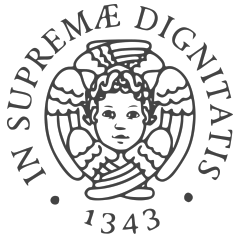
Design and implementation of parallel breadth-first search

Project Report

Academic year 2020-2021
Submitted for the examination of Parallel and distributed systems: paradigms and models

By

Giuseppe Grieco
g.grieco6@studenti.unipi.it



Department of Computer Science
University of Pisa, Italy

July, 2021

Contents

1 Introduction

The bread-first search is an algorithm visiting a graph in amplitude. It starts from a node, often called the root or source node, and continues visiting all its descendants level by level, whereas the i-th level will contain all the nodes at a distance i from the root. For each node visited the algorithm checks its label to count the occurrences of the target label. The assumption underlying all the work is that the input to the algorithm is a direct and acyclic graph. In addition, for the sake of clarity, the notation used is summarized below:

1.1 Notation

Let $\mathcal{G} = (V, E)$, $|V| = n$ is the number of node and $|E| = m$ is the number of edges. For all the node $v \in V$:

- $\mathcal{N}(v) = \{u : (u, v) \in E\}$ is the neighborhood of v;
- $k_{in}(v) = |\{e : e = (u, v) \in E\}|$ is the in-degree of v;
- $k_{out}(v) = |\mathcal{N}(v)|$ is the out-degree of v;
- $k(v) = k_{in}(v) + k_{out}(v)$ is the degree of v.
- $d(u, v)$ is the distance from u to v, i.e. the minimum path from u to v.
- $C(v)$ is the clustering coefficient, i.e.e $\frac{2 \cdot L_i}{k_i \cdot (k_i - 1)}$, where L_i represents the number of edges between the k_i neighbors of node i.

Using the node-focus notation above, it is possible to define general properties for the graph:

- $\bar{k}(G)$ is the average degree;
- $\bar{d}(G)$ is the average distance;
- $\bar{h}(G)$ is the diameter, hence the maximum distance among any two nodes.

In addition, we can define the graph inducted by a node v as the subgraph $G' = (V', E')$ containing all the nodes reachable from v.

1.2 Preliminary analysis

1.2.1 Data structure

Before entering in the algorithmic details let's first introduce the data structure used. There are many ways to represent a graph; among these the main ones are the adjacency list and the adjacency matrix. The choices among them is mainly a matter of the usage of the adjacency information and the expected nature of the graph. As for every node v of the graph, induced by the root, it will be necessary to go through each node $u \in \mathcal{N}(v)$, the adjacency list is way more efficient since for each node v the listing of $\mathcal{N}(v)$ takes a time proportional to $k_{out}(v)$, which is thus optimal. Moreover, if the expected input of the algorithm are "real" graphs, since they are very sparse, the representation as a adjacency list is way more efficient in terms of space complexity.
In particular, the solution implements a graph G as a vector of nodes. The node is a pair, the first component is the label of the node, while the second component is the adjacency list containing the indices of the nodes in the graph. In addition to mark each node as visited, when needed, a vector of boolean is used taking a space complexity of $O(log_2 n)$, this vector is called in the report *vector of visits*.

1.2.2 Sequential version

The sequential version is a straightforward implementation of the problem description. It uses two vectors F_i and F_{i+1}: the former is the frontier while the latter is used to add new nodes and therefore represents the frontier to be used in the next iteration. At the beginning, the algorithm initializes the vector of visits, the frontier with the neighborhood of the source node and marks each child as visited. Then it marks the source node as visited and updates occurrences if needed (i.e. if the root has the searched label). After this first phase, the algorithm starts the first loop which iterates on each level checking that F_i is not empty. Subsequently, for each level, it goes through each node $v \in F_i$, updates occurrences if needed and then goes through each node $u \in \mathcal{N}(v)$. For each node u in the neighborhood, if the node u is not marked as visited, it adds u to F_{i+1}. After having exhausted all the nodes of F_i, the algorithm swaps F_i with F_{i+1} and then clears F_{i+1}.

