SPM Project Report

D’Onofrio Matteo (561901)

Application project: Graph Search

1.Problem Definition

The problem to be solved is the following: given an acyclic graph *G* = *(V, E)* (such that each node *v* ∊ *V* has a unique associated nodeID, and an associated value *x*), a node value *X* and a source node *S* ∊ *V*, count how many occurrences of *X* are found in the graph during a breadth first search (BFS) on it, starting from *S*.
For the sake of clarity, from now on the test that checks if a node has a node value x = X will be called **TestProcedure.**

2.Design choices

2.1.Main idea

The main property of the BFS is: each *i-th* level in the graph must be explored at all before starting to explore the *(i+1)th* level.

Due to the nature of the problem, the solution can be modeled with the following iterative approach:

- for each *i-th* level it is used a frontier(*i*): a FIFO data structure containing all the nodes to be explored in the *i-th level*
- for each node *a* in the frontier(*i*), all its children nodes are checked, and if it’s found a child node not yet explored:
    - the child is inserted into the frontier(*i+1*)
    - if the TestProcedure on the child is positive, increment a counter
- continue until is obtained an empty frontier (no more nodes to be explored)

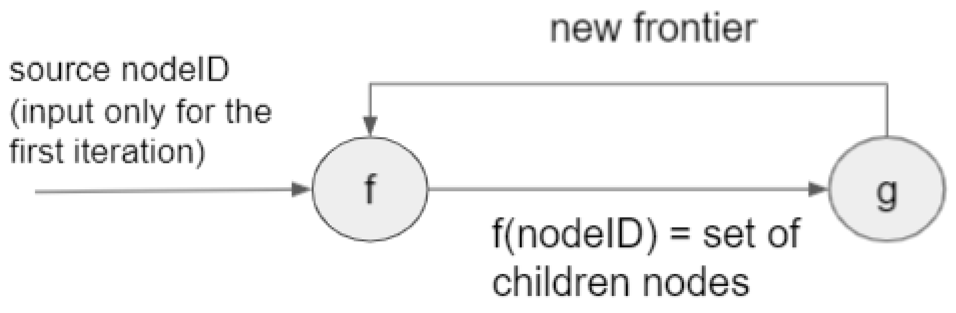
2.2.Chosen pattern

This main idea enable to see this problem as an “iterative” stream-data-parallel problem, because in each iteration we have as input the graph nodes (from the frontier) that are explored by the first stage, and the output of this first stage (children nodes) is the input of a second stage that apply on them the TestProcedure and create the new frontier.
So the first proposed pattern is a pipeline, **in which each iteration** is computed by two main stages:

- f (nodeID) -> {nodeID}
- g (nodeID) -> {nodeID} or {\}

The **f** function implements a stage that receives one node per time, from the frontier of the *i-th* level; then explores all its children nodes, and if finds some new child nodes (not yet explored) sends them to the next stage g.

The **g** function implements a stage that receives one node per time, from the stage f; then g checks the TestProcedure on that node and in positive case increments a counter;
In addition, while the function f explore the frontier(*i*), the g function builds the frontier(*i+1*) by inserting into this new frontier only the nodes received from the stage f, having at least one child node (since if the received node has no children, is not worth to insert it in the frontier). When the *i-th* level of the graph is explored at all, g sends in output the new frontier.
The following image clarify what said before:


new frontier
source nodeID (input only for the first iteration)
f
g
f(nodeID) = set of children nodes

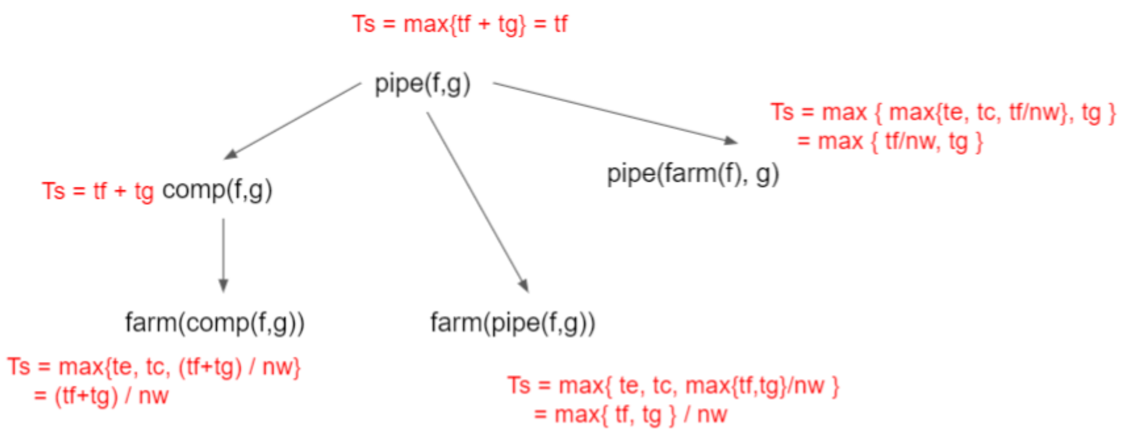
So the data flow is the following:

