SMP Report - Parallel BFS Search

Matteo Ceregini

July 2021

1 Main design choices

A *layer* k, denoted L_k, is defined as the set of all nodes (vertices) which are at a distance k from the source node x. The BFS search starts at a source node x in the graph and explores the current layer, i.e. all nodes that are at distance $d = 1$ from x before moving to the next layer, i.e. nodes that are at distance $d = 2$ from x. This process is repeated until there are no more nodes to explore. In general, the nodes in a layer L_d must all be explored before starting to explore the nodes in layer L_{d+1}, but the order in which the nodes of the current layer are explored is not relevant.

A parallel implementation of the BFS search must explore each layer sequentially in order to maintain the correctness of the algorithm. So we need to synchronize the parallel entities of our program (from now on workers or threads) in some way to ensure that the layers are explored sequentially, one at a time. To do this we need to distinguish between the nodes in the current layer L_d and those in the next layer L_{d+1}. One way to do this is to use two separate data structures to keep the nodes of L_d and those of L_{d+1}. The workers can then explore the nodes of L_d in parallel and insert the nodes at distance $d + 1$ in L_{d+1}. Just like in the serial version of BFS, when a node is visited for the first time, it is marked to ensure that this node is not visited more than once.

The implementation using the Fastflow framework and the one using C++ threads are briefly described below:

- **Fastflow:** it follows the design of a farm without a collector, where the emitter and the workers follow a master-worker schema. The emitter has two private data structures, representing the current and next layers, respectively. The emitter sends to each worker a node id (which uniquely identifies the node) at a time from the current layer. Each worker explores the node corresponding to the id that it received from the emitter. When a node is explored, its value is checked to count the occurrences of X. Then, the worker sent to the emitter a vector containing the nodes it visited. When the emitter receives a vector of node ids, it insert them into the next layer. When the emitter gets a number of vectors equal to the number of nodes of the current layer, it means that the current layer has been fully explored. In this case it checks to see if the next layer is empty. If it is empty, it means that the BFS search is finished since there are no more nodes to explore. If the next layer is not empty, the next layer becomes the new current layer and the whole process is repeated.

- **C++ threads:** threads share two data structures, representing the current and next layers respectively. Each thread is assigned a subset of nodes in the current layer to explore. Each thread explores its portion of the current layer. When a node is explored, its value is checked to count the occurrences of X. Then, each enters a synchronization phase in which it inserts the nodes it has visited into the next level and checks to see if there are any threads that have not finished exploring their portion of the current level. If there are, it waits for them to finish. Otherwise, it checks to see if the next layer is empty. If it is empty, it means that the BFS search is finished since there are no more nodes to explore. If the next layer is not empty, the next layer becomes the new current layer and the whole process is repeated.

2 Expected performances

A parallel implementation of the BFS search will introduce some overhead. Part of it is due to thread creation and destruction, but most of it is caused by the fact that the layers (not the nodes) must be explore sequentially. The overhead of the synchronization phase is mainly caused by the insertion of the nodes visited by the threads in the next layer. In the C++ threads implementation each thread inserts its visited nodes into the current layer after getting a lock, while in the fastflow implementation this is done by the emitter.

In addition, the shape of the graph determines how effective the parallelization of the BFS search will be. If many layers of the graph have less nodes than the number of threads then some of these threads will only cause overhead without any gain in speedup. If instead the graphs have very large layers it might be possible to gain some speedup as the work would be divided among the threads. If the nodes of the graph have an unbalanced number of edges, i.e. some nodes have many and others have very few, then some threads have more work to do then others. There may be some threads that are busy exploring nodes with many outgoing edges while other threads are waiting them, because they have already finished exploring the rest of the nodes in the current layer. Therefore, there would be threads doing no useful work as they have to wait for other threads to finish exploring their portion of the current layer. On the other hand, if the nodes of the graph have roughly the same amount of outgoing edges, each thread should have approximately the same amount of work for exploring each node and ideally all threads would finish their work at the same time.

Nonetheless, exploring a node is not computationally expensive. Exploring a node u of the graph involves the following steps:

1. Check if the value of u is equal to X and if it is, increment a counter.
2. For every node z that is adjacent to u, visit z if it has not been visited before.