2 Proposed solution

Before going into the details of the solution, let's describe the graphs used for the analysis. The dataset used is a set of synthetic graph generated using the Erdos-Renyi model. Therefore, the graph generates $ER(n, p)$ is parametric in p and n, whereas n is the number of node and, p can be interpreted formally as the probability of attaching a node regardless the previous realizations. On the another hand it can be seen as the density of the graph. In particular the set used by the experiments is composed by four different graphs: $ER(10000, 0.2), ER(10000, 0.4), ER(10000, 0.8), ER(100000, 0.002)$. The number of nodes was chosen as a tradeoff on a reasonable size, to evaluate the goodness of the algorithm and the cost in terms of memory and space required, since the space and time complexity are $O(n^2)$ which are not negligible[1].

2.1 Problem Analysis

2.1.1 Completion time

The sequential algorithm described in section .1.2.2 is an iterative algorithm with an initialization phase where the i-th iteration visits level i. The processing of the i-th level requires to go through each node $v : d(v, s) = i$ and visit its neighborhood. The time needed to process a node is independent of the level at which it is visited as it only depends on the size of its neighborhood. Hence, let the time taken by the node v be denoted with T_v. Then the time taken by the level i is $T_{L_i} = \sum_{v:d(v,s)=i} T_v$. Note that different levels require different times depending on the topology of the graph. In particular, the i-th level is influenced by the number of nodes at distance i from the root and by the size of the neighborhood of these nodes. The completion time of the i-th iteration is given by $T_i = T_{L_i} + T_{swap} + T_{clear}$, i.e. the time taken by the level i plus two additional factor: the former, T_{swap}, is the time taken to swap L and $\hat{L}$, which is the same for all the iterations since it is a swap of two pointers; the latter, T_{clear}, is time taken by the cleaning of the new $\hat{L}$ so it depends on number of elements. Let G be the graph induced by the root node then the completion time for the sequential algorithm is:

$$T_{seq}(G) = T_{init} + \sum_{i=1}^{\bar{d}(G)} T_i \leq T_{init} + (\bar{d}(G) \cdot max_i\ T_i)$$

[1] for low density a better algorithm were used provided in the networkx library, that were proposed by Vladimir Batagelj and Ulrik Brandes[1]

Where T_{init} is the time taken by the initialization phase, therefore it takes into account: the initialization of the vector of visits, the initialization of F_i and F_{i+1}, the analysis of the root node, the insertion of its neighbors in the frontier and the creation of the vector of visits.

There mainly are three kind of possible parallelism:

1. The frontier-level: which divides the work required by the computation of the single frontier among the workers;
2. The neighborhood-level: which divides the work required by the single node computation, hence the visit of its neighbors among the workers;
3. A combination of both 1 and 2.

The analysis is focused on the first approach since the second one is very sensitive to the local topology of a single node. Moreover, in real scenarios $\bar{k}$ is a very small value, which causes small units of work. In addition, still considering real scenarios, $\bar{d}$ is a very small number and n is a big value, whereby the frontiers will contain a very large number of nodes.

Ideally, considering nw workers one could think that the analysis of a single frontier L_i can be divided equally among the workers, obtaining a parallel completion time for a single iteration $T_{T_i}^{par} \approx \frac{T_{L_i}}{nw} + T_{swap} + T_{clear}$. However, this does not take into account that the analysis of L_{i+1} can start only when the analysis of L_i is terminated, which causes an additional time factor to synchronize all the workers. The parallel completion time for a single iteration becomes then: $T_{T_i}^{par} \approx \frac{T_{L_i}}{nw} + T_{sync} + T_{swap} + T_{clear}$. Hence, there are three tasks that can not be done concurrently, as they are in critical section:

1. Checking if a node has already been visited and, in the case it is necessary, updating the vector of visits;
2. Adding the nodes found to L_{i+1};
3. Update the number of occurrences.

Lets denote the additional factors listed above with: $T_{visited}$ due to 1, T_{merge} due to 2 and T_{update} due to 3. The first two are required at each iteration, the last one can be added directly once to T^{par} since each workers can keep a counter of the found occurrences and at the end sum it. The completion time of single iteration in parallel can be better approximated by: $T_{T_i}^{par} \approx \frac{T_{L_i}}{nw} + T_{sync} + T_{visited} + T_{merge} + T_{swap} + T_{clear}$, resulting in the following parallel completion time:

$$T^{par} \approx T_{init} + T_{update} + \sum_{i=1}^{\bar{d}(G)} T_{T_i}^{par}$$

Of course the above approximation considers the workload as perfectly balanced, that in principle is not easy to achieve, since it is not only a matter of the number of nodes (i.e. it is not enough to assign equal-size subset of the frontier to each worker). On the other hand, the completion time $T_{S \subseteq L_i}$ is influenced by the completion time of each node $v \in S$, namely T_v. A good scenario at iteration i, not necessary the best, is the one where the number of nodes at level L_{i+1} is equally distributed among the neighbors of nodes contained in L_i.