- initially f receives the source node, explore its children nodes and sends to g all the nodes not already visited
- g receives these nodes one by one, checks the TestProcedure (and eventually increments the counter), then inserts in the new frontier only the nodes having at least one child.
- if the i-th level is not explored at all (f is still receiving nodes from the frontier), g returns nothing.
- when the whole i-th level is explored, g sends out the frontier(i+1), restarting the process.

After this first proposed pattern, a tree of patterns has been created, using some rewrite rules, in order to find the best pattern (the one that minimizes the Service Time and Completion Time).

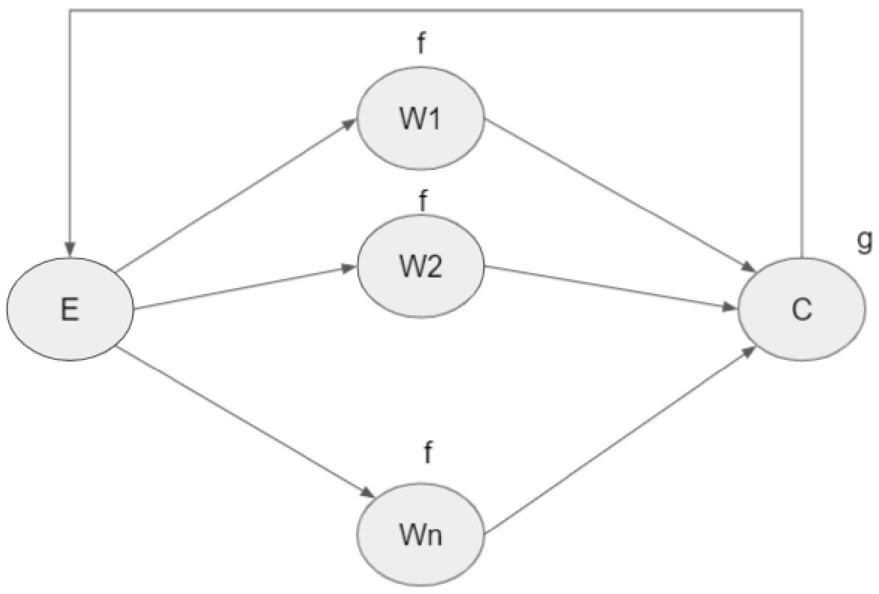
Before creating the tree of patterns, it has been estimated the average time spent from the function **f** and **g**, in order to determine the slowest among them: it turns out that, on average, the f function was around 5 times slower than the g function.

The following image shows the tree of patterns:



After this analysis the pattern chosen was **pipe(farm(f),g)**, because it has one of the lowest costs and its structure fits well the logic of the application's iterations.

The chosen pattern has a Farm whose Collector is a stage computing the g function.
All the Workers compute the f function.
The following image shows the chosen pattern:



This stream-data-parallel pattern will be the same for both the implementations (C++ Threads & FF): in FF the original Emitter will be replaced with a specific one; the original Collector will be replaced with a node computing the g function.

In this stream-parallel-pattern it's present also a state acces pattern which is composed by a binary vector called **visitedNodes**, having the same size of the #nodes in the graph, used in order to track if a node has been visited or not (if a node with nodeID = 2 is visited, visitedNodes[2] is set to 1). It's shared among all

Workers(threads), they continuously access and modify (if needed) the content of the vector.