Thus, even though the amount of work each thread has to do to explore a node is a bit unbalanced, the amount of work is still small. This fact can make it difficult to achieve a significant speedup using a parallel implementation of BFS search. Since the time needed to explore individual nodes is small, the overhead needed to ensure that the layers are explored sequentially has a greater impact on the performance. Having said that, we cannot achieve the ideal speedup of n when using n threads. A possible approximation of the speedup that can be achieved is the following:

$$speedup(n) = \frac{T_{seq}}{T_{par}(n)} = \frac{T_{seq}}{\frac{T_{seq}}{n} + t_{thread} + t_{synch} \times L} \tag{1}$$

where:

- t_{thread} is the time taken to create and destroy the threads.
- t_{synch} is the average time spent synchronizing the threads before the next layer can be explored.
- L is the number of layers in the graph.

If we consider the Amdahl Law, the term $t_{thread} + t_{synch} \times L$ represents the part of the computation that cannot be parallelized and therefore the serial fraction represented by this term limits the maximum achievable speedup.

3 Actual implementation and results achieved

3.1 Node class and Graph class

In both the C++ threads and fastflow implementations, each node in the graph is represented as an object that has the following attributes:

- An *id*, which uniquely identifies the node. Node ids start from zero and are assigned incrementally.
- A *value*, which is checked when the node is explored during BFS to count the occurrences of a certain value X.
- The *adjacency list* of the node that is the ids of the nodes that are adjacent to it.
- A boolean variable *color*, which indicates whether the node has already been visited or not.
- A boolean variable *alreadySeen*, which indicates whether the node is already present in the next layer or not.

A graph is basically a vector of Node objects, where a node with id $= i$ is inserted in the i-th position of the vector. The purpose of the *Graph class* is to create the graph from a file and accessing individual nodes.

3.2 FastFlow implementation

The implementation follows the design of a farm without a collector, where the emitter and the workers follow a master-worker schema. The behavior of the emitter and the workers is described as follows:

1. The emitter sends to each worker one node id at a time according to a round-robin policy. These ids are taken from a private vector of integers representing the current layer. Each time the emitter sends a node id to a worker, it increments the variable *sentMessages*, which is used to keep track of how many messages it has sent to the workers during the exploration of the current layer.

2. Each worker, when it receives a node id, accesses the corresponding node u of the graph and checks if the value of u is equal to X. If it is, it increments a global atomic variable used to count the occurrences of X in the graph. Next, it visits the nodes in the adjacency list of u and adds to a vector those nodes that have not been previously visited. Then, the worker sends this vector to the emitter.

   Note that the operation of visiting a node u, i.e. checking if *u.color* is equal to *true* and if it is adding *u.id* to the vector of visited nodes and setting *u.id* to *true*, is not done in mutual exclusion with other workers. So it may happen that two or more workers add the id of u to their vector of visited nodes ids, thinking that they were the first to visit u. This problem is automatically solved by the emitter, which even though it will receive the id of the node u from multiple workers, it will only add it to the next layer once.

3. When the emitter receives a vector of node ids, it checks, for each of these ids if it is already present in the next layer. This is done by looking at the value of the variable *alredySeen* of the corresponding node. If *alredySeen* = *false*, the id is inserted into a private vector of integers representing the next level and *alredySeen* is set to *true.*

   Each time the emitter receives a vector, it increments the variable *receivedMessages*, which is used together with the variable *sentMessages* to see if the exploration of the current layer is finished. If *receivedMessages* is equal to *sentMessages* it means that the emitter has received as many messages as it has sent and therefore the workers have finished exploring the current layer. In this case, the emitter checks if the next layer is empty.

   If the next layer is empty, it means that there are no more nodes to explore, so the emitter notifies the workers that the BFS search is finished and both the emitter and the workers finish their execution.

If the next layer is not empty, the emitter sets *receivedMessages* and *sentMessages* to zero and then swaps the current layer (which is empty) with the next layer. Then, the algorithm continues as described in step 1.

