Graph Search Project Report

Sara Montemaggi

Introduction

The applications presented in this report were designed to solve the “Graph Search” problem (i.e. return the number of occurances in a graph of the nodes with value X, found during a breadth first search of the graph starting from a certain node S).

In this report are described and compared two parallel versions of the application: the fisrt version has been implemented using plain C++ threads (it will be called from now on *bfs_PAR*) the second version has been implemented using the Fastflow library (it will be called from now on *bfs_FF*). Finally, it has also been written a simple sequential version of the application (from now on called *bfs_SEQ*) to compare the two parallel versions with.

Main design choices

Given a starting vertex S, all the vertices of the graph that can be reached from S must be visited in breadth first order. This means guaranteeing that for all distances *d*, all vertices that are *d* away from S must be visited before any vertex of distance *d+1*.

The graph can be considered in layers: the *d*th layer is the set V_d of vertices that are at distance *d* from S. Breadth first ordering means that all vertices in V_d are visited before any vertex in V_{d+1}. This implies that the layers must be examined serially, but the vertices in a given layer can be processed in parallel.

Next are presented the main characteristics of the two parallel applications *bfs_PAR* and *bfs_FF.*

- In the application *bfs_PAR* two shared data structures are used to represent the current layer to be analyzed and the next layer, that is computed by analyzing the current one and that will have to be analyzed in the next iteration.
  For each layer, each thread accesses its part of the layer, analyzes its part of nodes and updates its part of the next layer data structure. At this point the thread signals that it has finished its part of work for the current iteration and it waits for all other threads to finish. When all threads have finished they can start analyzing the next layer.
- The application *bfs_FF* is designed as a master-worker farm with one thread that plays both the roles of emitter and collector. For each layer the emitter-collector thread

schedules nodes to be analyzed to the worker threads in a round robin fashion, then it waits for partial results from the workers (i.e. subset of nodes of the next layer to be explored) and merges them. Finally, when all the workers have done analyzing their nodes and the next layer has been computed, the emitter-collector starts again scheduling nodes and the cycle is repeated until all layers have been explored.

Expected performances

The sequential implementation of the BFS visit of a graph requires only the sequential exploration of each node, in BFS order. In particular, for each node *u* the sequential program has to check if its value is equal to X, and if it is a counter is incremented. Then, the program iterates over *u*'s neighbors (i.e. iterates over the values stored in a vector), and for each neighbor it checks and possibly changes its color (i.e. checks and possibly changes the value of an integer). Finally, *u*'s color is set to black to signify that *u* has been completely explored. All these are operations that don't require much time.

A parallel implementation of the BFS visit will introduce some overheads. The overheads are due to the creation and the destruction of the threads and to the synchronization mechanisms that have to be implemented to guarantee that the program visits the graph in the correct order and outputs the correct result.

In particular, the parallel versions of the program must explore the layers of the input graph sequentially. For each layer a divide-and-conquer approach can be adopted: divide evenly among threads the nodes of the current layer that is being explored, each thread will then find a subset of nodes of the next layer to be explored, finally, those subsets will have to be merged to construct the complete next layer.

The divide phase and the merge phase that have to be executed sequentially for each layer should represent the main overheads that are introduced in the parallel versions of the application. Beacause of those overheads, the ideal speedup of n when using n threads cannot be obtained and a possible approximation of the obtained speedup is given by the formula (that does not take into account the time to create and destroy threads):

$$\frac{t_{nodes}}{\frac{t_{nodes}}{n} + L \times \left(t_{div} + t_{merge}\right)}$$

Where n is the number of threads, L is the number of layers of the graph, t_{nodes} is the time needed to explore every node sequentially in BFS order, t_{div} is the time needed to divide the nodes in a layer between the threads, t_{merge} is the time needed to merge the partial results returned by the threads.

The fraction of the total execution time that is needed for each divide and merge phase represents the serial fraction of the program and, as stated by the Amdahl law, it limits the maximum speedup that is possible to obtain. Exactly how large this fraction is depends on the implementation. However, considering the fact that the actual exploration of the nodes takes little time, it could be in practice quite large.

Moreover, the effectiveness of parallelization depends also on the structure of the input graph. For example, if many or all layers of the graph are made up by less then n nodes, some threads will only cause overhead and not speed up the actual computation. On the other hand, parallelization will be more effective in graphs that have larger layers, and only in those cases it could be possible to achieve some speedup.

