Rotundo Mariagiovanna. Matricola: 560765

Graph Search

a) the main design choices

The chosen problem is the "Graph Search" which is the implementation of a parallel BFS.
Considering a parallel implementation, the considerations done for this problem are mainly 3:

1. we can analyze only one level at a time. To do this, we need synchronization between threads because, without it, a thread can start to explore the next level while another thread is still exploring a node of the current level. Because of this, a **barrier is implemented** to ensure that if a thread finishes its computations on nodes of a level it waits until all the other workers finish the exploration of nodes of that level.
2. for each node, we must check its value and we must see what the new reachable nodes are to explore in the next level of the BFS. The check of the value of a node is an independent operation because it does not depend on the other values, so we can do this in parallel without any synchronization between threads. Instead, about the new reachable nodes, we must avoid that the same node is seen more than once because it can be explored only by a thread and only once. But a node can be reached by more than one node, so more threads (that explore different nodes) can see the same node as reachable. For this reason, we need **synchronization to avoid** that a new node will be **explored twice** (or more) in the next level.
3. When we start to explore a level, we already know what are the nodes that we must explore at that level because we obtain this information by the exploration of the previous level.

Because of point 3, we have used **two data structures**, one for the **input**, i.e. with the nodes to analyze in the current level, and one for the **output**, i.e. for the new nodes reachable by the nodes of the current level that must be explored in the next level. Two thread workers never take the same node from the input data structure and a node is taken only once by a thread. The **input data structure does not contain duplicates** to avoid synchronization between threads to check if someone else is computing the same node. To avoid consistency problems because of the access to this data structure by more threads, the threads can only read the elements to analyze, but they do not modify the data structure after they read an element. Because of point 2, the creation of the input data structure for the next level, containing the new nodes, is done after the exploration of all the nodes. To do this, the threads calculate, in **parallel, partial results** that are merged after the threads finish the exploration of the current level. In the partial results, there can be duplicates so, when they are merged, the duplicates are removed. In this case, we can see what the new nodes are in a parallel way, but we need a sequential computation to merge the results of the parallel computation.

b) the expected performances

The things that we can parallelize are **the check on the value** of the nodes (to check if it is equal to X), and the calculation of the partial results about the nodes of the next level. The check is a very simple operation that requires one comparison and a sum if the comparison is true, so it is very fast to execute.

The **computation of the new nodes** may be longer. Even if we parallelize the calculation using partial results, we spend time putting them together because we must check if in the final result there are duplicated nodes. So, we, in parallel, select only new nodes for the merge phase. If most reachable nodes by the nodes of the level we are exploring were already seen in past levels, the creation of the new input data structure is faster, because the merging phase, which is sequential, considers only a few nodes, and the workers check these old nodes in parallel without synchronization. In the phase of the merge of the partial results, we look at each node in these results. For this reason, the time of this phase depends on the graph and by the explored level. If many new nodes are seen by many threads or by the same thread more than one time, we must see these nodes many times to remove duplicates.
So, looking at these observations, we can notice that **the performances of the implementations depend on the considered graph**:

1. If the **graph has few nodes** (it is a small graph), parallelizing the BFS we have some overhead to parallelize the problem (the overhead to fork/join threads, the one for the barrier to synchronize the threads and to merge the results of the workers) and the parallel BFS requires more time than the sequential BFS, because this overhead is more than the time needed to execute the sequential computation.
2. If the graph **has few nodes in the levels** and these nodes are **less than the number of workers**, we have some inefficiency in the use of the resources because we pay overhead to create the threads, but some of them do not do useful work, because they do not explore any node. So, in this case, increasing more and more the number of workers, we expect worse performances because we increase the number of useless threads, but we have additional overhead because of them. For example, if the considered graph has only one node for each level, we have only one worker that explores all the nodes and all the other threads do nothing, but we have overhead to fork and join them and to synchronize all workers. So, in this case, we should use at most 1 worker, and for this reason, we should not parallelize the BFS to not paying overhead because of the parallelization.
3. Assuming big graphs with the same number of nodes. If in a graph, the nodes have **many connections** (**arcs**) with the other nodes, in general, the parallel BFS has better performance than the parallel BFS where nodes have **few arcs**. This because, in general, if in the graph many arcs connect nodes, the number of levels of the BFS decreases and the levels have many nodes (so the probability of having useless threads is smaller). So, in the whole execution of BFS, we execute fewer times the barrier to synchronize the workers and we can pay a smaller total overhead for it. Furthermore, if there are many connections the probability of seeing nodes already explored is higher, so the time needed to create the input data structure using the partial results decreases because we have partial results with fewer nodes.