2.1.2 Sequential analysis

In order to design a proper parallel solution, the times of the various operations and phases that the sequential version requires were measured, to identify where there are, any bottlenecks. The expected ones are the memory accesses, more in details:

- The time to access an element of the graph contained in the frontier, $T_{read(v[i] \in G)}$, that is random and unpredictable, since it only depends on the topology of the graph. One could imagine that a possible improvement is to sort the frontier, this probably causes an overhead that exceeds the gain, however, more considerations require an additional analysis that this work does not take into account.
- The same reasoning on accessing graph elements also applies to the time to access the vector of visits, $T_{visited[v]}$, where, however, the access order is influenced by the organization of the neighborhood of the nodes in L_i.

The accesses to the current frontier L_i and the neighborhood $\mathcal{N}(v)$ are among the memory access the one made more efficiently, since they are a scan from the first to the last elements of the vector, thus optimal in number of I/Os. Some of the measures to support this are presented in Table 2.1.2[2], note the time taken by $T_{read(v \in \mathcal{N}(v))}, T_{read(visited[v])}, T_{write(visited[v])}$ were not measured since the sum togheter took at most $\approx 5ns$.

	p		
	0.2	0.4	0.8
T_{clear}	$\approx 164ns$	$\approx 176ns$	$\approx 167ns$
T_{swap}	$\approx 154ns$	$\approx 168ns$	$\approx 163ns$
$T_{read(i \in F_i)}$	$\approx 160ns$	$\approx 164ns$	$\approx 166ns$
$T_{read(v[i] \in G)}$	$\approx 1874ns$	$\approx 2837ns$	$\approx 4233ns$

Table 1: Sequential measurements

As can be observed, as the density increase the time taken by the I/O increases. This happens because the bigger is the vector to read the higher is the number of pages it takes in the upper-level cache, i.e. higher is the number of access to those pages for a scan. The sequential completion times measured is shown in Table 2.1.2 as average on 10 runs.

	p		
	0.2	0.4	0.8
T_{seq}	$94862586ns$	$126302631ns$	$243904063ns$
$\sigma(T_{seq})$	$18246006ns$	$422329ns$	$548079ns$

Table 2: Sequential results

2.2 Solution Description

In order to prevent the load balancing issues due to the variance of k_{out}, the proposed solution uses a dynamic scheduling, which can be easily implemented in a Master-Worker fashion. In

[2]The results are obtained as an average over 10 runs

the present case, an auto-scheduling policy is implemented: all the workers have access to the frontier and obtain a new task of work using a shared data structure. Retrieving a task has the cost of an atomic fetch+add on an integer. Here a task correspond to a chunk, i.e. start and end indexes of F_i, note that the size of the chunk cw, is a parameter of the solution. Thanks to this shared data structure the master does not need to prepare the tasks for the workers. Instead it performs the same work as the worker with the difference that is also responsible for the swap and the cleaning of the new F_{i+1}.

2.2.1 Local next frontier

The trivial way to produce F_{i+1} is to share an array among the workers and insert in mutually exclusive, each new element found to the array. This solution is as easy as inefficient, since it produces different problems. First of all, false sharing occurs more frequently and the overhead due the atomicity is really high. In the proposed solution each worker has a local version of F_i, which contains only the nodes found by the worker at the end of the visit. The worker atomically adds all the elements of the local frontier to the global one. This reduces the number of false sharing and decreases the overhead, since the processing is much faster.

2.2.2 Expected performance

The solution mentioned above mitigates, thanks to dynamic scheduling, the problems due to the variance of k_{out}. However, it does not eliminate the problem in its entirely, because it remains a parallelism only at the frontier level. The worst case is when an huge hub occurs, since the worker who has to process it will take much longer. Instead, a good scenario for the algorithm is the one in which fairly large frontiers are generated.