3.3 C++ threads implementation

The current and the next layers are represented using two vectors that are share among the threads. The behavior of the each thread is described as follows:

1. The current layer is divided between the threads in almost equal parts. If the current level is not large enough (fewer node ids than threads), some threads will have no nodes to explore. Each thread then explores the nodes of the portion of the current layer assigned to it. For each node u of its portion, each thread accesses the corresponding node of the graph and checks if the value of is equal to X. If it is, it increments a global atomic variable used to count the occurrences of X in the graph. It then visits the nodes in the adjacency list of u and adds to a private vector called *nextLayerIds* the ids of those nodes that have not been previously visited.

   Just like in the fastflow implementation, the operation of visiting a node u, i.e. checking if $u.color$ is equal to *true* and if it is adding $u.id$ to *nextLayerIds* and setting $u.id$ to *true*, is not done in mutual exclusion with other workers. So it can happen that two or more workers add the id of u to their own *nextLayerIds*, thinking that they were the first to visit u.

2. Each thread tries to get a lock, which is shared among all threads. When a thread gets the lock, it does the following steps:

   (a) It checks, for each id in *nextLayerIds*, if it is already present in the next layer. This is done by looking at the value of the variable *alreadySeen* of the corresponding node. If *alreadySeen* is *false*, the id is inserted into the next layer and the variable *alreadySeen* is set to *true.*

   (b) It increments the value of the shared counter *countWorkers.* If the value *countWorkers* is less then the number of workers then the thread enters a wait state on the shared condition variable *cv.*

   Otherwise, if the value of *countWorkers* is equal to the number of threads, it means that the thread is the only one that is still active, the others are all waiting on the shared condition variable *cv.*

   The thread checks if the next layer is empty or not. If it is, then there are no more node to explore therefore the BFS search is over and the thread set to *true* the shared variable *finished.*

   If instead the next layer is not empty, it swaps the current layer (which is empty) with the next layer. The thread calls *cv.notify_all()* waking up the other threads and then it releases the lock.

3. If the BFS search is not finished (*finished* = *false*), each thread goes back to step 1. Otherwise, each thread finish its execution.

3.4 Previous attempts

Before developing the two implementations described above, some preliminary versions were developed, but they proved to be too slow. With these earlier versions it was possible to achieve a much lower speedup than with the versions described above. These previous attempts can be summarized as follows:

1. In this version, access to the nodes of the graph was done in a mutually exclusive manner (using the *try_lock* function) to avoid duplicates. Threads/workers would access the current layer in a mutually exclusive manner. Then, after exploring all nodes in the current layer, threads/workers would mutually exclusively access the next layer to insert the visited nodes.

2. In attempt to improve performance, access to the nodes of the graph was implemented by associating each node with an atomic variable and then allowing a node to be visited atomically using the *atomic_compare_exchange_strong* function. But again, the performances were not that high.

3. Another approach was to allow multiple threads to visit the same node (thus no exclusive access to nodes) and solve the duplicate problem by using a *set* or *unordered_set* data structure to represent the layers, but these solutions provided worse speedup than the “final” implementations.

3.5 Results

To test the performance of the two implementations, three direct acyclic graphs were generated using the program whose source file is *generate_graph.cpp*. These graphs were generated as follows:

1. A tree with n nodes is generated where each non-leaf node has at least *min_children* children nodes and at most *max_children* children nodes. An id is incrementally assigned to each node starting from the root with $id = 0$. The integer value of each node is chosen randomly between zero and five (inclusive).

2. For each node u in the tree, an edge is added with every other node x such that $u.id < z.id$ according to the probability specified by the parameter *percentage_other_edges*. This process ensures that the resulting graph is a DAG. Also, if the BFS search starts from the root, is possible to explore all the nodes of the graph.

The three graphs used to test the performance of the two implementations have:

- $n = 30000$.
- *max_children = 1.*
- *max_children = 15.*
- *percentage_other_edges* between 15% and 35% depending on the graph.

More information about the graphs is given in Table 1.

Graph name	**percent_other_edges**	**size (MB)**
DAG-15	15%	256
DAG-25	25%	426
DAG-35	35%	596

Table 1: DAGs used to test the performance.

Each data entry shown in the charts in the Figures 1, 2 and 3 was obtained by averaging the measurements obtained from 100 different runs.

