied in any graphs paying attention on the different parameters we set for the different models such as: nodes or workers, and last but not least the number of edges.

Graph_thread	Mutex	Mutex-Steal	Edges	Barrier	Barrier-Steal
10000_100_300_2	1.45845	1.24402	100.31	9095.5	493.0
10000_100_300_4	1.1987	1.66677	122.254	3574.12	3738.5
10000_100_300_8	1.5389	2.59138	167.803	4269.12	4133.38
10000_100_300_16	2.13879	4.31707	227.528	7437.97	6861.66
10000_100_300_32	3.72593	6.44511	378.286	21533.6	11737.5
10000_100_300_64	6.1332	13.919	427.908	65449.0	22551.2
10000_100_300_128	8.31173	14.2443	478.989	55003.6	23274.7
10000_100_300_256	5.70106	23.2237	398.751	81314.6	36130.1
20000_100_300_2	1.12535	1.22431	104.277	3399.8	1350.2
20000_100_300_4	1.17411	1.60542	124.419	9569.7	2965.8
20000_100_300_8	1.33742	2.26036	168.532	8917.89	4765.17
20000_100_300_16	2.10234	3.2046	243.315	8689.21	7493.68
20000_100_300_32	3.28535	4.99835	374.456	21101.8	11136.5
20000_100_300_64	6.18109	10.4845	629.949	65408.7	20817.7
20000_100_300_128	8.30482	16.8798	653.713	177021.0	45367.4
20000_100_300_256	6.67737	24.5184	580.672	227280.0	75546.0
20000_100_900_2	1.84519	2.0017	258.929	5826.25	2719.75
20000_100_900_4	1.88546	2.4068	325.82	5812.5	2633.38
20000_100_900_8	2.4906	3.57465	419.075	28111.1	5718.5
20000_100_900_16	4.22389	6.29684	611.913	24439.8	15912.2
20000_100_900_32	5.25807	9.41961	897.741	17829.4	13057.2
20000_100_900_64	10.3682	21.0563	1522.66	41328.4	27656.8
20000_100_900_128	20.8103	37.5923	2270.25	123949.0	49443.1
20000_100_900_256	34.4169	40.7432	2245.32	102097.0	45818.2
100000_400_1000_2	2.55737	2.95025	443.784	15872.2	28953.2
100000_400_1000_4	2.73338	3.25598	496.098	79784.1	15029.1
100000_400_1000_8	2.73749	3.1923	576.43	68546.0	74060.1
100000_400_1000_16	3.10627	3.80332	821.446	35377.5	13408.1
100000_400_1000_32	3.91884	4.74332	1190.92	32668.5	16026.0
100000_400_1000_64	7.5528	8.45224	2198.73	67516.2	36307.2
100000_400_1000_128	15.0559	16.5235	4051.56	149536.0	66250.3
100000_400_1000_256	35.0531	49.0627	7073.24	560094.0	141885.0