3. Algorithm implementation

3.1. Sequential version

The sequential version just extracts from the frontier one node per time, explores its children and for each child checks if it has been already visited or not: if the child is not already visited, it's applied the TestProcedure, then the child it's pushed into the frontier.
Since the TestProcedure is done on child nodes before pushing them into the new frontier, necessarily the check for the source node must be done initially, before starting the visit.

input: the source node S, the binary array visitedNodes, the value *X*
output: the number of occurrences of the node value *X* in the graph

```
TestProcedure_on_BFS(S, visitedNodes, X){
    frontier.push(S)
    counter = 0
    if(S.nodeValue==X)
        counter ++
    while(fronteer.size > 0){
        nodeID = frontier.pop
        for_each(child of nodeID){
            if (visitedNodes[child]==0){
                visitedNodes[child]=1
                frontier.push(child)
                if(child.nodeValue==X)
                    counter ++
            }
        }
    return counter
}
```

3.2. Parallel version (C++ Threads)

In the parallel version, in addition to the business logic shown in the sequential version, **is introduced a mechanism that allows to synchronize the Emitter with the Collector** and ensure that a level *i* is explored at all before starting to explore the next level *i+1.*

This mechanism works in the following manner: before an iteration (exploration of the frontier*(i)*) start, the Collector scans the frontier(*i*) and for each node in the frontier counts how many children it has, adding this value to a counter (called **expectedOutput**), in this way the Collector knows how many output have to

receive from all the Workers, in order to declare a whole level explored; then the Collector sends the frontier to the Emitter, after that the Emitter spreads the nodes into the Workers;
Each Worker receive a node and visit all its children:

- if a child wasn't already visited, it's pushed to the Collector
- if a child was already visited, the Worker just send a **-2** value to the Collector

So the Collector each time receives an output (which can be a nodeID to be visited or just a -2) from a Worker, decreases the counter. When the counter reaches 0, a new iteration can start.
Thanks to this, when all the Workers finish exploring a graph level, a new iteration can start directly after the Emitter receives the new frontier.
Finally, when the Collector finds an empty frontier (or a frontier composed by all nodes having no children), sends an EOS to the Emitter; the EOS is then propagated and the application gives in output the number of occurrences found before termination.
In the following are shown respectively the Emitter, Worker and Collector algorithms:

input: c2e is a single queue btw Collector and Emitter, e2w is a vector of queues btw Emitter and Workers, nw is the #Workers

```
Emitter(c2e, e2w, nw, S){
    turn = 0
    nodeID = S
    while(nodeID != EOS){
        e2w[turn].push(nodeID)
        turn = (turn +1 )%nw
        nodeID= c2e.pop()
    }
    send EOS to all Workers
}
```

input: e2w is a queue between Emitter and Workers, w2c is a queue between the Worker and the Collector, the binary array visitedNodes

```
Worker(e2w, w2c, visitedNodes){
    nodeID = e2w.pop()
    while(nodeID != EOS){
        for_each (child of nodeID){
            if(visitedNodes[child]==0){
                visitedNodes[child]=1
                w2c.push(child)
            }
            else{
                w2c.push(-2)
            }
        }
```

```
            nodeID = e2w.pop()
        }
        w2c.push(EOS)
    }
```

input: w2c is a vector of queues between the Workers and the Collector, c2e is a single queue btw Collector and Emitter, S is the source node, *X* is the value for the TestProcedure, nw is the #Workers

```
Collector(w2c, c2e, S, X, nw){
    counter = 0
    frontier = {}
    turn = 0
    expectedOutput = S.#children
    countEOS = 0

    if(S.nodeValue == X)
        counter ++

    while(true){
        nodeID = w2c[turn].pop
        expectedOutput --
        turn = (turn +1 )%nw
        if(nodeID == EOS){
            countEOS ++
            if(countEOS == nw)
                break
            continue
        }
        if (nodeID != -2){
            if(nodeID.nodeValue == X)
                counter ++
            if(nodeID.#children > 0)
                frontier.push(nodeID)
        }
        if(expectedOutput == 0){
            for_each (nodeID in frontier)
                expectedOutput+=nodeID.#children

            if(expectedOutput == 0){
                c2e.push(EOS)
                break
            }

            c2e.push(frontier)
            frontier = {}
            }
        }
        return counter
    }
```

3.3.FastFlow version

For the FastFlow version, the implementation follows exactly the same whole business logic of the C++ Threads version (with also the same stream-parallel-pattern) . The only difference here is that when the Collector find an empty frontier (or a frontier composed by all nodes having no children), in order to terminate correctly the application, send a **-3** value to the Emitter; the Emitter when receives the -3 value, start sending the EOS and terminating the application.