So, looking at these considerations, in general, we expect more speedup more the nodes and the arcs in the graph are, and less speedup less the nodes and the arcs of the graph are. These considerations are general, but we can see examples with graphs with more specific structures.

Considering the same graphs, if the exploration of nodes would require a **more complex task**, we would obtain better performances (bigger speedups) than now. This because the overhead due to the parallelism may become negligible with respect to the time needed to execute the exploration of the nodes.

c) the actual implementation and results achieved

Implementation

Both for the proposed parallel solution that uses C++ threads and for the one with FastFlow, the main design choices described in the first paragraph are used in the implementations. For both implementations, a farm is used.

In the implementation with **C++ threads** there are no threads for emitter and collector, but only the threads workers. Here, for each level, the workers take one node at a time from the input data structure that is a vector. The workers do not extract the node to explore removing it from the input vector, because otherwise, we need synchronization because of possible problems of consistency. This because more workers can extract nodes at the same time changing the vector, so we should allow access to the vector only to one worker at a time. But in this way the extraction of the nodes becomes sequential. To avoid this, to explore the nodes, the workers read the nodes without removing them from the input vector and, to avoid that two (or more) workers explore the same node, the nodes are assigned to the workers using a cyclic strategy (looking at their indexes). So, assuming *n* workers, the first worker read the first element, the second worker read the second element, and so on until the n^{th} worker read the n^{th} element. After the n^{th} element, the reads restart from the first worker and so on for all the elements in the vector.

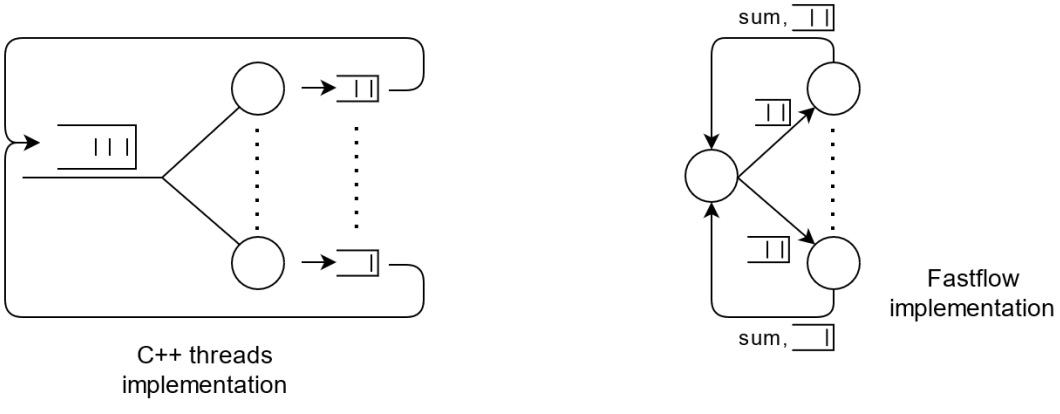
In this way, we assign the same number of nodes to explore (at most one node more for some workers) to the workers and the extraction of the element from the vector can be done in parallel. For each node that a worker explores, if its value is equal to X, the worker increases a local variable (not shared). Here it **counts the occurrences** of X that it finds considering all the nodes that it sees (in all the levels). These partial results of the workers are computed in parallel. At the end of the BFS, each worker adds to a global variable the partial result that it has calculated. To avoid problems due to the concurrency the variable that they update is an atomic variable. We have used an atomic variable instead of the mutex because it is more efficient (its use requires less time respect the time we need using a mutex). For each node, the workers look for what **the reachable nodes** that must be explored in the next level are (nodes that are not already explored). A new node must be inserted in the input vector only once (because this does not contain duplicates) but it can be reachable by more than one node. So, a node can be seen more than once by a worker or by different workers. Because of this, we need to remove duplicates.