To discuss with more specificity the expected performance, it is necessary to evaluate it with respect to the topology and statistical properties of the input graph:

- In general, the lower the value of $\bar{d}$ is, the higher the expected gain of the parallel version, because the overhead due to the level-synchronization will be the smaller and, the number of nodes in the frontiers will be higher, since n should be divided among few frontiers.
- In addition, fixing $\bar{d}$ and enlarging n increase the probably that some levels will contain a number of nodes high enough to achieve a good speedup increase.
- In addition to the previous property, if the k_{out} is considered, it is possible to better estimate how the frontiers will be made. Namely, as the variance in k_{out} is low, the confidence in the fact the nodes between the frontiers will be better distributed increases. It is not true in general, because actually it depends on which node are presents in the different neighborhood: the fewer are the duplicates among the nodes neighborhood the better distributed the nodes will be across the frontiers.
- Instead, the density itself does not say too much about about the distribution of the nodes among the levels. This is not always true: in fact if we take specific cases like the synthetic dataset generated through the Erdos-Renyi, this is false. This happen because increasing the value of p, i.e. the density, the algorithm tends to generate a small number of levels due to independency in drawing random edges: for small value of p the probability of finding node in higher level increases.

For example in real graphs is known that n is a big number and $\bar{d}$ is really small number. Even though, in real graphs often hub occurs and this generates an high variance in k_{out} and increments the load balancing issue.

2.2.3 The influence of chunksize

As mentioned before the proposed solution is parametric in cw, namely the chunksize, this measure refers to the number of nodes of F_i that each workers will consider as a single task. The pop operation is as written in the previous sections with an efficient data structure that implements it at the cost of an atomic `fetch-add`. The choice of cw value should be a trade-off between the advantage of having small tasking and the time taken by the pop operation. Small chunk size guarantees a more fairly distribution among the workers, as cons it increments the number of pop, hence the overhead. Moreover if the expected variance in k_{out} is high the advantage of having small chunk increase.

2.3 Analysis and measurements

All the experiments have been performed fixing cw as 32, this number has been chosed according to what is wrote in section 2.2.3 observing the overhead better described in section 2.3.3.

2.3.1 Build and execution

The source code have been complied with `g++17` using the commands:

- `g++ -std=c++17 ./src/bfs-sequential.cpp -o ./build/bfs-sequential -O3 -Wall -I include/ -ftree-vectorize`
- `g++ -std=c++17 ./src/bfs-pthread.cpp -o ./build/bfs-pthread -O3 -Wall -pthread -I include/ -ftree-vectorize`
- `g++ -std=c++17 ./src/bfs-fastflow.cpp -o ./build/bfs-fastflow -O3 -Wall -pthread -I include/ -ftree-vectorize`

The execution of the sequential code requires 3 positional arguments:

1. `inputFile` : string, the path to the graph
2. `startingNodeId` : integer, the id of the from which the bfs will start
3. `labelTarget` : integer, label whose occurrences are to be counted

The parallel versions requires 2 additional positional arguments:

1. `nw` : integer, the number of workers to use
2. `k`: integer, chunk size

All the version print by default the completion time.

2.3.2 pthread implementation

In the `pthread` implementation, the main thread acts as master and so, it executes the initialization phase, creates the threads and then starts its work as described. As soon as the visit of the graph is completed the master collects the result and prints it together with the completion time. The synchronization at the end of each level, among the workers and the master, is implemented using an active wait barrier implemented through mutex and conditional variable.
To handle the critical section in the check and update of the visited nodes a vector of atomic booleans were used as the vector of visits. The check and update was accomplished via an

`compare and exchange` instruction, which were done at the node level in the frontier, while at the neighborhood level (the one implemented in the sequential version) to avoid inserting duplicates a non-atomic vector of boolean were used, namely the *vector of insertion*. Since the vector of insertion is composed by non-atomic boolean this does not guarantee that duplicates will not be added, but the checks at node level guarantee that the node will not be visited more than once. This were made, since as vector of insertion generates few duplicates in the frontier the checks at node level will have often the same result, hence this helps to exploiting better the pipeline of the new processor through the branch prediction. The probability of adding duplicates grows as the $C(v)$ of the nodes v in the previous frontier increase. A cons of the introduction of this vector is that it increases the probability that false sharing occurs.

2.3.3 Overhead analysis

To better evaluate the performance as the number of threads increases, it is useful to observe the trend that the overheads show. With this type of analysis it is also possible to make a choice on the effective number of threads to use in case the nature (e.g. using the approximation of the completion time provided in section 2.1.1), intended as topology and statistical properties, of the graph is known.