4. implementation details

The graph is stored into two files:

- graph_stored_one_#nodes.txt
- graph_stored_two_#nodes.txt

The first one file contains all the graph's structure, encoded as a list of pairs **(i,j)**, each one defining an edge going from the node with nodeID=**i** to a node with nodeID=**j**

The second file contains a list of node value, each value $x \in (0,20)$.

These files are used by the functions contained in the **build_graph.hpp**, in order to build and fill the graph.

5. Performances evaluation

The experiments have been done on graphs by varying the graph size in this range of nodes [10.000, 50.000, 100.000]; each graph node has a maximum outgoing degree of 50.

In order to avoid confusion only the experiments on 10.000 and 100.000 are shown, since they are the most significant.

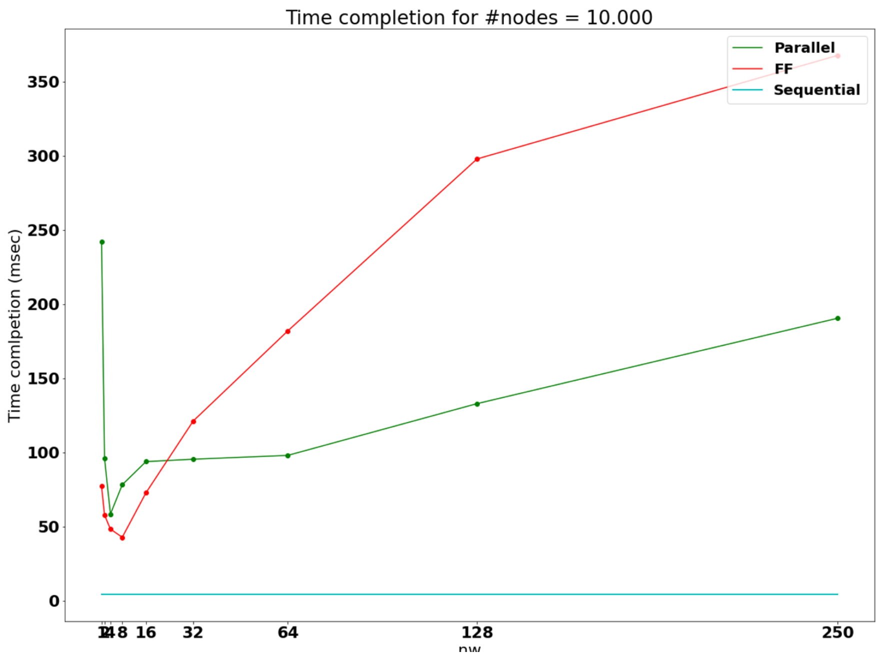
The experiments have been run on the Xeon PHI machine.

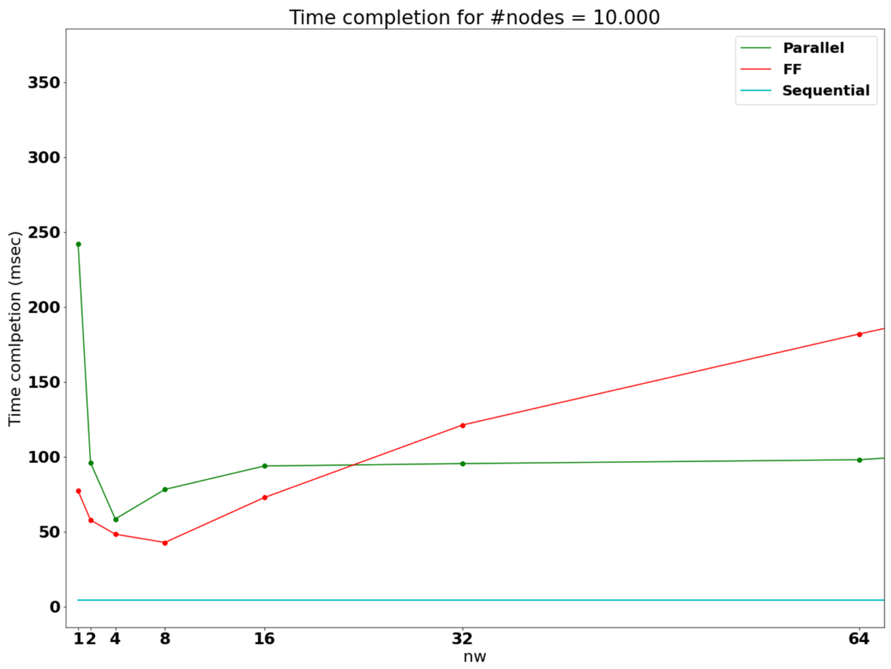
In order to get reliable results, for each graph size was taken a different number of cores in this range [1, 2, 4, 8, 16, 32, 64, 128, 250] and did at least 5 tests for each of them; then was computed the average for that specific pair (graph size - number of cores).