Implementation description and results achieved

In all applications the graph is represented with a set of vectors of integers. Each vector has size equal to the number of nodes. The ID of a node is its corresponding position in the vectors. In particular, one vector is used to store nodes values, one vector is used to store nodes colors (colors are updated when nodes are visited) and one vector is used to store nodes adjacency lists (adjacency lists are also stored as vectors of integers, representing lists of node IDs).

In both *bfs_PAR* and *bfs_FF* it was chosen not to take any lock before visiting a node because doing so would have introduced huge overheads. This has the consequence that some new nodes could be discovered at the same time by different threads and hence duplicates must be taken into account to compute the correct number of occurances of value X. Having to analyze some nodes multiple times also introduces overheads. However, it has been observed that it is rare for a layer to contain duplicate nodes and hence, in practice, the programs performed better without locking nodes.

Description of the application *bfs_PAR*

A layer is the set that contains the node IDs of the nodes that have to be explored during each iteration. The application represents a layer as a vector with size equal to the number of threads. Each position in the vector contains the reference to a vector of integers. In particular, position *i* has a reference to the vector containing the node IDs of the nodes found by thread *i* while exploring the graph, during the previous iteration.

Each thread behaves in the following way:

1. The thread computes a range that defines the part of the data structure representing the current layer that it must access. Ranges computed by different threads are guaranteed to be non-overlapping. Also, ranges are computed in such a way that threads have more or less the same number of nodes to explore.

2. The thread analyzes the nodes contained in the layer part assigned to him. When a node is found with value X, the thread puts its node ID in a private vector. Then, for each analyzed node *u* the thread updates the next layer data structure with the IDs of nodes adjacent to *u* that will have to be explored in the next iteration. Thread *i* will write its nodes IDs in the vector at position *i* of the next layer vector.
3. The thread acquires a lock and increments a shared counter by one, meaning that it has finished its work. Then, the thread checks if the counter is equal to the total number of threads:
    - If it is not it means that not all threads have finished working. In this case the thread releases the lock and waits on a condition variable to be woken up later to start the next iteration.
    - If it is it means that the thread is the last thread to have finished working. In this case it prepares the next iteration (it swaps current and next layer, it clears the next layer, it updates some shared variables that the threads will later have to read). Finally, it notifys all the other waiting threads, releases the lock and starts the next iteration.

    This procedure simulates a barrier which allows to synchronize the threads before starting each layer exploration.
4. When all layers have been explored each thread merges its vector containig IDs of nodes with value X that it has found with a shared vector, that will contain the final result. This last step is done in mutual exclusion and duplicate node IDs will not be inserted. The thread has now finished its execution.

When all threads have finished executing the program outputs the shared vector size.

Since the current layer data structure is only read by the threads there is not need for the threads to access it in mutual exclusion. The next layer data structure is written by all threads, but each thread writes in a different position so, again, there is no need to protect access.

Description of the application *bfs_FF*

In this case, a master-worker farm parallel pattern was used to implement the application. One thread plays both the role of emitter and of collector and the *nw* other threads are called worker threads.

- The emitter-collector thread keeps the list of node IDs of the current layer to be explored in a vector and distributes them to the worker threads in a round robin fashion. For each ID *u* sent to a worker, it receives back a vector containing node IDs of *u*'s neighbors that will have to be explored in the next iteration. The emitter-collector stores these IDs in a vector representing the next layer, guaranteeing also that no duplicate IDs will be stored. When the next layer has been completed this becomes the new current layer and the cycle is repeated until there are no more nodes to explore. When there are no more nodes to explore, the emitter-collector will send and EOS to all workers.

- Each worker thread receives node IDs from the emitter-collector. For each ID *u* received, the worker checks if the node's value is equal to X and if it is it increments an atomic counter, that in the end will contain the final sum. Since there are no duplicates in the IDs sent to the workers, the final sum will be correct. Then, the worker sends back to the emitter a pointer to a vector containig node IDs of *u*'s neighbors that will have to be explored in the next iteration (it will send a reference to an empty vector if it finds no neighbors).