To parallelize the search of the new reachable nodes, the workers do not write on the same data structure, but each of them writes in a different output vector and, at the end of the exploration of all the nodes of the level, these vectors are merged. So, when a worker explores a node, for each reachable node, it sees if the node was already seen in the past levels and, in this case, it ignores this node, otherwise it inserts the node in its output vector. At the end of the exploration of all nodes of the level, the last worker that terminates the exploration creates the input data structure using all the partial results computed by all the workers removing the duplicates and setting these new nodes as seen.

For **Fastflow**, the implementation is very similar to the one that uses C++ threads (just described). Also in this case a farm is used and this is a not ordered farm because it does not matter the exploration order of the nodes in a level. One of the main differences between these 2 implementations is that the Fastflow solution is implemented with a master-worker scheme. So, there is an emitter that manages an input vector for each worker and merges the output vectors. Each input vector contains the nodes

that will be explored by a worker. Each output vector will contain the new reachable nodes find by a worker. The emitter inserts in each input vector the nodes that a worker must explore, passes them to the workers, and then waits for the results by the workers. At the start, the emitter only inserts S in the first vector. A worker, when it receives a vector, analyses all the nodes in the vector counting the occurrences of X and the new reachable nodes from these nodes. Each vector is seen by only one worker. They do not insert in the output vector nodes with the flag that indicated that they were already seen, but a new node may be reachable by more nodes in the vector, so the output can contain duplicates. In the end, the worker sends to the emitter the number of occurrences that it found and the output vector. When the emitter receives this information, it adds to the variable for count occurrences, the number of found occurrences and for each node in the vector that it receives, it sees if the node is seen for the first time. In this case, it set it as seen and insert it in one of the new input vectors for the next level using a cyclic policy. Otherwise, it ignores this node. In this way, it removes eventual duplicates in all the output vectors that it receives. When all the workers finish their computation on the nodes of the level, the emitter assigns the new vectors to the workers. In this implementation, no mutexes or atomic variables are used, and the merge of the output vectors can start before reaching the barrier.

In the following picture, we can see the scheme about the behavior of the 2 parallel implementations.