2.3.4 Fastflow implementation

In the `FastFlow` implementation implements the same mechanisms as in the `pthread` version, except for the synchronization mechanisms, it uses the Master-Worker with feedback queue. The master after having performed the initialization phase and indicated to the threads which is the pointer to the queue where to write the nodes they find, performs the same work of the workers and orchestrates the level-synchronization. The level-synchronization is implemented throw the task queues and feedback queue, that were use to notify the master with the number of occurrences found in the level. Whenever the master receives all the feedbacks from all the workers, swap the pointers of L_i and L_{i+1} and informs the threads to work on the next level by indicating the new pointers pushing them on the task queues of each worker. Initially to starts the worker the master push the initial pointers on the task queues.

2.3.5 Results

The following section reports all the results obtained by both the `pthread` and the `FastFlow` implementation. All the synthetic dataset have been exploited in order to show empirically what have been discussed until now. The results have been shown plotting the performance in terms of completion time and speedup reached. The results obtained by the `FastFlow` version are worse in general than the `pthread` implementation, the gap with increasing number of threads shows an increasing trend. This gap is probably due to the fact that the `pthread` version can be more specific and implement the bare minimum in a more efficient way for the specific problem, especially in the synchronization mechanisms.
As discussed earlier as the number of nodes in the frontiers increases, an increase in speedup is expected. In the Erdos-Renyi model fixed the number of nodes n, as p increases we obtain graphs with a smaller number of frontiers, i.e. $\bar{d}$ smaller, and generating very large frontiers. This is well highlighted comparing the different plots in Figures 1, 2 and 3. Interestingly, as the number of nodes in the frontiers increases and, the number of frontiers decrease (because n is fixed), the gap between the performances of the two versions shows a decreasing trend.

Moreover, as mentioned in section 2.2.2 it is not only a matter of density, instead different factors must be taken into account, such as the number of nodes and $\bar{d}$. This is clearly looking

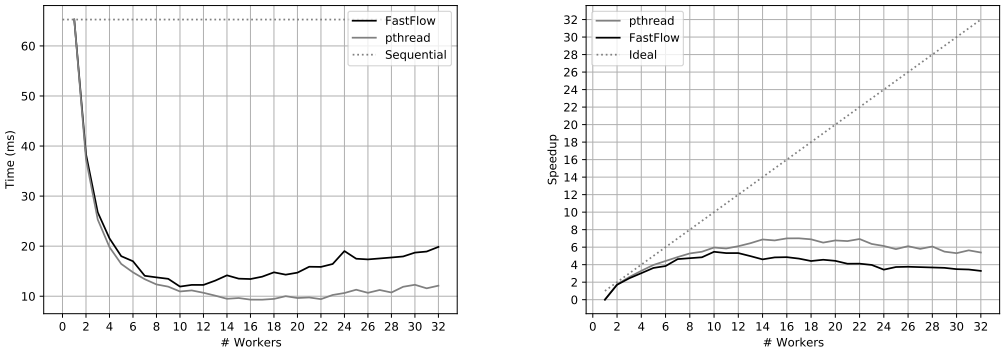


Figure 1: Performance and speedup using ER(10000, 0.2)

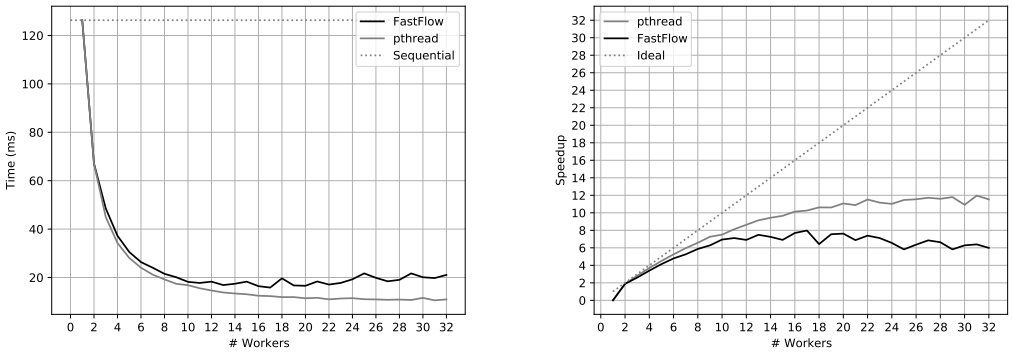


Figure 2: Performance and speedup using ER(10000, 0.4)