Both implementations are far from achieving the ideal speedup for a number of threads greater than 8. It can also be noticed that in the fastflow implementation there is a decline in speedup with a number of threads greater than 32, while in the C++ threads implementation this seems to happen with a number of threads greater than 64, although the speedup obtained is generally lower than in the fastflow implementation.

In addition, as shown in the charts in Figure 4, the fastflow implementation generally achieves higher efficiency than the C++ thread implementation using the same number of threads/workers. It can also be seen that the efficiency of the thread implementation almost always steadily decreases as the number of threads increases. On the other hand, the fastflow implementation first grows to an efficiency of about 0.85 and then decreases.

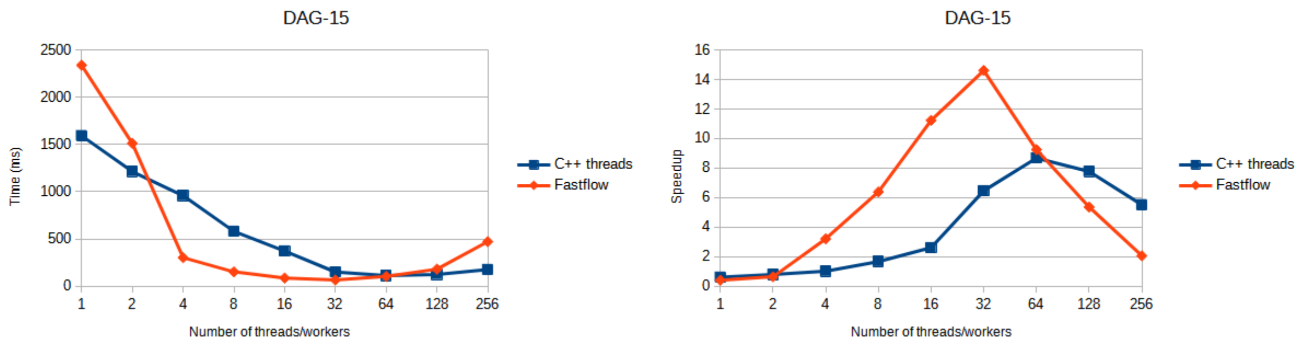

Figure 1: Time and speedup achieved with DAG-15

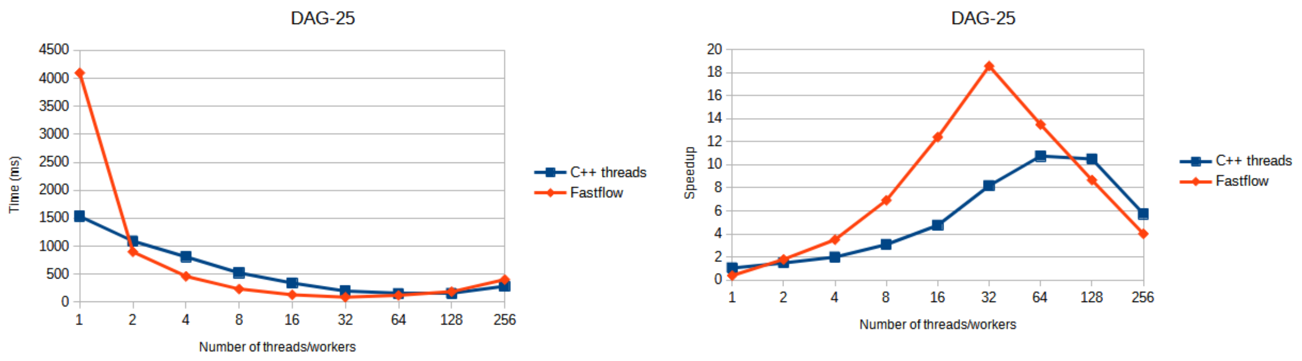

Figure 2: Time and speedup achieved with DAG-25

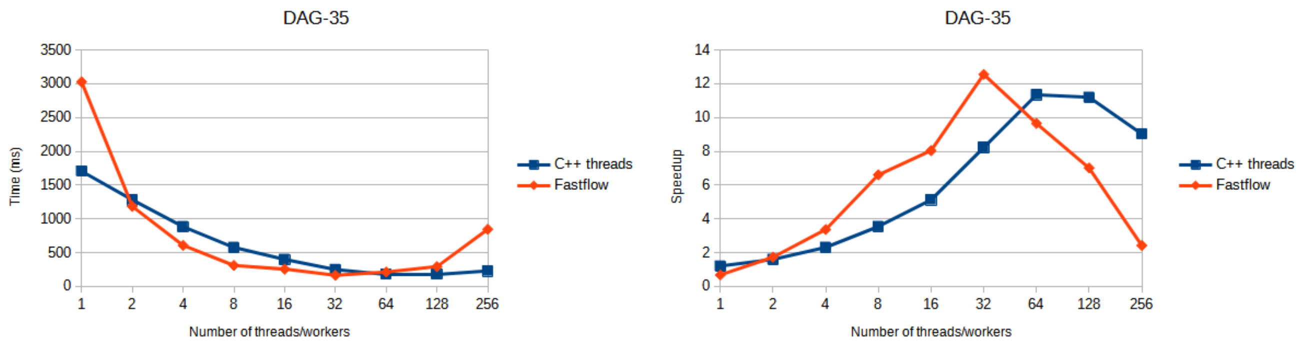

Figure 3: Time and speedup achieved with DAG-35

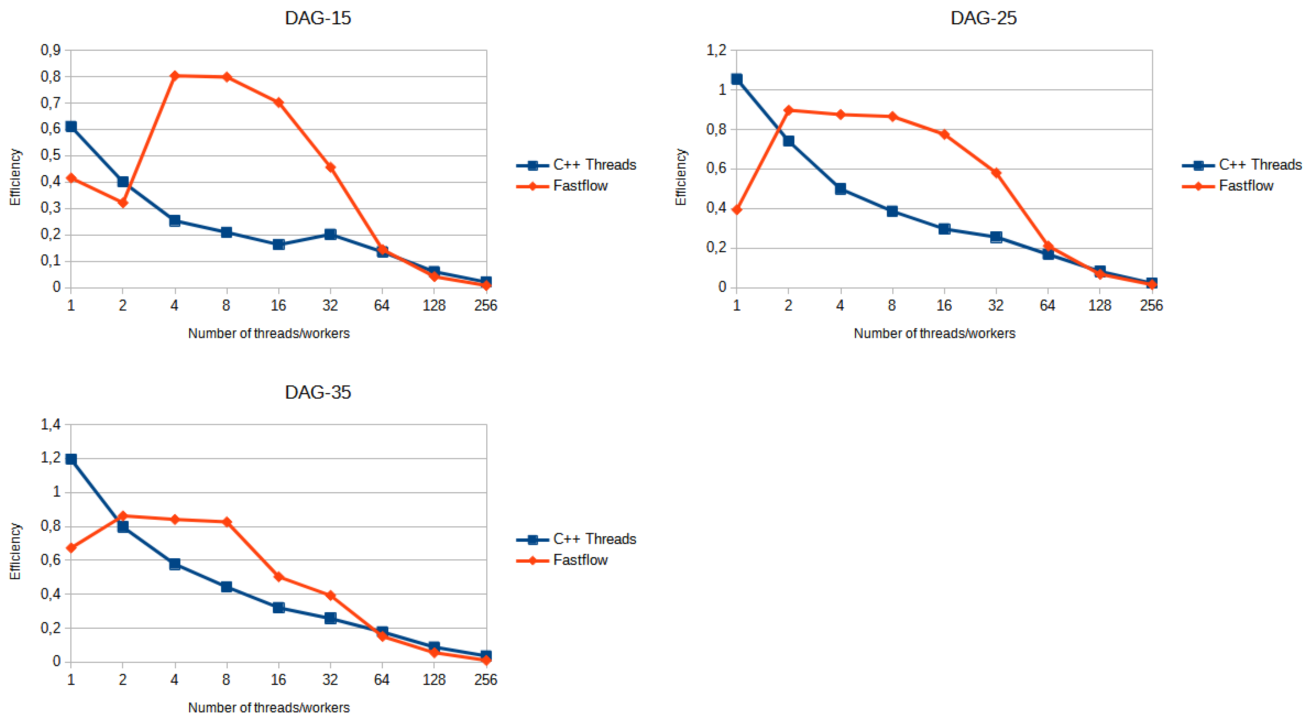

Figure 4: Efficiency achieved.