Table 2: Timing operations relative to parallel code

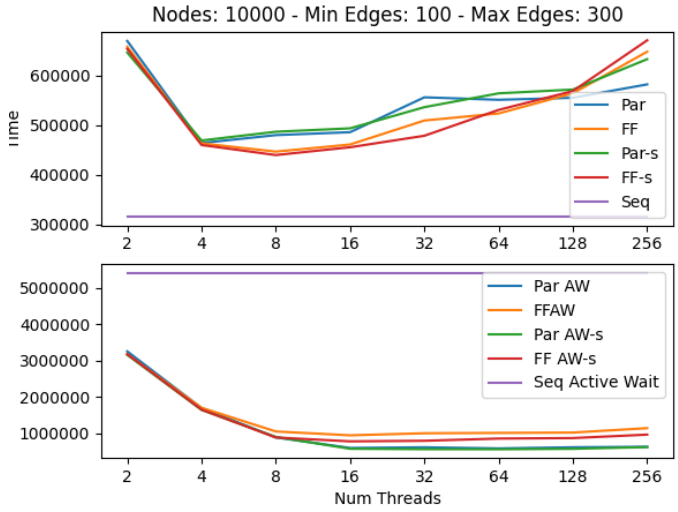

Figure 2: Comparison performances by varing the number of threads

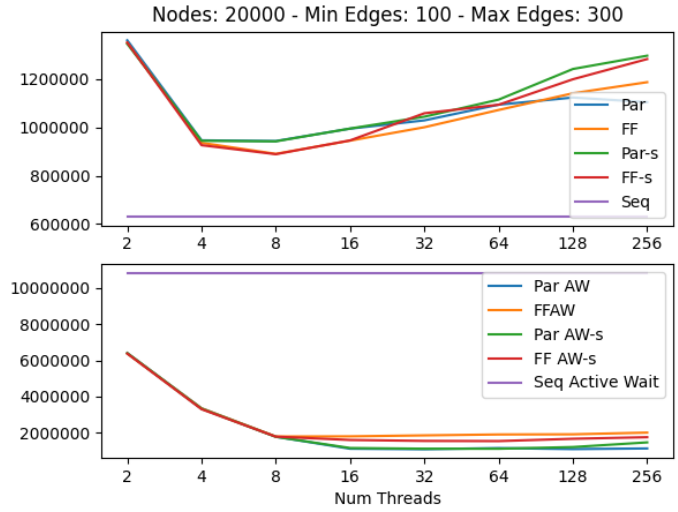

Figure 3: Comparison performances by varing the number of threads

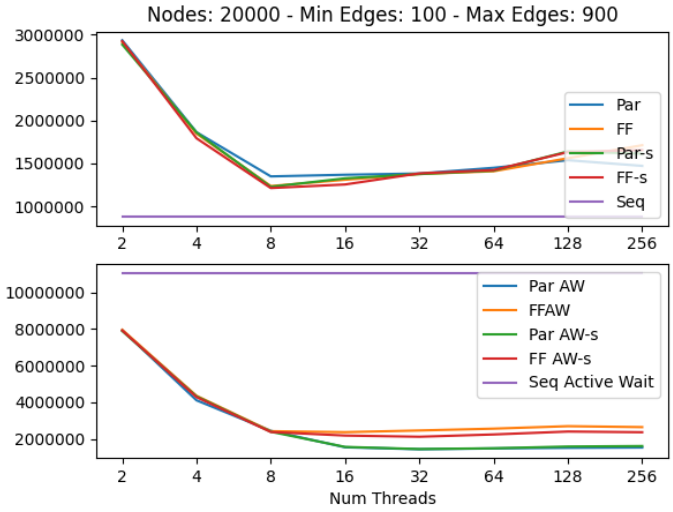

Figure 4: Comparison performances by varing the number of threads

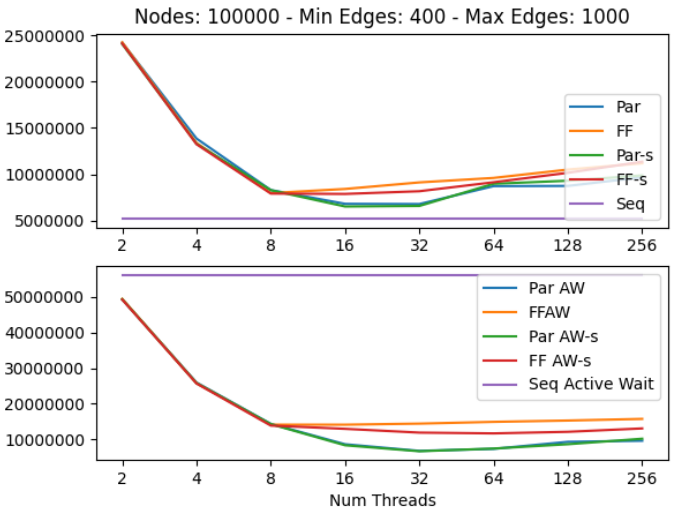

Figure 5: Comparison performances by varing the number of threads