Results

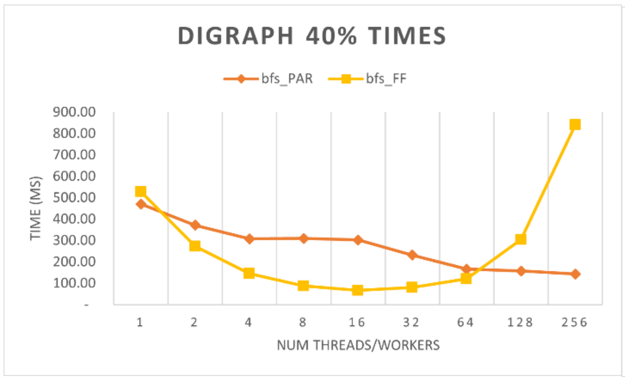
Graphs with different characteristics were generated to test the applications (the source code of the program used to generate the graphs is the file graphgen.cpp). All graphs were generated starting from the same tree structure: a tree with depth 5 and where every node has 15 children. Then, arcs from upper level nodes to lower level nodes were added according to a specified probability. Graphs were generated in this way to guarantee the properties of aciclicity and the one that starting from at least a node (the root) it is possible to visit all other nodes in the graph.

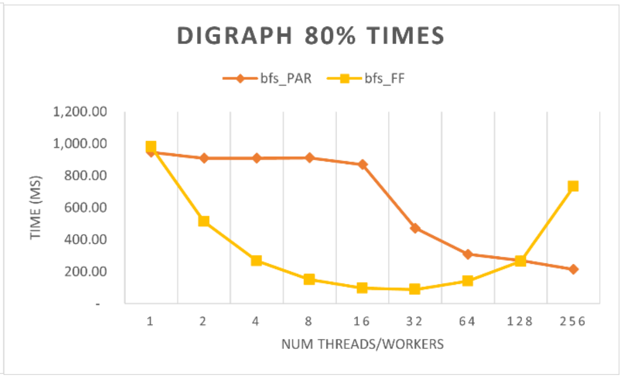
Graphs used to test the applications are: one directed acyclic graph with 0% of additional arcs added (i.e. a tree), one directed acyclic graph with 40% of additional arcs added (called *digraph 40%*), one directed acyclic graph with 80% of additional arcs added (*digraph 80%*) and one directed acyclic graph with 100% of additional arcs added (*digraph 100%*).

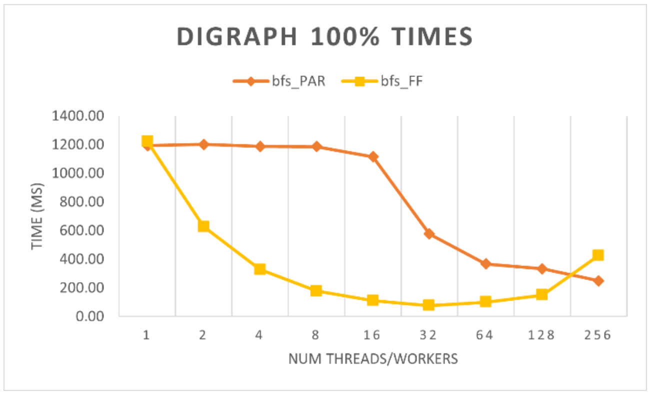
Times reported in the following paragraphs have been obtained by averaging the measurements obtained from 100 different runs.

By testing the applications *bfs_PAR* and *bfs_FF* on the simple tree no speedup was achieved: by increasing the number of threads used by the applications the time to visit the graph and compute the result increased too. This effect is due to the fact that the initial layers of the graph are made up by few nodes (1, 15, 15*15…), therefore, using more than one thread in the initial part of the tree exploration causes only overheads. Since, in this case, the number of nodes in successive layers increases exponentially, some speedup could be achieved only by exploring trees with greater depth.