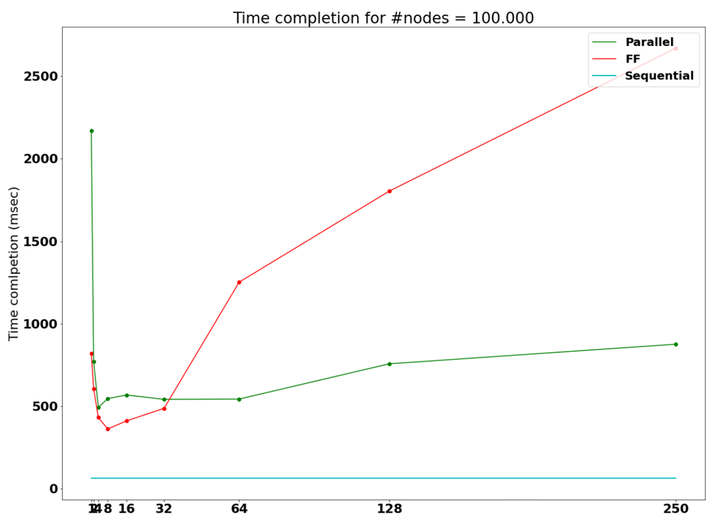
The following images show the four metrics computed:

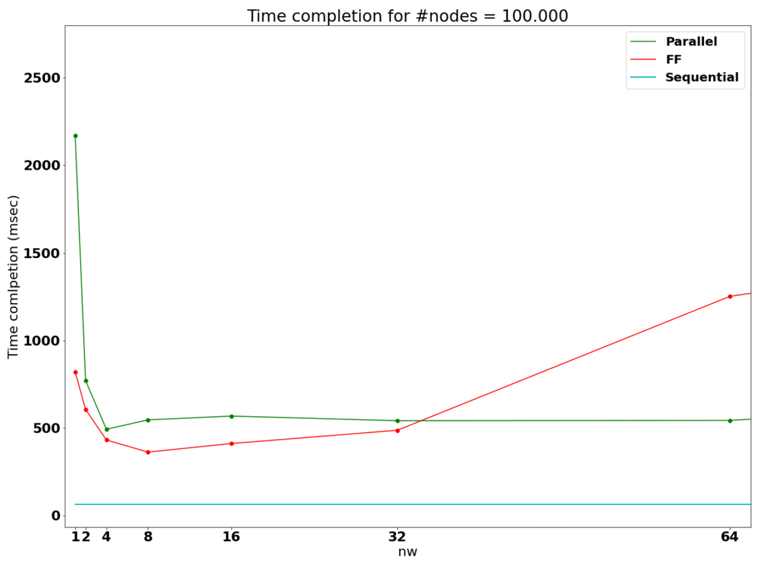
- Completion Time (Tc): the time (msec) spent from application start to application completion
- Speedup (Sp(n)): the ratio between the best known sequential execution time (msec) and the parallel (n) execution time (msec) **(Tseq/Tpar(n))**

