A Graph Search implementation using C++ threads and FastFlow

By Bolot Kasybekov

April 15, 2022

Contents

1 Introduction

The purpose of this report is to analyze the speedup which can be achieved using the parallelization of the BFS (*Breadth-First Traversal*) search traversal given the randomly generated *Acyclic Directed Graph (G = (V, E))*. Given the value of vertex $s \in V$ which serves as the starting point (*source node*) of *BFS* search, the aim of the project is to count the number of times a specific vertex $v \in V$ was found. The *BFS* search traversal will include the following solutions:

1. *Sequential BFS traversal*
2. *BFS traversal using standard C++ threads*
3. *BFS traversal using FastFlow*

In order to generate a graph *(G = (V, E))*, a Erdős-Rényi model was used which has been implemented in a small Python script. The ultimate goal is to compare the performances of all above-mentioned solutions and analyze their efficiency.

2 Solution

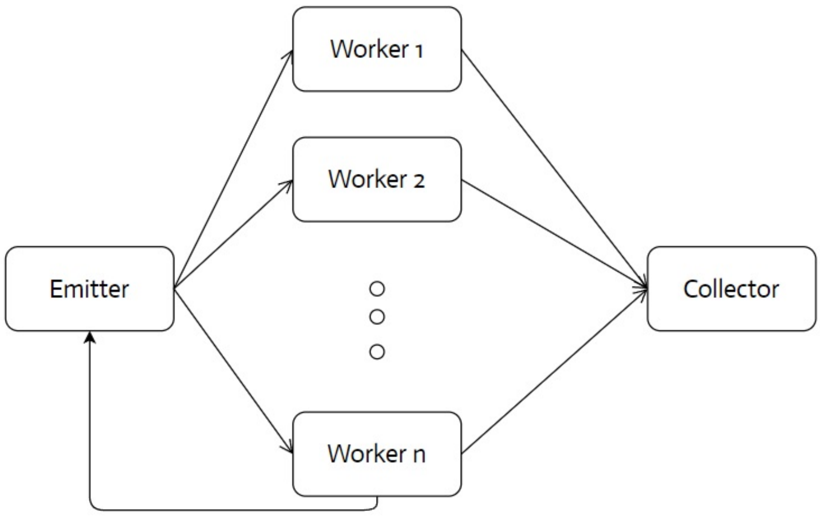
2.1 Sequential BFS traversal

The algorithm of BFS traversal based on the notion of an adjacency list is implemented using following steps:

1. the stl queue (**queue<int> q**) is initialized before traversal begins and initial vertex is pushed into the queue
2. within while loop the front element is popped off **q.pop()**
3. if starting vertex $s \in V$ is poped from queue, then all vertices connected to vertex $s \in V$ in this pass are accessed
4. if the vertex has not yet been visited **visited[n] != 1**, the processing goes further on
5. the vertex is marked as visisted **visited[n] = 1**
6. if vertex **n** equals to $v \in V$, the counter is increased by one
7. the current vertex **n** is pushed to the queue **q.push(n)**
8. new iteration of loop is started

2.2 C++ thread

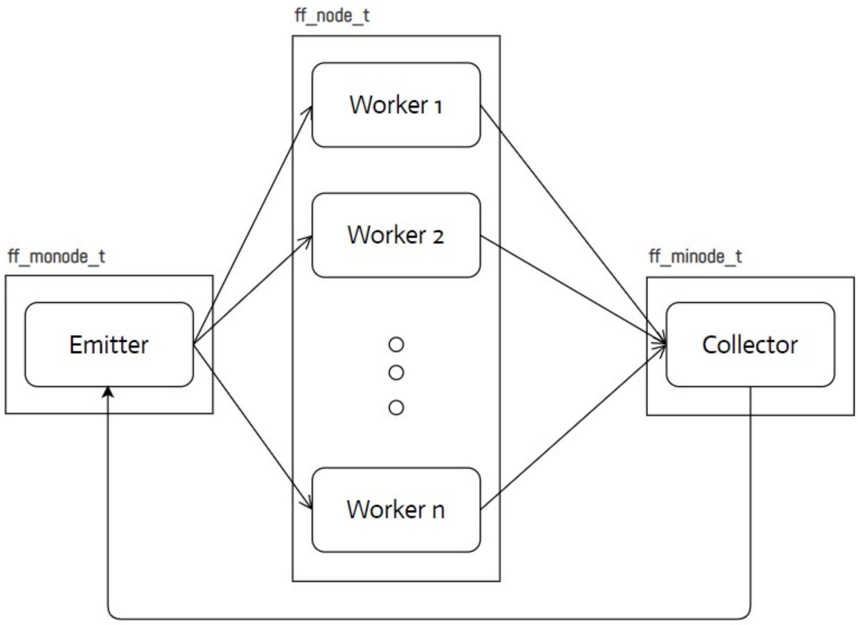
The following algorithm is based on the above-mentioned algorithm of traversing a graph. However, to efficiently increase the parallelization technique *Farm* is used in the context of this solution. There are three main elements used in this solution: *Emitter*, *Worker* and *Collector*.