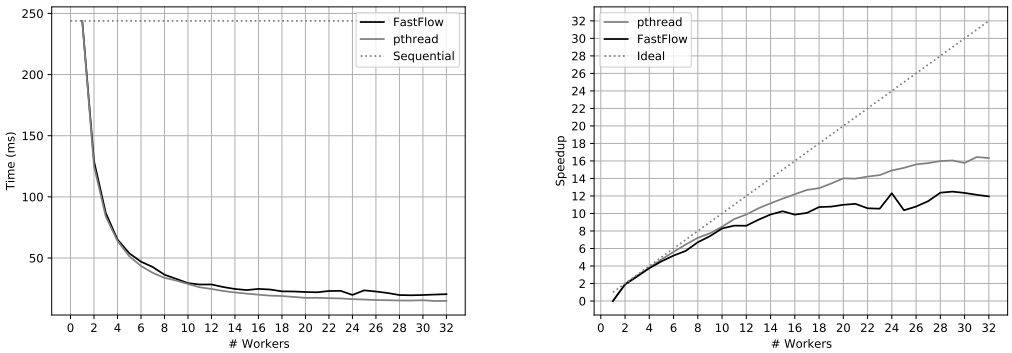


Figure 3: Performance and speedup using ER(10000, 0.8)

at the result obtained in Figure 3 and 5.

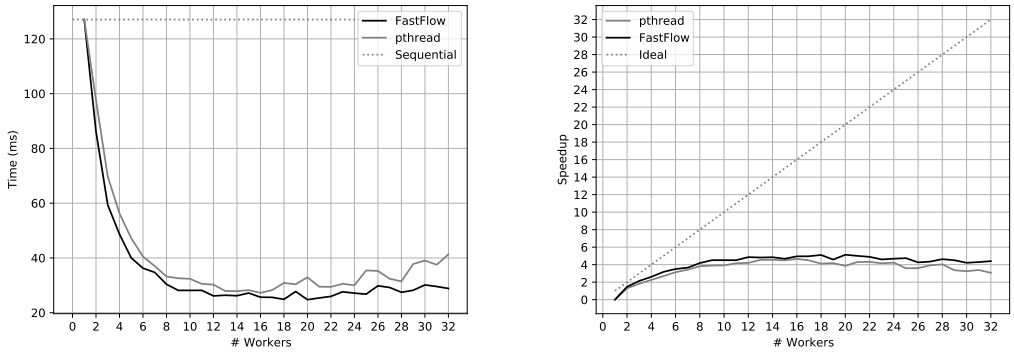


Figure 4: Performance and speedup using ER(100000, 0.002)

2.4 A real use case

To further evaluate the solutions, a real case is shown, using as input a real network of 149279 nodes and $\approx$ 10530774 edges, hence 0.00095 as density. The network was built as a network of interactions happened in the month of october 2020 in the /r/politics subreddit. The nodes represent users who have commented or written posts, while the edges from user A to user B indicates that the former has posted a comment to the latter. The results obtained are reported in terms of performance and speedup in the Figure 5.

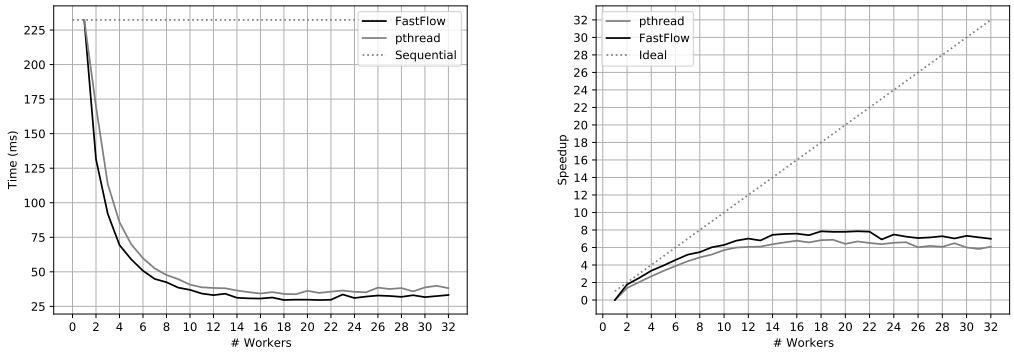


Figure 5: Real use case speedup

3 Conclusion

TODO: scrivere

References

[1] Vladimir Batagelj and Ulrik Brandes. “Efficient generation of large random networks”. In: *Phys. Rev. E* 71 (3 Mar. 2005), p. 036113. DOI: 10.1103/PhysRevE.71.036113. URL: https://link.aps.org/doi/10.1103/PhysRevE.71.036113.