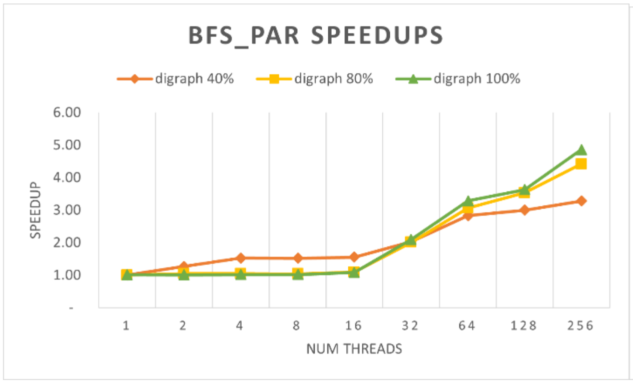
In the graphs below are reported execution times relative to *digraph 40%*, *digraph 80%* and *digraph 100%*. The graphs with a greater number of arcs are the ones that benefit more from prallelization since, in those cases, layers are made up by a greater number of nodes. The graph *digraph 100%* represents an extreme case since it is actually composed by only one layer (excluding the root layer), being the root connected to all other nodes

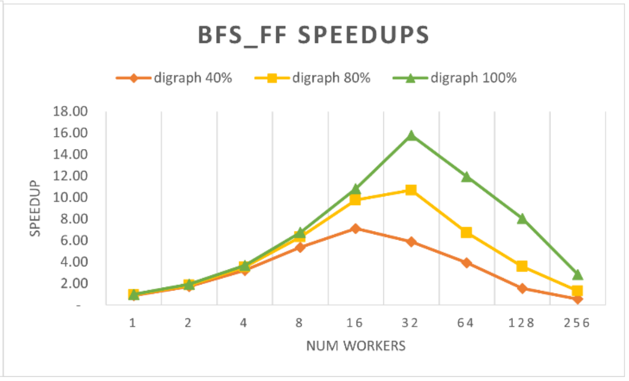






By testing the applications *bfs_PAR* and *bfs_FF* on the graphs with more arcs it was possible to achieve some speedups, as reported in the following graphs.



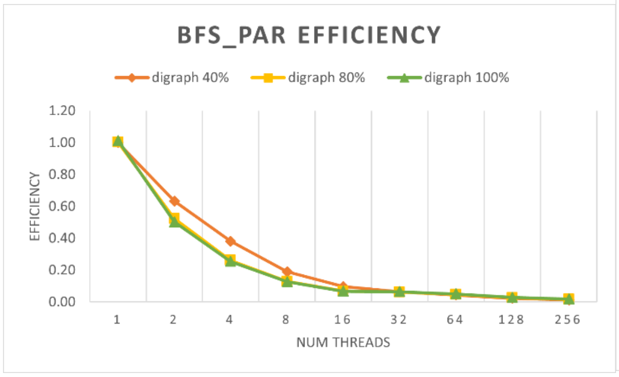


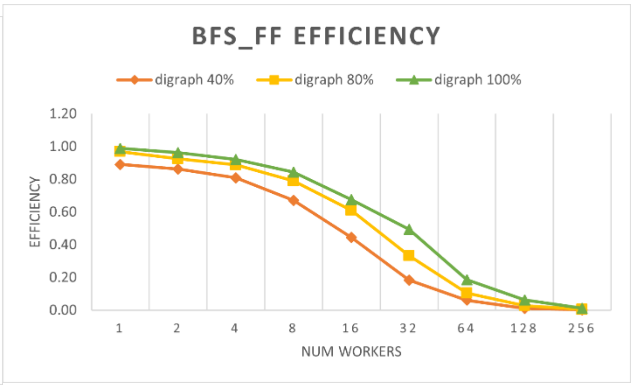
The application *bfs_PAR* is far from achieving an ideal speedup, however, by increasing the number of threads the speedup increases too. The maximum speedup was achieved by using 256 threads to explore *digraph 100%*.

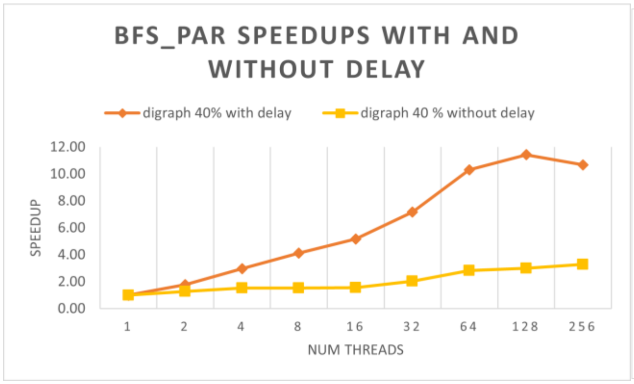
On the other hand, the application *bfs_FF* achieves an almost optimal speedup with 2 and 4 workers, then the speedup keeps increasing but it is further from the optimum value. Finally, the speedup starts decreasing when using more then 32 workers on *digraph 80%* and *digraph 100%*, and when using more then 16 workers on *digraph 40%*. The overall maximum speedup was achieved by using 32 workers to explore *digraph 100%*.