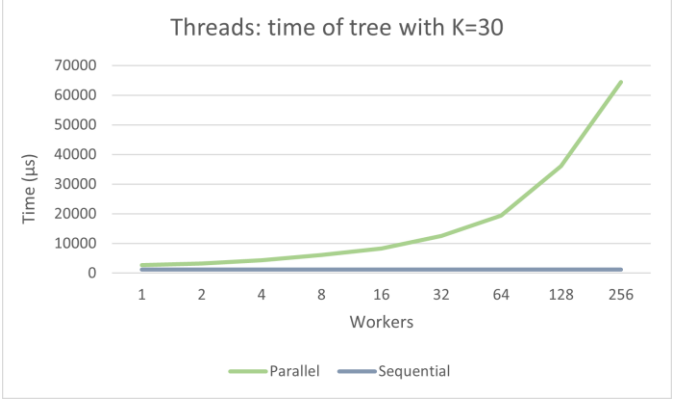


Result achieved

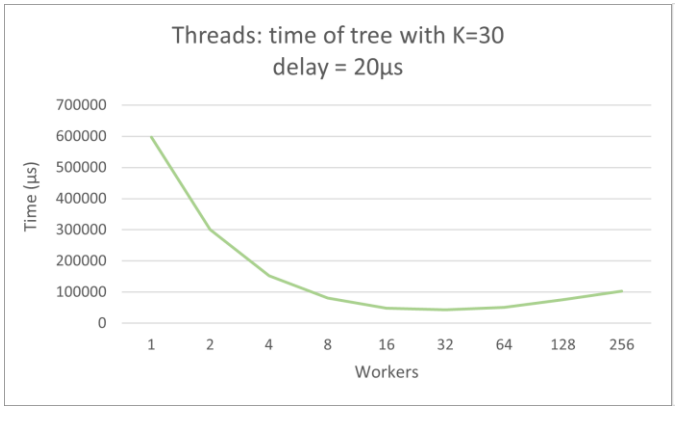
Looking at the considerations and the results that we expect described in the previous paragraph, we can see some examples and results, and consider some specific structures of graphs.

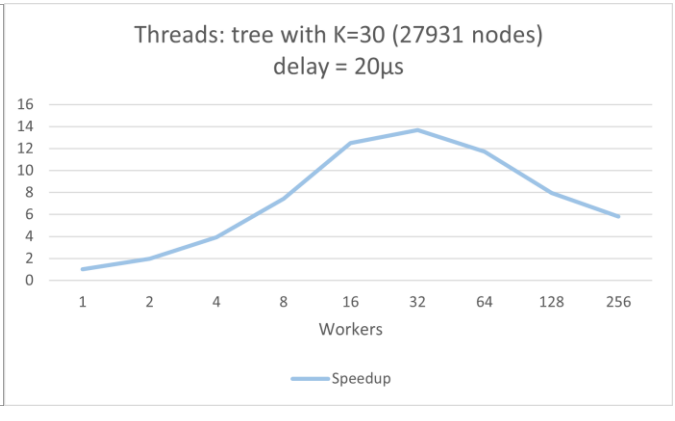
As the first example, we can see a comparison between performances on a tree and a graph represented by an upper triangular matrix with all 1s in the upper part and 0 on the diagonal (so, with as many arcs as possible to guarantee that it is acyclic). For this example, the graphs are analyzed using the C++ threads implementation, but the FastFlow implementation has a similar behavior/results. The tree used as an example is a **tree** where each node has **30 child nodes (K = 30)**, and the total nodes are 27931. Ignoring the first level of the tree because it has always only one node, let us consider the following ones. Here each worker, for each node, looks at its value and finds the new nodes that are exactly K. All these nodes are seen exactly once because they have only one input arc. So, in the parallel implementation, in this case, we see each node of the next level in parallel and then we see again all these nodes sequentially to merge the results even if we are sure that there are no duplicates because the graph is a tree. So, we do useless work, and we spend time because of the barrier to synchronize the threads. In the sequential implementation, we do not do this additional work and we do not have overhead because of the barrier and the creation of the threads. Supposing that the time

to count of the occurrences is negligible, the sequential implementation is very fast to execute the BFS, and we expect that the parallel implementation has worse performances with respect to the sequential one because the overhead is not negligible. For this reason, we expect that in the parallel version, more are the threads, and more is the time of the BFS because the overhead increases. In the following picture, we can see the time that we need to execute the BFS on the considered tree using different numbers of workers. We can notice that increasing the number of workers, the time to execute the BFS increases as we expect, and that it is always bigger than the sequential time. The sequential time does not depend on the number of workers because there is no parallelism. Since the parallel time grows up increasing the number of workers, we do not have speedup.