4 Comparison of C++ threads and FastFlow solutions

The implementation that performed the best is the one that uses the fastflow framework. The main reason is the fact that in the fastflow library the thread management is better optimized compared to the C++ threads implementation and this reduces the overall overhead.

In the C++ threads implementation, each thread must calculate its own range of the current layer. However it turns out that this cost is negligible, since on average each thread calculates its range in less than one microsecond. On the other hand, the overhead due to the thread synchronization phase is significant, since that part of the code is executed sequentially by each thread. This overhead also depends on the size of the next layer, because when a thread enters the synchronization phase, it adds to the next layer the nodes it visited.

On the other hand, in the fastflow implementation, when a worker has explored the node assigned to it, it sends the vector of visited nodes to the emitter which inserts them into the next layer. In this case, the worker is free to continue to explore other nodes assigned to him, while the emitter inserts the nodes of the vector in the next layer. Also, since both the workers and the emitter send pointers, the communication cost is negligible.

This turns out to be a better approach than the C++ threads implementation, where threads compete for the lock to be able to add the visited nodes into the next level and thus resulting in a larger overhead.

Despite these differences, the two implementations have many things in common. Accessing the nodes of the graph and exploring them is done the same way, in fact both implementations use the same class *Node* and *Graph*. Also, the occurrences of X are counted in the same way, using an atomic variable that is incremented whenever a node with value X is explored. Thus, the time required to access, explore a node and visit its neighbors is similar in both implementations.

5 Analysis between expected and actual performance

The ideal speedup could not be achieved in either implementation. In addition, it was chosen to not test the two implementations with much larger graphs for reasons of space (user space limited to a 1 GB on the server). By using larger graphs it might have been possible to make better use of parallelization and get better speedups.

In the C++ threads implementation, as discussed in Section 2, the overhead caused by the synchronization phase proved to be the main obstacle to achieving a significant speedup. This is the reason why, in the graphs shown in Figures 1 2 and 3, as the number of threads grows first the speedup increases and then decreases. This is because the amount of work each thread has to do becomes too small to offset the overhead of having to synchronize the threads before exploring the next layer and and thus there is a decline in speedup.

On the other hand, in the fastflow implementation, this overhead turns out to be less significant than assumed, because as mentioned before, the emitter can insert the nodes visited by the workers into the next layer while the workers can explore other nodes in the current layer. Therefore, this reduces the amount of work that needs to be done sequentially by the emitter (after the workers have finished exploring the current layer) to prepare the next iteration of the BFS search.

An overhead that was not initially taken into account, but which depends on the way the BFS search is implemented, is the time taken to remove duplicate nodes. In both implementations this overhead is not that big in itself, since it consists of checking the value of a variable. If one had used data structures such as sets or unordered sets for representing the layers in order to automatically avoid duplicates, then the time taken to remove duplicates would have been greater (as describe briefly in the Previous attempts subsection).

6 How to build the programs

Programs can be build using the `make` command:

- `make serial` can be used to compile the source files of the serial version of the BFS search into an executable named `BFS_serial.out`.
- `make cpp_threads` can be used to compile the source files of the C++ threads implementation of the BFS search into an executable named `BFS_cpp_threads.out`.
- `make ff` can be used to compile the source files of the fastflow implementation of the BFS search into an executable named `BFS_fastflow.out`.
  - It is possible to specify the path to the fastflow directory by adding `FF_ROOT=path_fastflow_directory` to the make command. By default `FF_ROOT` is set to `./fastflow`.
- `make generate_graph` can be used to compile the source file of the program used to generate DAGs into an executable named `generate_graph`.
- `make all` can be used to compile all the programs.
- `make clean` can be used to delete the executables.

7 How to run the programs

Programs can be run using the following commands:

- The serial version of the BFS search:

```
./BFS_serial.out [starting node] [value] [path to graph file]
```

- The C++ threads implementation of the BFS search:

```
./BFS_cpp_threads.out [num. of threads] [starting node] [value] [path to graph file]
```

- The fastflow implementation of the BFS search:

```
./BFS_fastflow.out [num. of workers] [starting node] [value] [path to graph file]
```

- The program used to generate DAGs:

`./generate_graph.out` (It will generate a file called *graph* inside the current directory.)