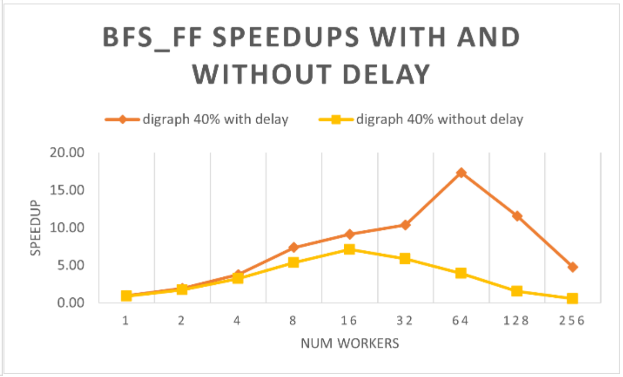
Overall, the application *bfs_FF* achieves greater speedups than *bfs_PAR* when using a number of worker less or equal than 128, hence, *bfs_FF* is able to achieve a better speedup in almost every tested case. Moreover, *bfs_FF* achieves also a greater efficiency than *bfs_PAR* with the same number of workers used. Also, *bfs_FF*'s efficiency decreases more slowly than *bfs_PAR*'s efficiency. Efficiency's values are reported in the below graphs.

Greater speedups could be obtained by exploring graphs with even bigger layers (however, bigger graphs would have required more than 1GB of disk space to be stored so they were not used), or they could be obtained if exploring a node required more work. As an example of this last observation, in the last two graphs the speedup values obtained by testing the applications with *digraph 40%* are compared with the speedup values obtained by exploring the same graph with a delay of 30µs inserted each time a node is explored. As expected, by adding the delay greater speedups were obtained.









Comparison among the implementation and performances of *bfs_PAR* and *bfs_FF*

The application *bfs_FF* is the one which overall achieves better results. This is due to the fact that the FastFlow library is able to better optimize the threads menagement. Moreover, less overheads are introduced in the FastFlow version, as shown in the following paragraphs.

At the start of every layer exploration, in the application *bfs_PAR* each thread has to compute its range before starting to analyze its nodes. This operation took on average 2-3µs. Hence, the operation does not represent in practice a huge overhead, taking into account that exploring a node took on average 20µs and that one thread has usually to explore many nodes (20µs to visit a node is only an approximation since this time varies a lot depending on the number of neighbors of the node). On the other hand, in *bfs_FF* the emitter-collector simply distributes node IDs to the workers in a round robin fashion. As soon as a worker receives the first ID it can start exploring that node. Therefore, it can be assumed that in the latter case less overhead is introduced to divide a layer between the threads.

In both applications, the next layer data structure is updated in parallel with the current layer exploration. In *bfs_PAR* each threads updates its part of the next layer, while, in *bfs_FF* it is the emitter-collector that is in charge of merging the partial results returned to him by the workers. Ideally, results are merged as soon as they arrive, hence also in this case the next layer should be constructed at the same time the current one is explored. Then, when all nodes in the current layer have been explored, both programs perform a sequential phase in which the next iteration is prepared. In *bfs_FF* this requires only swapping two vectors, while, in *bfs_PAR* this phase involves iterating over a vector with size equal to the number of threads , hence the time required increases if more threads are added. In fact, it was observed that this operation took on average 25µs when using only one thread and on average 150µs with 256 threads. Therefore, less overheads are introduced in *bfs_FF* to switch from one iteration to another.

Moreover, in *bfs_PAR* each layer could contain duplicate node IDs, hence some nodes could be explored more times. This does not happen in *bfs_FF*. Although it has been observed that in practice it is rare to have duplicate IDs, those could sometimes cause additional useless work for the application *bfs_PAR*.