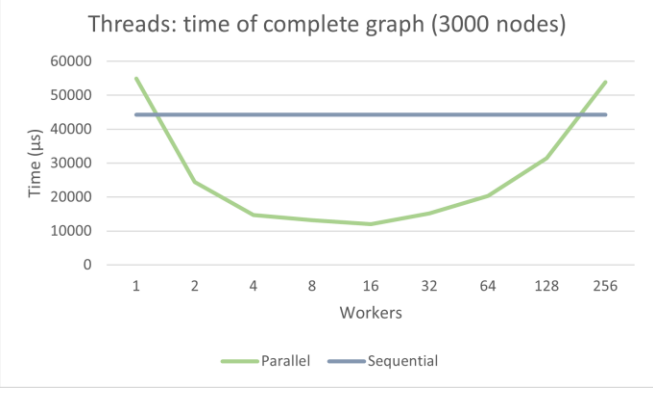


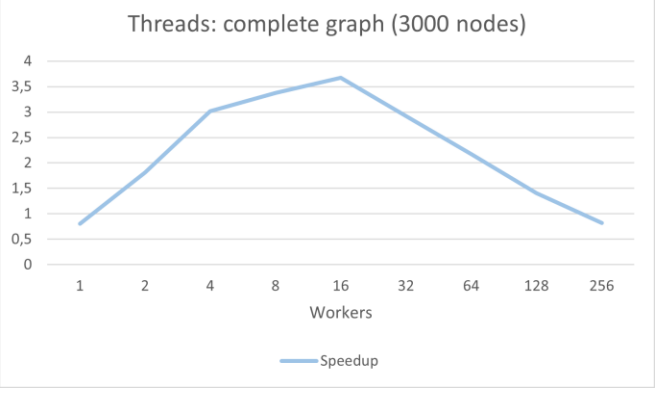
The results may be different if the work to execute in parallel on each node is bigger. In this case, we may have speedup because the time due to the overhead to manage the parallelism become negligible with respect to the time to execute the work on the nodes. With the parallel solution, we can execute this work in parallel and reach good speedups. We can see an example in the following pictures considering the same tree of before. In the following picture on the left, it is shown the time needed to complete the BFS, using the parallel solution, on the tree described above but with, a delay (to simulate more works) of 20µs for each node. On the right, it is shown the obtained speedup knowing the sequential time that is about 600000µs. In this case, we have speedup and using few threads, it is near to the optimal one (*n* workers gives a speedup that is close to *n*). We have used a delay of 20µs because it is time bigger enough to have a negligible overhead with few threads but smaller enough to show that by a certain number of threads the overhead starts to be significant. So, consequently, the speedup increases until a certain point where it starts to decrease.