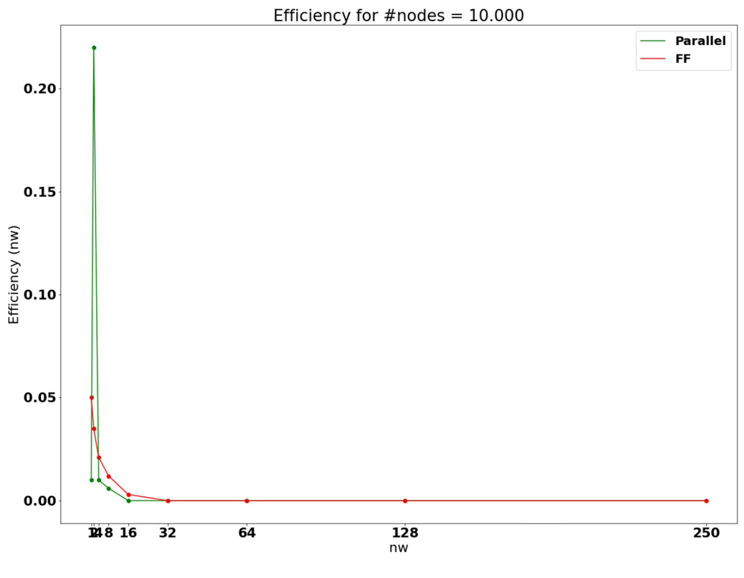
- Scalability (n): the ratio between the parallel execution time with parallelism degree equal to 1 and the parallel execution time with parallelism degree equal to n **(Tpar(1)/Tpar(n))**
- Efficiency(n): the ratio between the ideal execution time and actual execution time **(Tseq/ n * Tpar(n))**