The load balancing is implemented by using *round-robin scheduling* which means that *emitter* (**queue<int> q**) sends one element of data to each *worker* at a time. Within each worker, the vertex is checked if it has been visited before (**visited[n] != 1**). If it hasn't, it is marked as visited (**visited[n] = 1**). This particular element is pushed to the queue **srcToFarm.push(n)** and returned to the *Emitter* to keep on traversing the graph. Once the traversal is over, **EOS** value is sent to all *Workers* to finish the computation. The *Collector* also gets **EOS** value and keeps on increasing **countEos** variable to the value equal of the number of workers. Once it is reached, the computation is over.

2.3 FastFlow

The FastFlow solution is designed in the very same way as the one implemented in C++ thread. However, there are slight differences which are important to mention and reflected in the design.



As it is shown in the picture above, there are 3 key elements which are added to the solution: *monode*, *node*, *minode*. It means that solution consists of three blocks. *monode* matches the role of *Emitter* and distributes the elements to the *Worker*. *node* matches the role of *Worker* and rettrieves all child elements. *minode* matches the role of *Collector* gathers all child elements and forwards them back to *Emitter*

3 Results

In this section of the report, the results of the solution are discovered to provide a clear picture of the observation. The number of nodes n of the graph $G = (V, E)$ are [10.000, 50.000, 100.000]. The results are generated by runing each solution 5 times to get an average value of the BFS time traversal. The *parallelization degree* is 2^w where $w \in [0, 1, 2, 3, 4, 5, 6, 7, 8]$. The graph density according to Erdős-Rényi model is $p = 0.5$ The expeirement are conducted on the *Xeon PHI* machine. Within this report, following metrics are used to retrieve a necessary information: **speedup**, **scalability** and **efficiency**.

$$speedup(w) = \frac{Tseq}{Tpar(w)}$$

$$\sigma(w) = \frac{Tpar(1)}{Tpar(w)}$$

$$\varepsilon(w) = \frac{Tseq(1)}{w * Tpar(w)}$$

3.1 10k nodes

The starting node $s \in V$ is 0. The node $v \in V$ which is searched is 17. The results of experiment are shown in Table 1

Parallelism degree	Sequential BFS (μs)	C++ Thread (μs)	FastFlow (μs)
1	133	231	152
2	134	127	88
4	134	60	55
8	134	43	38
16	133	50	33
32	133	50	56
64	134	110	95
128	133	130	224
256	134	195	869

Table 1: BFS traversal on 10k nodes

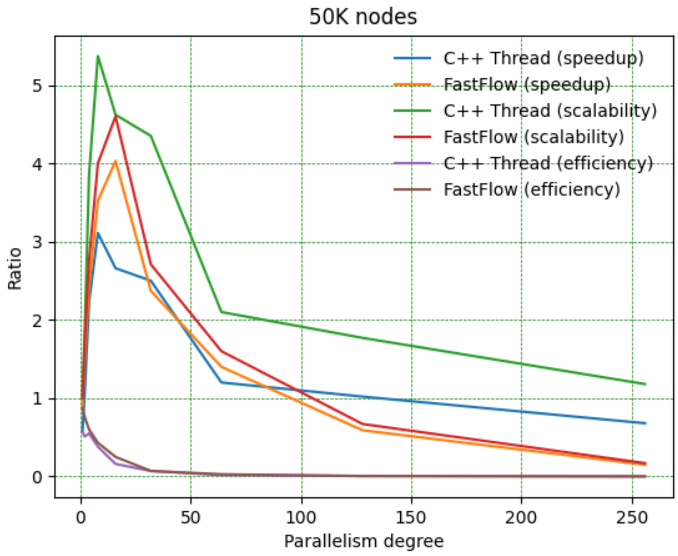


Figure 1: Metrics based on C++ thread and FastFlow

3.2 50k nodes

Parallelism degree	Sequential BFS (μs)	C++ Thread (μs)	FastFlow (μs)
1	1002	1620	1061
2	1003	871	625
4	1003	480	514
8	1003	261	515
16	1002	263	215
32	1003	225	215
64	1003	201	205
128	1002	361	336
256	1003	427	762