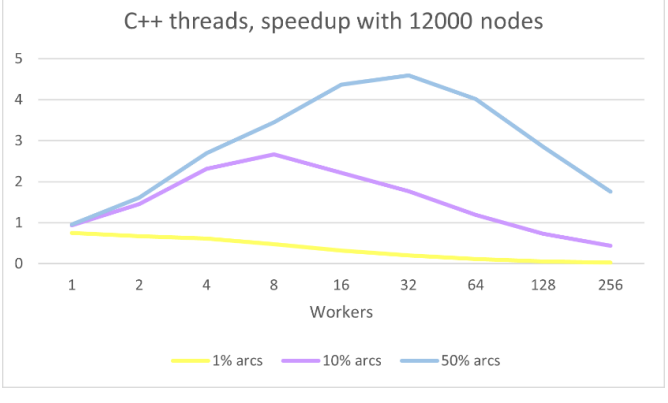


Using a graph with many arcs the results may be different. For example, consider a graph with 3000 nodes where each node has an arc towards each node with a greater id of its id. For these graphs with all arcs, the BFS has only 2 levels to visit: the first with the starting node (S) and the second with all the other nodes because they are all reachable from S. When the second level is explored, for each node, each worker see that the reachable nodes are all already seen so, it put no node in the output vector This is done in parallel while in the sequential computation we must see all these reachable nodes sequentially, so we need more time. After that, the merge phase considers only empty output vectors because there are no new nodes. So, the creation of the new input vector after the termination of all the workers is very fast. In the following pictures, we can see the time of the BFS on the **complete graph of 3000 nodes** and the speedup that we have on it. We can notice that by increasing the number of workers, the time of the BFS starts to decrease because we have less sequential work and more parallel work to execute. By a certain number of threads, the time of the BFS starts to go up because of the overhead that becomes more and more significant (because of the management and the synchronization of the threads) with respect to the time to execute the exploration of the nodes in parallel that becomes smaller and smaller. Because of these performances, increasing the number of workers the speedup increases and by a certain point, it starts to decrease. The sequential BFS does not depend on the number of workers.





With a generic graph the behavior depends on the number of the nodes, and for graphs with the same number of nodes, depends on the number of arcs in the graph. More the arcs of the graph are and more the parallel BFS scales. With a big number of arcs, the sequential time increases because nodes are seen more times. The parallel program may have better performance than the sequential one because the analysis of the reachable nodes, to see if they are new or not, is done in parallel and the probability of find nodes already seen is higher with many arcs between nodes.



In the picture on the left, we can see the speedup on graphs of 12000 nodes. We consider 3 graphs, with a different number of arcs. Here, higher is the percentage and more arcs are in the graphs. So, the graph with 1% of arcs is the graph with the smaller number of arcs, the graph with 10% has more arcs than 1%, but less than the graph with 50%. The percentage indicates the maximum number of outgoing arcs that a node can have. 50% indicates that a node can have at most the 50% of the maximum number of outgoing arcs. In this

picture, we can notice that more are the arcs, and more the parallel BFS scale and, as a consequence, we have better speedups. This is for the points 2 and 3 saw in the *b* paragraph.

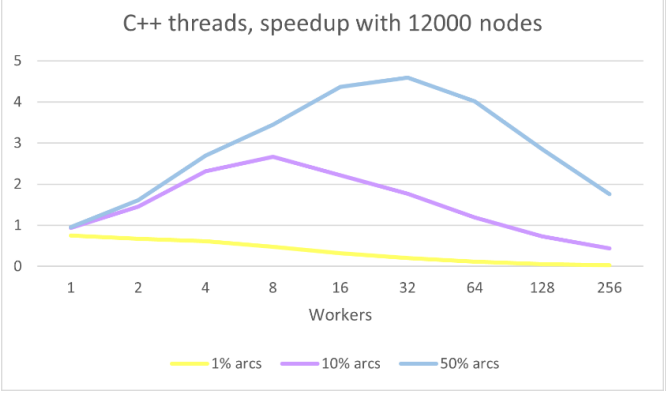
d) a comparison among the implementation and performances of the C++ and the FastFlow solutions