Finally, *bfs_PAR* performs a last sequential phase to compute the final output (i.e. the number of occurances of X). During this phase, threads have to access a shared vector in mutual exclusion to insert the IDs of nodes with value X that they have found in it, performing duplicate elimination when necessary. The time required by each thread to perform this operation varied depending on the number of threads used: with less threads each thread took more time to update the final result vector since it had more values to insert in it (because the thread had explored more nodes), with more threads each thread would take less time to update the data structure. Since the updates were done in mutual exclusion, to approximate the total time required to construct the final result, the time required by a single thread should be multiplied by the number of threads. As an example, it was observed that when using one thread only, that thread took on average 250µs to update the final result, while, when using 256 threads each one required on average $5.5\mu s$, that gives approximatly a total of $5.5 * 256 = 1408\mu s$. On the other hand, in *bfs_FF* an atomic counter is simply updated by the workers during the graph exploration, therefore, no final overhead is introduced (here this strategy works because in this case it is guaranteed that workers do not receive duplicate node IDs to explore).

Differences among expected and measured performances

As previously anticipated, some overheads would have to be introduced in the parallel versions of the Graph Search application. Moreover, it was not possible to test the applications on graphs composed by many huge layers due to limited disk space. Hence, it could not be expected to obtain an ideal speedup. However, some overheads that were initially considered turned out to be of minor importance, while other overheads were found to have a greater impact on performances.

In both *bfs_PAR* and *bfs_FF* the t_{div} that was taken into account in the preliminary analysis does not represent a major overhead.

In *bfs_PAR* the next layer is constructed as the current one is explored, so t_{merge} becomes only the time that is required to prepare the data structure and initialize the variables needed to start the next layer exploration. However, this preparation phase introduces an overhead that increases with the number of threads used. A similar reasoning holds for *bfs_FF*, also, in this case the switch between current and next layer exploration requires less time. However, if the workers are faster in exploring nodes and delivering partial results than the emitter-collector is in merging them, the emitter-collector becomes the bottleneck. This is the reason why, in the experiments, by increasing the number of workers the speedup also increases for a while but then it starts decreasing: when an increasing number of workers is used, more results will be ready at the same time for the emitter-collector to process them.

An overhead that was not initially taken into account is the one introduced in *bfs_PAR* to perform the final duplicate elimination in order to compute correctly the program output. When testing the application on the test graphs, this turned out to be the operation that individually required more time to be computed.

Finally, other overheads have been introduced but have not been taken into account in this analysis. For example overheads produced by threads fork and join operations and communication overheads between emitter-collector and farm workers in *bfs_FF*.

Commands to build and execute the programs

Build and execute the BFS Graph Search programs

To compile all the executables to perform the BFS Graph Search go in the directory where the Makefile and the source file are (they should be in the same directory) and use the command:

```
make all
```

This will produce the three executables: bfs_SEQ.out, bfs_PAR.out and bfs_FF.out.

To compile only bfs_SEQ.out type the command:

```
make serial
```

To compile only bfs_PAR.out type the command:

```
make cpp_threads
```

To compile only bfs_FF.out type the command:

```
make fastflow
```

Fastflow headers will be searched by default in the directory ../fastflow/, this can be changed with the make command option FF_ROOT=path_to_fastflow.

To delete all .out files type the command:

```
make clean
```

To execute bfs_SEQ.out type the command:

```
./bfs_SEQ.out [path_to_graph_file] [value_to_count] [start_node]
```

To execute bfs_PAR.out or bfs_FF.out type the command:

```
./bfs_PAR.out [path_to_graph_file] [number_of_threads] [value_to_count] [start_node]
```

or

```
./bfs_FF.out [path_to_graph_file] [number_of_workers] [value_to_count] [start_node]
```

Graphs to test the applications with can be generated with the graph generator program.

Build and execute the graph generator program

To compile the graph generator program type the command:

```
g++ graphgen.cpp -o executable_name -O3
```

Then, to run the graph generator program simply type:

```
./executable_name
```

This will then generate a file called digraph, containing the generated graph, that can be used to test the BFS Graph Search programs.

In graphgen.cpp file the macros MIN_CHILDREN, MAX_CHILDREN, MIN_RANKS, MAX_RANKS and PERCENT can be redefined to generate graphs with different characteristics.