Table 2: BFS traversal on 50k nodes

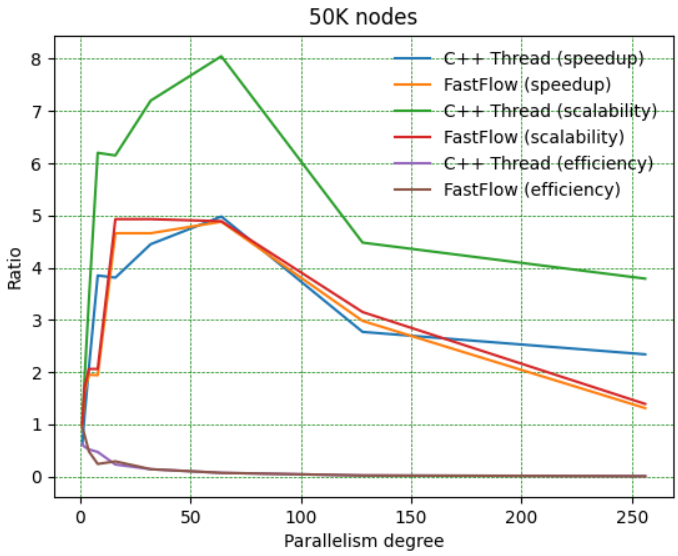


Figure 2: Metrics based on C++ thread and FastFlow

3.3 100k nodes

Parallelism degree	Sequential BFS (μs)	C++ Thread (μs)	FastFlow (μs)
1	2258	3601	2382
2	2258	1955	1382
4	2260	1040	1150
8	2259	747	1136
16	2258	489	1120
32	2261	382	680
64	2259	359	390
128	2260	466	740
256	2260	646	1271

Table 3: BFS traversal on 100k nodes

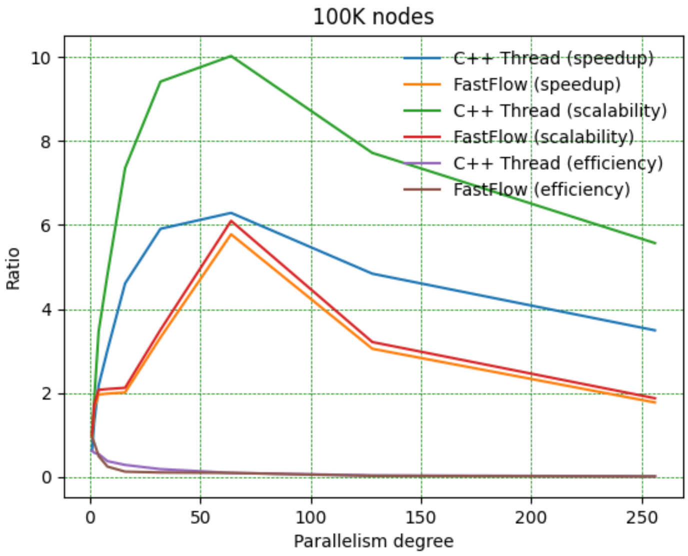


Figure 3: Metrics based on C++ thread and FastFlow

4 Overhead Analysis

As it can be observed from Tables 1, 2 and 3 above and Figures 1, 2 and 3, **speedup** and **scalability** drop as the **parallelism degree** increases. There are multiplicity of reasons:

- Due to the fact that the *Emitter* implements static *round-robin scheduling* policy, the traversal might take a longer time to complete in case of the particular node having many children while other other threads have to stay idle. This, in turn, contributes to longer **Completion Time**

- In both *C++ threads* and *FastFlow* implementations, *synchronization* policies used as **unique_lock** and **lock_guard** allow to accept one element at a time. As the number of threads grow exponentially, the coordination among threads using *synchronization* also increases exponentially. Which, in turn, impacts overall **Completion Time**

- As each thread alone without any code implementation and synchronization require a time for its initialization, the increasing number of threads will increase the overall time of execution. As it has been tested without any code implementation and synchronization, on average, each thread require **7 milliseconds**. For instance, the code with

parallelism degree of 256 requires approximately **36 milliseconds** to be initialized.

- Both solution implement **vectorization** . It means as the **parallelism degree** grows exponentialy, the **overhead** grows alongside. As it is shown on figures 1, 2, 3, the overall *speedup* and *efficiency* fall dramatically with higher number of **parallelism degree** .

5 Instructions

In the root folder **Makefile** is setup for compiling.
Instructions for compiling:

- **make GraphFF** - compile *FastFlow* implementation
- **make Graph** - compile *C++ thread* implementation

Instructions for running:

- *./executable_name [text_file] [start_node] [search_node] [workers_num]*
- *executable_name* - graph_search_ff or graph_search
- *text_file* - the file names are: random_10000.txt, random_50000.txt, random_100000.txt
- *start_node* - the node value from which *BFS search* starts
- *search_node* - the node value which is searched by *BFS*
- *workers_num* - the number of workers