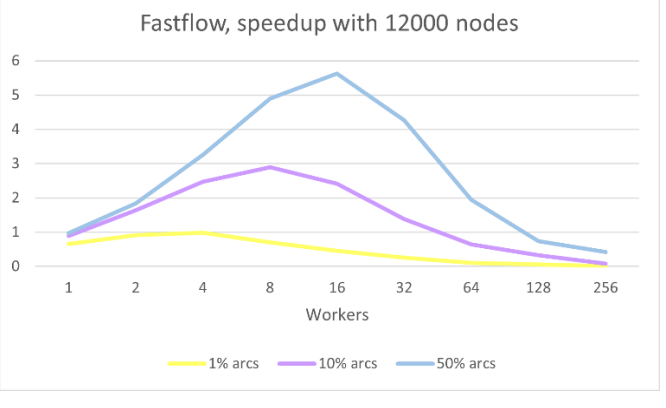
Implementation

The implementation of the two solutions with the C++ threads and Fastflow are very similar. Both use a farm and the behavior of a worker to explore a node it is the same for both the implementations. The main difference is that the barrier in the C++ threads solution is implemented in the threads themselves, while in FastFlow it is implemented in the emitter. In the implementation with C++ threads, there is not an explicit emitter. In the FastFlow implementation, the emitter sends nodes to explore to a worker passing it a vector with these nodes, doing that a node is only in one vector and only once. Another important difference is that, in the implementation with C++ threads, the removing of duplicated for the new reachable nodes is done after that all workers finish to explore all nodes that are assigned with them, so it is done sequentially. In FastFlow, this is done by the emitter every time it receives back new nodes by a worker. So, when it receives a partial result, the emitter sums the found occurrences that it receives to its local variable to count the occurrences and checks if in the vector with the new nodes that it receives there are nodes that were already seen or duplicates. In this case, it removes them, and it sets the new nodes as seen. This is done sequentially in the emitter (because it analyzes one result at a time), but these partial results can be analyzed in parallel to the execution of the workers that have not finished yet. This, instead, cannot be done in the other implementation. The same is true for the total number of occurrences that in the C++ threads solution it is calculated at the end of the BFS sequentially by all workers that update an atomic variable.

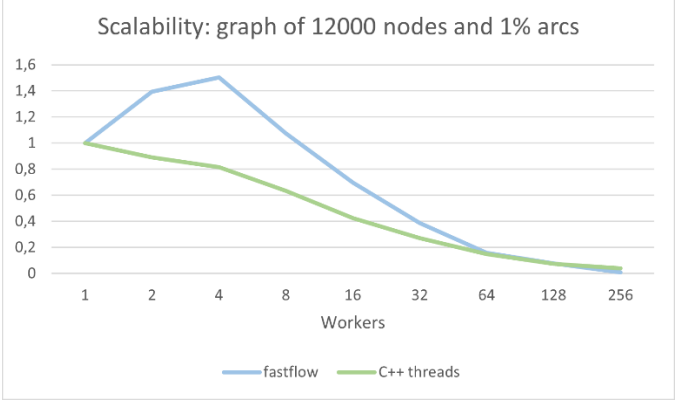
Performances

The Fastflow implementation and the implementation with the C++ threads have similar behavior on the same graph, but on graphs where nodes have many arcs, the FastFlow implementation usually is faster than the other one. Instead, for graphs with very few arcs between nodes, like, for example, the trees, the FastFlow implementation may be slower. On some graphs, the FastFlow solution requires fewer workers to have a similar speedup to the one that we obtain from the C++ thread implementation. For example, considering the same graphs of before with 12000 nodes and a different number of arcs, the speedups that we obtain are the ones in the following pictures.





In the picture on the left, we can see the speedup of the 3 graphs using the C++ threads implementation, while on the right we can see the speedup of the same graphs but using the FastFlow implementation.
We can notice that in the case of 10% of arcs, both the implementations have no speedup. For 50% of arcs, FastFlow has better performances. It reaches a bigger speedup with fewer workers with respect to the C++ threads implementation. The FastFlow implementation may have higher scalability than C++ thread implementation, but from a certain number of workers, the scalability decreases faster than the other solution. For example, we can notice this when we use 256 threads (but also with 64 and 128) for the graph with 50% of arcs (but also for the one with 10% of arcs).
For 1% of arcs, the C++ thread implementation does not scale, instead, for the FastFlow implementation, until 4 threads the time of the BFS decreases before starts to increase again with more than 4 threads. We can see the scalability on the graph with 1% of arcs, for both the implementation, in the next picture.



e) an analysis of the reasons of differences observed among expected and measured performance

For the following observations, we consider the C++ thread implementation.
We know that the maximum speedup that we can have using *n* threads is *n*.

The speedup using n threads is defined as: $sp(n) = \frac{Tseq}{Tpar(n)}$

Here, $Tpar(n)$ includes the time to execute the BFS splitting the work between all the n workers, but also the overhead due to the parallelism.
For simplicity, we analyze execution times considering one generic level of the graph for the BFS and not the whole graph.