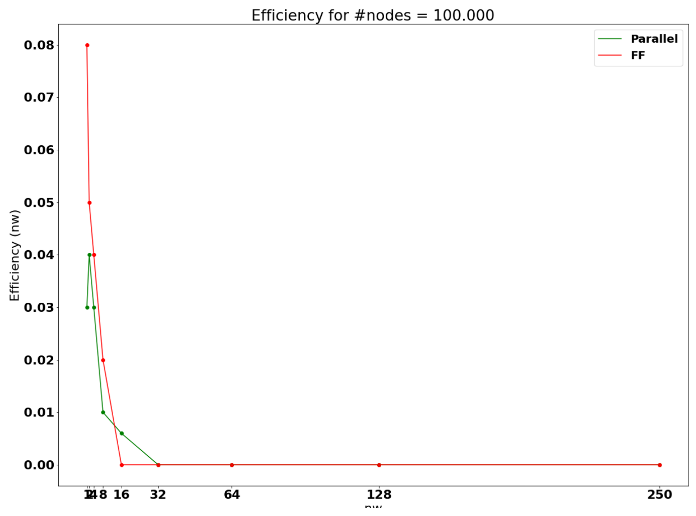


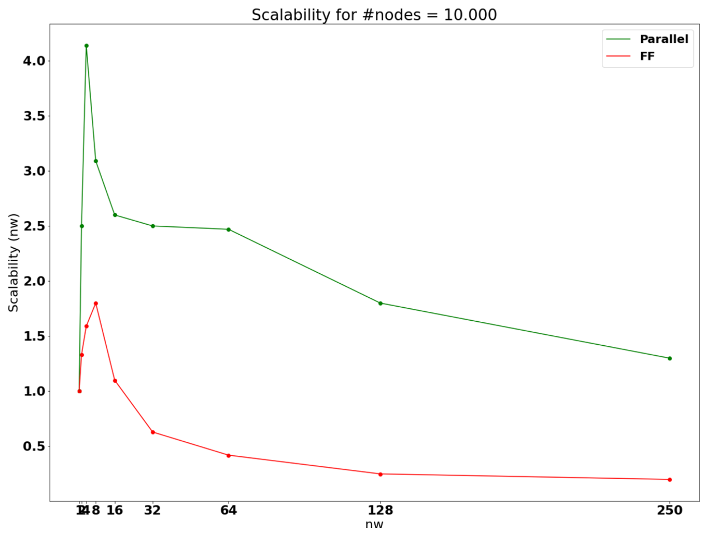


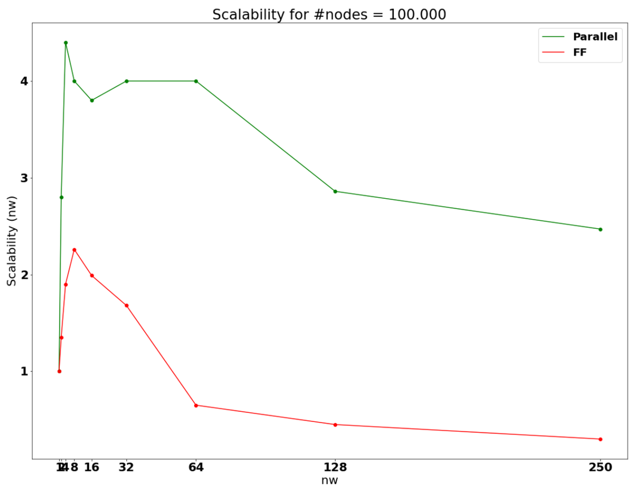

Time completion for #nodes = 100.000
Parallel
FF
Sequential
Time comlpetion (msec)
0
500
1000
1500
2000
2500
1
2
4
8
16
32
64
128
250


Time completion for #nodes = 100.000
Parallel
FF
Sequential
Time comlpetion (msec)
0
500
1000
1500
2000
2500
1
2
4
8
16
32
64
nw


Efficiency for #nodes = 10.000
Parallel
FF
Efficiency (nw)
0.00
0.05
0.10
0.15
0.20
1
2
4
8
16
32
64
128
250
nw


Efficiency for #nodes = 100.000
Parallel
FF
Efficiency (nw)
0.08
0.07
0.06
0.05
0.04
0.03
0.02
0.01
0.00
1
2
4
8
16
32
64
128
250
nw


Scalability for #nodes = 10.000
Parallel
FF
Scalability (nw)
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
1
2
4
8
16
32
64
128
250
nw


Scalability for #nodes = 100.000
Parallel
FF
Scalability (nw)
4
3
2
1
1
2
4
8
16
32
64
128
250
nw

As we can see, the best Completion Time is given from the sequential version, since just by introducing any parallel degree (in both C++ Threads & FastFlow version), the Completion time starts to grow up very quickly.
This behaviour strongly influences the other metrics.
The causes of this results are the following:

- All the queues among stages are blocking queues, using a lock & unlock mechanism each time an element must be pushed/pop; This introduce a large overhead to each node transfer and so to the whole computation
- Due to the necessity of exploring an entire level before starting the subsequent, the synchronization mechanism impose a passive wait of the workers: since the Workers proceed in parallel, some of them can obviously terminate before others, in this case the ones that terminates earlier continue to call a pop on their input queues, but since these queues are empty they must stay in a passive wait, until all the Workers end up their job. So the whole iteration will proceed at the speed of the slowest worker.
- Due to the time spent for the Threads initialization, which is around 100-150 msec per thread; so it becomes non-negligible when the number of threads growing up.
- Finally, the avg time spent by the f and functions for each node, is around 5 msec; the time spent by stages for sending/receiving the nodes through the queues is on avg 80 msec. So Tsend, Treceive >> Tf, Tg: the application spends more time in data transfer than in computation.
  This leads to an increase of the Completion time obtained by enlarging the parallelism degree.

A comparison between C++ Threads version and the FastFlow version, shows that the FastFlow version behaves better in Completion time when the parallelism degree is low, due to optimized handling of the RTS and to the use of non-blocking queues among nodes.
Even here when the parallelism degree grows too much, it starts to take a huge time, due to the overheads.

About the scalability: the C++ Threads version has a better one because the Tpar(1) of the C++ Threads has a completion time much worse than the Tpar(1) of the FastFlow version, enabling better scalability.

About the efficiency, since Tsend, Treceive >> Tf, Tg it’s obtained a very bad efficiency.

6. Commands

To compile the C++ Thread version: **g++ -std=c++17 main_cppT.cpp -o main_cppT.out -pthread -O3**

To compile the FastFlow version: **g++ -std=c++17 -I ${HOME}/fastflow -O3 main_ff.cpp -o main_ff.out -pthread**

To run the C++ Thread version: **./main_cppT.out numWorkers X graph_size sourceNode**

To run the FastFlow version: **./main_ff.out numWorkers X graph_size sourceNode**

NB:

- **passing the arg numWorkers=0 to the C++Thread, run the sequential version**
- **the X value must be in this range (0, ,19)**
- **the sourceNode must be in this range (0, , graph_size-1)**