Time to execute a level of the BFS

Ideally, the best time to explore a level using *n* threads is $\frac{TseqLevel}{n}$ (where $TseqLevel$ is the time to explore that level in a sequential way), but we can have a time different from this:

1. Because the maximum number of workers that we can really use may be smaller than *n* because the level has a number of nodes smaller than *n*. For example, if in a level we have 2 nodes and *n*=4, we can use at most 2 workers of the 4 that we have, and the best time becomes $\frac{TseqLevel}{2}$.
2. Even if we can use all the threads, some nodes can require more work, and as consequence, more time to be explored than other nodes. For example, suppose again that n=4. If in a level there are 4 nodes where the first has 10000 child nodes and the other 3 has no child nodes, the first thread, that executes the exploration on the first node, has more computations to do, so require more time than the other workers to finish the exploration of its node. A worker may have many nodes to explore that require more time and other workers, instead, may have only nodes that require a smaller time to be explored. So, the load assigned to the workers may be unbalanced and for this reason $TparLevel(n) > \frac{TseqLevel}{n}$ (where $TparLevel(n)$ is the time to explore the level in parallel having *n* workers). We have tried to balance the load, assigning nodes to the workers using cyclic scheduling, but there is no guarantee that the load is the same for each worker.

Overhead

We have overhead due:

1. The fork and join of the thread workers.
2. The barrier.
3. The merge of the results computed in parallel.

The 1st time depends on the number of threads: more the threads to create are and more time we need.

The 2nd instead depends on the level we are exploring, from its nodes, and from the number of workers we are using. If the load due to the nodes to explore is well balanced the time to reach the barrier is smaller than the time we need instead if the load is not balanced. This is because if it is not balanced, some workers finish their computations fast, and some workers instead require more time. So, we have some workers that, from a certain point, do nothing and other workers that have still much work to do. So, the "inactive" workers must wait before starting the exploration on the next level.

The 3rd point depends on the partial results computed in parallel about the new nodes. More are the new nodes found and more time we need to merge them.

Total time

So, the time to execute the parallel BFS is equal to the time for fork and join a thread for the number of threads workers, plus, for each level of the BFS: the time to execute in parallel the exploration of the nodes (that as said before, can be more than $\frac{TseqLevel}{n}$ for the 2 reason above), the time due to the barrier, and the time to merge the results that were computed in parallel.

With FastFlow, we can have a smaller overhead due to the merge, because this can be done by the emitter in parallel to the computation of the workers, before the barrier, but we must consider that we have one thread more with respect to the other implementation, and we have some overhead because of this thread.

Because of all these considerations, looking at the structures of the graphs, we expect better or worse performances, and we have done some considerations of some specific graphs in the *c* paragraph.

f) the commands needed to rebuild and run the project

To run the projects, use the provided **makefile**. Compile using the command "*make*" or "*make all*". The makefile supposes that the directory of FastFlow is in the home. If the path of the FastFlow directory is different, change the path "***${HOME}/fastflow***" with the correct one. To remove the compiled programs, use the command "*make clean*" or "*make cleanall*".

To run the sequential program (***sequential.cpp***) you need to pass 3 parameters to the main: S (the initial node for the BFS), X (the value to look for), and the file with the graph to consider.

To run the parallel programs (with the C++ threads (***parallel_threads.cpp***) or with FastFlow (**master_worker.cpp**)), you need to pass 4 parameters to the main: S, X, the number of threads to use for the workers, and the file with graph to consider.
The number of threads passed corresponds to the number of workers but there are also other threads as the main thread and the one for the emitter of the farm (for FastFlow solution).

To run the projects with **delay**, remove the comment:

- at line 78 for the file *sequential.cpp*,
- at line 89 for the file *parallel_threads.cpp*,
- at line 176 for the file *master_worker.cpp*.

By default, the delay is a delay of 20 μs.

In the provided graphs the node from which you can reach all the other nodes is node 0.